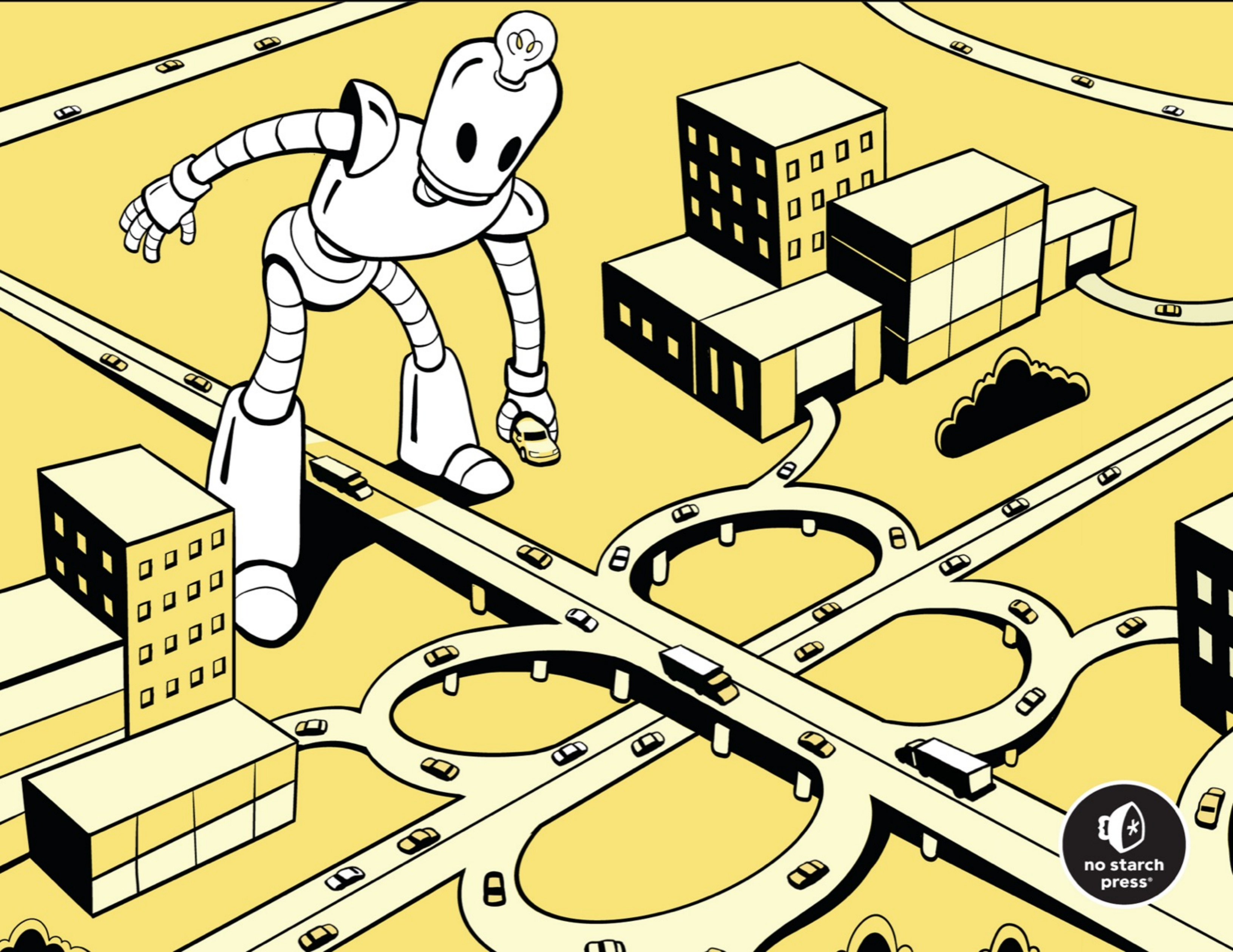
GRAPH ALGORITHMS THE FUN WAY

POWERFUL ALGORITHMS DECODED, NOT OVERSIMPLIFIED

JEREMY KUBICA



no starch press®

CONTENTS IN DETAIL

GRAPH ALGORITHMS THE FUN WAY

Powerful Algorithms Decoded, Not Oversimplified

by Jeremy Kubica

no starch press®

San Francisco

First printing

28 27 26 25 24   1 2 3 4 5

ISBN-13: 978-1-7185-0386-1 (print)
ISBN-13: 978-1-7185-0387-8 (ebook)

Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Production Editor: Sydney Cromwell
Developmental Editor: Abigail Schott-Rosenfield
Cover Illustrator: James L. Barry
Interior Design: Octopod Studios
Technical Reviewer: Daniel Zingaro
Copyeditor: Céline Parent
Proofreader: Debbie Greenberg

Library of Congress Cataloging-in-Publication Data

```
Name: Kubica, Jeremy, author.
Title: Graph algorithms the fun way / by Jeremy Kubica.
Description: San Francisco, CA : No Starch Press, [2025] | Includes index.
Identifiers: LCCN 2024017136 (print) | LCCN 2024017137 (ebook) | ISBN 9781718503861
  (print) | ISBN 9781718503878 (ebook)
Subjects: LCSH: Graph algorithms. | Python (Computer program language)
Classification: LCC QA166.245 .K83 2025 (print) | LCC QA166.245 (ebook) | DDC 518/.1—
  dc23/eng/20240628
LC record available at https://lccn.loc.gov/2024017136
LC ebook record available at https://lccn.loc.gov/2024017137
```

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com.

[E]

To Nathan and Julie

About the Author

Jeremy Kubica is an engineering director. He received a PhD in robotics from Carnegie Mellon University and a BS in computer science from Cornell University. He spent his graduate school years creating algorithms to detect killer asteroids (actually stopping them was, of course, left as “future work”). He is the author of multiple books designed to introduce people to computer science, including *Computational Fairy Tales* (2012), *The CS Detective* (No Starch Press, 2016), and *Data Structures the Fun Way* (No Starch Press, 2022).

About the Technical Reviewer

Dr. Daniel Zingaro is an award-winning associate professor of computer science at the University of Toronto Mississauga. He is well known for his uniquely interactive approach to teaching and internationally recognized for his expertise in active learning. He is the author of *Algorithmic Thinking*, 2nd edition (No Starch Press, 2024), and *Learn to Code by Solving Problems* (No Starch Press, 2021), and co-author of *Learn AI-Assisted Python Programming* (2023) and *Start Competitive Programming!* (2024).

ACKNOWLEDGMENTS

I would like to start by thanking the whole team at No Starch Press who helped make this book a reality: Bill Pollock, Jill Franklin, Abigail Schott-Rosenfield, Sabrina Plomitallo-González, Sydney Cromwell, Céline Parent, and Debbie Greenberg. A particular thank-you to my amazing editor, Abigail Schott-Rosenfield, for their excellent help, guidance, and suggestions throughout the process. I would also like to thank Carlos Bueno, who originally pointed me to the team at No Starch.

Thank you to Daniel Zingaro for his thorough and insightful technical review. His work was vital in improving both the accuracy and understandability of the material.

A tremendous thanks goes out to all the people who provided early thoughts and comments: Andrew Moore, Theodore Kim, Eleanor Rieffel, and Justin Carlson. Their suggestions helped guide my approach and the direction of the book.

I am grateful for the many great teachers, mentors, and colleagues who introduced me to the fascinating world of graph algorithms.

Thank you to my parents for their support and encouragement. Thank you to Regan, Nathan, and Julie for their support, encouragement, and patience.

INTRODUCTION

This book is an introduction to graphs and their algorithms for programmers who want to understand and apply them. Graphs are a type of data structure used throughout mathematics, computer science, and numerous other fields to model and solve a wide range of real-world problems. The structure of a graph allows us to represent connections or associations between items. Understanding this structure is critical to harnessing the power of graphs and using them efficiently.

Graph Algorithms the Fun Way grew out of the chapter on graphs in my previous book, *Data Structures the Fun Way* (No Starch Press, 2022), where I wrote, “We could devote an entire book to this single vastly impactful data structure.” Yet this book still only scratches the surface of the exciting and powerful world of graph algorithms, an area of study with a long history and ongoing research. A comprehensive coverage of all graph techniques and their relative advantages would require numerous volumes and be out of date the moment it was printed. Instead, this book is meant to serve as a foundation for people approaching this exciting field for the first time.

The book starts by introducing the components of graphs, then dives into exploring a variety of graph algorithms and how they apply to real-world problems. It is more than a cookbook of common algorithms. Its goal is to help readers understand the ideas behind the algorithms and build the intuitions to adapt the concepts covered here to techniques beyond this book.

Who Is This Book For?

This book is for programmers who want to learn more about graphs, graph algorithms, and the computational thinking behind such techniques. I assume no prior knowledge of graphs or graph algorithms. However, readers should have the kind of basic familiarity with Python that can be expected after an introductory course, book, or boot camp. They should be familiar with fundamental Python programming concepts, including basic data structures such as lists and dictionaries.

I hope this book will be useful to a wide range of audiences, not just programmers learning graph algorithms for the first time. The examples and metaphors used throughout the book are designed to provide an alternative way to view the topics from their standard mathematical definitions. Advanced students and experienced computer scientists may find a new perspective to understand particularly difficult or tricky topics.

Analogies and Examples

This book supplements formal descriptions and code with a range of real-world and absurd examples and analogies. The structure of graphs makes them perfect for illustrating algorithms with stories of adventurers searching labyrinths or planning vacations through unknown cities. The goal of these examples and analogies is twofold. First, they motivate the algorithms themselves and why we care about the problems they solve. Second, they provide an alternate approach to visualizing these problems that will help readers break free of technicalities and minutiae.

Language and Coding Conventions

I chose to present example code in Python due to the language's wide use and readability. However, aficionados of other languages need not fear, as the concepts behind the code are language-agnostic. Graph algorithms have been implemented in a wide range of languages, and all code examples in this book can be adapted beyond Python.

The code throughout the book uses common Python conventions. To make the code clearer, I use type hints, as in the following code block:

```
def is_edge(self, from_node: int, to_node: int) -> bool:
    return self.get_edge(from_node, to_node) is not None
```

The input arguments list the expected types, such as `int`, and the function definition describes the expected return type (`bool`).

The code in this book uses multiple core Python libraries. Since functions throughout a file often use the same library, individual code snippets do not explicitly include the `import` statements. Users implementing the code will need to make sure to import the relevant libraries. Where ambiguous, I identify the needed libraries in the code's text description.

Standard Python libraries used in this book include:

`csv`

`copy`

`itertools`

`math`

`queue`

`random`

`typing` (for `Union`)

The `typing` library in particular is needed for a number of code snippets, in order to support type hints for functions with multiple return values.

In addition, Appendix B defines a custom `PriorityQueue` class used in multiple examples, and Appendix C defines a minimal `UnionFind` data structure.

The code in this book is structured to stand alone as much as possible and requires only these core Python libraries. This means I sometimes don't take advantage of good existing libraries. For example, in Chapter 1 I represent a matrix as a list of lists instead of leveraging the `numpy` library optimized for matrix operations. I call out instances where existing libraries would be a good fit, but I leave their integration into the code as an exercise for the reader.

I've also made many of the implementations in the book more verbose than strictly necessary in order to focus on the computational ideas behind them. This means that individual implementations may be broken into extra stages to illustrate the computational concepts, structured in a way that matches the explanation, or otherwise may not be fully optimized. In addition, to keep the examples simple, I often leave out the basic validity checks that are vital for production programs. Treat the examples as illustrations of general concepts rather than code to use verbatim in your own projects.

Terminology and Definitions

Since graph algorithms have been studied in a variety of fields, multiple terms sometimes exist for the same underlying concept. For example, links in a graph are also commonly referred to as *edges* or *arcs*. I define each concept when it is introduced and note some of the alternative names that readers might find in other references.

In other cases, the same term is used differently within different fields. In particular, the definitions of several key terms have deviated over the years between formal graph theory in mathematics and computer science study. For example, in mathematics, a *path* through a graph cannot include repeated nodes, while in computer science it often can. Where definitions differ, I default to the common computer science usages and note the difference in the text.

How to Use This Book

This book is structured progressively, with later chapters building on earlier ones. Part I sets up the conceptual foundations on which later chapters rely:

Chapter 1: **Representing Graphs** Introduces the structure of graphs, discusses the graph representations of adjacency lists and adjacency matrices, and provides the implementations used throughout the rest of the book.

Chapter 2: **Neighbors and Neighborhoods** Covers the core concept of neighboring nodes, basic algorithms to build sets of neighbors, and some basic metrics for understanding the local connectivity around a node.

Chapter 3: **Paths Through Graphs** Discusses paths through graphs and introduces multiple representations including lists of nodes, lists of edges, and lists of back pointers.

Later sections are less interdependent but still call upon concepts in earlier chapters. Each is organized around a theme. Part II focuses on searches and shortest paths in a graph:

Chapter 4: Depth-First Search Introduces two implementations of depth-first search, a recursive approach and an iterative stack-based approach, and also discusses how search information can be encoded in a depth-first search tree.

Chapter 5: Breadth-First Search Explores breadth-first search, discusses its properties, and shows how we can use it to find the shortest paths through unweighted graphs.

Chapter 6: Solving Puzzles Shows how we can use graphs to encode puzzles and use the search algorithms from Chapters 4 and 5 to solve these puzzles.

Chapter 7: Shortest Paths Introduces three algorithms for finding shortest paths through weighted graphs: Dijkstra's algorithm, the Bellman-Ford algorithm, and the Floyd-Warshall algorithm.

Chapter 8: Heuristic-Guided Searches Describes two heuristic-based searches, heuristic greedy search and A* search, and shows how they can make use of heuristic information about how promising the nodes are.

Part III focuses on connectivity and ordering in graphs:

Chapter 9: Topological Sort Discusses the problem of sorting a graph's nodes in topological order and introduces two algorithms for this task: Kahn's algorithm and an extension of depth-first search.

Chapter 10: Minimum Spanning Trees Describes two algorithms for finding minimum spanning trees on graphs, Prim's algorithm and Kruskal's algorithm, and also shows how the ideas behind Kruskal's algorithm can be extended to problems such as generating solvable mazes or clustering spatial data.

Chapter 11: Bridges and Articulation Points Examines algorithms based on depth-first search for finding bridges and articulation points in graphs.

Chapter 12: **Strongly Connected Components** Explores Kosaraju-Sharir's algorithm to identify strongly connected components in directed graphs.

Chapter 13: **Random Walks** Introduces into random walks on graphs and discusses the concept of Markov chains, then shows how to implement random walk behavior on graphs and estimate the underlying graphs from observed data.

Part IV introduces the concept of flow within graphs and uses it to solve a particular matching problem:

Chapter 14: **Max-Flow Algorithms** Defines the concepts of flow through a graph and the max-flow problem, introduces an extended version of the graph data structure to support this problem, and describes the Ford-Fulkerson and Edmond-Karp algorithms for solving the maximum-flow problem.

Chapter 15: **Bipartite Graph Matching** Introduces the task of matching in graphs and the concept of bipartite graphs before focusing on the specialization of matching within bipartite graphs. We show how to use maximum-flow algorithms to solve one variety of the matching problem on bipartite graphs.

Part V covers various node assignment and path planning problems through graphs:

Chapter 16: **Graph Coloring** Introduces the problem of assigning colors to graph nodes such that no two neighbors share a color and considers a range of algorithms to solve this problem.

Chapter 17: **Cliques, Independent Sets, and Vertex Covers** Introduces algorithms for three computationally challenging assignment problems: finding a maximum clique, finding a maximum independent set, and finding a minimum vertex cover.

Chapter 18: **Tours Through Graphs** Considers three path-planning problems: finding paths that visit each node exactly once, finding paths that visit each node exactly once while minimizing the edge weights traversed, and finding paths that cross each edge exactly once. We describe why the first two problems are difficult, but there exists an efficient solution for the third.

The appendices provide additional functions and data structures that are helpful for implementing the algorithms in this book:

Appendix A Describes functions for programmatically creating graphs, including loading them from files.

Appendix B Defines the modifiable priority queue data structure used in algorithms throughout the book.

Appendix C Introduces a minimal `UnionFind` data structure necessary to implement some of the algorithms in Chapter 10

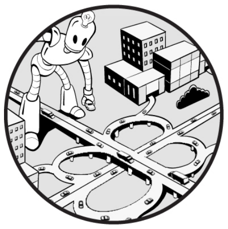
Throughout the book, the reader should focus on the questions *How?* and *Why? How* does this real-world problem map onto a graph formulation? *Why* does a certain approach help us compute the solution? *How* does an algorithm use the graph's structure to solve the problem? *Why* do we care about this problem? *How* do these algorithms apply to different problems? *Why* is the author using that ridiculous analogy? Understanding the answers to these questions will provide the foundation you need to effectively use existing algorithms and develop your own techniques in the future.

PART I

GRAPH BASICS

1

REPRESENTING GRAPHS



A *graph* is an abstract data type that can be implemented with a variety of data structures. This chapter introduces the fundamental components of a graph, nodes and edges, then shows how to build the two most common graph representations: adjacency lists and adjacency matrices. Understanding the structure and composition of graphs is critical to harnessing their power and designing algorithms to use them efficiently.

To implement the graphs, we define the `Edge`, `Node`, and `Graph` classes upon which almost every algorithm in this book relies. We discuss what information the classes store and provide functions for interacting with these objects. We also discuss the trade-offs involved in different implementations, along with possible alternatives and hybrids.

Graph Structure

A graph consists of two components: nodes and edges. A *node* (also called a *vertex*) represents a location or item within the graph. Nodes are often used to model concrete entities like people, cities, or computers. *Edges* (also called *links* or *arcs*) link together pairs of nodes, defining the relative connections within the graph. They are used to represent both concrete items, such as roads between cities, and abstract concepts, such as the friendship of two people.

Figure 1-1 shows an example graph with five nodes and seven edges. We use the standard graphical representation with nodes as circles and edges as lines connecting two circles. Many figures in the book also include a label within each circle to identify individual nodes.

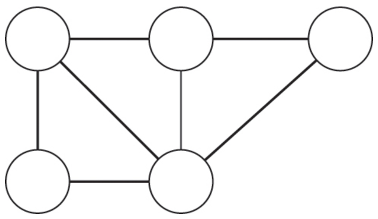


Figure 1-1: A graph with five nodes and seven edges

To describe graphs in mathematical notation, we use V to represent the set of nodes and E to represent the set of edges. The number of nodes and edges are represented using the mathematical notation for the size of a set, meaning the number of nodes is $|V|$ and the number of edges is $|E|$.

Using these simple components, we can represent a surprisingly large number of real-world systems and answer a range of real-world practical questions. For example, graphs allow us to model the following scenarios:

Transportation networks Nodes are cities, and edges represent paths. We can compute the shortest path between points or look for single points of failure that would cut off one part of the network from another.

Mazes Nodes are intersections, and edges are the halls linking them. We can search for paths through the maze.

Educational topics Each node is a topic, and the edges link two related topics. We can sort topics by prerequisite knowledge.

Social networks Nodes are people, and edges are their friendship connections. We can model information flow through the network to predict how rumors will travel.

We can further increase the power of our graphs by allowing the edges to provide additional information like directionality and weight, as discussed in the following subsections and in later chapters.

Weighted Edges

In almost every real-world transportation network, there is a different *cost* for traversing different edges. We might measure this cost in distance or the price of gas, for example; either way, it is cheaper to drive from Boston to New York than it is to drive from Boston to Los Angeles. Cost measures can also be more complicated, such as factoring in the stress of navigating a narrow, winding road through the mountains. Alternatively, for some problems, we want to consider the inverse of a cost, such as the strength of a connection between nodes or the benefit of following a particular edge. Accounting for the edge's costs or benefits is critical in accurately solving many graph problems, such as finding the shortest (or least scary) path between two locations.

Weighted edges capture not only the links between nodes but also the costs or benefits of traversing those links. For some applications, the weightings are obvious and easy to obtain, such as the distance between cities. For example, we could assign an edge between Pittsburgh and Cleveland the weighting of 133 to reflect 133 miles of highway between the two cities. In other contexts, we might use weightings to represent more abstract concepts like the strength of a friendship. A weighting of 10 for the connection between Tina and Bob could indicate that the two are best friends, while a weighting of 1 for Tina's connection with Alice would indicate that they are mere acquaintances. It is usually obvious from context whether a weight represents a cost or benefit.

We call graphs with weighted edges *weighted graphs* and those without such edges *unweighted graphs*. We visually represent edge weights as numeric labels adjacent to the line representing the edge itself. In Figure 1-2, for example, three of the edges have weight 1.0, one edge has weight 2.0, and the remaining three have weight 3.0.

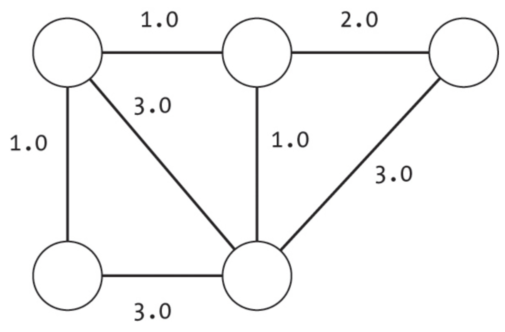


Figure 1-2: A weighted graph

If necessary, we can use weighted graphs to model unweighted edges by either using a single weight for all edges, such as 1.0, or ignoring the weight attribute in our algorithms.

Directed Edges

In some systems, connections between nodes are not symmetric. For example, consider the pipe from a building's water heater to the kitchen faucet. Unless the plumbing is very broken, it is not possible for the water to flow into the faucet and back to the water heater.

Directed edges indicate such directionality in the connection between two nodes. We use terminology that mirrors real-world transportation networks: the node from which a directed edge begins is the *origin* or `from_node`, while the node to which the directed edge points is the *destination* or `to_node`.

While directed edges can represent physical directionality, such as a one-way road, we can also use them to model more abstract directionality, such as prerequisite courses in an educational institution. If each node is a course, a directed edge might indicate that we need to take Introduction to Computer Science before Advanced Graph Algorithms, as shown in Figure 1-3.

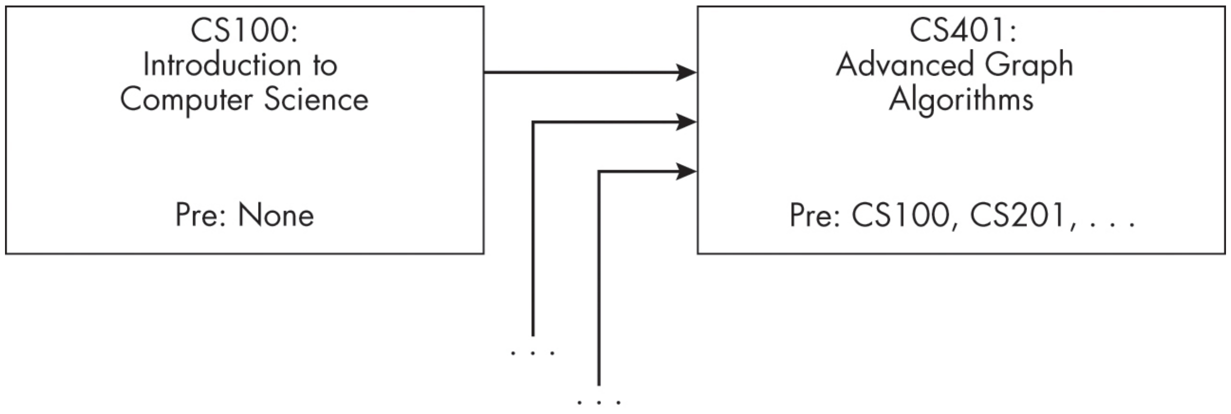


Figure 1-3: Arrows showing the directionality of course prerequisites

We call graphs with directed edges *directed graphs*. Graphs without directed edges (such as those in Figures 1-1 and 1-2) are known as *undirected graphs* with *undirected edges*.

We can use directed edges to extend our earlier social network model. While it would be ideal if all friendships were reciprocated, this is sadly not always the case. Tina and Bob might call each other best friends. However, while Alice considers Tina a dear friend, Tina thinks of Alice simply as an acquaintance from work.

Figure 1-4 shows an example graph with directed edges, where each directed edge is shown as a single arrow indicating its direction.

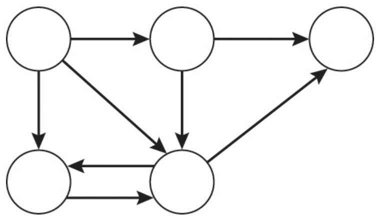


Figure 1-4: A directed graph

We can represent symmetric or undirected relationships between nodes in a directed graph by using pairs of directed edges, one in each direction, as shown between the bottom two nodes in Figure 1-4. This allows us to model systems with a mix of directed and undirected relationships. For example, real-world transportation networks contain a mix of one-way and two-way roads, and many social networks contain mutual friendships. By using a directed graph and corresponding edge pairs, we can fully model these systems.

Edges with Both Weight and Direction

To maximize the representational power of the graph, we can combine the use of weighted edges and directed edges, as shown in Figure 1-5. This representation allows the graph to capture both the directionality and the cost versus benefit of each connection.

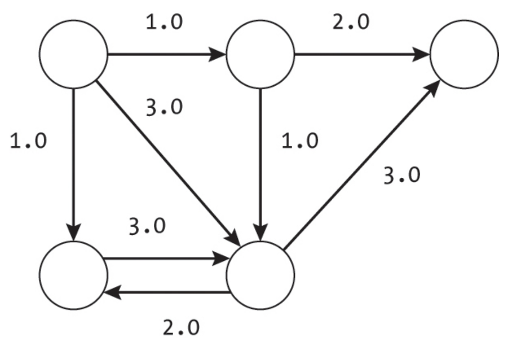


Figure 1-5: A directed and weighted graph

We must specify a separate weight for each directed edge between two nodes, but, as with the bottom pair of nodes in Figure 1-5, these weights need not be equal. When modeling the cost of traversing a road, for example, we might choose a much higher cost for the uphill direction than for the downhill one. Depending on the application, such as planning a cycling trip, the cost of uphill roads might be significantly higher. Similarly, Tina and Alice assign different levels of importance to their friendship.

Throughout this book, we will use graph implementations that support both weighting and directionality. If necessary, these data structures can still be used to store simpler graphs without weighted or directed edges. While this generality adds some small complexity to the data structure and may potentially add overhead to algorithms that do not use all the information, this approach results in a flexible data structure that can be used by a range of algorithms.

The Adjacency List Representation

The graph representation used throughout the majority of this book is the *adjacency list representation*, which stores a graph's structure as a set of individual lists of neighbors for each node. This allows us to mirror real-world phenomena where each node maintains information on its local connections, such as in a social network where individuals maintain contact information for their immediate friends.

There are a variety of ways to implement the adjacency list representation. Nodes and edges can be represented implicitly through associations or explicitly as data structures in their own right. In the simplest implementation, we could implicitly store a graph using a single list of lists, where each node has a numeric ID and each entry in the list corresponds to a given node's neighbors. For example, consider the following line of code:

```
g: list = [[1,3,4], [0,2,4], [1,4], [0,4], [0,1,2,3]]
```

This list of lists `g` represents the undirected, unweighted graph with five nodes and seven edges shown in Figure 1-6. The first entry in the list on the right-hand side of the figure indicates that node 0 has three neighbors: nodes 1, 3, and 4. Each undirected edge is represented in two different neighbor lists, one for the node on each end.

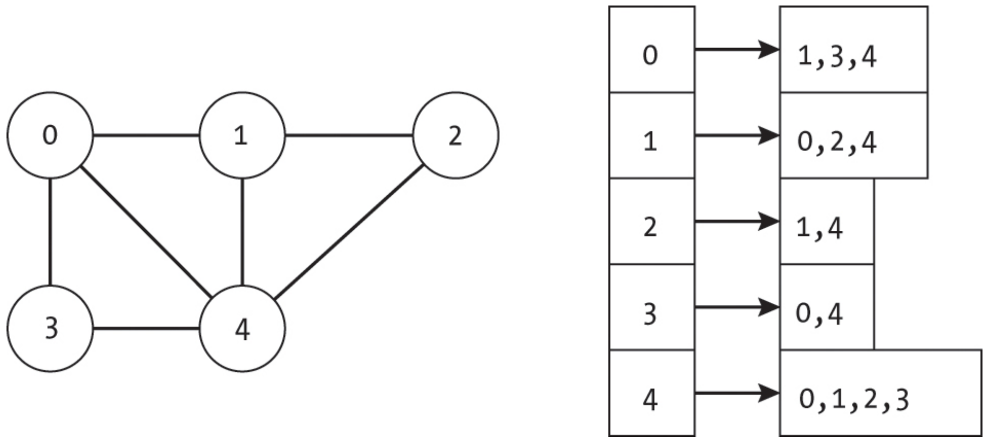


Figure 1-6: A graph (left) and its adjacency list representation (right)

Alternatively, we could create a `Node` data structure that contains not only the adjacency list but also supplementary information. This might include a label to identify the node, a Boolean indicating whether the node has been processed, or an integer indicating the time at which we first saw the node. We could also make the representation more detailed by defining an `Edge` data structure with information about directionality and weight, then storing a list of adjacent edges within each node.

The optimal representation for any given use case depends highly on the purpose of the data structure. For large graphs in limited memory environments, a sparser representation like the list of lists in Figure 1-6 might be ideal. However, when modeling more complex problems, such as directional traffic flows over different road conditions, we may need to store more information.

The rest of this section introduces a highly structured adjacency list representation that prioritizes generality and understandability so that we can reuse it throughout the different algorithms in this book. We use both `Edge` and `Node` objects to facilitate the storage of a variety of auxiliary information for both pieces. Each `Node` object maintains its own list of adjacent `Edge` objects that store the information needed to encode weights and directionality.

An important aspect of this implementation is that each node has a unique *numeric index* that indicates its location within the overall `Graph` data structure. Throughout this book, we will refer to a node and its index relatively interchangeably. For example, we refer to the node at index 0 as node 0. We might also say that a function returns a list of nodes visited when the implementation returns a list of indices.

As we will see throughout the book, this graph representation lends itself to algorithms that traverse the graph node by node, such as the majority of algorithms in this book. While this implementation is effective for illustrating a range of graph algorithms, the reader may want to use more memory-efficient or computationally efficient representations that are better optimized for specific problems.

Edges

We define an `Edge` object as little more than a container that stores information for directed and weighted edges:

`to_node (int)` Stores the node index of the edge's destination

`from_node (int)` Stores the node index of the edge's origin

`weight (float)` Stores the edge's weight. Where necessary for a particular use case, we'll use a value of 1 to represent unweighted edges

As shown in Figure 1-7, the `Edge` object stores all the information we may need to work with an edge independent from other classes. The inclusion of `from_node` in the `Edge` class may seem redundant because we are storing the edges in a list at each node and thus can retrieve that information from the node. However, explicitly storing this information will enable us to use algorithms later in the book that work with sets of edges independent of the nodes.

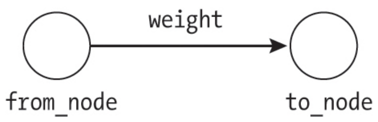


Figure 1-7: The information contained in the `Edge` *class*

Using the attributes of the `Edge` class, we define a constructor that copies in the data:

```
class Edge:
    def __init__(self, from_node: int, to_node: int, weigh
t: float):
        self.from_node: int = from_node
        self.to_node: int = to_node
        self.weight: float = weight
```

Since the `Edge` class is used primarily for storage, it does not include any additional functions. Attributes are accessed directly. We can model undirected edges in the graph by storing a pair of directed edges from each node. That is, an undirected edge between nodes A and B would materialize as a directed edge from node A to node B and a directed edge from node B to node A. While this doubles the number of edges stored in an undirected graph, it emphasizes flexibility and allows us to use the same class for a range of applications.

The `Edge` class illustrates how we use the numeric node identifier throughout the code. Instead of storing an explicit link to the node, `to_node` and `from_node`, we store the integer index of the corresponding nodes. When we need to access additional attributes within the node, we use these indices to directly look up the `Node` object from the graph's `nodes` list.

Nodes

We define a `Node` object to both store the information relevant to the node and provide basic operations on that information. Each `Node` object contains the following attributes:

`index` (`int`) Stores the numeric index of the node

`edges` (`dict`) Stores the edges out of the node

`label` (`int`, `string`, or `object`) An optional label used to identify the node or mark its current state

Instead of using a list to store the edges, we use a dictionary keyed by the destination node's integer index and with `Edge` objects as values. This representation allows us to efficiently ask questions like "Is there an edge between node A and node B?" without iterating through all of node A's edges.

We can visualize the `Node` object as part of a high school social network. Each student is represented as a node with their student ID number as their index. The `edges` dictionary represents that student's personal list of friends. As noted earlier, each `Edge` object can be directional and weighted to fully capture the complexities of high school alliances and feuds. The `label` string can be used to store information about each student, such as whether they have heard the latest rumor.

As with the `Graph` and `Edge` classes, we define a constructor to set up the initial state of the node, as well as a series of helper functions:

```
class Node:
    def __init__(self, index: int, label=None):
        self.index: int = index
        self.edges: dict = {}
        self.label = label

    def num_edges(self) -> int:
        return len(self.edges)

    def get_edge(self, neighbor: int) -> Union[Edge, None]:
        if neighbor in self.edges:
            return self.edges[neighbor]
        return None
    def add_edge(self, neighbor: int, weight: float):
        self.edges[neighbor] = Edge(self.index, neighbor, w
eight)

    def remove_edge(self, neighbor: int):
        if neighbor in self.edges:
            del self.edges[neighbor]

    def get_edge_list(self) -> list:
        return list(self.edges.values())

    def get_sorted_edge_list(self) -> list:
        result = []
        neighbors = (list)(self.edges.keys())
        neighbors.sort()

        for n in neighbors:
            result.append(self.edges[n])
        return result
```

The constructor sets the integer index (`index`) to the given value. It creates an empty dictionary (`self.edges = {}`) to store future edges and starts with an empty label (`self.label = None`).

The `Node` class contains a variety of helper functions to facilitate working with it. When implementing the graph, the `Edge` class needs to be defined before the `Node` class. We also need to import `Union` from Python's `typing` library (by adding `from typing import Union` at the start of the file) to support the optional type hints used in the example code.

The first two functions provide access to the node's edges. The `num_edges()` function returns the number of edges. The `get_edge()` function returns a given edge, or `None` if no such edge exists. This allows us to combine lookups and existence checks into a single function.

The next two functions modify the node's connections. The `add_edge()` function takes a destination index and weight, then creates and inserts the corresponding `Edge` object. It overwrites the existing edge if the neighbor's index already appears in the dictionary, allowing us to update edge weights. The function `remove_edge()` drops an edge from the dictionary if it exists.

The final two functions are convenience functions for returning the node's edges as lists. The function `get_edge_list()` returns the edges in their dictionary ordering and is used whenever an algorithm needs access to the list. The function `get_sorted_edge_list()` returns the edges in order of increasing neighbor index and is primarily used throughout this book to provide a consistent ordering for examples.

While these functions use dictionaries to store a node's edges (indexed by the destination node), it's possible to adapt each function to store the node's edges as a list instead. Compact lists of just the outgoing edges prioritize memory usage over the time it takes to look up a specific edge. In contrast, to prioritize lookup speed for specific edges, each node could store a list of length $|V|$ with a space for each possible edge and store `None` for edges that do not occur. The dictionary-based approach balances these two competing aspects.

The Graph Class

The `Graph` class used in the majority of this book consists of a list of `Node` objects and some utility information that simplifies common computations:

`nodes (list)` Stores the graph's nodes

`num_nodes (int)` Stores the total number of nodes in the graph

`undirected (bool)` Indicates whether this is a directed or undirected graph

The `num_nodes` and `undirected` values can be computed from the structure of the graph itself but are stored for convenience.

We always store directed edges and use the Boolean `undirected` to modify behavior when working with directed and undirected graphs. Most notably, as demonstrated later in the section "Accessing, Building, and Modifying the Graph," we'll use `undirected` to insert a pair of directed edges when the graph itself is undirected. Other common implementations either use separate functions, such as an `insert_undirected_edge()` function, or create entirely different implementations for directed and undirected graphs. Again, we are prioritizing generality of the data structure over pure optimization.

Given this information, we can create a simple constructor for building a graph with a given number of nodes and no edges:

```
class Graph:
    def __init__(self, num_nodes: int, undirected: bool=Fal
se):
        self.num_nodes: int = num_nodes
        self.undirected: bool = undirected
        self.nodes: list = [Node(j) for j in range(num_node
s)]
```

The constructor initializes the convenience variables, then creates a list of `Node` objects. The function does not create any edges. Implicit in this implementation is the existence of a unique numeric identifier for each node that corresponds to its location in the graph's `nodes` list.

The `Graph` class also includes a variety of functions to create, search, access, and otherwise process graphs. Instead of providing a huge block of code for all graph functions in this section, we'll introduce the general functions throughout this section.

Accessing, Building, and Modifying the Graph

To facilitate accessing edges, we next define a series of helper functions within the `Graph` class:

```
def get_edge(self, from_node: int, to_node: int) -> Union[E
dge, None]:
    if from_node < 0 or from_node >= self.num_nodes:
        raise IndexError
    if to_node < 0 or to_node >= self.num_nodes:
        raise IndexError
    return self.nodes[from_node].get_edge(to_node)

def is_edge(self, from_node: int, to_node: int) -> bool:
    return self.get_edge(from_node, to_node) is not None

def make_edge_list(self) -> list:
    all_edges: list = []
    for node in self.nodes:
        for edge in node.edges.values():
            all_edges.append(edge)
    return all_edges
```

The `get_edge()` function takes an origin index and a destination index and returns the corresponding edge, if it exists. It performs basic bounds checking for validity, then uses the origin node's corresponding `get_edge()` function to retrieve an edge if one exists and return `None` otherwise. The `is_edge()` function simply checks for the existence of any edge with the given origin and destination. Finally, the `make_edge_list()` function dynamically constructs and returns a list of all edges in the graph.

The `Graph` class's constructor allocates a given number of nodes but does not create any edges. Obviously, this does not produce a remotely useful graph. To model any interesting problem, our graph needs to include both nodes and edges. We therefore add a few additional functions for creating and modifying our adjacency graph representation. First, in the `Graph` class, we provide the ability to add and remove edges given the indices of the origin and destination nodes:

```
def insert_edge(self, from_node: int, to_node: int, weight:
float):
 ❶ if from_node < 0 or from_node >= self.num_nodes:
        raise IndexError
    if to_node < 0 or to_node >= self.num_nodes:
        raise IndexError

    self.nodes[from_node].add_edge(to_node, weight)
 ❷ if self.undirected:
        self.nodes[to_node].add_edge(from_node, weight)

def remove_edge(self, from_node: int, to_node: int):
 ❸ if from_node < 0 or from_node >= self.num_nodes:
        raise IndexError
    if to_node < 0 or to_node >= self.num_nodes:
        raise IndexError

    self.nodes[from_node].remove_edge(to_node)
 ❹ if self.undirected:
        self.nodes[to_node].remove_edge(from_node)
```

Both the `insert_edge()` and `remove_edge()` functions follow the same flow: they start by checking that both the origin and destination indices correspond to nodes included in the graph ❶ ❸. If the node indices are invalid, the functions raise an `IndexError`.

If the indices are valid, the functions modify the adjacency list of the origin node. The insertion function uses the node's `add_edge()` function. The removal function uses the node's `remove_edge()` function. Because we are using a single class to represent both directed and undirected graphs, the functions also need to add ❷ or remove ❹ the corresponding inverse edge in the case of an undirected graph.

We can use these functions together to dynamically create graphs. For example, we can use the following code to create a directed graph with five nodes and then insert eight weighted edges:

```
g: Graph = Graph(5, undirected=False)
g.insert_edge(0, 1, 1.0)
g.insert_edge(0, 3, 1.0)
g.insert_edge(0, 4, 3.0)
g.insert_edge(1, 2, 2.0)
g.insert_edge(1, 4, 1.0)
g.insert_edge(3, 4, 3.0)
g.insert_edge(4, 2, 3.0)
g.insert_edge(4, 3, 3.0)
```

This generates the graph shown in Figure 1-8.

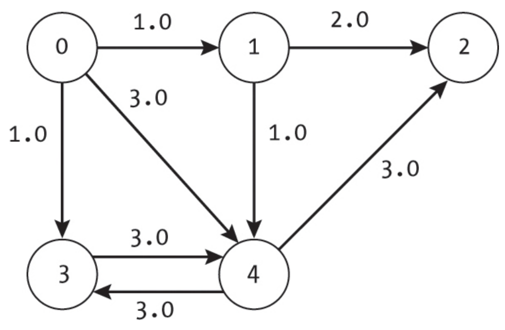


Figure 1-8: A directed and weighted graph with nodes labeled by their indices

While we provide the ability to pre-allocate nodes in the constructor, for some algorithms, we need to insert new nodes as we explore a graph. To facilitate this, we also provide a function for inserting new nodes:

```
def insert_node(self, label=None) -> Node:
    new_node: Node = Node(self.num_nodes, label=label)
    self.nodes.append(new_node)
    self.num_nodes += 1
    return new_node
```

The `insert_node()` function creates a new node and automatically assigns the identification number to the next index. The node is then appended to the `nodes` list, the count of nodes is incremented, and the new node is returned.

While the functions in this section provide the building blocks for constructing graphs, it would be tedious to manually specify graphs with a long sequence of `insert_node()` and `insert_edge()` calls. Appendix A examines a few sample algorithms that build off these initial functions to programmatically create graphs from files or common problem specifications.

Copying the Graph

Finally, we also define a helper function within the `Graph` class that produces a new copy of the graph for use with algorithms that modify the graph:

```
def make_copy(self):
    g2: Graph = Graph(self.num_nodes, undirected=self.undir
ected)
    for node in self.nodes:
      ❶ g2.nodes[node.index].label = node.label
        for edge in node.edges.values():
          ❷ g2.insert_edge(edge.from_node, edge.to_node, ed
ge.weight)
    return g2
```

The `make_copy()` code starts by creating a new `Graph` object (`g2`) with the same number of nodes and undirected setting as the current graph. It then uses two nested `for` loops to iterate through each node and its outgoing edges. For each node, it copies the label ❶. For each edge, it inserts an equivalent edge into `g2` ❷.

Copying the graph will allow us to employ algorithms that destructively modify the graph. For example, in Chapter 16 we will introduce an algorithm for assigning colors that iteratively removes nodes from the graph.

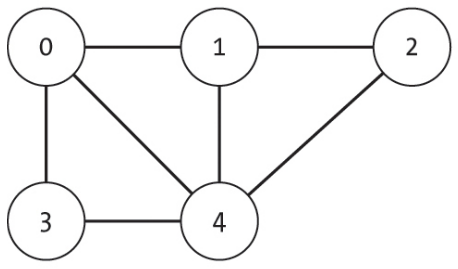
The Adjacency Matrix Representation

Another powerful graph representation is the *adjacency matrix*. While we will primarily rely on the previous adjacency list representation for most of the algorithms in this book, the adjacency matrix representation is important for an entire class of mathematically based algorithms. Many algorithms can be described or analyzed via matrix operations. We will make use of the matrix formulation in Chapter 13 when considering random walks on graphs.

The *adjacency matrix* representation of a graph uses a single matrix to indicate the edge weights between each pair of nodes. The value in row *i*, column *j*, represents the weight of the edge from node *i* to node *j*. A value of 0 indicates that no such edge exists. Represented as a list of lists, the following matrix would create an undirected, unweighted graph with five nodes and seven edges:

```
g = [[0, 1, 0, 1, 1],
     [1, 0, 1, 0, 1],
     [0, 1, 0, 0, 1],
     [1, 0, 0, 0, 1],
     [1, 1, 1, 1, 0]]
```

This corresponds to the graph shown in Figure 1-9, where node 0's three connections are represented by the corresponding nonzero entries in the matrix.



$$\begin{array}{c|ccccc|}
 & 0 & 1 & 2 & 3 & 4 \\
\hline
0 & 0 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 \\
2 & 0 & 1 & 0 & 0 & 1 \\
3 & 1 & 0 & 0 & 0 & 1 \\
4 & 1 & 1 & 1 & 1 & 0 \\
\hline
\end{array}$$

Figure 1-9: A graph (left) and its adjacency matrix representation (right)

The matrix of connection can use any setting of values. Floating-point entries can be used to represent weighted edges. Undirected edges are represented by a matching pair of values, making undirected graphs symmetric.

To create and store adjacency graphs, we'll use the basic `GraphMatrix` class presented in this section. As with the `Graph` class, we optimize the representation for understandability rather than computational cost or memory usage.

Our `GraphMatrix` class contains three pieces of information:

`connections` (`list` of `list`) Stores the adjacency matrix

`num_nodes` (`int`) Stores the total number of nodes in the graph

`undirected` (`bool`) Indicates whether this is a directed or undirected graph

As with the `Graph` data structure, we allow `GraphMatrix` to represent both directed and undirected graphs. We use the `undirected` attribute to specify which type of edges are included. We define a simple constructor for building a graph with a given number of nodes and no edges:

```
class GraphMatrix:
    def __init__(self, num_nodes: int, undirected: bool=Fal
se):
        self.num_nodes: int = num_nodes
        self.undirected: bool = undirected
        self.connections = [[0.0] * num_nodes for _ in rang
e(num_nodes)]
```

The code initializes every entry in `connections` to `0`, creating a graph without any edges.

We also define a getter function to retrieve the weight of a connection between two nodes:

```
def get_edge(self, from_node: int, to_node: int) -> float:
    if from_node < 0 or from_node >= self.num_nodes:
        raise IndexError
    if to_node < 0 or to_node >= self.num_nodes:
        raise IndexError
    return self.connections[from_node][to_node]
```

The code checks that both the origin and destination indices are valid. If so, the core returns the corresponding floating-point value from the array.

While we store the adjacency matrix in a list of lists to keep this illustration simple, it is often preferable to use a representation optimized for matrix operations, such as that provided in the popular numpy package. Such numerical packages will be faster and provide a range of helper functions. We leave the implementation of a `GraphMatrix` in numpy or a similar mathematical package as an exercise for the reader.

Unlike the `Graph` class, a new `GraphMatrix` object pre-allocates all the space to store edge information in the main `connections` matrix. We can directly set entries in this matrix to add or remove edges:

```
def set_edge(self, from_node: int, to_node: int, weight: fl
oat):
 ❶ if from_node < 0 or from_node >= self.num_nodes:
        raise IndexError
    if to_node < 0 or to_node >= self.num_nodes:
        raise IndexError

 ❷ self.connections[from_node][to_node] = weight
 ❸ if self.undirected:
        self.connections[to_node][from_node] = weight
```

The code checks that both the origin and destination indices are valid and, if not, raises an error ❶. If the indices are valid, the function sets the matrix entry corresponding to this edge ❷. If the graph is undirected, the function modifies the symmetric entry in the matrix ❸.

We can use the `set_edge()` function to add, remove, or modify edges. We add new edges by setting the entries to nonzero weights. If an edge already exists between those two nodes, the function updates the weight. We remove edges by setting an entry to 0. For example, we could create the graph in Figure 1-8 as:

```
g: GraphMatrix = GraphMatrix(5, undirected=False)
g.set_edge(0, 1, 1.0)
g.set_edge(0, 3, 1.0)
g.set_edge(0, 4, 3.0)
g.set_edge(1, 2, 2.0)
g.set_edge(1, 4, 1.0)
g.set_edge(3, 4, 3.0)
g.set_edge(4, 2, 3.0)
g.set_edge(4, 3, 3.0)
```

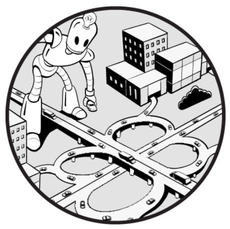
Why This Matters

The graph structure and its underlying implementations form the foundation of all algorithms in this book and drive their development. Deciding which representation to use requires us to consider trade-offs in memory usage, computational efficiency, and complexity, depending on what makes sense for the task at hand. In cases where we want only to iterate over a node's immediate neighbors, for example, the best choice might be an adjacency list representation because we can access the neighbor lists independently. In contrast, for algorithms that are more mathematical, we might prefer a matrix representation that can make use of existing mathematical libraries.

The goal of introducing the implementations in this chapter is not to provide a single canonical approach, but rather to introduce the different ways of thinking about graphs and the different trade-offs inherent in their representations. There are a huge variety of hybrid approaches or further adaptations we can make to the implementations presented in this chapter to optimize a graph representation to the problem of interest.

In the following chapters, we'll introduce a series of problems we can solve using graphs, building on the concepts and code introduced in this chapter as we do so. For each problem, we present a few practical algorithms that can readily be applied in real-world situations. We'll start in the next chapter by introducing the concept of neighboring nodes and using algorithms to construct neighborhoods.

2
NEIGHBORS AND NEIGHBORHOODS



Almost every algorithm in this book requires interacting with a node's *neighbors*. The idea of a neighbor is intuitively quite familiar; in an undirected graph, the neighbors of a given node are those nodes with which it shares an edge. The terminology for neighbors is a little more complex in directed graphs, where there are different types of neighbors depending on whether the edge is incoming or outgoing.

Identifying the set of neighbors for a given node is a foundational step in most graph algorithms, such as searching for paths through a new graph, and many real-world tasks. When planning a trip through a transportation network, for example, we might ask which cities we can reach directly from the current one.

This chapter introduces the formal definition of neighbors and presents some basic functions we'll use throughout the book. It also introduces two example neighbor-based metrics: a node's degree and its clustering coefficient. These metrics provide insights about a node's neighborhood that help us analyze the characteristics of graphs. A node's degree tells us its number of connections and its clustering coefficient tells us about the interconnectedness of its neighbors.

Neighbors in Undirected Graphs

Many metrics and algorithms require determining the set of nodes in immediate proximity to a given node *v*. In an undirected graph, the *neighbors* of node *v* are all nodes connected to *v* by an edge. Figure 2-1 shows an example graph and lists the neighbors for each node. Node 0 has three neighbors (1, 3, and 4), while node 3 has only one neighbor (0).

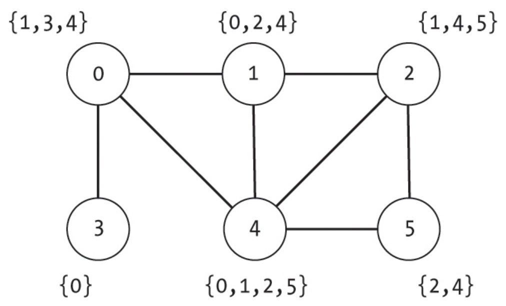


Figure 2-1: An undirected graph with neighbors listed for each node

We can add a short helper function to our `Node` class to compute the set of neighbors in an undirected graph, as shown in Listing 2-1.

```
def get_neighbors(self) -> set:
    neighbors: set = set()
    for edge in self.edges.values():
        neighbors.add(edge.to_node)
    return neighbors
```

Listing 2-1: Determining the set of neighboring nodes in an undirected graph

The `get_neighbors()` function in Listing 2-1 creates an empty `set` data structure, then iterates through each of the node's edges and adds the corresponding neighbor to the set.

Consider a graph representing a social network, where each person is a node and an undirected edge between node *v* and node *u* indicates those two people are friends. We might use a node's neighbors to compose a guest list for a party, or to model how a rumor spreads within the network.

As another application of determining a node's neighbors, consider the age-old question, "Who was in a movie with that particular star?" This is a question we will consider in more detail later in the chapter. We can build a co-occurrence graph that represents which actors appear together in a movie. Each node represents a single person. An edge indicates that two people have appeared in the same movie together. Since this relationship is always symmetric, we use an undirected graph to model these co-occurrences.

Neighbors in Directed Graphs

In directed graphs, we could consider several types of nodes to be neighbors: the nodes at the end of outgoing edges from *v*, the nodes at the beginning of incoming edges to *v*, or the nodes on either side of a directed edge. To resolve this ambiguity, we divide neighbors for such graphs into two main types. *In-neighbors* are all nodes that have edges with *v* as the destination; in other words, the edge is incoming from node *v*'s perspective. In a directed social network where friendships are not symmetrical, for example, *v*'s in-neighbors are the friends who would tell them the latest gossip. *Out-neighbors* are all nodes to which *v* has an outgoing edge, representing the friends to which *v* would pass gossip.

The code we add to `Node` class for computing out-neighbors in a directed graph is identical to the code presented in Listing 2-1 for undirected graphs, except for the name of the function, as shown in Listing 2-2.

```
def get_out_neighbors(self) -> set:
    neighbors: set = set()
    for edge in self.edges.values():
        neighbors.add(edge.to_node)
    return neighbors
```

Listing 2-2: Determining the set of out-neighbors in a directed graph

The `get_out_neighbors()` function iterates through all the edges and collects the destination nodes in a set that it then returns. The social equivalent would be compiling a list of people to whom one person sends messages.

In contrast, the code for computing the set of in-neighbors requires us to search through every node in the graph, because we do not maintain lists of edges pointing *into* a given node, as shown in Listing 2-3. This code is called from the `Graph` class so that it has access to the full list of nodes.

```
def get_in_neighbors(self, target: int) -> set:
    neighbors: set = set()
    for node in self.nodes:
      ❶ if target in node.edges:
            neighbors.add(node.index)
    return neighbors
```

Listing 2-3: Determining the set of in-neighbors

Like other neighbor algorithms, the code constructs the neighbors from an initially empty set. The function iterates over each node and checks whether the target node has an entry in that node's `edges` dictionary ❶. If the target node has an entry in the node's edges dictionary, the neighbor is added to the set.

What happens if we run Listing 2-3 on an undirected graph? Not only does the function not fail, but it also produces the correct set of neighbors. The `get_in_neighbors()` function considers the same neighboring nodes as the code in Listing 2-1, but from the opposite side of the edge. However, `get_in_neighbors()` is significantly less efficient than `get_neighbors()` on an undirected graph because it iterates over all nodes in the graph and not just the ones connected to the target node.

Self-Loops

One additional complexity when defining neighbors is the potential for *self-loops*, in which an edge links a node to itself. For example, in Figure 2-2, node 1 has an edge to itself.

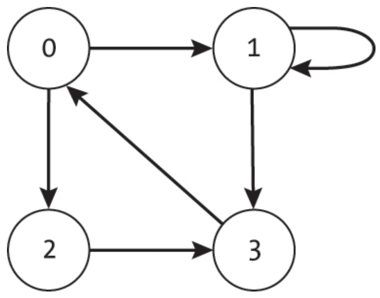


Figure 2-2: A graph with a self-loop

Self-loops work like circular roads that return to their starting points. More topically, we can visualize them in the context of my conversations about this very book. If we use a weighted graph to represent the number of conversations I had with various people, the largest weighted edge would be a self-loop indicating the number of times I mumbled to myself while trying to work something out.

In the adjacency list representation, a self-loop is represented by the inclusion of an edge whose destination matches its origin. In the adjacency matrix representation, a self-loop for node *v* is represented by a nonzero value along the diagonal of the matrix (row = *v*, column = *v*).

If node *v* has a self-loop, we consider it a neighbor of itself. In the case of a directed graph, this means node *v* is both its own in-neighbor and its own out-neighbor, since the edge starts and ends at node *v*.

Throughout this book, we use the common computer science convention of allowing self-loops only in directed graphs. While many algorithms can handle undirected graphs with self-loops, and most of the rest can easily be adapted to do so, these loops often do not make sense in the context of the problems that the undirected graphs are modeling. For example, the graph-coloring problem examined in Chapter 16 requires us to assign different colors to any two nodes connected by an edge. Self-loops make no sense in such a problem formulation.

Degree

One useful statistic for understanding a node's connectivity is its *degree*, the number of times edges connect to a node. Figure 2-3 shows an example undirected graph where each node is labeled with its degree. Node 0 has a degree of 3, while node 5 has a degree of 2.

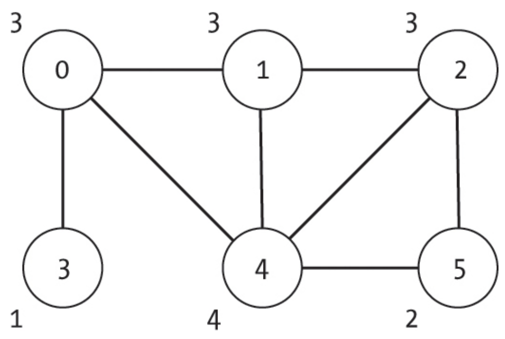


Figure 2-3: An undirected graph with each node’s degree shown

In social networks, a node’s degree would indicate the number of friends that person has. We can use this as a rough proxy for how popular or well-connected that person is.

From a mathematical standpoint, edges forming self-loops in undirected graphs are counted twice for the degree, since they contact the node on each end. While we do not use self-loops in undirected graphs for the algorithms in this book, we will include this check for completeness when computing degree in Chapter 18.

In directed graphs, we break the concept of degrees into two separate measures, as we do for neighbors. A node’s *out-degree* measures the number of connections out of that node, while its *in-degree* measures the number of edges from other nodes into the given node. In a social network, your in-degrees and out-degrees could represent the number of people with whom you share news and the number of people who share news with you, respectively. Good confidants are friends with a high in-degree and a low out-degree. Good sources of gossip have both a high in-degree, to collect tidbits, and a high out-degree, indicating their willingness to pass those tidbits along.

Edges forming self-loops in directed graphs have the same *origin* and *destination*. They count once toward the in-degree and once toward the out-degree. For example, Figure 2-4 shows a directed graph and the degrees for each node. The left-hand side of the figure shows each node’s out-degree, while the right-hand side shows their in-degrees.

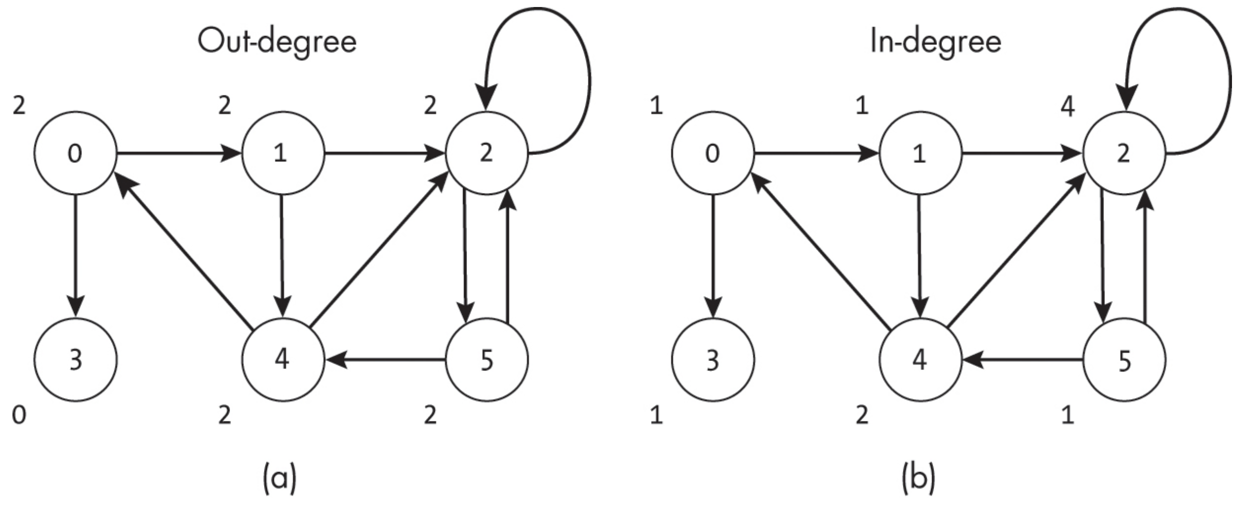


Figure 2-4: A directed graph labeling each node's out-degree (a) and in-degree (b)

Computing a node's in-degree or out-degree in a connected graph requires counting the number of incoming or outgoing edges. To do so, we can adapt the neighbor computation code from the previous section by keeping a counter instead of building a set.

Clustering Coefficient

The *clustering coefficient* of a node (sometimes called the *local clustering coefficient*) is a metric that characterizes how interconnected the node's neighbors are with each other. In the context of a social network, the clustering coefficient effectively asks, "To what extent are my friends also friends with each other?" At zero, the metric indicates that none of our friends like each other, leading to extremely awkward parties. At the other extreme, a value of one indicates that every one of our friends is connected to every other friend.

Formally, the clustering coefficient for a node *v* in an undirected graph is the fraction of possible edges among *v*'s neighbors that exist. We find the set of all neighbors (all nodes that share an edge with *v*), count how many of those neighbors share an edge with each other, and divide that by the total number of possible edges within that set. If node *v* has *k* neighbors, there could be up to $k\ (k - 1)\ /\ 2$ possible edges between them.

Nodes with one or fewer neighbors need special treatment because their neighbors have zero possible connections. If someone has no friends, it makes no sense to compute the percentage of their friends who like each other. For simplicity, we return a value of 0 in these cases, reflecting the lack of local connections.

We can define a function to compute the clustering coefficient of a given node with index `ind` in an undirected graph, as shown in Listing 2-4.

```
def clustering_coefficient(g: Graph, ind: int) -> float:
  ❶ neighbors: set = g.nodes[ind].get_neighbors()
    num_neighbors: int = len(neighbors)

    count: int = 0
    for n1 in neighbors:
        for edge in g.nodes[n1].get_edge_list():
          ❷ if edge.to_node > n1 and edge.to_node in neigh
bors:
                count += 1

    total_possible = (num_neighbors * (num_neighbors - 1))
/ 2.0
  ❸ if total_possible == 0.0:
        return 0.0
    return count / total_possible
```

Listing 2-4: Code to compute the local clustering coefficient

The code for the `clustering_coefficient()` function starts by using the `get_neighbors()` function ❶ from Listing 2-1 to generate the set of all neighboring nodes. It then uses a pair of nested `for` loops to check each unique pair of neighbors. The first `for` loop iterates over the node's neighbors, and the second iterates over the neighbor's edges.

For each edge that includes the neighboring node, the code checks that the node on the other side of the edge has an index greater than the current neighbor node and is also a neighbor of the original node ❷. The first check is necessary to avoid double-counting the neighbors. Undirected edges appear twice in the adjacency lists but should be counted only once. Each edge (u, v) is counted only when $u < v$. If the edge passes this test, it is counted.

The `clustering_coefficient()` function finishes by returning the fraction of total possible edges among neighbors that are observed, taking care to avoid a divide-by-zero if a node has one or zero neighbors ❸.

Figure 2-5 shows an example graph that lists the clustering coefficient for each node.

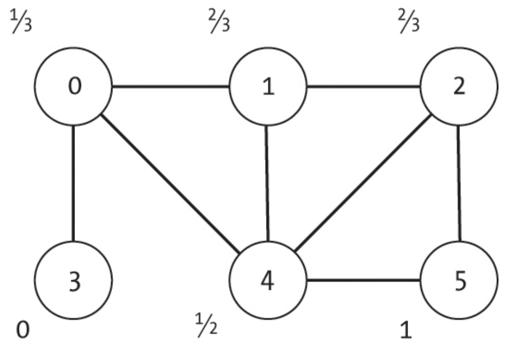


Figure 2-5: A graph with clustering coefficients

Node 0 has three neighbors (1, 3, and 4), which can have three edges among themselves, but only a single pair of its neighbors (1 and 4) are connected, giving it a clustering coefficient of 1/3. In contrast, node 5 has two neighbors that share an edge, giving it a clustering coefficient of 1. Node 3 has only one neighbor and is therefore assigned a value of 0.

Computing the Average Clustering Coefficient

The clustering coefficient tells us only about the characteristics of the graph around a single node. We can extend the insights provided by computing the *average local clustering coefficient* for all nodes of the graph, which provides a numerical measure of the local interconnectedness of an undirected graph.

We can calculate this measure for an undirected graph by computing the clustering coefficient for each neighbor, then taking the average, as shown in the following code:

```
def ave_clustering_coefficient(g: Graph) -> float:
    total: float = 0.0
    for n in range(g.num_nodes):
        total += clustering_coefficient(g, n)

    if g.num_nodes == 0:
        return 0.0
    return total / g.num_nodes
```

The `ave_clustering_coefficient()` function loops through each node, calls `clustering_coefficient()` on that node, and adds the result to a running total. As long as the function has seen at least one node, it returns the total divided by the number of nodes. For example, the graph in Figure 2-5 has a local clustering coefficient of approximately 0.5278.

Handling Limitations

The clustering coefficient only provides information about the neighboring nodes' connectivity relative to a single given node, without telling us anything about those nodes' connectivity in general. For example, consider the graph in Figure 2-6. Node 0 has a clustering coefficient of 1, indicating that all its neighbors are mutually connected. However, this doesn't tell us anything about the network one step farther away, much less *all* its neighbors' connections.

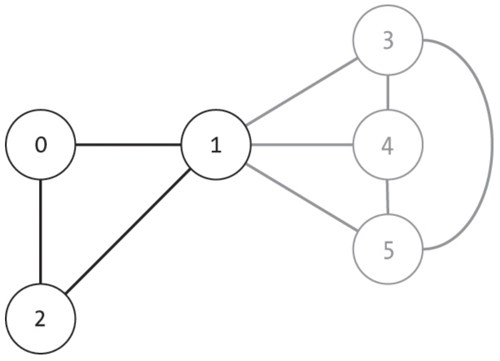


Figure 2-6: The interconnections of node 0 and its immediate neighbors versus those neighbors' connections

In Figure 2-6, node 1 has many additional connections that are not considered by the clustering coefficient because they are not directly connected to node 0's neighbors. These connections are shown in gray, while immediate neighbors are shown in black. In this case, node 1 is part of two different sets of interconnected nodes, {0, 1, 2} and {1, 3, 4, 5}.

In our social network example, this means that the clustering coefficient cannot tell us about our friends' friends. Our friends may get along with each other but also be part of other groups. Practically speaking, the local clustering coefficient can tell us if the people we invited to our party will get along, but it cannot tell us whether there is a bigger party they'll go to instead. For example, if both nodes 0 and 4 from Figure 2-6 throw a party, node 1 would enjoy either event but would have more friends at node 4's party.

Generating Neighborhood Subgraphs

We can extend the idea of neighbors to determine a *neighborhood subgraph* in an undirected graph, which includes both the neighboring nodes and the edges between them. We define two types of neighborhoods in undirected graphs depending on whether we want to include the original node. An *open-neighborhood* subgraph of node *v* consists of the neighbors of *v* and the edges between them. A *closed-neighborhood* subgraph of node *v* consists of node *v* and all its neighbors, as well as the edges between those nodes.

The Code

We can create a function within the `Graph` class to generate the open- or closed-neighborhood subgraph around a given node (with index `ind`) in an undirected graph. This function operates by determining the neighboring nodes and using them to seed a new graph, then adding the appropriate edges:

```
def make_undirected_neighborhood_subgraph(self, ind: int,
 closed: bool):
  ❶ if not self.undirected:
        raise ValueError

  ❷ nodes_to_use: set = self.nodes[ind].get_neighbors()
    if closed:
        nodes_to_use.add(ind)

    index_map = {}
  ❸ for new_index, old_index in enumerate(nodes_to_use):
        index_map[old_index] = new_index

    g_new: Graph = Graph(len(nodes_to_use), undirected=Tru
e)
```

```
    for n in nodes_to_use:
        for edge in self.nodes[n].get_edge_list():
          ❹ if edge.to_node in nodes_to_use and edge.to_no
de > n:
                ind1_new = index_map[n]
                ind2_new = index_map[edge.to_node]
                g_new.insert_edge(ind1_new, ind2_new, edg
e.weight)

    return g_new
```

The code for the `make_undirected_neighborhood_subgraph()` function starts by checking if the graph is undirected and, if not, raising a `ValueError` ❶. While this is not strictly necessary and the code will produce some results for directed graphs, it helps ensure the function is used as designed. Next, the code extracts the target node's set of neighbors with the `get_neighbors()` function from Listing 2-1 ❷. This set, `nodes_to_use`, comprises all the nodes that will be used in the subgraph. If the subgraph is a closed-neighborhood subgraph, the code adds the target node itself to that set.

The code for generating neighborhood subgraphs is complicated slightly by the way the `Graph` class indexes the nodes. Since our graph uses numeric indices in the range $[0, |V| - 1]$, where $|V|$ is the number of nodes, any subgraph might use different indices for a given node. To account for this, the code builds a dictionary `index_map` that maps the old node index to the new node index ❸. This allows the generated subgraph to use numeric indices without gaps. As we will see later in Figure 2-8, we can use alternate information, like the label, to preserve the identities of the nodes.

Finally, the code creates the new graph, using a pair of nested `for` loops to do so. This code mirrors the local clustering coefficient code from Listing 2-4. The first `for` loop iterates over the nodes in `nodes_to_use`, while the second iterates over that node's edges. By testing that the neighboring node's index (`edge.to_node`) is greater than the index of the current node `n`, the function guarantees it will insert each undirected edge only once ❹. A new edge is added only if both nodes are in `nodes_to_use` and the other node has not already been processed. The `Graph` class's `insert_edge()` function handles correctly inserting the undirected edge using the new node indices.

An Example

Consider what happens when we build a neighborhood subgraph from the graph in Figure 2-7. Returning to the earlier example of the movie star network, this graph could represent the seven stars (Alice, Bob, Carl, Dan, Edward, Fiona, and Gwen) who appear in the world-famous *Graph Theory* series of action thrillers: *Graph Theory* (with stars Alice and Bob), *Graph Theory 2: A New Node* (with stars Bob and Carl), *Graph Theory 3: The Lost Edge* (with stars Bob, Fiona, and Gwen), and so forth. Each node is labeled with the first letter of the star's name and maps their connections to their co-stars.

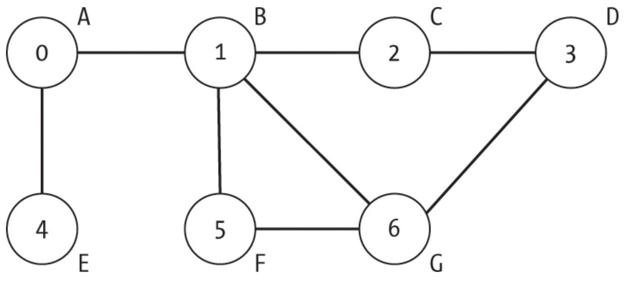


Figure 2-7: An undirected graph representing the stars of the Graph Theory *series*

To understand more about the appearances of Bob and his co-stars, we create a closed-neighborhood subgraph around Bob (node 1). This represents the stars with whom Bob has shared the screen and captures the interactions among them. Figure 2-8 shows the operation to build this graph. The left column shows the full graph, with the current node being processed indicated by a dashed circle, and the right column shows the new subgraph at that point. As noted earlier, the subgraph's nodes use different indices; in this case, we might store the star's name in the node's label.

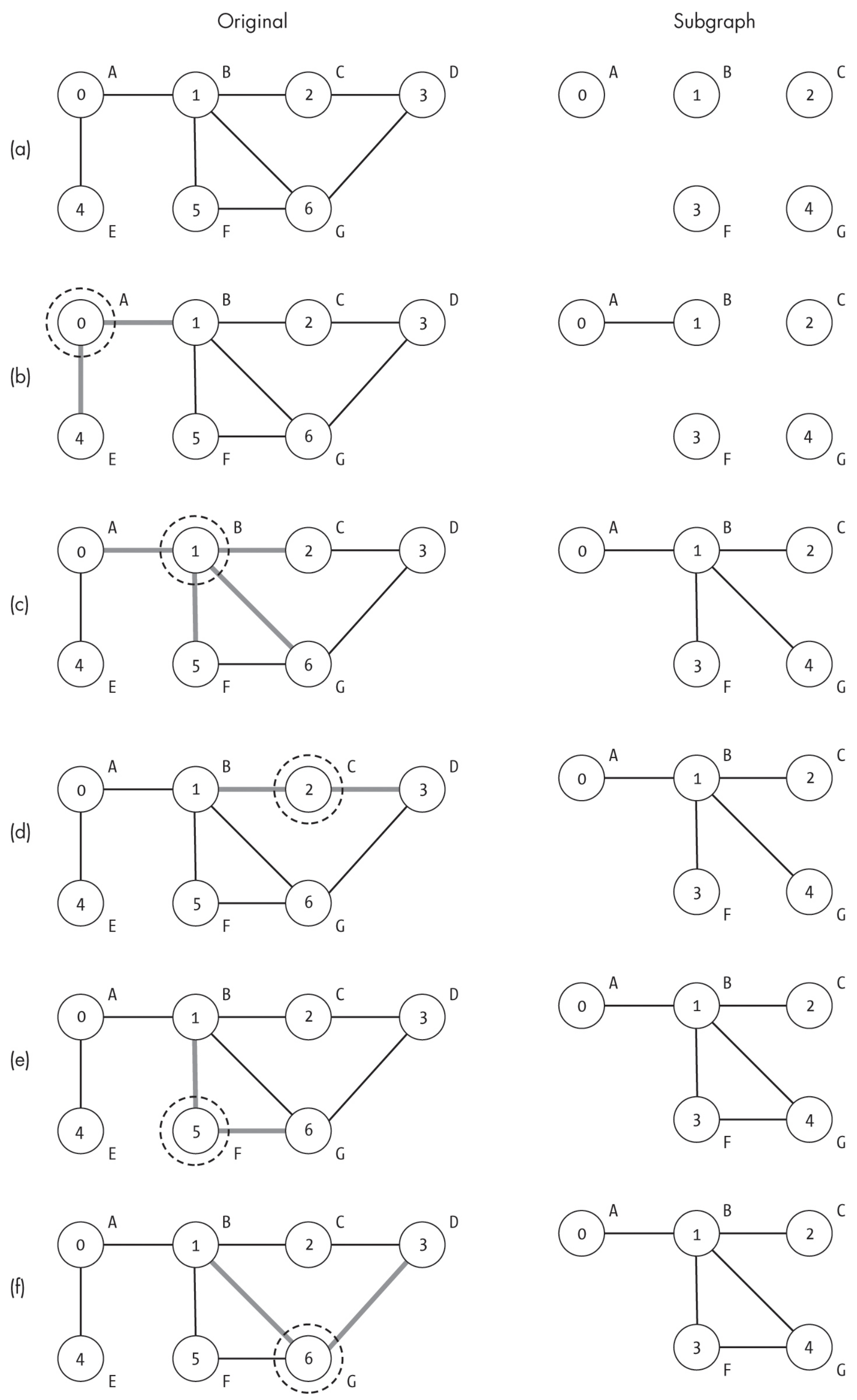


Figure 2-8: The steps to construct a closed-neighborhood subgraph around Bob

Figure 2-8(a) begins by creating a new graph containing just Bob and his co-stars. The set of neighbors includes everyone with whom Bob has appeared on-screen. Alice is included from their appearance together in the original *Graph Theory* film, while the links to stars Fiona and Gwen come from the third, and best reviewed, installment in the series.

The indexing also changes in the new graph. As shown in the figure, three of the people are given the same indices (nodes 0, 1, and 2), while two are assigned new indices (5 and 6). The node index for Fiona changes from 5 to 3 in the subgraph, and from 6 to 4 for Gwen. In Appendix A, we'll discuss how to expand the `Graph` structure to use string-based labels, which obviate the need for this index remapping.

After setting up the new graph, we iterate one by one through the people under consideration (Bob and his co-stars) and add new edges to the subgraph. When considering node 0 in Figure 2-8(b), we add only one of their two edges, (0, 1). This is because both Alice and Bob are being considered. In contrast, Edward (node 4) was only in the disastrous spin-off attempt *The Golden Vertex* with Alice. Since Edward never appeared on-screen with Bob, he is not part of Bob's neighborhood subgraph.

When we get to Bob, who represents the franchise staple, we add edges to three new co-stars in Figure 2-8(c). We do not add an edge back to Alice because we have already processed that node and its edges. The code continues through Carl in Figure 2-8(d), Fiona in Figure 2-8(e), and Gwen in Figure 2-8(f). Since Edward and Dan didn't co-star with Bob, they are not in the list of neighbors and are never considered. The final subgraph is shown on the right-hand side of Figure 2-8(f).

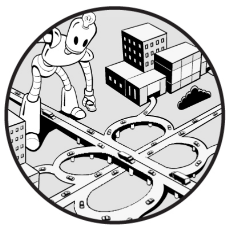
Why This Matters

A graph's neighbors provide fundamental information about the local structure and interconnections around a given node. For the most part, the formal definitions of these terms are conveniently intuitive. When traversing a graph, we ask which nodes are neighbors of the current node and are thus reachable. Neighbors will form the basis of our discussion of graph search algorithms in later chapters, as many of these algorithms share the core loop of iterating over a node's edges and seeing which other nodes share them.

Concepts like a node's degree and its local clustering coefficient provide concrete metrics about its immediate neighbors and neighborhood. These example metrics are only a fraction of the multitude of ways to quantify the properties of a graph. Numerous metrics have been developed to analyze the properties of real-world graphs, from their level of interconnectedness to their width. A comprehensive review of all graph metrics is well outside the scope of this book, but upcoming chapters discuss some additional analytics.

In the next chapter, we consider another foundational graph algorithm concept: paths. Paths provide a description of movements throughout the graph and allow us to record how to traverse from one node to another.

3
PATHS THROUGH GRAPHS



The concept of a *path* through a graph is another foundational building block we'll use throughout the book. For example, we may be interested in determining the lowest-cost path between two nodes (*shortest path algorithm*) or whether it is even possible to reach one node from another using any path at all. Multiple subsequent chapters are devoted to computing paths with different properties.

The general concept of a path mirrors its real-world counterpart. Just as the paths in your favorite park provide routes for moving from one place to the next, paths in graphs provide the same mechanism. They are sequences of nodes (or edges) that let us travel through the graph. When escaping a maze or navigating a road trip, we move from node to node using the graph's edges.

This chapter formally defines what we mean by paths through a graph and looks at different ways we can represent these structures. Whether finding paths or using them as part of an algorithm, we need to be able to represent them efficiently and unambiguously. If you've ever asked for directions and been met by a vague wave and a statement like "Go that way and turn right when you get to either the third or fourth intersection. You'll probably know it when you see it," you've experienced working with an incomplete path.

This chapter introduces three decidedly unambiguous representations. We also consider the properties of paths and a few tasks for which we can use them. We explore functions to check whether a path is valid and to compute the cost of a path in a weighted graph. Finally, we discuss how paths relate to the question of reachability in a graph.

Paths

A *path* through a graph is a sequence of nodes that are connected by edges. These are the waypoints we pass through when moving along the graph from one node to another. Just as we referred to the endpoints of individual edges as origins and destinations, we use those terms to describe the endpoints of an entire path: the first node in the path is the origin, and the last is the destination. Figure 3-1 shows the path consisting of the nodes [0, 1, 3, 2, 4, 7]. Each adjacent pair of nodes along the list corresponds to an edge in the graph.

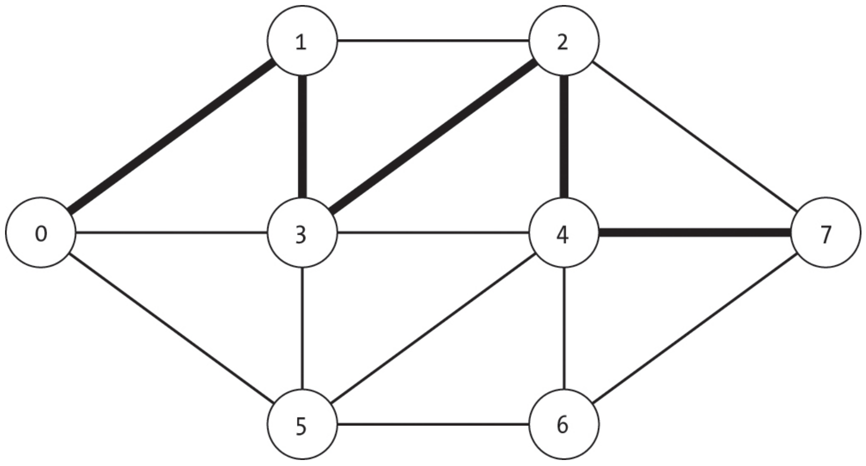


Figure 3-1: A valid path from node 0 to node 7

For the purposes of this book, we use the definition of a path (as a sequence of nodes) that is common in computer science and algorithm texts. This differs from the formal graph theory definition of a path, which does not allow for repeated nodes. In graph theory, a path allowing repeated nodes is called a *walk*. Using the more general definition of a path allows us both to stay consistent with the other algorithmic texts and to mirror the real-world paths.

Paths have directionality, even on undirected graphs. The list of nodes presents the order in which we travel the path. The path [0, 1, 2, 7] travels *from* node 0 *to* node 1 *to* node 2 *to* node 7. This directionality will be important in understanding the results of algorithms throughout the rest of this book.

Path Representation

There are a variety of ways to represent a path in code. As with all representations, we can tailor the data structures of a path to the problem or algorithm at hand. This section covers several approaches to storing paths in code, and their respective advantages and drawbacks. We also look at how each representation maps to a real-world counterpart in the context of planning a road trip.

Throughout the section, we also use the problem of checking path validity to illustrate how the storage representation works and how code traverses it. For the purposes of the following code, we will consider empty paths, containing no nodes or edges, as valid. You can easily adapt the code to exclude these cases depending on the problem domain.

Lists of Nodes

We typically use an ordered *list of nodes* to represent the path from a starting node to an ending node in the graph, a representation found in many computer science texts. We define the list of nodes inclusively, so `path[0]` is the index of the starting node (path origin), while `path[N-1]` is the index of the ending node (path destination) for a path of length `N`. For example, we represent the path in Figure 3-1 as [0, 1, 3, 2, 4, 7]. This representation can be used for a single path with a fixed origin and destination.

On our road trip, the list-of-nodes representation is equivalent to listing each city we will visit along the way. Suppose we plan to travel from Boston to Philadelphia to Pittsburgh to Columbus to Indianapolis. If we stop at a tourist center in each city, including the origin and destination, and purchase a commemorative postcard, the complete stack of cards will sum up the exciting journey.

We check whether such a path is valid by iterating through the list of nodes and confirming that an edge exists between each pair, as shown in the following code:

```
def check_node_path_valid(g: Graph, path: list) -> bool:
    num_nodes_on_path: int = len(path)
  ❶ if num_nodes_on_path == 0:
        return True
  ❷ prev_node: int = path[0]
    if prev_node < 0 or prev_node >= g.num_nodes:
        return False

    for step in range(1, num_nodes_on_path):
        next_node: int = path[step]
      ❸ if not g.is_edge(prev_node, next_node):
            return False
      ❹ prev_node = next_node
    return True
```

The `check_node_path_valid()` function first checks whether the path is empty and, if so, returns `True` ❶ because we defined a path of zero nodes as valid.

If the path is not empty, the code retrieves the starting node of the path and checks that it is valid ❷. The code then loops through the remaining nodes on the path, using the variable `step` to track the current step under consideration. Since the code already tested the first node, it starts at the second node in the list (`step=1`). For each new node, it uses the `is_edge()` function to check both that the node is valid and that there exists an edge from the previous node to this new node ❸. If the new node is invalid or no corresponding edge exists in the graph, the function immediately returns `False`. After checking the edge, the code proceeds to the next node in the path ❹. The function returns `True` if it makes it through the entire path without finding any invalid nodes or edges.

In the context of our road trip example, this code corresponds to iterating over the list of cities. We check that each city is a valid city and that there is a road directly from the previous city on the list (in other words, there is an edge). Given a list such as [Boston, New York, Philadelphia, Pittsburgh], we would return `True`, while [Boston, New York, Madrid, Philadelphia, Pittsburgh] would clearly return `False`.

Lists of Edges

Another natural approach to representing paths is to use a *list of edges*. The origin and destination of the path correspond to the origin of the first edge and the destination of the last edge, respectively. Each edge in the list represents the transition between two nodes.

This formulation requires the additional constraint that the origin of every edge (except the first one) equals the destination of the previous edge. We say that a set of edges [e_0, e_1, . . . , e_k] is a valid path only if:

$$e_{(i-1)}.\texttt{to\_node} = e_i.\texttt{from\_node} \text{ for all } i > 0$$

This definition restricts paths to mirror their real-world counterparts in that the start of every segment (edge) needs to align with the ending point of the previous segment (edge). We do not allow the path to stop at one node and magically restart from a different one. If a friend gave you instructions for a road trip from Boston to Seattle in which day one is spent driving from Boston to Pittsburgh and day two is spent driving from Cincinnati to St. Louis, you would soon realize there was a problem: somehow the proposed route skips the majority of Ohio. The path shown in Figure 3-2, like our friend's bad directions, is not valid.

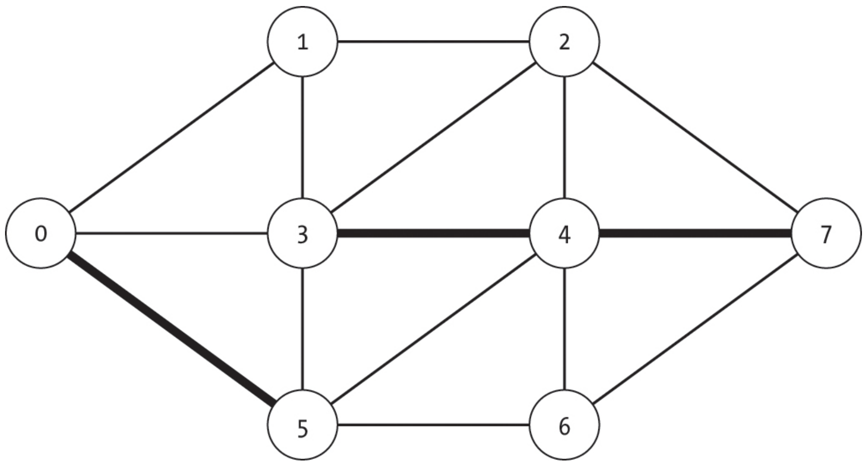


Figure 3-2: An invalid path through the graph

When writing paths as a list of edges, we denote each edge as the tuple (`from_node`, `to_node`) to mirror the `Edge` objects. For example, the path in Figure 3-1 corresponds to [(0, 1), (1, 3), (3, 2), (2, 4), (4, 7)].

We check whether such a path is valid by iterating through the list of edges and confirming both that the destination of the last edge matches the origin of the current edge and that the edge exists in the graph, as shown in the following code:

```
def check_edge_path_valid(g: Graph, path: list) -> bool:
  ❶ if len(path) == 0:
        return True

  ❷ prev_node: int = path[0].from_node
    if prev_node < 0 or prev_node >= g.num_nodes:
        return False

    for edge in path:
      ❸ if edge.from_node != prev_node:
            return False

        next_node: int = edge.to_node
        if not g.is_edge(prev_node, next_node):
            return False

        prev_node = next_node
    return True
```

The `check_edge_path_valid()` function first checks whether the path is empty and, if so, returns `True` ❶ because we previously defined a path of zero edges as valid. Otherwise, the code retrieves the starting node of the path as the origin of the first edge ❷. It checks that this node has a valid index and, if not, returns `False`.

The code then loops through all the edges on the path. For each edge, it first checks that the origin of the edge matches the last edge's destination ❸. It then extracts the index of the new destination node (`next_node`) and uses the `is_edge()` function to check both that the node is valid and that this edge exists. If the new node is invalid, or no corresponding edge exists in the graph, the function immediately returns `False`. After checking the edge, the code proceeds to the next edge along the path. The function returns `True` if it makes it through the entire path without finding any invalid nodes or edges.

Lists of Previous Nodes

For many of the algorithms in this book, we use a more specialized representation of paths as lists mapping nodes to the previous node along the path. This method lends itself well to how many of the algorithms in this book process the data by traveling from one node to the next. It has both significant advantages and significant disadvantages.

Consider the path [0, 1, 3, 2, 4, 7] shown in Figure 3-3. For each node along the path, we can indicate which node precedes it, as shown in the list `last`. The value `last[4]=2` indicates that we get *to* node 4 *from* node 2. We use the special value −1 to indicate that a node does not have a preceding node. This can be due to the node either being the first node on the path (the origin of the path) or not being part of the path at all.

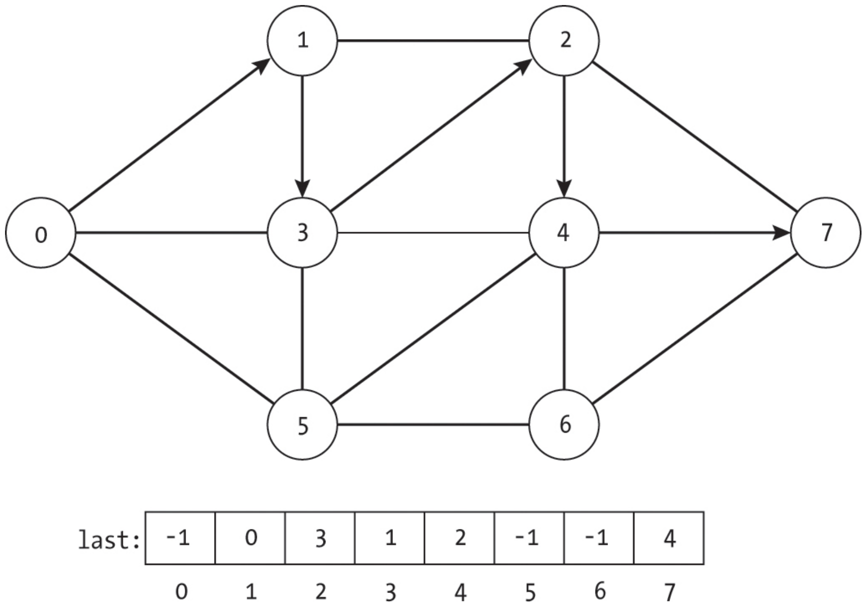


Figure 3-3: A path through the graph and its representation as an array of previous nodes

The major disadvantage of this representation is that it limits the types of paths we can represent. Each node can have, at most, one preceding node. We cannot represent many paths that revisit nodes, such as [0, 1, 3, 0, 5, 3, 4, 7]. If we were to try to represent this path as an array of previous nodes, we would run into a problem with node 3's `last` entry. Node 3 is preceded first by node 1 and then, later, by node 5. While it is possible to define more complex lists of lists that handle these cases, we use the single list of indices because it is well suited to the algorithms in this book.

The major advantage of this representation is that it's easy to update paths inside the algorithms because we need to change only a single index value. We will use this behavior again and again throughout later chapters to simplify the code. Another advantage is that this representation can capture multiple, branching paths. Unlike the previous two representations, which use a single fixed origin and destination nodes, the previous-node representation requires only a single fixed origin. The list can be used to extract a path from *any* destination node back to the origin.

In the road trip analogy, the list-of-previous-nodes representation is equivalent to listing each city you *could* visit and the city from which you would travel to get there. If your plans change and you decide to visit Pittsburgh, you can follow the chain of previous nodes through Philadelphia and back to Boston. The list of previous visits provides information for all possible stops on trips out of the origin city.

Another useful analogy for this representation is the set of chalk marks that an adventurer makes while exploring a labyrinth, such as shown in Figure 3-4.

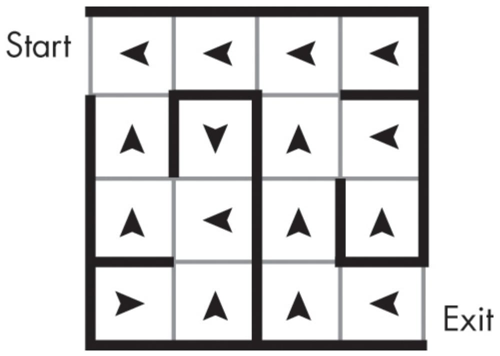


Figure 3-4: Chalk marks in a labyrinth indicating the direction you came from

Determined not to get lost, the adventurer draws a chalk arrow at every intersection, indicating the passage they used to get to the current room. These markings point backward along the path traveled. Such chalk marks have a distinct advantage over breadcrumbs in their overall information content, allowing the adventurer to find their way back to the start from any room in the dungeon. In particular, if they know the location of the exit, they can rebuild the path from the exit to the start.

Translating a Previous-Node List into a List of Nodes

To translate the previous-nodes representation into a list of nodes, we walk the pointers in the `last` list backward from a given destination, as shown in the following code:

```
def make_node_path_from_last(last: list, dest: int) -> lis
t:
  ❶ reverse_path: list = []
    current: int = dest

  ❷ while current != -1:
        reverse_path.append(current)
      ❸ current = last[current]

  ❹ path: list = list(reversed(reverse_path))
    return path
```

The `make_node_path_from_last()` function begins by compiling the path list in reverse and storing it in `reverse_path`, before reversing the order at the end. Initially, the code sets the `reverse_path` list as empty and starts at the end by setting the `current` node to the destination ❶. It uses a `while` loop to traverse the path until it hits `-1`, which indicates the lack of a previous node ❷. At each step in the loop, the code appends the current node to `reverse_path` and moves to the preceding node as defined by `last` ❸. Finally, the code creates a reversed copy of `reverse_path` (the path in the correct order) and returns that list ❹.

Let's consider what happens when we apply the code to the graph in Figure 3-5(a) and a `last` array indicating paths starting from node 0:

$$[-1, 0, 1, 2, 2, 0, 5, 0, 5, 8]$$

Figure 3-5(b) shows the backward pointers on the graph. As we can see, the arrows point backward to the origin node. Figure 3-5(c) reverses each point to demonstrate how we can reconstruct the forward path.

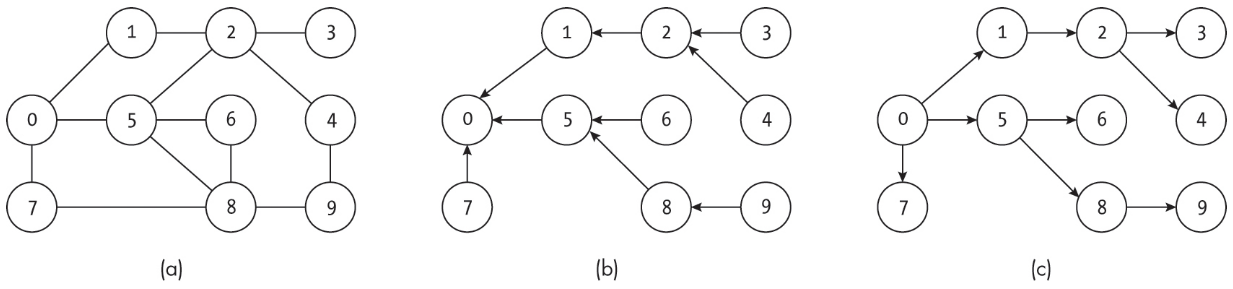


Figure 3-5: An example graph (a), the last pointers created by a breadth-first search (b), and the corresponding forward paths through the graph (c)

If we use node 9 as our destination (`dest=9`), we build up the reverse path (represented as a list of nodes) as follows:

[9]

[9, 8]

[9, 8, 5]

[9, 8, 5, 0]

The returned final path in this case is [0, 5, 8, 9].

As noted earlier, the previous-node representation allows us to represent paths from a single origin to all destinations in the graph. Because of this, it is often not possible to convert from a list-of-nodes or list-of-edges representation into a full list-of-previous-nodes representation. Returning to Figure 3-5, a list-of-nodes path from node 0 to node 3 would be [0, 1, 2, 3]. While this tells us everything we need to know about the path from node 0 to node 3, it does not tell us about paths from node 0 to any nodes that are *not* in the list. We could not use it to derive a path from node 0 to node 5.

Checking the Validity of a Previous-Node List

To test the validity of this type of path, we iterate over all entries in the list and check that either the previous node is −1, or the edge exists:

```
def check_last_path_valid(g: Graph, last: list) -> bool:
 ❶ if len(last) != g.num_nodes:
        return False

    for to_node, from_node in enumerate(last):
     ❷ if from_node != -1 and not g.is_edge(from_node, to
_node):
                return False
    return True
```

The `check_last_path_valid()` code first checks that the `last` list has the correct number of entries ❶. Even if all the entries are −1, there needs to be an entry for every node.

The code then iterates through the list using Python's `enumerate()` function, where `to_node` is the index of `last` currently being examined and `from_node` is the corresponding value. The code checks that either `from_node` is –1, indicating that there is no previous node on the path, or that the combination of `to_node` and `from_node` corresponds to a valid edge ❷. If neither case holds, the code immediately returns `False`. If every entry in `last` is valid, the code returns `True`.

In the context of our road trip example, this function iterates over each city in our list and asks, "Can we get there directly from the listed previous city?" We would approve an entry for Pittsburgh that has a previous node of Philadelphia or Erie. However, we would soundly reject an entry for Boston with a previous node of Santa Fe.

Calculating Path Cost

For many use cases, we may be interested not only in which edges to use to compose the path but also an overall measure of either the benefits or cost of the path as a whole. As we saw in Chapter 1, we can use edge weights to capture the cost of traversing between two nodes, such as using them to model the distance between cities on our road trip. When planning a flight itinerary from Boston to Seattle, we might skip the edge from Boston to Miami because of its nearly 1,500-mile cost.

We define the *cost of a path* in a weighted graph as the sum of the edge weights along that path. Formally, we say that for a path $[e_0, e_1, \ldots, e_k]$:

$$PathCost([e_0, e_1, \ldots, e_k]) = \sum_{i = 1 \text{ to } k} e_i.\texttt{weight}$$

The cost of the path [0, 3, 4, 2] in Figure 3-6 would be 1.0 + 3.0 + 5.0 = 9.0.

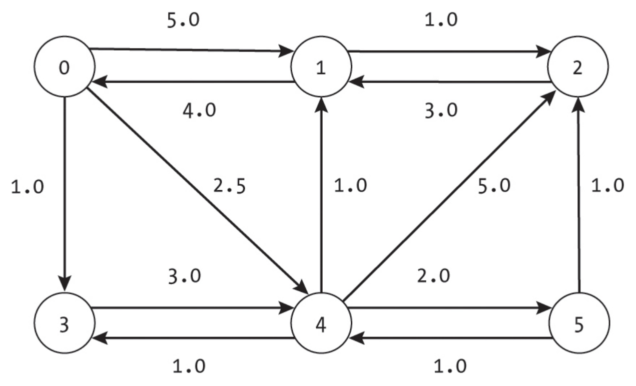


Figure 3-6: A weighted, directed graph with six nodes

For unweighted graphs, we often use a unit value of 1.0 for each edge and thus compute the cost as the number of edges used.

If we have a list of edges representing our path, we compute the cost of the path by iterating over those edges, as shown in Listing 3-1.

```
def compute_path_cost_from_edges(path: list) -> float:
 ❶ if len(path) == 0:
        return 0.0

    cost: float = 0.0
    prev_node: int = path[0].from_node
    for edge in path:
     ❷ if edge.from_node != prev_node:
            cost = math.inf
        else:
            cost = cost + edge.weight
        prev_node = edge.to_node

    return cost
```

Listing 3-1: A function to compute the total path cost

The code starts by checking that the path has at least one edge and, if not, returning a cost of `0.0` ❶. It also initializes variables to track the total cost (`cost`) and the previous node seen (`prev_node`). The `prev_node` variable is used to assess the validity of the path.

The `compute_path_cost_from_edges()` function computes the cost of a path iterating through each edge in the list. It checks that the origin of the current edge matches the last node ❷. If the nodes do not match, the path is invalid. For example, an edge list of [(0, 1), (2, 3), (3, 4)] would be invalid because you cannot jump from node 1 to node 2 without an edge between the two nodes. If the transition is valid, the edge's weight is added to the cost. If the transition is invalid, we use an infinite weight. Depending on the implementation, the programmer might want to raise an exception, exit the program, or use some other method to indicate an error. The code updates the `prev_node` variable to track the new current location.

The code continues through the list, checking each edge and summing their weights. When it has finished each edge in the list, it returns the total cost with `return cost`.

Unlike other functions in this chapter, we intentionally do not pass in a graph to this implementation to demonstrate how we can operate on a pure list of edges. The downside to this simplicity is that the function cannot validate the path against the graph itself. We could extend the function to perform an additional check by passing in the graph and using it to validate both the node indices and the edges' existence. The code for these additional checks follows the approach shown in other functions.

Reachability

We can use the formulations of paths to formalize another important question on graphs: "Is node *v* reachable from node *u*?" This question is vital to a range of real-world problems. In a transportation network, it translates into the question, "Can we get to city *v* from city *u*?" In a social network, it translates into the question, "Can a rumor spread from person *u* to person *v*?" And in a dungeon labyrinth, it translates into the vital question, "Can I get from here to the exit?"

We say that node *v* is *reachable* from node *u* if there exists a path that starts at node *u* and ends at node *v*. Given a candidate path, we can use any one of our validity checkers from earlier in this chapter to test whether it is valid.

Imagine you are trapped in a castle dungeon represented by the graph in Figure 3-7. The edges indicate which adjacent rooms have an unlocked door between them. To escape from the evil wizard, you need to navigate to a room with a staircase to the upper floor. Here, the reachability question is vital. If you start in room 0 and need to navigate to node 15, you are out of luck. Node 15 is not reachable from node 0.

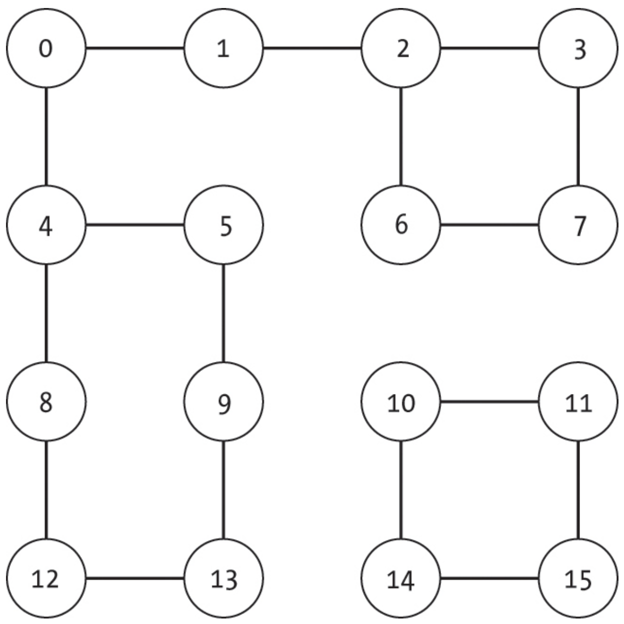


Figure 3-7: A graph representing connectivity among rooms in a castle dungeon

In an undirected graph, we can divide the nodes into disjoint sets called *connected components* such that any node in a connected component is reachable from any other node. Given a subset of nodes $V' \subseteq V$, we say that a connected component is a maximal set of nodes such that:

$$\textit{reachable}(u, v) \text{ for all } u \in V' \text{ and } v \in V'$$

In the context of our dungeon example from Figure 3-7, the map consists of the two connected components {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13} and {10, 11, 14, 15}. We are okay if the stairs out of the dungeon are in the same connected component as our current room. Otherwise, we are trapped.

Why This Matters

We will use the concept of paths throughout the book to solve problems spanning from path planning to optimizing flow through a capacity-limited network. Paths will be a fundamental unit of data used throughout many of the algorithms and will be one of the most common results computed by the functions. Concepts like reachability and path cost will be foundational in numerous algorithms, from finding strongly connected components in Chapter 11 to performing matching on bipartite graphs in Chapter 15.

In addition to their computational usefulness, paths can help us visualize the operation of algorithms in a real-world context. Instead of an abstract "sequence of edges," it helps to visualize paths as their real-world counterparts. This chapter repeatedly made use of the road trip analogy to map edges to roads across the country. We can similarly imagine physically walking paths through many of the algorithms discussed in this book.

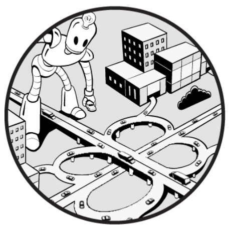
The next chapter builds upon the concept of paths, considering multiple algorithms that explore graphs and return the paths taken. These paths provide important information about both the functionality of the search and the ability to navigate through the graph.

PART II

SEARCH AND SHORTEST PATHS

4

DEPTH-FIRST SEARCH



A few natural questions arise when we consider the realm of graph search algorithms. Why do we want to search a graph? What are we looking for? Didn't we already find all the nodes when we created the graph? To some extent, the term *graph search* undersells the generality of these algorithms. Graph search algorithms provide a mechanism for systematically traversing all the nodes in a graph. We could use this ability to search for a particular node, such as finding treasure hidden in a maze, or to otherwise enumerate and analyze the graph.

We'll begin our exploration of graph search with *depth-first search*. This algorithm traverses the nodes of a graph by exploring one edge at a time leading out from the current node, progressing deeper and deeper into the graph before backtracking and trying alternate paths. It is arguably one of the most powerful, flexible, and useful graph algorithms covered in this book, supplying the core logic for many of the more advanced algorithms in later chapters.

What makes depth-first search so useful is its simplicity and adaptability. It can be implemented in a relatively simple recursive function, and, with minor additions, it can compile an extensive amount of information about the graph. This allows it to serve the role of the trusty kitchen mixer, helping us create everything from basic bread to a wedding cake.

This chapter presents possible use cases for depth-first search, then covers the recursive and stack-based algorithms for this search. We show how depth-first search can be used to determine a graph's connected components, then discuss two useful extensions to this search: depth-first search trees and iterative deepening.

Use Cases

To provide an overview of how depth-first search works and why it's useful, let's go over a few cases where we might use this search in our daily lives.

Exploring a Hedge Maze

Imagine standing at the entrance of a vast hedge maze. As nervousness sets in, you remind yourself that this isn't a mythical Greek labyrinth with a minotaur waiting to attack the unsuspecting adventurer. You're only facing a multi-acre challenge of your spatial and navigational abilities. The bored teenage attendant mutters a not-so-reassuring consolation: "They usually remember to patrol the maze and pick up lost hikers before closing time."

In this case, graph search corresponds to searching the graph for one special node, the exit, starting at the entrance node. As shown in Figure 4-1, there are multiple valid ways you could represent the maze as a graph. Figure 4-1(a) shows the shape of the maze itself. As shown in Figure 4-1(b), you could partition the physical space into cells and call each one a node with edges to reachable adjacent spaces. Alternatively, as in Figure 4-1(c), you could create nodes only for the entrance, exit, and decision points. The paths between these special points become the graph's edges.

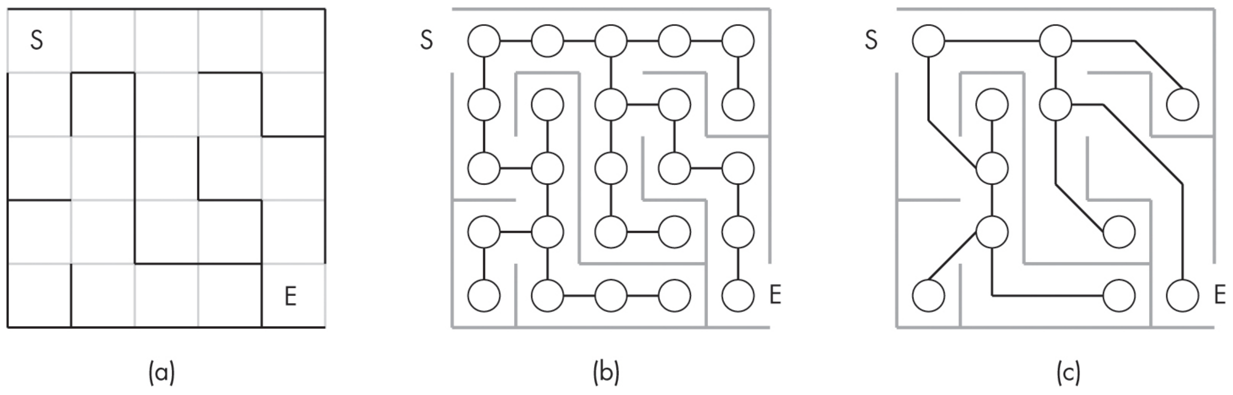


Figure 4-1: A maze (a) with two different graph representations (b) and (c)

We'll return to this maze example throughout the chapter as a fun and easy way to visualize ourselves wandering the graph. The maze example also provides amusing real-world counterparts to marking nodes or choosing which edge to take. More importantly, we can add monsters at any point for a little excitement: all the best labyrinths involve some danger.

Learning a New Subject

Learning a new subject can be viewed as a graph exploration problem. Each node represents a subtopic of interest and edges represent the references between them. The graph search represents the learning journey through the various subtopics. The goal is not to reach a particular node, but rather to cover relevant parts of the topic graph.

For example, consider the general topic of geology. An intrepid student sets out to learn everything they can about the subject, starting with the topic of rocks. As they read about each subtopic, they build a graph of related knowledge, as shown in Figure 4-2. Their studies follow threads deep into the details. A reference to igneous rocks leads to an interest in obsidian, then volcanoes, and then making the classic baking soda and vinegar volcano science project. Another path takes them through metamorphic rocks to marble, where they venture down a related path of interior decorating and flooring installation.

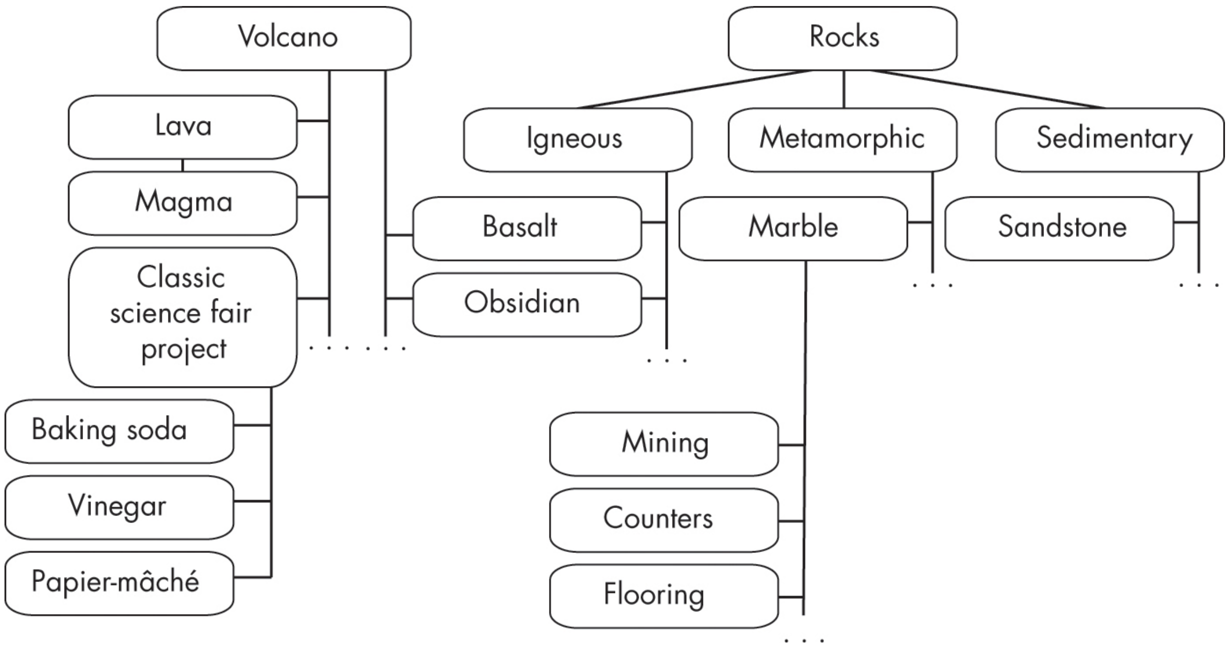


Figure 4-2: A graph of subtopics starting from the topic of rocks

The graph in Figure 4-2 is woefully incomplete. Not only are many fascinating subjects (such as subduction zones and bauxite) omitted, but there are also more connections linking the subjects than the figure can show. Many different rocks would share links to common elements or minerals. Exploring the entire topic graph for this one area could take a lifetime. As we will see in this chapter and the next, the type of search we use can have a profound impact on the order in which we approach the topics.

Checking Reachability

In our daily lives, we often want to know whether some path exists to a certain node from a given starting node. For example, we might use a graph of flights to check whether we can travel between two cities, or we might use a social network to check if a rumor will spread from one person to another.

Consider the network of people represented in Figure 4-3. Each node represents a person, and an edge from node *u* to node *v* indicates that person *u* is willing to share information with person *v*. A useful factoid discovered by person 0 can be passed through the graph to persons 1, 3, and 4. In the case of person 3, the information first travels through persons 1 and 4. However, persons 2 and 5 are left out altogether, because there is no path of knowledge-sharing that goes to either of them.

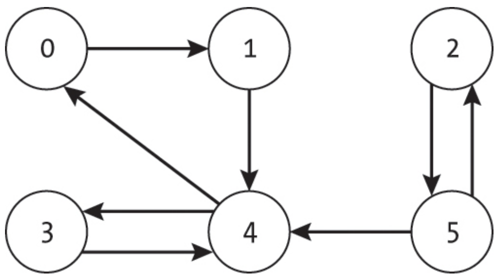


Figure 4-3: An example directed graph with six nodes

We will explore the concept of *reachability* and *connected components* for undirected graphs later in this chapter, and we'll discuss algorithms for those problems on directed graphs in Chapter 12. For now, it is sufficient to see how a graph search can answer the reachability question. The starting node *S* and goal node *G* are given. All we need to do is start a search from node *S* and check whether it finds node *G*. If so, there must exist some path between them.

Recursive Depth-First Search

We commonly implement depth-first search as a *recursive algorithm*, where the core functionality is called once for each node. This section presents the code for this search and shows how it progresses through an example graph.

The Code

Listing 4-1 shows a bare-bones version of depth-first search.

```
def dfs_recursive_basic(g: Graph, ind: int, seen: list):
  ❶ seen[ind] = True
    current: Node = g.nodes[ind]

    for edge in current.get_edge_list():
        neighbor: int = edge.to_node
      ❷ if not seen[neighbor]:
            dfs_recursive_basic(g, neighbor, seen)

def depth_first_search_basic(g: Graph, start: int):
    seen: list = [False] * g.num_nodes
    dfs_recursive_basic(g, start, seen)
```

Listing 4-1: The core depth-first search recursive function

The recursive helper function takes several pieces of information: the graph itself (`g`), the index of the current node being explored (`ind`), and a list mapping each node to whether it has already been visited (`seen`). The code starts by marking the current node as visited ❶ and retrieving the `Node` data structure. It then checks each of the node's neighbors by iterating through the list of edges. For any node that has not already been visited ❷, it recursively calls the search on that node.

The outer function sets up the `seen` list and starts the depth-first search from a specific starting node index (`start`). This outer function searches from only a single starting node and thus visits only nodes that are reachable from that node. If we want to visit every node, we need to start a search from every previously unseen node. As shown in Listing 4-2, we expand the outer function to iterate over the nodes in order and call the recursive depth-first search from every unseen node.

```
def depth_first_search_basic_all(g: Graph):
    seen: list = [False] * g.num_nodes
    for ind in range(g.num_nodes):
      ❶ if not seen[ind]:
            dfs_recursive_basic(g, ind, seen)
```

Listing 4-2: A depth-first search that explores all nodes in the graph

After initializing the `seen` list, the code loops over each node index, checks whether it has been visited in a previous depth-first search ❶, and, if not, starts a new depth-first search from that node.

While this code performs a depth-first search, it doesn't do anything interesting with the search. This is like taking a nice walk through the maze, but not recording anything about the solution. Let's consider the simple addition of keeping track of our path. This is equivalent to bringing a notebook into the maze and recording which directions we take.

The code for recording the path traveled during the depth-first search uses an additional list—the `last` node index visited immediately before the current one:

```
def dfs_recursive_path(g: Graph, ind: int, seen: list, las
t: list):
    seen[ind] = True
    current: Node = g.nodes[ind]

    for edge in current.get_edge_list():
        neighbor: int = edge.to_node
        if not seen[neighbor]:
          ❶ last[neighbor] = ind
            dfs_recursive_path(g, neighbor, seen, last)

def depth_first_search_path(g: Graph) -> list:
    seen: list = [False] * g.num_nodes
    last: list = [-1] * g.num_nodes

    for ind in range(g.num_nodes):
        if not seen[ind]:
            dfs_recursive_path(g, ind, seen, last)
    return last
```

The recursive function starts the same way as the earlier basic version. The current node's index is marked as seen, the current node is retrieved, and a `for` loop checks whether each of the node's neighbors has already been visited. It is only when exploring new nodes that there is a difference in behavior. Before recursively calling the depth-first search on the new node, the code records that the current node (`ind`) immediately preceded the next node (`neighbor`) ❶. As discussed in Chapter 3, the `last` list provides all the information needed to reconstruct the path taken by the search.

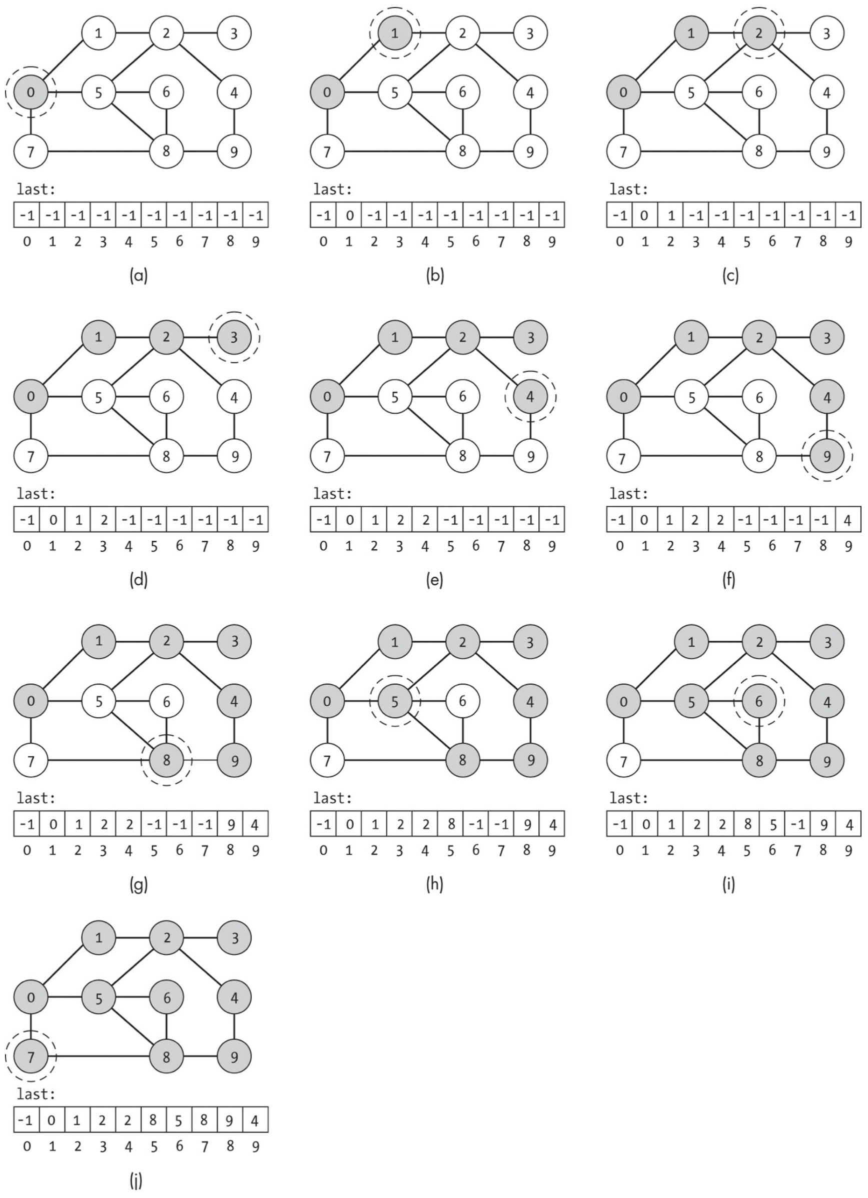
The outer function is similarly modified to initialize and pass in this list of previous nodes `last`. The `last` list uses an indicator value of `-1` to indicate that there is no preceding node. Nodes with values of `-1` at the end of the search were the starting nodes for the various depth-first searches.

An Example

Figure 4-4 shows an example of recursive depth-first search on an undirected graph with 10 nodes. Each subfigure shows the state immediately after the current node is marked seen in the function. The dashed circle indicates the current node being explored. The shaded nodes are the ones that have been visited (and thus marked seen). The `last` vector shows how the path through the graph evolves during the search.

The search starts in Figure 4-4(a) at node 0, which has three neighbors: nodes 1, 5, and 7. We can visualize this in the context of an adventurer exploring a labyrinth (a little more exciting than a hedge maze). Node 0 represents the adventurer standing at the first intersection, considering the three possible branches ahead. They do not know which one will lead to the exit and which will result in a dead end.

The search chooses the first neighbor, node 1, and recursively triggers a depth-first search. During the exploration of node 1 in Figure 4-4(b), we can see that `last` has been updated to indicate that node 1 was reached from node 0. As our adventurer walks from intersection 0 to intersection 1, they record this step in a little notebook to preserve the information for future generations.



(a) last: -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
(b) last: -1 0 -1 -1 -1 -1 -1 -1 -1 -1
(c) last: -1 0 1 -1 -1 -1 -1 -1 -1 -1
(d) last: -1 0 1 2 -1 -1 -1 -1 -1 -1
(e) last: -1 0 1 2 2 -1 -1 -1 -1 -1
(f) last: -1 0 1 2 2 -1 -1 -1 -1 4
(g) last: -1 0 1 2 2 -1 -1 -1 9 4
(h) last: -1 0 1 2 2 8 -1 -1 9 4
(i) last: -1 0 1 2 2 8 5 -1 9 4
(j) last: -1 0 1 2 2 8 5 8 9 4

Figure 4-4: A recursive depth-first search on a graph with 10 nodes

Depth-first search always moves from one node to a directly connected one. Similarly, our adventurer continues down one path until they hit a dead end. Then, possibly full of panic and dread, they backtrack to try alternate loops while firmly hoping they do not run into any monsters or sarcastic maze attendants. Since backtracking consists of returning to an adjacent room, it makes physical sense.

The search continues through the entire graph, recursively exploring the neighbors in order of increasing node index. In the labyrinth example, this corresponds to our explorer wandering deeper into the maze and backtracking at dead ends. Because of the structure of both the graph and the depth-first search, the path to reach a node is not necessarily the shortest. For example, while node 5 can be reached directly from node 0, the search encounters it via the path [0, 1, 2, 4, 9, 8, 5].

In this example, all the nodes were reachable from a single starting node. However, as noted in the last section, this might not always be the case. As shown in Listing 4-2, we might need to start multiple depth- first searches from different initial nodes in order to fully cover the graph.

The simplicity of depth-first search can be a drawback. The decision of which neighbor to explore is arbitrary (here based on index ordering), instead of using information we might have about the world. If our adventurer is in a labyrinth with an exit to the west, they might prioritize moving west to moving east. We will see some approaches for incorporating such heuristic information in Chapter 8.

Depth-First Search with a Stack

Instead of using recursion, we can also implement depth-first search as an iterative function by using a *stack*.

The Code

Listing 4-3 uses the standard Python `list` as our stack (with `append` serving as the traditional stack `push`).

```
def depth_first_search_stack(g: Graph, start: int) -> lis
t:
    seen: list = [False] * g.num_nodes
    last: list = [-1] * g.num_nodes
    to_explore: list = []

  ❶ to_explore.append(start)
  ❷ while to_explore:
      ❸ ind = to_explore.pop()
        if not seen[ind]:
            current: Node = g.nodes[ind]
            seen[ind] = True

          ❹ all_edges: list = current.get_sorted_edge_list
```

```
()
           ❺ all_edges.reverse()
             for edge in all_edges:
                 neighbor: int = edge.to_node
                 if not seen[neighbor]:
                     last[neighbor] = ind
                     to_explore.append(neighbor)
    return last
```

Listing 4-3: A stack-based depth-first search

The code for the iterative depth-first search starts by initializing our helper data structures. In addition to the `seen` and `last` lists, the function also uses a stack named `to_explore` to track which node indices it needs to explore in the future. The function starts by pushing the initial node onto the `to_explore` stack ❶.

The majority of the work in the function is done within a `while` loop that iterates over the elements in `to_explore` until the stack is empty ❷. At each iteration, the top index is popped from the stack ❸ and, if it hasn't previously been seen, is explored. As in the recursive function, the code retrieves the node data structure and marks the index as having been seen. It then retrieves a list of all edges ❹. A `for` loop iterates over all outgoing edges; the code sets the `last` value of the as-yet-unseen nodes and adds them to the stack.

For consistency of ordering with other examples, the code in Listing 4-3 reverses the list to examine neighbors in *decreasing* order of the neighbor's index ❺. This is not a necessary component of the algorithm.

An Example

Figure 4-5 shows the execution of the iterative depth-first search using a stack. As in Figure 4-4, the current node is indicated with a dashed circle and the visited nodes are shaded.

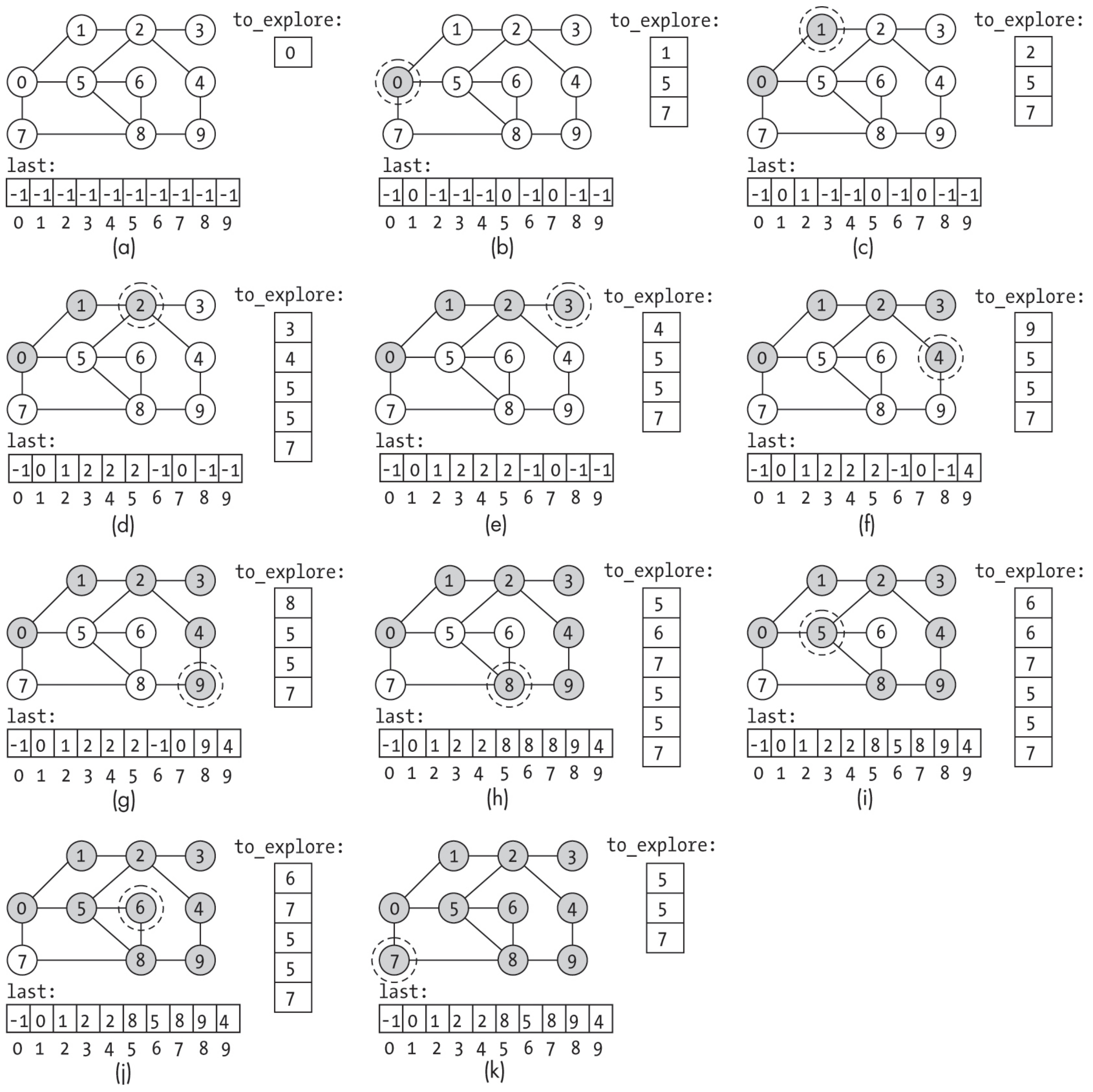


Figure 4-5: An iterative depth-first search on a graph with 10 nodes

This implementation of the stack-based approach differs from the recursive implementation in two interesting ways. First, the last array changes to reflect the *latest* path leading to the node before the node is visited. In Figure 4-5(b), the last list indicates the path to node 5 comes from node 0, because the search has seen node 5 is a neighbor of node 0. However, as the depth-first search progresses, the entry for node 5 is updated. In Figure 4-5(d), the search finds a more recent path to node 5 through node 2. In Figure 4-5(h), it finds another path through node 8.

Second, the stack of nodes to explore in this case includes *duplicates*, such as the three instances of node 5 in Figure 4-5(h). This is because, as described in the previous paragraph, depth-first search may see multiple paths to a node as it explores more deeply. These duplicates do not affect the accuracy of the algorithm because we check that a node is unvisited whenever we pop an index off the stack. However, they can increase memory usage. With modifications, and an additional running-time cost, we could extend the code to keep only the instance of the index that is highest on the stack.

The difference between the recursive and stack-based approaches corresponds to how our explorer tracks their journey through the labyrinth. In both cases, they note which rooms they have visited in their `seen` notebook. In the stack-based approach, however, they maintain a second notebook labeled `to_explore`. Instead of just walking into unvisited adjacent rooms, as in the recursive approach, the explorer carefully writes out all unvisited rooms adjacent to their current room. Before changing rooms, they check which room they most recently added to their notebook and proceed to that one.

Finding Connected Components

We can use depth-first search to find the sets of connected components in an undirected graph. As discussed in Chapter 3, a connected component in an undirected graph is a set of nodes such that each node in the set can reach every other node in the set. If we start a depth-first search from a single node in the graph, it will visit only the nodes reachable from that starting node. In an undirected graph, these visited nodes make up a connected component. By rerunning the depth-first search from any unseen nodes, as in Listing 4-2, we can map all connected components in a graph.

The Code

The following code conducts a depth-first search from each unseen node, while also maintaining information on which node is in which connected component:

```
def dfs_recursive_cc(g: Graph, ind: int, component: list,
 curr_comp: int):
  ❶ component[ind] = curr_comp
    current: Node = g.nodes[ind]

    for edge in current.get_edge_list():
        neighbor: int = edge.to_node
      ❷ if component[neighbor] == -1:
            dfs_recursive_cc(g, neighbor, component, curr_
comp)

def dfs_connected_components(g: Graph) -> list:
    component: list = [-1] * g.num_nodes
    curr_comp: int = 0

    for ind in range(g.num_nodes):
        if component[ind] == -1:
          ❸ dfs_recursive_cc(g, ind, component, curr_comp)
            curr_comp += 1

    return component
```

The code modifies the recursive depth-first search function so that it uses a single list (`component`) to track both whether a node has been visited and, if so, which component it is in. The recursive function starts by setting the current node's component ❶. Before exploring a neighbor, it checks whether it is already part of an existing component (and thus visited) ❷.

The outer function starts by setting up the helper data structures, including a list mapping each node to its component number (`component`) and a counter of the current component number (`curr_comp`). As with the exhaustive depth-first search from Listing 4-2, the code then iterates over each node and checks whether it has been visited. If not, it starts a depth-first search from that node ❸. During each depth-first search, it fills in more values of the `component` list.

An Example

Figure 4-6 shows an example of this algorithm on a graph with three connected components. Shaded circles indicate the nodes seen after each iteration, and the dashed circle indicates the starting node for that iteration's search.

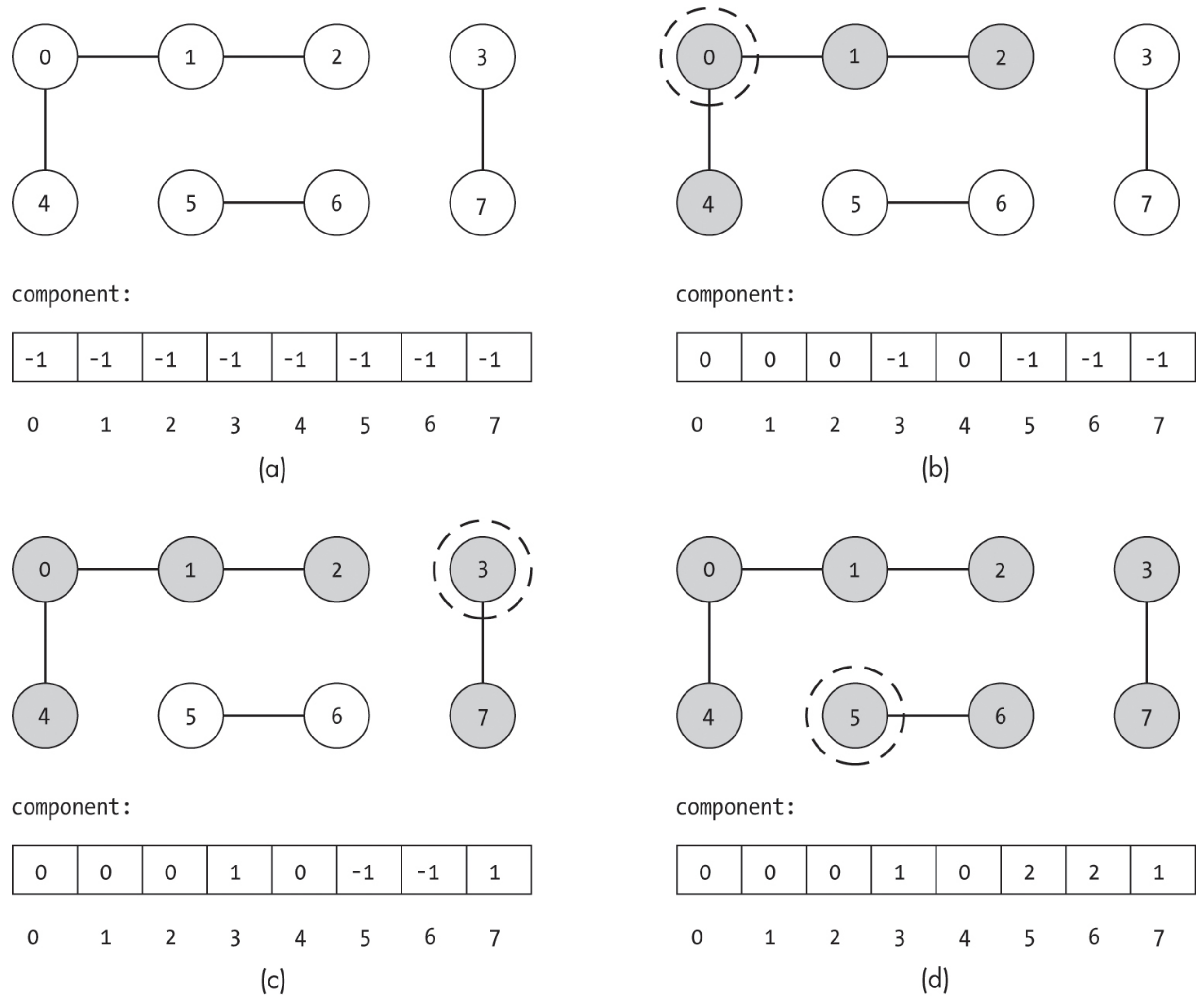


Figure 4-6: The steps of connected component detection based on depth-first search

Figure 4-6(a) shows the state of the graph before the first search, when none of the nodes have been seen or assigned a component number. As shown in Figure 4-6(b), the first search starts at node 0 and finds the component {0, 1, 2, 4}. Figure 4-6(c) shows the second search starting at node 3, the first unseen node, and finding the component {3, 7}. The final search in Figure 4-6(d) starts at node 5 and finds the component {5, 6}.

Depth-First Search Trees and Forests

If we save the edges traversed during a depth-first search, we can capture useful information about the structure of both the search and the graph itself. The connected components from the previous section are only one such type of information. Consider searching an undirected graph like the one shown in Figure 4-7(a).

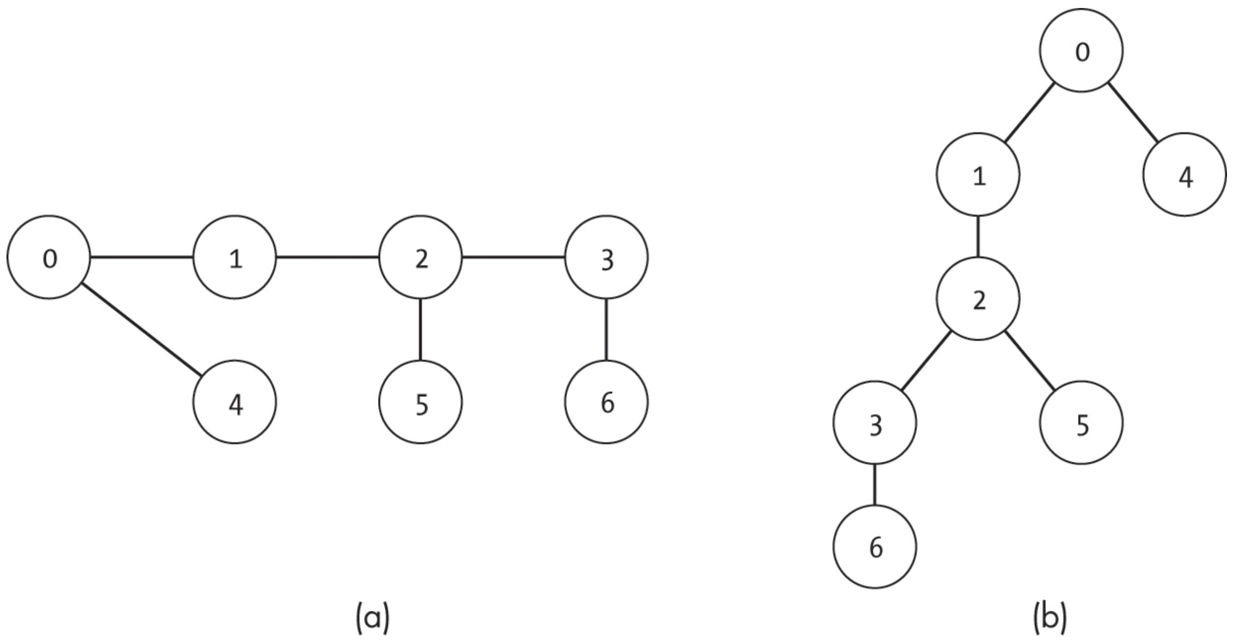


Figure 4-7: An undirected graph (a) and an example depth-first search tree (b)

The order in which a particular search explores the nodes (and traverses the edges) defines a tree structure called a *depth-first search tree* (or sometimes just a *depth-first tree*) that summarizes the search. Each edge traversed is included in the tree. The hierarchy of nodes is determined by the order in which depth-first search encounters them. If the search progresses from node *u* to an unvisited node *v*, then *u* will be the parent of *v* in the tree. Alternatively, using the concept of our `last` array from the code in this chapter, the parent of a tree node within index *i* is `last[i]`. Figure 4-7(b) shows the depth-first search tree for a search starting at node 0.

Depth-first search trees are not unique, but rather they depend on where the search starts, as shown in Figure 4-8. Starting the search at node 2 of the same undirected graph in Figure 4-8(a) leads to a different depth-first search tree in Figure 4-8(b).

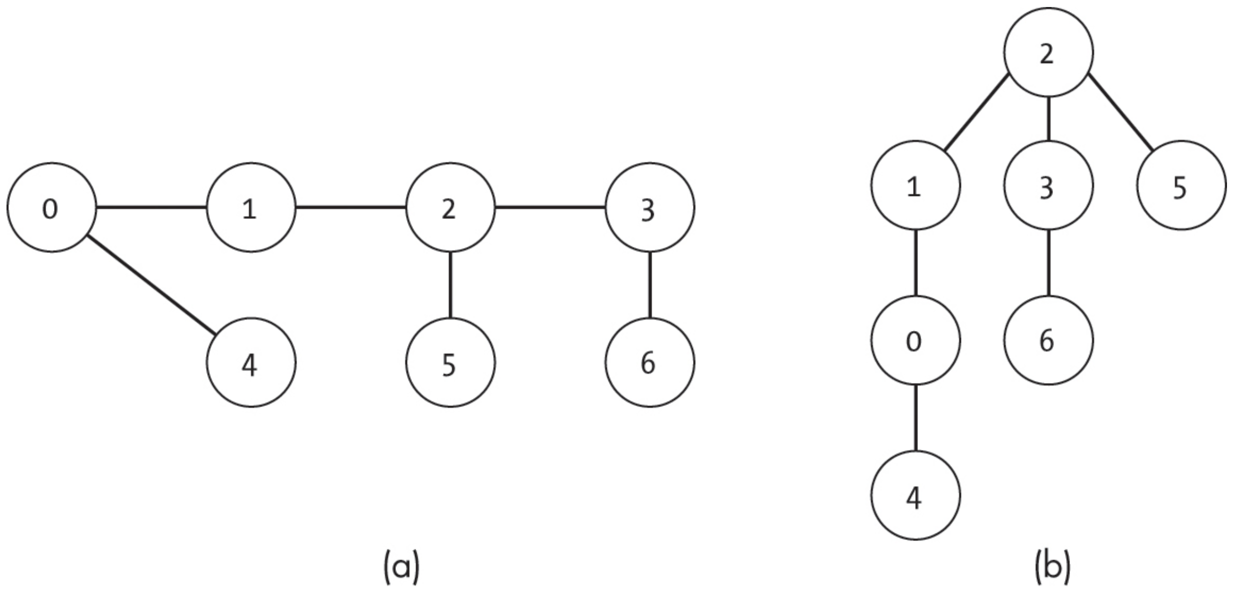


Figure 4-8: An undirected graph (a) and an alternate depth-first search tree (b)

As noted earlier, a single depth-first search might not explore the entire graph, meaning we might need to run multiple depth-first searches for completeness. This gives rise to the concept of a *depth-first search forest* (or, alternately, just a *depth-first forest*), in which each individual depth-first search generates a single tree data structure with the initial node as the root. The forest is the collection of individual trees. As shown in Figure 4-9, this arises naturally for undirected graphs anytime there are disconnected components. The two disconnected components {0, 1, 2, 3, 4, 6} and {5, 7, 8} in Figure 4-9(a) form two different trees in Figure 4-9(b).

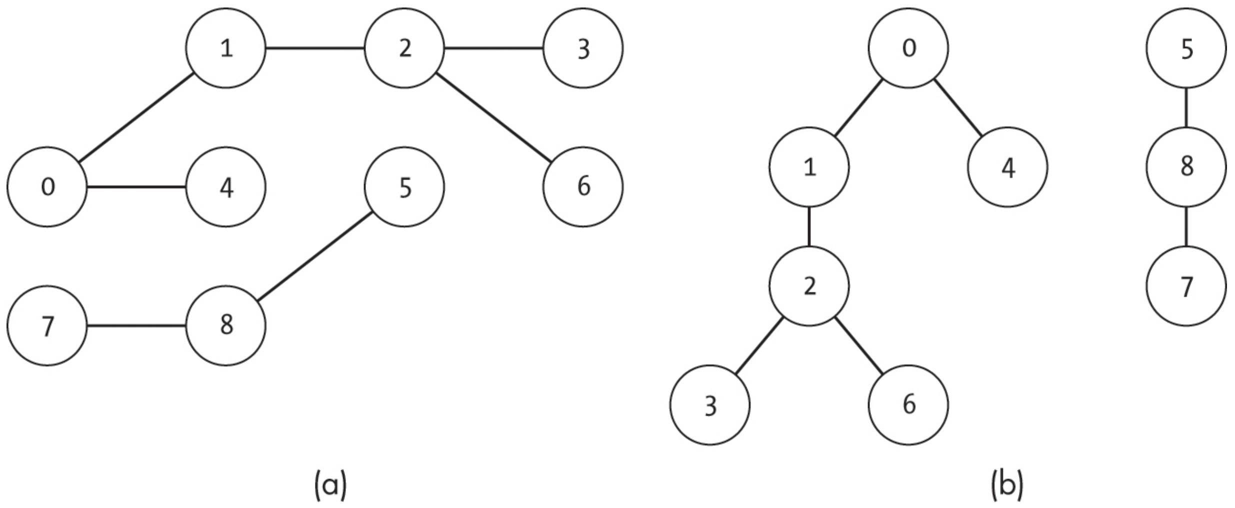


Figure 4-9: An undirected graph with two non-connected components (a) and example depth-first search trees (b)

In directed graphs, whether we need to run multiple searches can depend on our chosen starting node. Figure 4-10 shows an example directed graph and its depth-first search forest. Figure 4-10(a) depicts the original directed graph, while Figure 4-10(b) shows the depth-first search forest that results from checking the nodes in order of increasing index.

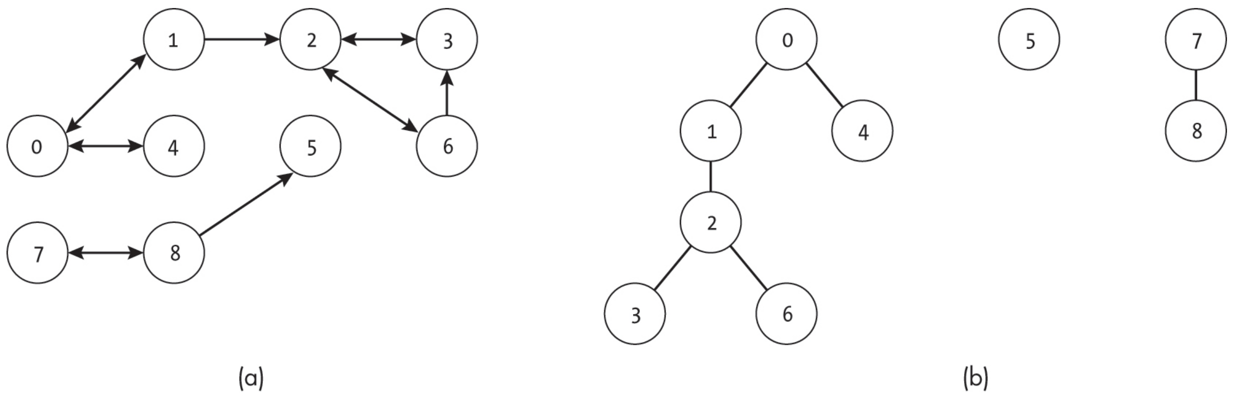


Figure 4-10: An example graph (a) and a corresponding depth-first search forest (b)

Because we are ordering by increasing index, we check node 5 before either node 7 or 8. This results in a separate tree for 5 in our forest because we cannot reach any other node from 5. However, 5 is reachable by nodes 7 and 8. If we had checked either of those nodes first, then node 5 would have been in their tree. The structure of the forest is determined by both the graph and the order in which we search the nodes.

In later chapters we will use the structure of depth-first search trees to help us understand the behavior of depth-first search itself. For now, just know that these trees capture information about the progression of depth-first search through a given graph.

Iterative Deepening

One major disadvantage of depth-first search is that it can waste time on long (or deep) dead ends when there is a closer state of interest. Imagine a spelunker lost in an underground cave system. Multiple paths branch out ahead of them, some leading toward the surface, others turning deeper into the caves. To make it out alive, they would like to use a search strategy that doesn't require a 10-mile trip farther underground only to hit a dead end, forcing them to backtrack and try a different option.

Iterative deepening is a strategy to limit excessively deep paths during a depth-first search. Instead of continuing along one path until it ends, the algorithm starts with a predefined depth and cuts off exploration when that depth is reached. If the entire search returns without finding the objective, iterative deepening increases the maximum depth and reruns the search. It continues this process until the objective is found or the entire graph is searched.

Consider what happens if our lost spelunker tethers themselves to a fixed-length rope. The spelunker uses the rope to limit how far into the cave system they're willing to venture. They follow one path until they hit the end of the rope. Even if there are more passages ahead, they backtrack and explore alternative paths that are still reachable with the current tether. Only when they have exhausted all possible paths do they upgrade to a longer tether. This prevents them from going too far in the wrong direction before trying some of the other options.

At first glance, iterative deepening might seem like a colossal waste. It ends up exploring nearby nodes multiple times (with multiple max depths). A node one step from the starting node would be explored every iteration. Similarly, our spelunker will visit the first intersection multiple times.

Yet this approach can be useful in some situations. Consider a tree-like graph as shown in Figure 4-11. A normal depth-first search will progress down a single branch until the very end. If the tree is deep, this can be quite a few nodes.

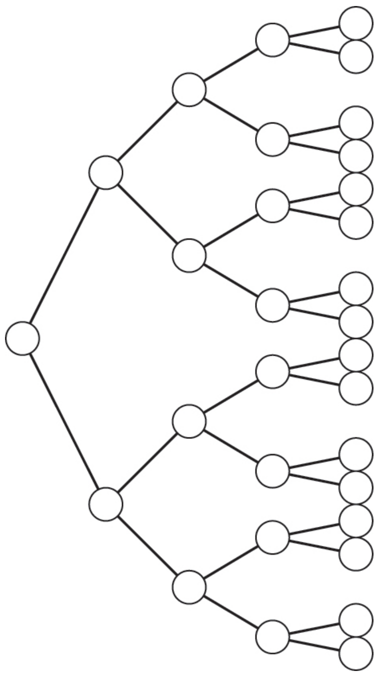


Figure 4-11: A graph that branches out like a tree

In contrast, as shown in Figure 4-12, iterative deepening effectively searches the tree level by level. During the first iteration (max depth of 1), only three nodes are explored. During the second iteration (max depth of 2), seven nodes are explored, including four new nodes.

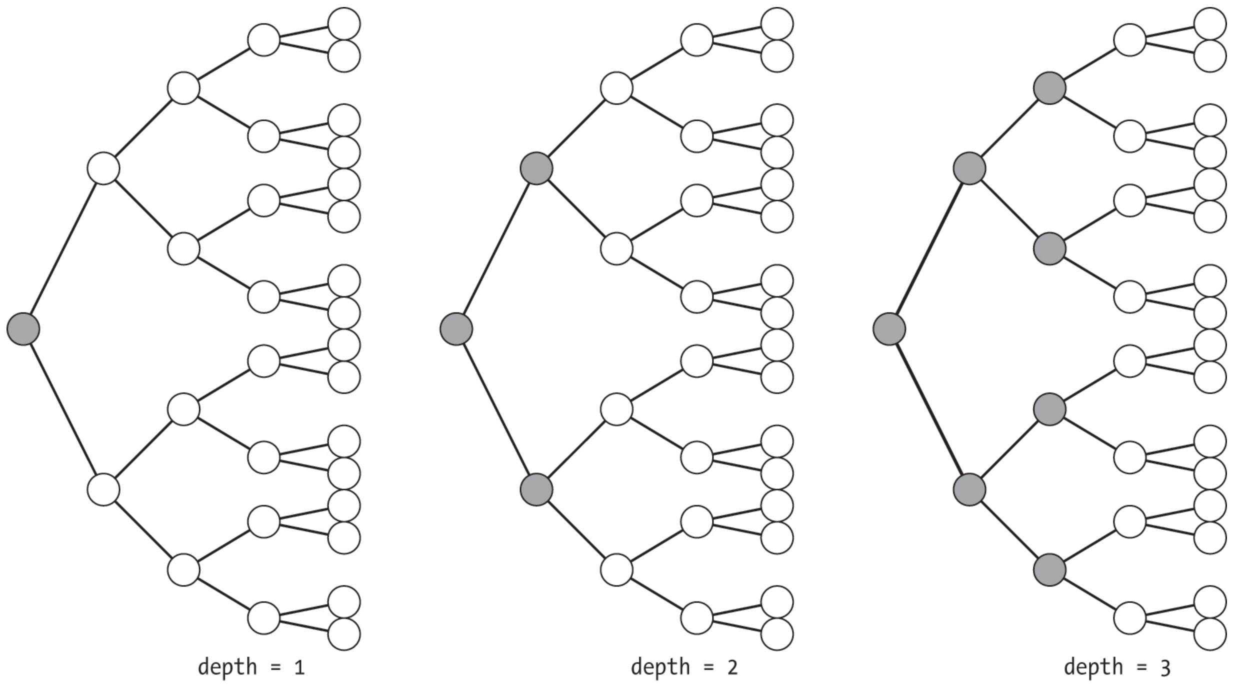


Figure 4-12: The first three iterations of iterative deepening

For a balanced, complete binary tree like the one in Figure 4-11, each iteration takes approximately twice the time and explores twice the nodes of the iteration before it. As we will see in the next chapter, iterative deepening results in a search pattern similar to breadth-first search.

Why This Matters

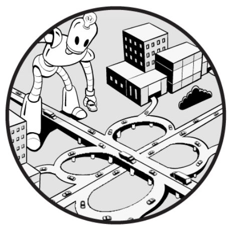
Depth-first search is a core graph algorithm that we will use throughout the rest of this book, building numerous extensions from its simple recursive formulation. In the world of graph algorithms, this search is a fundamental building block. Later in the book, we'll use graph search algorithms, including many modifications of depth-first search, to uncover the inherent ordering of nodes in directed graphs or propose node labeling in graph coloring.

Unfortunately, depth-first search is not always the perfect solution. It does not use heuristics when choosing the next node to explore. Worse, it is prone to traversing long dead ends.

Upcoming chapters will cover both techniques that build from depth-first search and alternate search algorithms that avoid some of its pitfalls. First, however, we consider a different type of search: breadth-first search.

5

BREADTH-FIRST SEARCH



Breadth-first search is an alternative approach for exploring graphs that progresses like a wave from a starting node. Whereas depth-first search prioritizes recently discovered nodes, breadth-first search prioritizes exploring nodes discovered earlier in the search. This simple change in prioritization leads to radically different behavior of the search algorithm, along with a variety of useful properties.

The key idea behind breadth-first search is that it explores nodes using a first-in, first-out ordering, such as that provided by a queue. Each time the search encounters a previously unseen node, it places that node in a queue of nodes to explore later. When it's ready to move on to the next node, it doesn't look at the current node's neighbors, but rather it extracts the node from the front of the queue, meaning that it always picks the node that has been waiting the longest.

In the previous chapter, we visualized depth-first search as an adventurer exploring a labyrinth. We can picture breadth-first search as the same adventurer using a different strategy, meticulously working their way through a list of future rooms to explore. Determined to visit uncharted territories as soon as possible, the explorer enumerates all unexplored rooms in a list. Each time they find a new room, they append it to the bottom of the list. Resisting the temptation to ditch their plans and rush into this new location, they consult their list and turn to the topmost unvisited option instead.

This chapter introduces breadth-first search and examines its properties. In particular, breadth-first search finds the shortest path from a node to all other reachable nodes in unweighted graphs, making it a useful component in a variety of more complicated algorithms.

Use Cases

Breadth-first search maps naturally onto numerous real-world tasks, such as learning new concepts or exploring a new city.

Learning New Topics

Breadth-first search provides a systematic approach to learning new concepts that focuses on building up the foundations before investigating any one area too deeply. Suppose you're learning a new programming language. Each node in the graph represents a concept you must learn, while the edges between the nodes represent pointers between the concepts. Perhaps you're reading a chapter on Python that discusses both its syntax and its execution model. These concepts become neighbors of the current chapter, which we can either explore immediately or put on the list to explore later.

A breadth-first search approach to learning prioritizes the concepts that have been on our "to learn" list the longest. You might start at the general concept of the Python language, then note the immediate neighbor topics of syntax, execution model, and running a sample program. Each topic goes on your list to explore, and you proceed through them one at a time. While learning about syntax, you come across references to lists, sets, and dictionaries. Instead of flipping ahead to those chapters, you add each concept to the bottom of your list to explore later and continue to the next topic at the top of the list. This means you're able to complete a simple "Hello, world!" program before delving into lambda expressions.

Exploring a New City

Suppose you're exploring a new city, building your knowledge base by establishing a known area of explored nodes, then expanding the frontier into the unknown. One day after work, you travel that extra block to try the coffee shop you've seen down the street. Another day, you ask the age-old question, "What's over that hill?" As you discover new neighborhoods, you write down newly discovered but as-yet-unvisited areas for later adventures.

Contrast this breadth-first approach with the admittedly more adventurous depth-first method of walking in one direction until you hit the city limits. The latter tactic will efficiently expand the areas seen, but at the cost of deferring closer options. If you always go north when you have the option, you might end up 50 blocks north, yet remain completely unaware of the amazing coffee shop one block south of your apartment.

The Breadth-First Search Algorithm

Breadth-first search operates by maintaining a *queue of nodes* and iteratively exploring them until the queue is empty. The nodes in the queue are reachable from a visited node but have not been visited themselves.

Breadth-first search begins by inserting a *starting node* into the queue. In many applications, the choice of the starting node is obvious. If you are browsing an online encyclopedia of coffee grinders, for example, the starting node is the first page you open. If you are searching for a path from your hotel to the closest coffee shop, the starting node will be the hotel. If you are searching through your social network for someone who can get you concert tickets, the starting node is yourself, the center of the network. In the examples in this chapter, we arbitrarily use node 0 as our starting point.

At the start of each iteration of a breadth-first search, the algorithm dequeues the first node and visits it. It then checks that node's outgoing edges and adds any previously unvisited neighbors to the queue. The process continues, one node at a time, until the queue is empty.

If the graph being explored is not fully connected, the search terminates before visiting every node. In many cases, this is exactly what we want. If we are looking for a path from the hotel to a coffee shop, we do not care about unreachable coffee shops. Perhaps we are on a tropical island with a road network connecting our hotel to 10 coffee shops. Our search would spread out to find coffee shops across the island but would stop short of suggesting cafés on neighboring islands.

However, in some cases, we need a more comprehensive search. If we are searching for information on coffee grinders, we do not want to miss important context simply because someone neglected to add a hyperlink. We can extend the breadth-first search to exhaustively explore every node by adding the first unexplored node to the queue whenever the queue is empty.

The Code

The code for breadth-first search consists of a `while` loop exploring new nodes until the queue of pending nodes is empty:

```
def breadth_first_search(g: Graph, start: int) -> list:
    seen: list = [False] * g.num_nodes
    last: list = [-1] * g.num_nodes
    pending: queue.Queue = queue.Queue()

  ❶ pending.put(start)
    seen[start] = True

    while not pending.empty():
        index: int = pending.get()
        current: Node = g.nodes[index]

        for edge in current.get_edge_list():
            neighbor: int = edge.to_node
          ❷ if not seen[neighbor]:
                pending.put(neighbor)
                seen[neighbor] = True
                last[neighbor] = index

    return last
```

The `breadth_first_search()` code starts by creating the helper data structures, including the list of which nodes have been seen (`seen`), a list of previous nodes in the search to represent the path (`last`), and a queue (`pending`). For the queue, we use the `Queue` data structure defined in Python's `queue` library, which requires an additional `import queue` in the file. However, it's also possible to use a built-in data structure such as a `list`. Before the main loop begins, the code inserts the starting node into the `pending` queue and marks it as seen ❶.

The function uses a `while` loop to continue exploring nodes until the queue is empty. During each iteration, it takes the node from the front of the queue and uses a `for` loop to check each of its neighbors. If the function has not already seen the neighbor ❷, it adds it to the queue, marks it as seen, and updates the pointer to the previous node. The function finishes by returning the `last` list, which captures the path taken by the search.

Unlike the implementations of depth-first search in Chapter 4, which marked a node as seen when the search visited it, this breadth-first search implementation marks nodes as seen when it first adds them to the queue. This prevents repeat entries in the queue.

An Example

Figure 5-1 shows an example of the steps of a breadth-first search through a graph with 10 nodes. The shaded nodes have been marked seen. Each subfigure shows the settings of the `last` array and the state of the queue with the front on the left-hand side.

Figure 5-1(a) represents the state of the search before we start the `while` loop. The search has marked the starting node (0) as seen and placed it on the queue. All other nodes remain marked unseen. Every node has a back pointer (`last`) of `-1`, including the starting node.

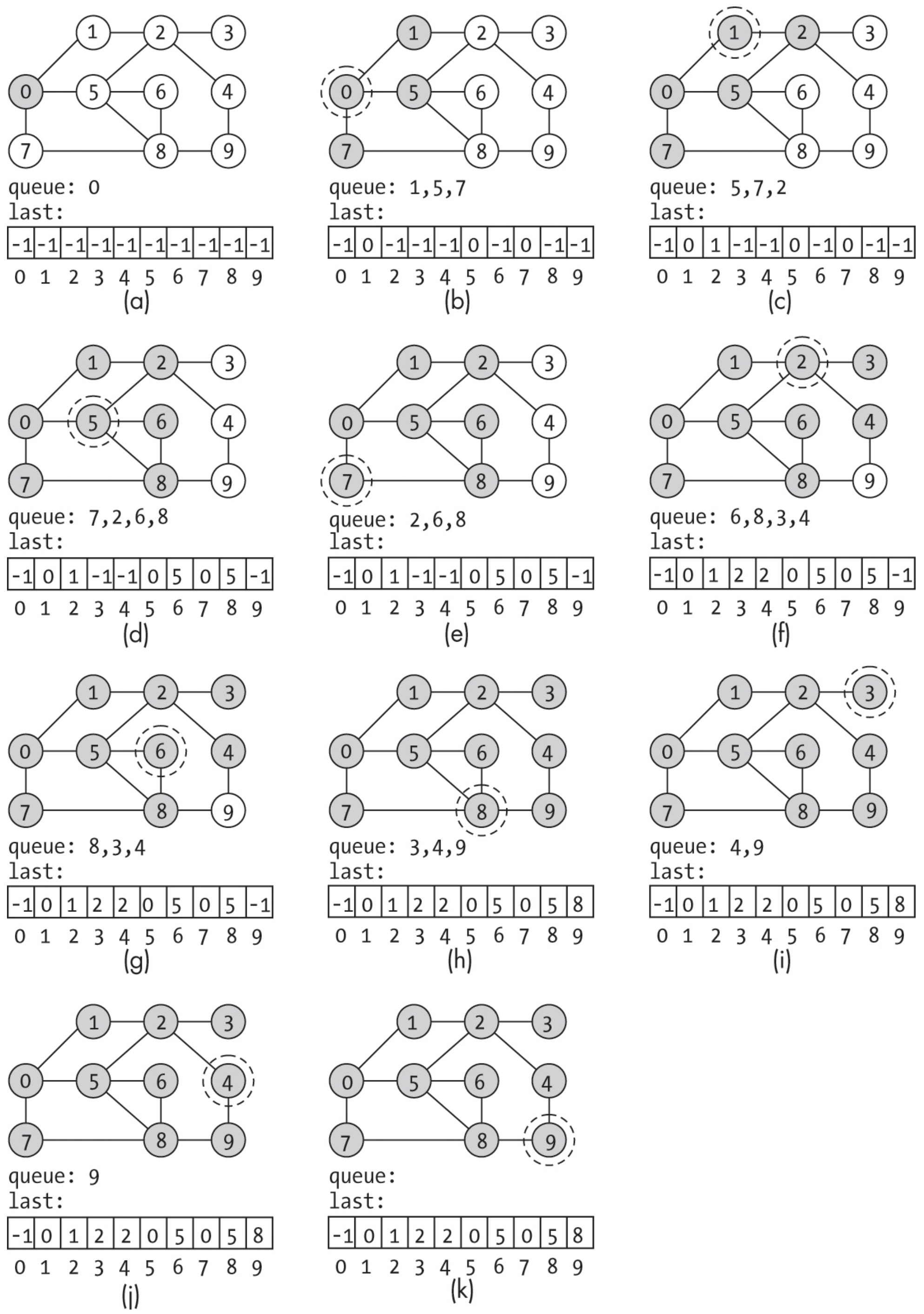


Figure 5-1: The steps of a breadth-first search

The search begins in earnest in Figure 5-1(b), when it removes node 0 from the queue and explores it. It finds three neighbors (nodes 1, 5, and 7), marks each of these as seen, and places them in the queue to explore later. In this way, the queue is much like a daily to-do list: we cross an item off, realize that it leads to more tasks, and add those to the end of the list. We also update the last array to indicate the path we will take to each of these nodes. The value 0 indicates that node 0 precedes each of the nodes on the path.

The search next visits node 1 in Figure 5-1(c). This node has only one unseen neighbor (node 2), since we have already seen node 0. The algorithm marks node 2 as seen, adds it to the queue, and sets its entry in the `last` array to `1`.

In Figure 5-1(d), we begin to see how this search diverges from the depth-first search. Rather than continuing down the current path, breadth-first search explores the earliest-seen node that is still unexplored. In this case, it moves to node 5. In the context of the adventurer exploring a labyrinth, this corresponds to exploring the room at the top of our hero's list. Admittedly, this could be inefficient in the physical world. The adventurer may need to backtrack through much of the labyrinth to return to that room. However, this is not a problem in the computational realm. Once we have the index of a node, we can easily load it into memory.

In Figure 5-1(d), the algorithm explores node 5 and finds two new neighbors, nodes 6 and 8. It marks each of these as seen and updates their last entries to point back to node 5. It then places 6 and 8 at the end of our list of nodes to explore; it will investigate them in due time.

The search continues in Figure 5-1(e) by exploring the last of node 0's neighbors with a visit to node 7. While we have now seen a significant fraction of the nodes in the example graph, we have visited only nodes that are one or fewer steps from node 0. Our search is spreading like a wave from the starting node, visiting all the close nodes before hitting ones farther away.

The search continues through the rest of the figure. At each step, it retrieves the node at the front of the queue for exploration. This is the node that has been waiting longest on its list. It visits that node, checking for and processing any new neighbors. The search completes in Figure 5-1(k) when it extracts the final node from the queue.

Finding Shortest Paths

A major benefit of breadth-first search is that it will find the shortest paths from the starting node to all reachable nodes on an unweighted graph. When working with unweighted graphs, we use the term *shortest paths* to indicate paths with the fewest number of edges. Breadth-first search is able to find such paths thanks to how the algorithm prioritizes which paths it explores. By using a first-in, first-out data structure, the algorithm effectively prioritizes unexplored nodes that are closest to the start node.

Figure 5-2 illustrates this behavior, depicting the graph from Figure 5-1 with dashed lines indicating the number of steps to each node. While exploring the starting node, breadth-first search enqueues all nodes that can be reached in one step from this node. Next, it explores each of those nodes in order. While exploring a node one step away, it might find new nodes that are two steps away. However, these are always added to the end of the queue and thus explored after all the nodes that are one step away have been visited. As a result, breadth-first search sweeps through all nodes *k* steps away before considering any that are *k* + 1 steps away.

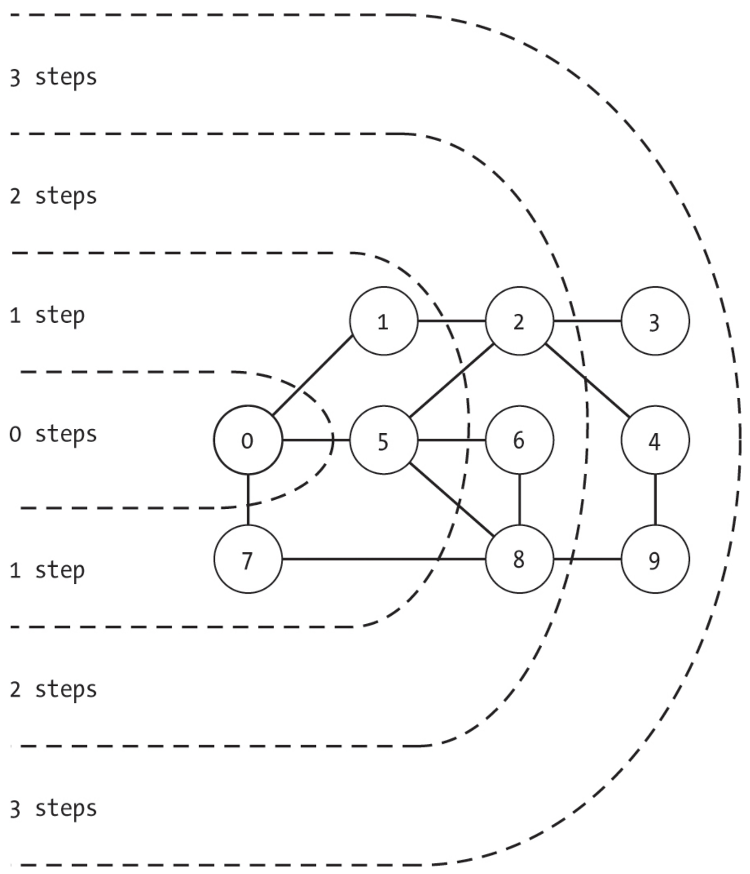


Figure 5-2: Contour lines showing the expansion of breadth-first search

In weighted graphs, the concept of *shortest paths* is often used to describe the paths with the lowest sum of edge weights. Since breadth-first search does not consider edge weight, a breadth-first search will not find the shortest path in terms of cost.

As an example, consider the graph in Figure 5-3. Both nodes 1 and 2 will be seen by breadth-first search during the exploration of node 2. Both will be marked as seen, added to the queue, and assigned 0 as the previous node on their path, regardless of the edge weight to that node. The algorithm does not look for or find the lower weight path through node 1 to node 2, but it still finds the one with the fewest number of edges.

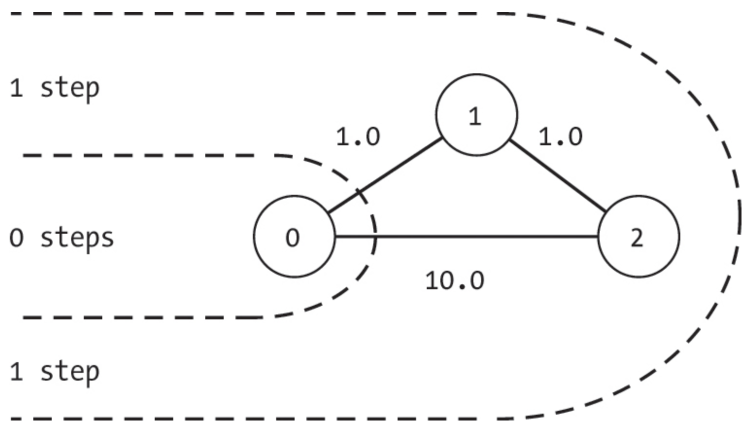


Figure 5-3: Contour lines showing the expansion of breadth-first search on a weighted graph

We'll consider the question of edge weights and shortest paths later in the book. Chapter 7 introduces a variety of shortest-path algorithms on weighted graphs, while Chapter 8 examines heuristic search algorithms that account for edge weights.

Simple Path Planning

The fact that breadth-first search finds paths with the smallest number of edges makes it useful for a limited set of *path-planning* tasks. Consider the old-time video game task of planning a path on a flat two-dimensional grid where some squares are obstructed by boulders. This section discusses how to create a graph to represent this problem and the solution we get when running breadth-first search.

Breadth-first search path planning on a flat plane with obstacles provides a useful illustration of the operation of breadth-first search. It also introduces how we can think about representing path-planning problems in graph form, preparing us for later chapters that cover better algorithms for path planning. These include *lowest-cost path algorithms* that account for edge weights to simulate differing costs of various terrain and *heuristically guided search algorithms* that improve the running time of the search itself by prioritizing the most promising paths.

Constructing a Graph from a Grid

Constructing a graph representation of a regular grid has a variety of uses, from path planning to computer vision. The underlying grid might represent spatial regions on a map (for path planning or scientific computation), pixels in an image (for computer vision), or even just an arrangement of nodes. For the purposes of this section, we'll focus on a video game map.

We generate a grid-based graph by creating a single node for each grid square and a single undirected edge linking each pair of adjacent squares. For a grid where the number of rows = *height* and number of columns = *width*, we start by allocating *height* × *width* nodes. We can visually represent these in a grid pattern, as shown in Figure 5-4(b), but technically they are stored in a single list within a graph data structure.

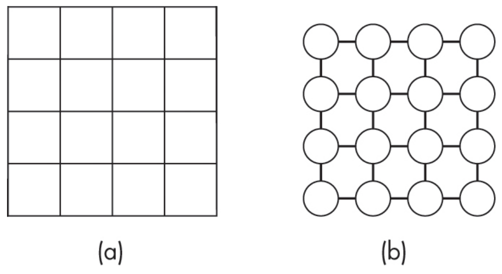


Figure 5-4: A grid (a) and a graph representation of the grid (b)

We can map the grid coordinates for row r and column c to a corresponding node index as follows:

$$index = r \times width + c$$

Because the edges are undirected, we can scan the nodes from top left to bottom right and insert edges to the nodes to the right of or below the current node. If we specify this as a loop over the *r* and *c* values of the grid, our tests of whether to include an edge are as follows:

If *c* < *width* – 1, the node has a neighbor to its right at *index* + 1.

If *r* < *height* – 1, the node has a neighbor below it at *index* + *width*.

Listing 5-1 shows the code for constructing a grid-based graph.

```
def make_grid_graph(width: int, height: int) -> Graph:
    num_nodes: int = width * height

    g: Graph = Graph(num_nodes, undirected=True)
    for r in range(height):
        for c in range(width):
          ❶ index: int = r * width + c

          ❷ if (c < width - 1):
                g.insert_edge(index, index + 1, 1.0)
          ❸ if (r < height - 1):
                g.insert_edge(index, index + width, 1.0)
    return g
```

Listing 5-1: Creating a graph representation of a grid

The `make_grid_graph()` code starts by creating an undirected graph `g` with one node for each grid square. It then loops over all the grid cells with two `for` loops, computes the corresponding node index ❶, and checks whether there should be an edge to the right of ❷ or below ❸ the current node. The code completes by returning the graph `g`.

Adding Obstacles

Path planning across a flat plane is not a particularly exciting task. Even in the context of describing breadth-first search, it amounts to little more than watching the frontier of visited nodes expand across the grid. To make the example more interesting, let's add obstacles to the grid. We pass these to the code in the form of tuples (*r*, *c*) that indicate the obstacle's row and column in the grid. This allows us to make grids like the one in Figure 5-5, where the open cells are traversable and the shaded circles represent obstacles.

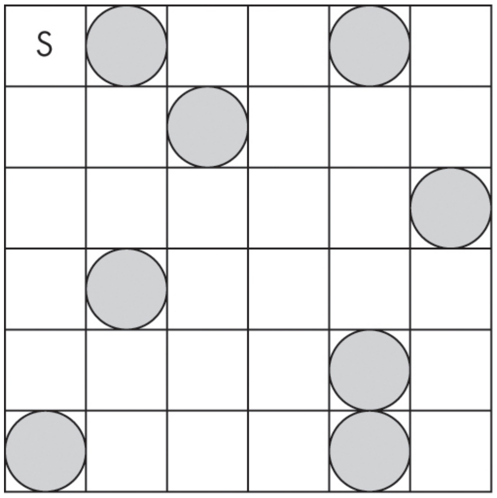


Figure 5-5: A 6×6 grid with eight obstacles

The code follows the same form as Listing 5-1 but uses this extra information:

```
def make_grid_with_obstacles(width: int, height: int,
                             obstacles: set) -> Graph:
    num_nodes: int = width * height

    g: Graph = Graph(num_nodes, undirected=True)
    for r in range(height):
        for c in range(width):
          ❶ if (r, c) not in obstacles:
                index: int = r * width + c
              ❷ if (c < width - 1) and (r, c + 1) not in o
bstacles:
                    g.insert_edge(index, index + 1, 1.0)
              ❸ if (r < height - 1) and (r + 1, c) not in
 obstacles:
                    g.insert_edge(index, index + width, 1.
0)
    return g
```

As in Listing 5-1, the code starts by creating an undirected graph `g` with one node for each grid square. It then loops over all the grid cells with two `for` loops. At each grid square, it checks whether the current cell is blocked by an obstacle ❶. If it is blocked, the node has no edges into or out of it. If it isn't blocked, the code computes the index in the node list, checks whether there should be an edge to the right of the current node ❷, and checks whether there should be an edge below the current node ❸. Both edge checks include the additional constraint that the neighboring cell must not be blocked by an obstacle.

This function demonstrates the power of general graph representation, which allows us to fully capture the structure of the environment, including valid transitions and obstacles. If we can transition directly from one point to another, the graph has an edge between the corresponding nodes. Otherwise, no transition is allowed. We do not need to save the dimensions of the graph or the list of obstacles. We can use similar approaches to model walls in a maze. As we will see in later chapters, we can use edge weights and directionality to further increase modeling power.

Running Breadth-First Search

We can run breadth-first search on the grid-based graphs from the previous two sections without any modifications. After all, both functions are producing standard `Graph` objects. Figure 5-6 shows the results of running breadth-first search on the grid from Figure 5-5, where *S* denotes the starting node at row 0, column 0.

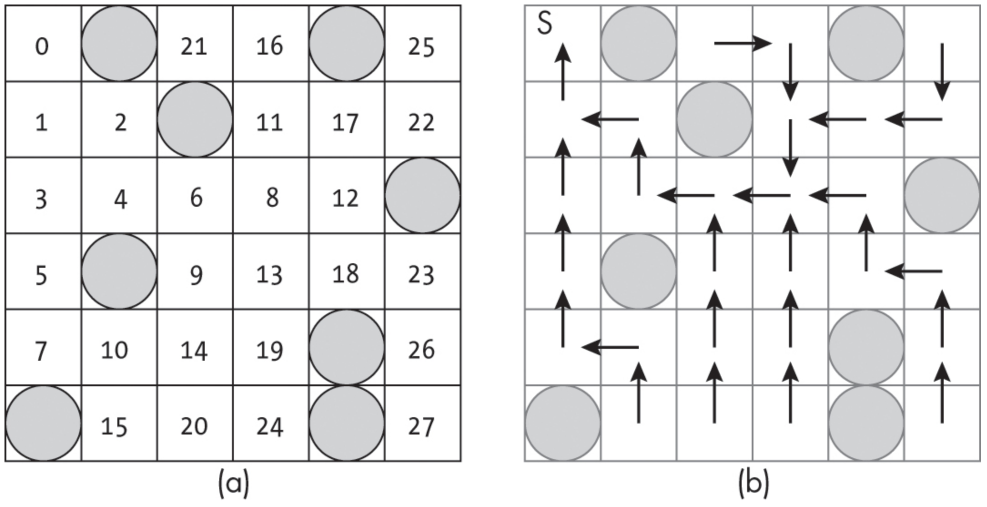


Figure 5-6: Grids with the exploration order (a) and last pointers (b) after breadth-first search

The grid in Figure 5-6(a) shows the order in which the search visits each grid cell. It starts in the upper left-hand corner and expands outward like an oozing blob creature attempting to absorb the world. While the other cells are close by straight-line distance metrics, it takes a while for the search to visit them, because the search needs to navigate around the obstacles.

The grid in Figure 5-6(b) shows the `last` pointers for each node in the graph. Using these, we can reconstruct the shortest path from any destination node back to the start node. For example, the cell in row 3, column 2 (labeled with a 9) can reach the origin by moving up, left, up, left, and up. We can reverse these pointers to get the shortest path *from* the start node to any reachable destination.

Why This Matters

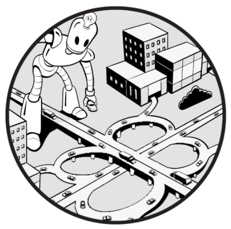
Breadth-first search provides an alternate mechanism for searching graphs with different behavior. Due to the way it orders nodes to explore, it prioritizes nodes that are closest to the starting node as measured by the number of edges. As a result, breadth-first search explores outward from the starting node in a frontier and will find paths to each node with the fewest edges. This behavior makes this search a useful component of various more complex graph algorithms.

In addition, breadth-first search is both simple and efficient. Unlike the common recursive implementation of depth-first search, the standard implementation of breadth-first search uses an iterative `while` loop rather than recursive function calls to operate on the queue.

The next chapters explore a different type of graph search: algorithms that find the shortest paths on weighted graphs. These algorithms build on the basic searches we have introduced so far and unlock a range of new applications.

6

SOLVING PUZZLES



The uses of graph search algorithms extend beyond searching paths between physical places or virtual links between items. They also apply to a wide range of more abstract problems, such as solving puzzles or devising strategies in games.

Many puzzles can be represented by a set of discrete states that capture the different configurations of the puzzle. *Solving the puzzle* might then consist of using a sequence of steps to transition from the initial state to some predefined goal state. This could correspond to moving discs on a Tower of Hanoi puzzle, moving people in a river-crossing puzzle, or rearranging tiles in a slider puzzle. In this chapter, we transform each of these puzzles into graph search problems by modeling the states of the puzzle as nodes and the transitions between them as edges. We then solve them by searching for a path to the puzzle's target state.

State Spaces and Graphs

This section describes the state space representations and graph representations of our three classic puzzles.

The Tower of Hanoi

The Tower of Hanoi puzzle consists of three pegs, along with discs of different diameters that can be stacked on the pegs. Initially, all the discs are on the leftmost peg and are stacked in order of decreasing diameter, as shown for a three-disc version in Figure 6-1, where the widest disc is on the bottom of the pile and the smallest is on top. The goal of the puzzle is to move the discs one at a time such that they all end up on the rightmost peg in the same order.

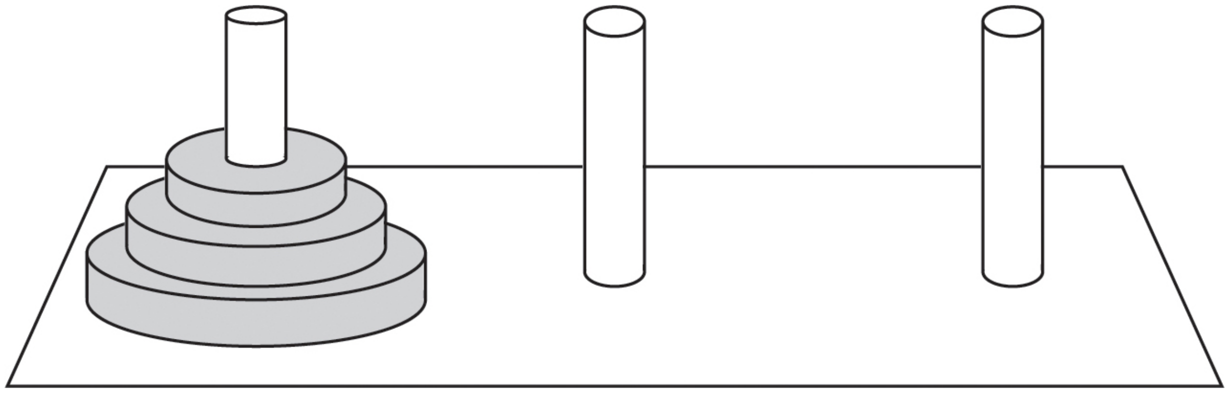


Figure 6-1: The initial state of the Tower of Hanoi puzzle

Two constraints make this puzzle interesting. First, you can move only the topmost disc of each stack. Second, at every step, each stack must maintain the sorted ordering. You are never allowed to place a wider disc on top of a smaller one. For example, Figure 6-2 shows a move from the initial state to the valid second state.

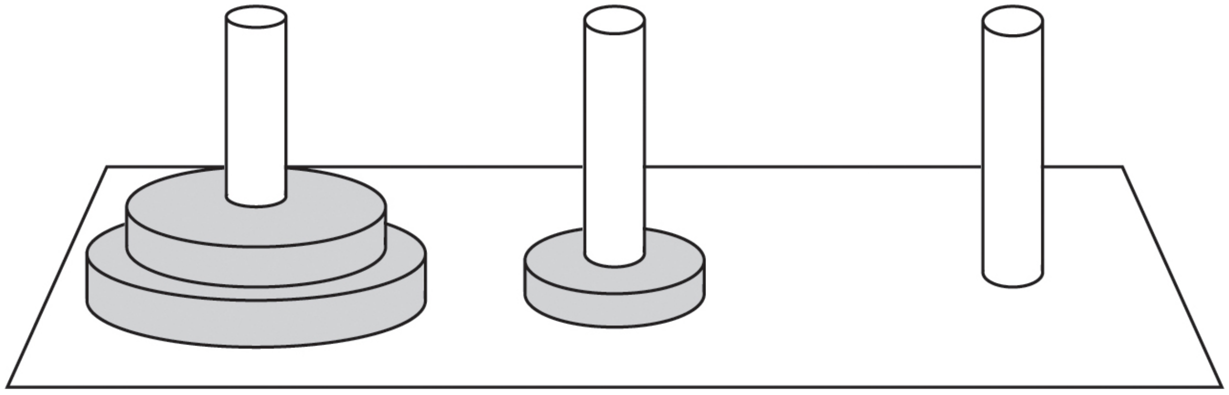


Figure 6-2: The Tower of Hanoi puzzle after moving one disc

We can represent the different configurations of discs as *states* in a variety of ways. To illustrate this, let’s consider a set of three vectors that track the ordered set of discs on each peg, where the combined vector set represents the state of the puzzle. We denote states within angle brackets. The state of the puzzle in Figure 6-1 is represented with the vectors `<[3,2,1], [],[]>`, while the state of Figure 6-2 is represented as `<[3,2],[1],[]>`.

We might think of these states as enumerating the universe of possibilities for the puzzle. The current arrangement of discs might be `<[3], [],[2,1]>`. However, if we had made different moves, we might have the arrangement `<[3],[1],[2]>`. We can sit back and envision the wide range of possible options, how to move between them, and the impact of future moves.

With the ability to enumerate the puzzle’s states as vectors, we can transform the problem into an undirected graph by using a single node for each state and edges for valid actions. This is illustrated in Figure 6-3 for the Tower of Hanoi with three discs, where the nodes are potential destinations along our problem-solving journey. One node represents our starting state of `<[3,2,1],[],[]>`, and one represents our target state of `<[],[],[3,2,1]>`, with a range of other states in between.

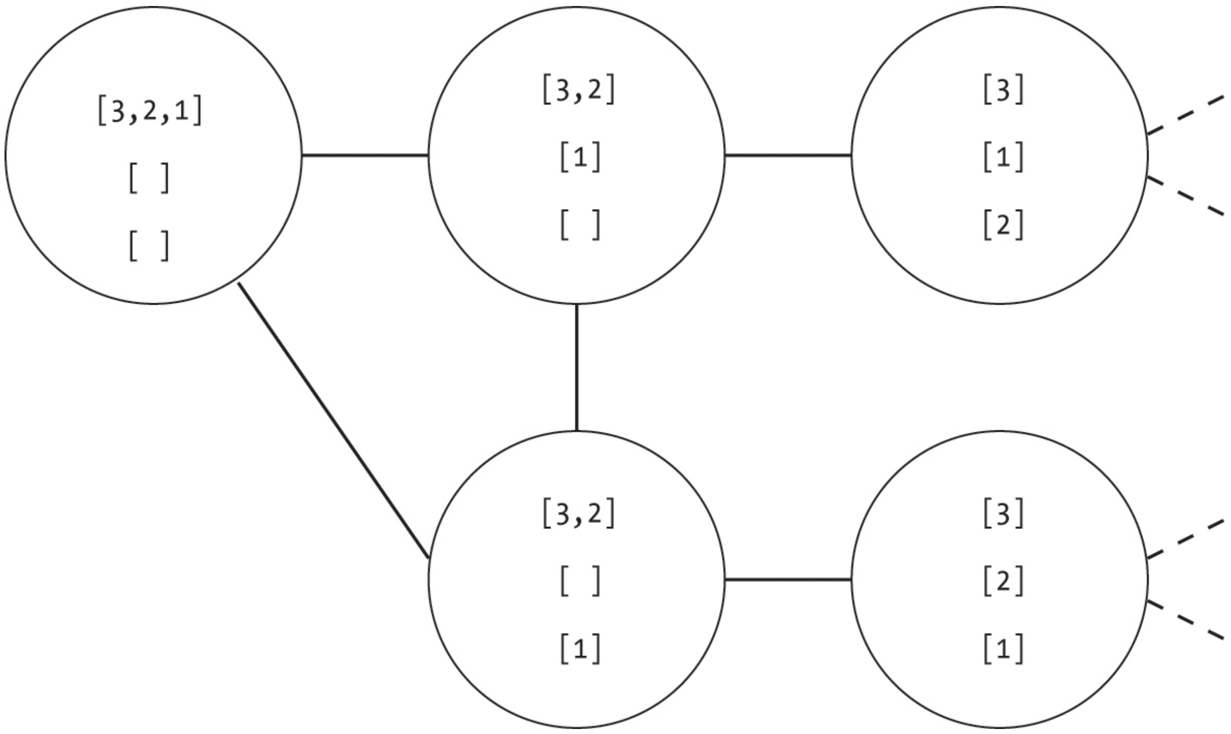


Figure 6-3: Part of the graph representation of the Tower of Hanoi puzzle

Since we can move from state `<[3,2,1],[],[]>` to state `<[3,2],[1],[]>` by moving the smallest disc from the first to the second peg, the graph contains an undirected edge between those nodes. In contrast, we cannot move from state `<[3,2,1],[],[]>` to state `<[2,1],[3],[]>` by moving a single disc, so there is no edge between those states.

Until now, example graph nodes in this book have mostly represented concrete items like physical locations, computer nodes, web pages, tasks, or people. Now, however, our nodes represent potential configurations of the world. One immediate consequence of this is an explosion in the number of nodes. There are many more potential configurations of discs in the Tower of Hanoi than there are physical discs or pegs in the puzzle. This means we may encounter much larger graphs than before and the performance of the algorithms becomes ever more important.

River-Crossing Puzzles

River-crossing puzzles are a class of brain teasers that ask about transporting a set of people or animals across a river in a boat with limited capacity.

Challenges arise from constraints on which entities are allowed to be left alone together on either bank.

Let's consider one classic river puzzle that we will describe as the prisoners-and-guards puzzle, in which three guards and three prisoners need to cross a river. At their disposal is a boat that can carry two people at most. The prisoners are handcuffed and cannot escape if left alone. However, if there are more prisoners than guards on a shore, the prisoners will gang up on the guards and steal their keys. Thus, on each shore, a guard must be accompanied by *at most* the same number of prisoners.

We can represent the puzzle as a *state space graph*, where each node represents a state of the puzzle. The puzzle's state consists of three pieces of information: the number of guards on the left bank, the number of prisoners on the left bank, and whether the boat is on the left or right bank. The number of guards and prisoners on the right bank can be derived from the number on the left, so we do not need to explicitly store that information. The starting state for the problem is `<3,3,L>`, with all six people and the boat on the left bank.

Each node in the graph represents a single valid state, while edges link states that are reachable via a single move. Valid moves include sending any combination of one or two people across in the boat: two guards, one guard, one guard and one prisoner, two prisoners, or one prisoner. The boat cannot travel across without anyone, as someone needs to steer it. Since we can always undo any move by sending the same configuration back across, the graph is undirected.

Figure 6-4 shows the entire graph of 16 possible states. Each state is a single node with both a graphical and a textual representation of the state. The letters `G` and `P` represent the locations of guards and prisoners, respectively. The position of the boat is shown and represented with an `R` or `L` in the state at the bottom. For many states there are only two valid moves, but for others there are multiple options.

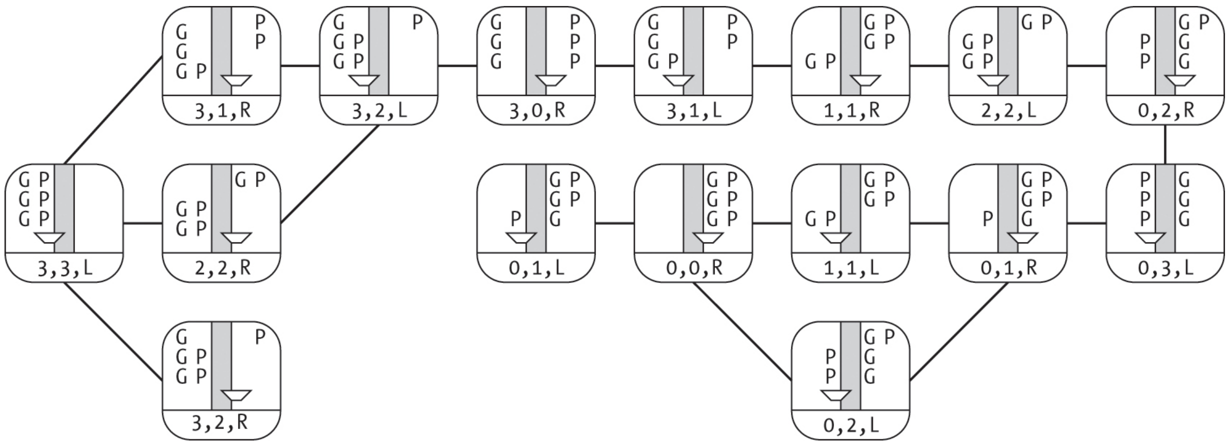


Figure 6-4: The graph representation of the prisoners-and-guards river-crossing puzzle

This puzzle illustrates the use of state representations while using a relatively small graph and makes it easy to visualize and analyze the reachable set of states. In later sections we'll use this puzzle to show how we can programmatically create graphs and search for solutions.

Slider Puzzles

A slider puzzle consists of a grid of square tiles on a board with one square missing to leave a gap. Tiles can be rearranged by sliding an adjacent tile into the gap, effectively allowing the tile and gap to switch places. The goal of the puzzle is to move each of the tiles into the correct position. Depending on the puzzle, we might be tasked with ordering a sequence of numbers or unscrambling a picture. A classic example of this game is the 15-square, shown in Figure 6-5, where each tile is labeled with an integer from 1 to 15 and the correct state has all the tiles arranged in ascending order from top left to bottom right.

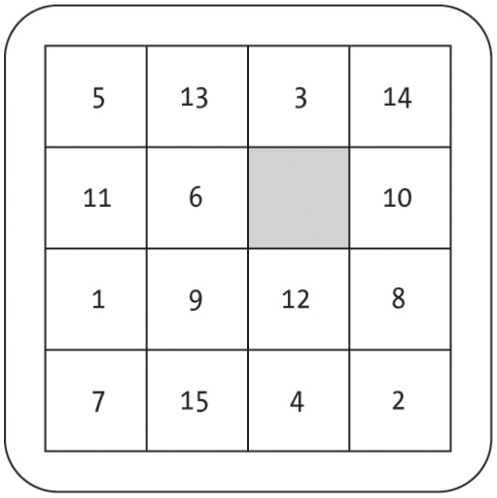


Figure 6-5: The 15-square puzzle

The slider puzzle lends itself well to graph representation. Each possible arrangement of tiles is a unique puzzle state and can be represented by a graph node. Edges indicate possible traversals between states. At most, each state has four undirected edges, representing the neighbors that can be reached by filling that state's gap position with each of the four adjacent tiles. We can call these edges Up, Down, Left, and Right.

Figure 6-6 shows an example state and its four neighbors. By searching through the graph for a path from the initial state's node to the goal state's node, we can find a sequence of moves that solves the puzzle.

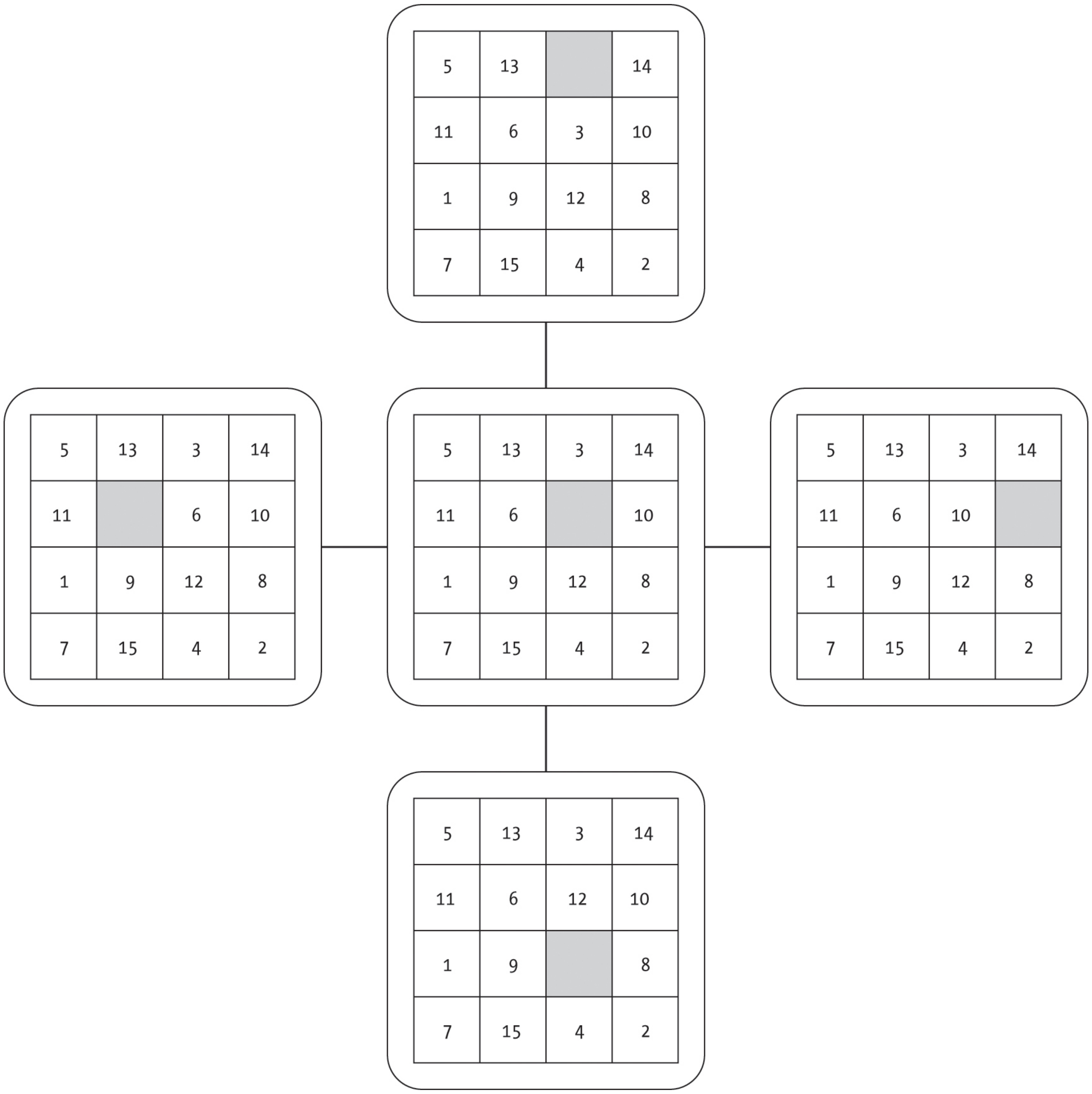


Figure 6-6: A single state of the 15-square puzzle (center) and its four neighbor states

The 15-square puzzle demonstrates how quickly the state space can explode. This apparently simple puzzle has over 20 trillion states, many of which will never need to be visited on our path to the solution.

Constructing a Graph with Search

The graph search algorithms from the previous chapters required us to provide a fully specified graph. For puzzle problems, this might not be feasible. We don’t want to laboriously enumerate their many, many states by hand before we can start the search. Not only would this be time-consuming, but it would also likely lead to errors and accidentally induce the puzzle equivalent of cheating—adding an edge that allows an illegal move. Even worse, we would waste a huge amount of effort generating states that either are not used in the solution or are not reachable at all.

Instead, we can dynamically create a graph using a search algorithm by extending the breadth-first and depth-first search algorithms to explore the state space, adding nodes and edges on the fly. Each time we discover a new state, we add the corresponding node. Each time we test a move between states, we add the corresponding edge. Unlike previous searches that iterated over each node or each edge out of a node, using a search to build the graph requires the algorithm to iterate over puzzle states and the valid moves out of each state.

Throughout the rest of this section, we examine how we can build out such graphs using the river-crossing puzzle. We start with an initial state `<3,3,L>` and explore outward from there. At each step, we ask, “What is the next state if we send some combination of people (two guards, one guard, one guard and one prisoner, two prisoners, or one prisoner) across the river?” We compute the next state with simple arithmetic and check its validity using the puzzle’s rules. If the new state is valid, we add it and the corresponding edge to our graph. Throughout the next few sections, we will build out the code to store the state space, define valid transitions, and construct the graph. While we focus on the river-crossing puzzle, the approaches apply to a range of puzzles.

Representing the Puzzle’s States

In order to define our search, we first need to represent the puzzle’s state. The following code shows how to define a simple class `PGState` (where `PG` stands for “prisoners and guards”) to store the current state of the puzzle and provide some helper functions:

```
class PGState:
    def __init__(self, guards_left: int = 3, prisoners_lef
t: int = 3,
                 boat_side: str = "L"):
```

```
        self.guards_left = guards_left
        self.prisoners_left = prisoners_left
        self.boat_side = boat_side

    def __str__(self) -> str:
        return (f"{self.guards_left},{self.prisoners_lef
t},{self.boat_side}")
```

The variables `guards_left` and `prisoners_left` store the number of guards and prisoners on the left riverbank. The string `boat_side` indicates whether the boat is currently on the left (`L`) or the right (`R`) bank. The `__str__()` function allows us to easily convert the state into a string representation for storage and display.

Given the `PGState` data structure, it is easy to programmatically compute the result of moving a given number of guards and prisoners as part of the next boat trip, as shown in the following code:

```
def pg_result_of_move(state: PGState, num_guards: int,
                      num_prisoners: int) -> Union[PGStat
e, None]:
  ❶ if num_guards < 0 or num_prisoners < 0:
        return None
    if num_guards + num_prisoners == 0:
        return None
    if num_guards + num_prisoners > 2:
        return None

  ❷ G_L: int = state.guards_left
    G_R: int = (3 - state.guards_left)
    P_L: int = state.prisoners_left
    P_R: int = (3 - state.prisoners_left)
    if state.boat_side == "L":
        G_L -= num_guards
        G_R += num_guards
        P_L -= num_prisoners
        P_R += num_prisoners
        new_side: str = "R"
    else:
        G_L += num_guards
```

```
            G_R -= num_guards
            P_L += num_prisoners
            P_R -= num_prisoners
            new_side = "L"

❸   if G_L < 0 or P_L < 0 or G_R < 0 or P_R < 0:
        return None

❹   if G_L > 0 and G_L < P_L:
        return None
    if G_R > 0 and G_R < P_R:
        return None
    return PGState(G_L, P_L, new_side)
```

The majority of the `pg_result_of_move()` function checks whether the move is valid. If the move is invalid, the function returns `None`. Otherwise, it will return the new `PGState` corresponding to the result of the move. Note that this requires us to import `Union` from Python's `typing` library to support type hints for multiple return types. The code checks that the number of prisoners and number of guards are both nonnegative, that there is at least one person in the boat, and that there are at most two people in the boat ❶. If any of the validity checks fail, the move is not valid, and the function returns `None` to indicate there is no next valid state.

If the move passes these first validity checks, the code computes the resulting number of prisoners and guards on the left shore (`P_L` and `G_L`, respectively) and the number of prisoners and guards on the right shore (`P_R` and `G_R`) ❷. These four counts are used to check whether the new state is valid. The code checks that the move is not relocating more people than are on the current shore by confirming that none of the counts become negative ❸. It also checks that the new state has a valid balance of guards and prisoners ❹. If there is at least one guard on a shore, then there cannot be more prisoners than guards on that shore. However, it is valid to have only prisoners on a shore. Once again, if any of the validity checks fail, the new state is not valid, and the function returns `None`. If all checks pass, the code returns a `PGState` data structure representing the new state.

While `pg_result_of_move()` checks and computes the result of a single move, we need to construct edges for each valid move out of a state.

We can define a helper function to generate and test all possible neighbors of the current state:

```
def pg_neighbors(state: PGState) -> list:
    neighbors: list = []
  ❶ for move in [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]:
      ❷ n: Union[PGState, None] = pg_result_of_move(state,
move[0], move[1])
        if n is not None:
            neighbors.append(n)
    return neighbors
```

This code creates an empty list of neighbors (`neighbors`), then systematically tries all five possible moves: one guard, two guards, one prisoner, two prisoners, and one guard plus one prisoner ❶. Each time, the code calls `pg_result_of_move()` and checks whether a valid neighboring state is returned ❷. If so, it adds the new state to the list of neighbors.

Generating the Graph

Now that we have the components to algorithmically determine which states neighbor the current state, we can use a modified breadth-first search to generate the state space graph for the prisoners-and-guards puzzle. This algorithm will start from the initial state and explore outward along edges to adjacent states. We'll use the `pg_neighbors()` helper function from the last section to determine the set of valid neighboring states from the current state. As new states are discovered by the neighbor generation function, we'll add those states to the graph as new nodes.

We track the state information in the `PGState` data structure. For convenience, we will link to this state information as a `PGState` object assigned to the node's label. This makes the current state data structure readily accessible during the search. We use the `__str__()` method to produce a string representation of the data structure for use in helper data structures.

In addition to the data structures used in previous breadth-first searches, we need to track one additional piece of information: a mapping from the state to its node in the graph. It doesn't help us to know that an edge exists between `<2,2,R>` and `<3,3,L>` if we can't then find the corresponding nodes and create the edge. We store this information in a dictionary (`indices`) that maps a string representation of the state to the corresponding node's index in the graph.

The code to create the state graph for prisoners and guards combines the pieces we've previously assembled:

```
def create_prisoners_and_guards() -> Graph:
    indices: dict = {}
    next_node: queue.Queue = queue.Queue()
    g: Graph = Graph(0, undirected=True)

  ❶ initial_state: PGState = PGState(3, 3, "L")
    initial: Node = g.insert_node(label=initial_state)
    next_node.put(initial.index)
    indices[str(initial_state)] = initial.index

    while not next_node.empty():
      ❷ current_ind: int = next_node.get()
        current_node: Node = g.nodes[current_ind]
        current_state = current_node.label

      ❸ neighbors: list = pg_neighbors(current_state)
        for state in neighbors:
            state_str: str = str(state)
          ❹ if not state_str in indices:
                new_node: Node = g.insert_node(label=stat
e)
                indices[state_str] = new_node.index
                next_node.put(new_node.index)
          ❺ new_ind: int = indices[str(state)]
            g.insert_edge(current_ind, new_ind, 1.0)

    return g
```

The code for generating the graph starts by setting up the necessary data structures: an empty dictionary (`indices`), an empty queue (`next_node`), and an empty graph (`g`). It creates a new `PGState` object for the initial state, inserts the corresponding node into the graph with the `Graph` class's `insert_node()` function, adds the initial state's node index to the queue, and adds the string to index mapping to the dictionary ❶.

We are now ready to start the search itself. Like other breadth-first searches, our prisoners-and-guards graph generation uses a queue of node indices (`next_node`) to control the search. The next node index is dequeued, and the corresponding node and state are retrieved ❷. Unlike previous breadth-first search examples, the algorithm cannot rely on the node's edge list to determine neighbors. Instead, the code uses the `pg_neighbors()` function to generate possible neighboring states ❸.

The code checks whether each state has been seen before by searching for its string representation in the `indices` dictionary ❹. If the state does not have an entry in the table (and a valid node index), we have neither seen it nor added it to the graph yet. New nodes are created for any previously unseen states, and new edges are generated between the current node and its neighbor ❺.

The code concludes by returning the completed graph `g`. Since the algorithm only searches outward from the initial state, the returned graph will include only states that are reachable from the initial state by valid moves. Nodes for invalid or unreachable states are not included.

Figure 6-7 shows the first few steps of the algorithm's progress. The single node corresponding to the graph's initial `<3,3,L>` state is shown in Figure 6-7(a). Figure 6-7(b) shows the result after the first node is visited: the `pg_neighbors()` function finds three valid neighbors for the current state, and the algorithm creates new nodes for each one. After exploring the `<3,1,R>` state, the code creates a `<3,2,L>` node and the corresponding edge, as shown in Figure 6-7(c).

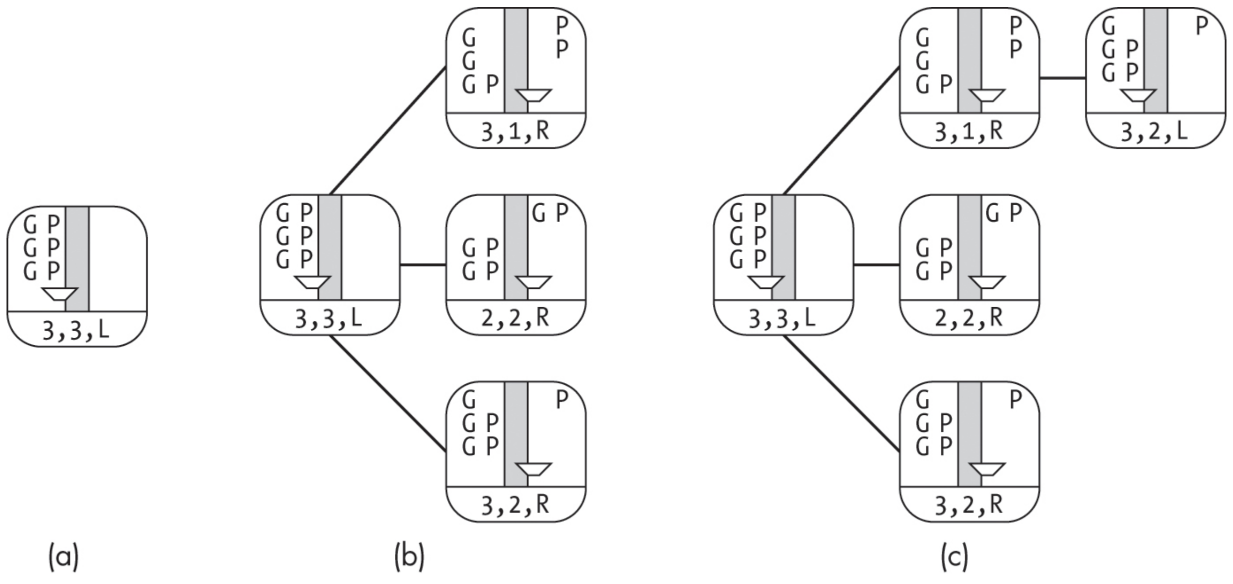


Figure 6-7: The first three steps of generating the graph for the prisoners-and-guards puzzle

The algorithm generates new nodes whenever it first sees them. However, it does not necessarily generate all the node's edges until it visits that node, which is why there is no edge from state `<3,2,L>` to state `<2,2,R>` in Figure 6-7(c). The code will generate the edge between those nodes only when it visits either `<2,2,R>` or `<3,2,L>`.

Solving a Puzzle with Search

We can apply the searches from past chapters directly to the prisoners-and-guards puzzle graph. We add the search functions to the same prisoners-and-guards program, building off the previous section's functions.

To simplify the logic, we start with a simple helper function that creates a dictionary mapping the state's string to the corresponding node's index:

```
def pg_state_to_index_map(g: Graph) -> dict:
    state_to_index: dict = {}
    for node in g.nodes:
        state: str = str(node.label)
        state_to_index[state] = node.index
    return state_to_index
```

The resulting map of state string to node index allows us to look up the start and goal indices without tracing through the graph. We can look up the index of the starting node (0) from the string `"3,3,L"`. Similarly, we can look up the goal node's index (14) from its state string `"0,0,R"`.

Here's the code for searching the puzzle:

```
def solve_pg_bfs():
  ❶ g: Graph = create_prisoners_and_guards()

  ❷ state_to_index: dict = pg_state_to_index_map(g)
    start_index: int = state_to_index["3,3,L"]
    end_index: int = state_to_index["0,0,R"]

  ❸ last: int = breadth_first_search(g, start_index)

  ❹ current: int = end_index
    path_reversed: list = []
    while current != -1:
        path_reversed.append(current)
        current = last[current]

  ❺ if path_reversed[-1] != start_index:
        print("No solution")
        return

  ❻ for i, n in enumerate(reversed(path_reversed)):
        print(f"Step {i}: {g.nodes[n].label}")
```

The code starts by creating the puzzle's graph representation ❶. It then builds a dictionary `state_to_index`, which maps the state string to the index, and uses that to look up the indices of the starting and goal nodes ❷.

The code uses a standard breadth-first search to explore the graph, returning the `last` list ❸. Finally, it traverses the `last` list backward from the goal node until it reaches the starting node or a dead end ❹. If the path dead-ends before reaching the starting node, the function displays the message `No solution` ❺. Otherwise, the code walks the path forward and displays the list of states visited in the correct order ❻.

Figure 6-8 shows the generated graph with node indices. Each node is labeled with both its node index (top) and its state string (bottom).

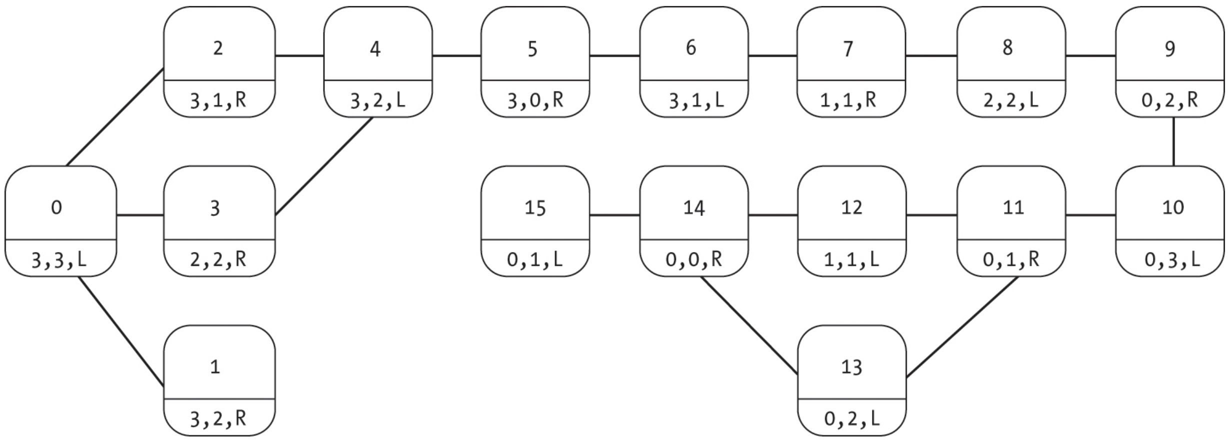


Figure 6-8: The graph for prisoners and guards with node indices shown

Given the generated puzzle graph, we can directly run the breadth-first search from Chapter 5. Table 6-1 shows the state of the `last` vector after each node is explored. The first row corresponds to iteration 0 and state `<3,3,L>`. The goal (state `<0,0,R>`) is visited during iteration 14.

Table 6-1: The Progression of the `last` Vector

Step (node)	**3 3 L**	**3 2 R**	**3 1 R**	**2 2 R**	**3 2 L**	**3 0 R**	**3 1 L**	**1 1 R**	**2 2 L**	**0 2 R**	**0 3 L**	**0 1 R**	**1 1 L**	**0 2 L**	**0 0 R**	**0 1 L**
0 (3,3,L)	–1	0	0	0	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
1 (3,2,R)	–1	0	0	0	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
2 (3,1,R)	–1	0	0	0	2	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
3 (2,2,R)	–1	0	0	0	2	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
4 (3,2,L)	–1	0	0	0	2	4	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
5 (3,0,R)	–1	0	0	0	2	4	5	–1	–1	–1	–1	–1	–1	–1	–1	–1
6 (3,1,L)	–1	0	0	0	2	4	5	6	–1	–1	–1	–1	–1	–1	–1	–1
7 (1,1,R)	–1	0	0	0	2	4	5	6	7	–1	–1	–1	–1	–1	–1	–1
8 (2,2,L)	–1	0	0	0	2	4	5	6	7	8	–1	–1	–1	–1	–1	–1
9 (0,2,R)	–1	0	0	0	2	4	5	6	7	8	9	–1	–1	–1	–1	–1
10 (0,3,L)	–1	0	0	0	2	4	5	6	7	8	9	10	–1	–1	–1	–1
11 (0,1,R)	–1	0	0	0	2	4	5	6	7	8	9	10	11	11	–1	–1
12 (1,1,L)	–1	0	0	0	2	4	5	6	7	8	9	10	11	11	12	–1
13 (0,2,L)	–1	0	0	0	2	4	5	6	7	8	9	10	11	11	12	–1
14 (0,0,R)	–1	0	0	0	2	4	5	6	7	8	9	10	11	11	12	14

	3	**3**	**3**	**2**	**3**	**3**	**3**	**1**	**2**	**0**	**0**	**0**	**1**	**0**	**0**	**0**
	3	**2**	**1**	**2**	**2**	**0**	**1**	**1**	**2**	**2**	**3**	**1**	**1**	**2**	**0**	**1**
Step (node)	**L**	**R**	**R**	**R**	**L**	**R**	**L**	**R**	**L**	**R**	**L**	**R**	**L**	**L**	**R**	**L**
15 (0,1,L)	–1	0	0	0	2	4	5	6	7	8	9	10	11	11	12	14

Based on the `last` vector that breadth-first search produces, we can track the moves necessary to take us from the initial state to the goal. Remember that breadth-first search returns the shortest path on unweighted graphs, so it has found us the solution requiring the fewest moves.

We can use similar approaches for the Tower of Hanoi and the slider puzzle. Each time, we would start by defining the data structure for the state space and a function to algorithmically generate a state's neighbors.

Why This Matters

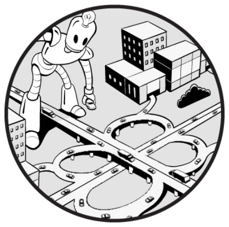
This chapter introduced how to apply graph search to the abstract world of solving puzzles. Beyond the relatively simple puzzles covered here, we can model increasingly complex problems by incorporating the rich structure of the graph, including directed and weighted edges. While the puzzles in this chapter used undirected edges to represent reversible moves, for example, we can represent a move that cannot be taken back with a directed edge. If we're trying to solve a puzzle that involves crossing a bridge that will collapse after one trip, we cannot directly return to the state where we are on the other side of the ravine with the bridge intact. Similarly, weighted edges allow us to account for the cost of moves.

The code in this chapter also demonstrated that we do not have to generate graphs before starting our searches, but instead we can use a search to create the graph data structure as we explore the various states. In many cases, we might not need to explicitly create the graph data structure at all. Once we have defined the states and transitions, we can apply graph algorithms like breadth-first search directly to them. For the remainder of this book, we will refer to modeling problems as graph problems even in cases where we do not explicitly build a graph.

In the next chapter, we will return to the question of computing paths through graphs, expanding beyond the search-based approaches we have seen so far to find the least-cost path through graphs with weighted edges.

7

SHORTEST PATHS



The problem of finding the lowest-cost path through a graph, as measured by the sum of edge weights along the path, is naturally analogous to a range of real-world path planning and optimization tasks. For example, we may be interested in routing trucks between two cities such that we minimize the total cost of fuel. In this chapter, we consider algorithms to find paths with the minimum cost starting at a given origin.

While the problem of finding these paths is often called the *shortest-path problem*, it is more accurate to think of it in terms of lowest-cost path, since a path's cost is not always a function of distance. For example, this chapter will also consider versions of the problem that allow negative edge weights. We will use the terms *shortest path* and *lowest-cost path* interchangeably throughout the chapter; the formulations and implementations are the same.

This chapter covers three algorithms for finding shortest paths. We start with *Dijkstra's algorithm*, which, like the search algorithms in the previous chapters, explores outward from a starting node. The *Bellman-Ford algorithm* iteratively refines the best paths by considering individual edges.

Finally, the *Floyd-Warshall algorithm* allows us to find the shortest paths between all pairs of nodes.

Lowest-Cost Paths

Before we can dive into algorithms to determine the lowest-cost paths through graphs, we must formalize what we mean by lowest-cost path. Recall from Chapter 3 that the total cost of a path is the sum of all edge weights along the path. For a path $p = [e_0, e_1, \ldots, e_k]$, we define the cost as:

$$PathCost(p) = \sum_{i = 0 \text{ to } k} e_i.\texttt{weight}$$

We take the *shortest path* to be a sequence of edges $p = [e_0, e_1, \ldots, e_k]$ from a given origin node u to a given destination node v that minimizes the path cost:

$$ShortestPath(u, v) = MIN_p\ (PathCost(p)) \text{ such that } e_0.\texttt{from\_node} = u \text{ and } e_k.\texttt{to\_node} = v$$

We then define the *distance* between two nodes as the cost of the shortest path between those nodes:

$$dist(u, v) = PathCost(ShortestPath(u, v))$$

As a concrete example, consider paths from node 0 to node 5 through the six-node graph with directed edges in Figure 7-1. The weights are shown for each edge. Our goal is to find the sequence of edges from node 0 to node 5 that incurs the least cost.

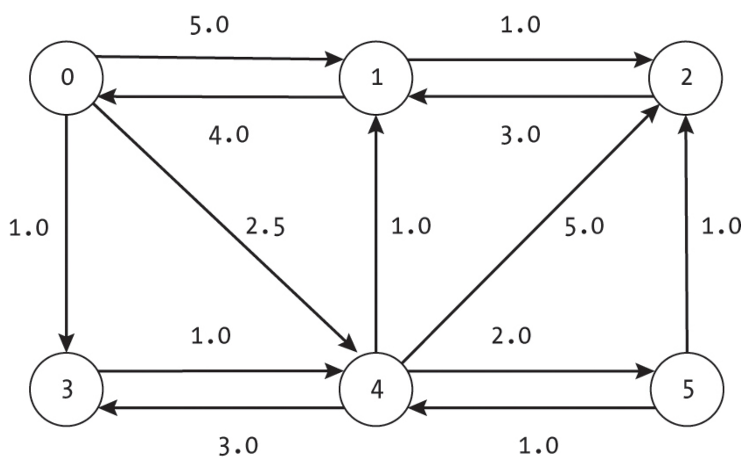


Figure 7-1: A weighted, directed graph with six nodes

Table 7-1 lists the cost of a few possible paths from node 0 to node 5 in Figure 7-1, showing that we can define a variety of paths with differing costs. In this example, the shortest path is [0, 3, 4, 5] with *dist*(0, 5) = *PathCost* ([0, 3, 4, 5]) = 4.0. The lowest cost between two nodes may not use the minimum number of edges but may instead take more lower-cost steps.

Table 7-1: Path Costs from Node 0 to Node 5

Path	Cost
0, 3, 4, 5	4.0
0, 4, 5	4.5
0, 1, 0, 3, 4, 5	13.0
0, 3, 4, 3, 4, 5	8.0

As Table 7-1 demonstrates, paths can contain loops. If we restrict the problem to using only positive edge weights, loops will strictly increase the cost of a path. Shortest-path algorithms will thus avoid them. This is immediately apparent from real-world examples such as the adventurer exploring a labyrinth in Chapter 4. Looping around the same rooms in a labyrinth not only requires extra walking but also repeatedly incurs the cost of facing any respawned monsters.

However, this problem becomes more complex (and less intuitive to visualize) if we allow negative edge weights. For this reason, the algorithms in this chapter place constraints on the use of negative weights, requiring that paths include no loops with a negative cost.

A graph may have multiple lowest-cost paths between two nodes. The algorithms in this chapter produce one of the lowest-cost paths.

Dijkstra’s Algorithm

Dijkstra’s algorithm, invented by the computer scientist Edsger W. Dijkstra, finds the lowest-cost path from a given starting node to all other nodes in the graph. It can work on either unweighted graphs or weighted graphs, with the constraint that none of the edge weights are negative. This constraint mirrors real-world path-planning problems in that we cannot decrease the total path length (cost) by adding another step. The clearest example of this is planning a road trip to minimize the total distance traveled. Since it is impossible to have negative distance, we can never shorten the trip by adding another step to the current path.

Dijkstra's algorithm operates by maintaining a set of unvisited nodes and continually updating each one's current estimated cost. It continually reduces the number of unvisited nodes by choosing the closest (lowest-cost) unvisited node and visiting it. It then explores this node and checks whether it provides a better path to each of its unvisited neighbors. In more detail, the algorithm computes the cost of the new proposed path by taking the cost to the current node and adding the cost to the neighbor. If this new cost is less than the best cost seen so far, the algorithm updates the cost.

Because the algorithm always chooses the closest (least-cost) node to explore next, we can be sure we're taking the shortest possible path every time we visit each node. This arises from the fact we do not allow negative edge weights, so the addition of steps always increases the cost of the path.

To visualize this, consider the state of the algorithm before visiting node *v*. We might worry that there is a better path to *v* through some unvisited node. After all, we haven't seen all the paths in the graph. We might not have even visited all of *v*'s neighbors. However, any such path would have to travel through an unvisited node *w*. Since *v* is selected first, the cost of the path from the origin *u* to *w* is at least as large as the cost from *u* to *v*, and that gets us only part of the way to *v*. The subsequent path from *w* to *v* would only further increase the cost.

For our adventurer mapping out a labyrinth from Chapter 4, Dijkstra's algorithm mirrors their clearing the maze room by room. The adventurer, a former cartography student, keeps meticulous notes on the shortest path found to each room they visit, since their retirement plan is to build dungeon guides they can sell to future adventurers. As the adventurer plans their next move, they consider which unvisited rooms they could reach, then try to determine the best path to each room from the dungeon entrance (the origin node). The path lengths are the cost of getting to an adjacent room, then transitioning to the unexplored room.

The Code

The code for Dijkstra's algorithm uses a priority queue to manage the set of unvisited nodes, as shown in the following implementation:

```
def Dijkstras(g: Graph, start_index: int) -> list:
    cost: list = [math.inf] * g.num_nodes
    last: list = [-1] * g.num_nodes
    pq: PriorityQueue = PriorityQueue(min_heap=True)

  ❶ pq.enqueue(start_index, 0.0)
    for i in range(g.num_nodes):
        if i != start_index:
            pq.enqueue(i, math.inf)
    cost[start_index] = 0.0

    while not pq.is_empty():
      ❷ index: int = pq.dequeue()

        for edge in g.nodes[index].get_edge_list():
            neighbor: int = edge.to_node

          ❸ if pq.in_queue(neighbor):
                new_cost: float = cost[index] + edge.weigh
t
              ❹ if new_cost < cost[neighbor]:
                  ❺ pq.update_priority(neighbor, new_cost)
                    last[neighbor] = index
                    cost[neighbor] = new_cost

    return last
```

This code relies on a custom implementation of the priority queue described in Appendix B that allows for the dynamic updating of priorities. Interested readers can find the details in that appendix. For now, it is sufficient to view `PriorityQueue` as a data structure that allows the efficient insertion of prioritized elements, removal of the element with the minimum priority, lookup of elements, and updates to elements' priority.

The code starts by creating several helper data structures, including the following: a list of best costs so far to each node (`cost`), a list indicating the last node visited before a given node (`last`), and a min-heap priority queue of unvisited nodes (`pq`). The code places the starting index (`start_index`) into the priority queue with a priority of `0.0` and all other nodes into the priority queue with an infinite priority ❶. The cost to the starting node is marked as `0.0`.

The code then processes the nodes in the priority queue one by one, iterating through them with a `while` loop until the priority queue is empty. During each iteration, the code extracts the minimum-cost node from the priority queue and explores it ❷.

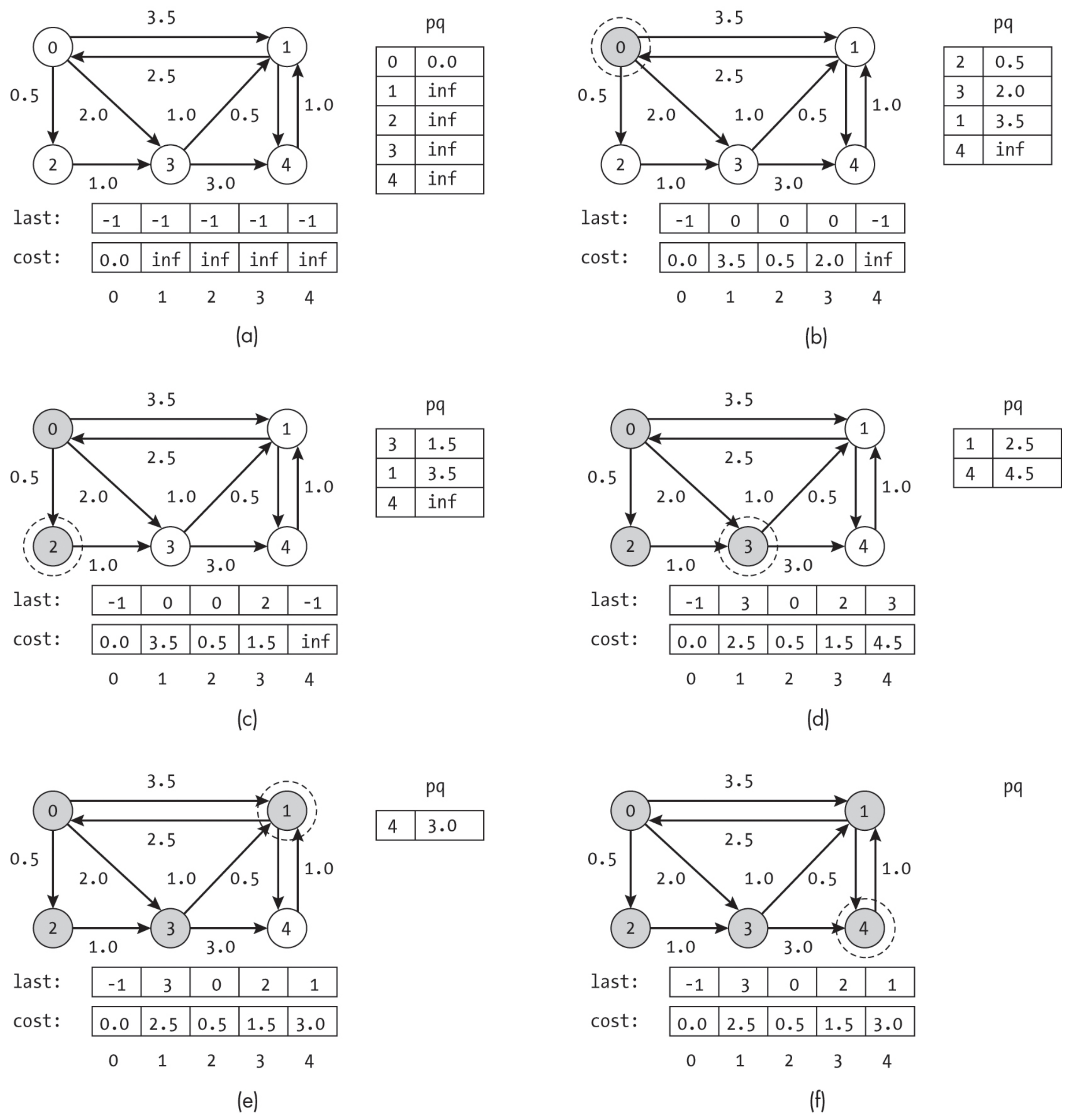
The code uses a `for` loop to consider each of the current node's neighbors, checking whether the neighbor is still in the priority queue with the `in_queue()` function ❸. If the node is in the queue, the code has not yet visited it. The code then checks whether it has found a better path to that neighbor through the current node ❹. This check corresponds to comparing the best-cost path to the neighbor found so far (`cost[neighbor]`) with the cost of the best path going through the current node (`cost[index] + edge .weight`). Since the code initially sets all non-starting nodes to an infinite cost, they will at least be updated the first time they are seen. If the code has found a better path to the neighbor, it updates the neighbor node's priority in the queue, its previous node in `last`, and the best cost ❺.

The code continues exploring unvisited nodes (those still in the priority queue) until all nodes have been visited. As noted previously, once the code visits a node, it has found the shortest path and does not need to reconsider any visited nodes. When the code has visited all nodes, it returns the list of previous nodes on the path.

Given the *heap-based* implementation of Dijkstra's algorithm presented here, we can imagine how the algorithm scales to larger graph sizes. The algorithm visits each node exactly once by inserting all the nodes into a priority queue and then removing them one by one. Using a heap-based priority queue, each operation scales as $\log(|V|)$, so the entire time for iterating over these nodes scales as $|V| \log(|V|)$. During the visit to each node, we check whether each neighbor is in the priority queue and potentially update the priority. Since we are using the custom `PriorityQueue` described in Appendix B, the lookup uses a dictionary and takes constant time on average. The update scales as $\log(|V|)$. Since we visit each node only once and consider its outgoing edges once, we will perform at most one update per edge, giving a cost that scales as $|E| \log(|V|)$. Thus, the total cost of the algorithm scales as $|V| \log(|V|) + |E| \log(|V|) = (|V| + |E|) \log(|V|)$.

An Example

Figure 7-2 illustrates Dijkstra’s algorithm operating on a five-node graph. Each subfigure represents the state of the algorithm after finishing a step. The dashed circle indicates the node that was just processed, while shaded nodes are those that have been visited. The priority queue (`pq`) is shown in sorted order to make it easy to see the relative priorities, although it is stored in heap order.



(a)

pq	
0	0.0
1	inf
2	inf
3	inf
4	inf

	0	1	2	3	4
last:	-1	-1	-1	-1	-1
cost:	0.0	inf	inf	inf	inf

(b)

pq	
2	0.5
3	2.0
1	3.5
4	inf

	0	1	2	3	4
last:	-1	0	0	0	-1
cost:	0.0	3.5	0.5	2.0	inf

(c)

pq	
3	1.5
1	3.5
4	inf

	0	1	2	3	4
last:	-1	0	0	2	-1
cost:	0.0	3.5	0.5	1.5	inf

(d)

pq	
1	2.5
4	4.5

	0	1	2	3	4
last:	-1	3	0	2	3
cost:	0.0	2.5	0.5	1.5	4.5

(e)

pq	
4	3.0

	0	1	2	3	4
last:	-1	3	0	2	1
cost:	0.0	2.5	0.5	1.5	3.0

(f)

pq

	0	1	2	3	4
last:	-1	3	0	2	1
cost:	0.0	2.5	0.5	1.5	3.0

Figure 7-2: The steps of Dijkstra's algorithm

Figure 7-2(a) represents the state of the algorithm before the first node is explored. The adventurer is standing at the entrance, ready to begin their quest. All nodes have `last` entries of `-1`, indicating we do not know the path to get to them. Node 0 has a cost of `0`, since our search begins there, and all other nodes have an infinite estimated cost because we do not yet know *any* possible path. Unlike the algorithms for depth-first search and breadth-first search, we start with all nodes in the priority queue.

At each step, the algorithm explores the best node remaining in the queue. In Figure 7-2(b), it selects node 0 (the only node without an infinite cost) and visits it. It finds edges to three neighbors: nodes 1, 2, and 3. The search compares the cost of a path through node 0 to the current costs. Since the new cost is less than infinity for all three nodes, it updates their entries in `last`, `cost`, and `pq`. This update in the priority queue reorders the list of nodes to explore next.

Figure 7-2(c) shows what happens when the search visits node 2. This node has a single neighbor (node 3) with an estimated cost of `2.0`. However, the path to node 3 through node 2 now provides a better option with a total cost of 0.5 + 1.0 = 1.5. The algorithm updates the `cost` entry of node 3 to `1.5` and its `last` entry to `2`.

Through the lens of our adventurer, room 2 provides a better path to room 3. Maybe there is a particularly powerful monster guarding the passage from room 0 to room 3. The adventurer, cognizant of the needs of future generations of explorers, takes this into account and redirects the suggested path to room 3 to go through room 2.

The search continues through the remaining nodes. When the algorithm visits node 3 in Figure 7-2(d), it finds a better path to node 1. Similarly, visiting node 1 in Figure 7-2(e) provides a better path to node 4. The search completes in Figure 7-2(f) after visiting the final node.

Disconnected Graphs

Asking what happens if some nodes are not reachable from the starting node can help us understand how Dijkstra's algorithm performs on a disconnected graph. Consider the four-node graph in Figure 7-3, where only nodes 0 and 1 are reachable when starting from node 0.

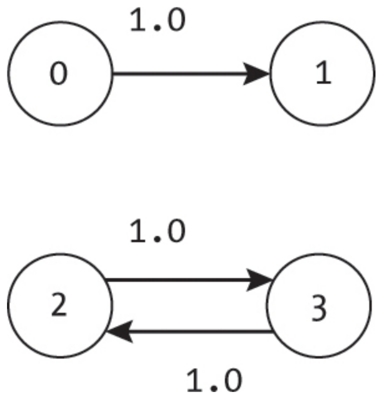


Figure 7-3: A graph with unreachable nodes

This corresponds to unreachable rooms in the labyrinth. From legend, the adventurer knows the labyrinth has four rooms, but they can reach only two. There is no path from room 0 to room 2 or room 3, so the adventurer has no choice but to indicate this in their notes.

Dijkstra's algorithm can easily handle this case. Both nodes 2 and 3 are initially assigned a `last` value of `-1` and an infinite cost. Because there is no path from node 0 to either of these nodes, when either is extracted from the priority queue, it still has an infinite cost. When the algorithm considers the node's neighbors, the estimated cost through that node will be infinite, so the algorithm never updates either `cost` or `last`. At the end of the algorithm, both nodes 2 and 3 will have last pointers of `-1`.

Negative Edge Weights

In real-world problems, edge weights can be negative, representing a negative cost (a benefit). Consider the example of communications within a social network, where each connection between friends is an edge. The weight of each edge represents the cost of passing a rumor from one person to the next. This cost might be the loss of the time required to send a text or to chat. However, in some cases, the edge weight can be negative and represent a benefit to using that channel of communication. If two friends have not spoken in a while, the cost of reengaging to pass along some gossip may very well be negative.

Alternatively, we can envision path planning to minimize the battery usage of an electric vehicle. If an edge represents a steep downhill road, we can use the combination of gravity and regenerative braking to charge the battery. The cost in terms of battery usage for this segment is negative.

Note that in the context of negative edge weights, the term *shortest path* does not really make sense, as distances cannot be negative. No matter how skilled you are at path planning, you cannot organize a cycling trip that gets you home before you set out. However, we continue to refer to these problems as *shortest path* for consistency with the wider literature.

When considering the shortest path through a graph with negative edges, we still need to maintain one constraint: the graph cannot contain negative cycles. A *negative cycle* occurs whenever there is a path from a node back to itself where the sum of edge weights is negative. In the presence of such cycles, the entire concept of lowest-cost path breaks down. For example, consider the graph in Figure 7-4, where edge (0, 1) has a weight of `1.0` and edge (1, 0) has a weight of `-2.0`.

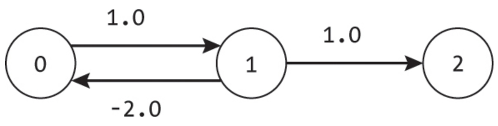


Figure 7-4: A graph with a negative cycle

If we try to find the lowest-cost path from node 0 to node 2 in Figure 7-4, we immediately run into problems. As shown in Table 7-2, we can keep adding another loop from node 0 to node 1 to node 0 to further reduce the cost. The lowest-cost path would loop back and forth forever.

Table 7-2: The Cost of Paths in Figure 7-4

Path	Cost
0, 1, 2	2
0, 1, 0, 1, 2	1
0, 1, 0, 1, 0, 1, 2	0
0, 1, 0, 1, 0, 1, 0, 1, 2	–1
. . .	. . .

In contrast, Figure 7-5 shows a graph with a negative edge weight but no negative cycles. It is possible to travel from node 1 to node 0 with a negative cost. However, any path from node 0 back to itself or node 1 back to itself will have a total positive cost.

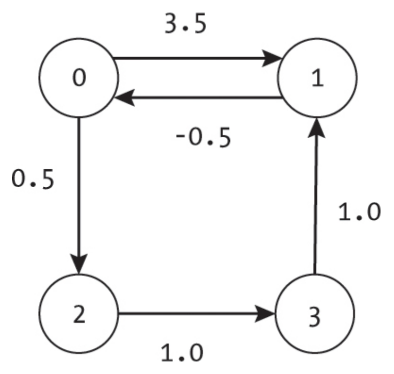


Figure 7-5: A graph with a negative edge weight but no negative cycles

How can we tell whether a graph has negative cycles? A negative cycle could be incredibly long, looping through every node of the graph and thus not be immediately obvious. The Bellman-Ford algorithm solves this problem for us by checking for the existence of negative cycles in a graph.

Bellman-Ford Algorithm

A major drawback of Dijkstra's algorithm is that it is limited to graphs with positive edge weights. The Bellman-Ford algorithm removes this limitation, but it must add computational cost to do so.

The *Bellman-Ford algorithm* operates by repeatedly iterating over the list of edges and using them to update the best cost to each node (a process called *relaxation*). Like Dijkstra's algorithm, it maintains a list `cost` that stores the best cost seen so far from the origin to each node. Each time Bellman-Ford considers an edge, it asks if this edge presents a better path to the edge's destination node by comparing that node's current cost estimate (entry in `cost`) to the one provided by traveling from the edge's origin (using the origin's entry in `cost` plus the edge weight). It repeats this test over and over, improving the estimates of the best paths.

We can visualize the algorithm as an extremely thorough travel agent considering flight options for the coming travel season. The agent starts with an origin like Chicago and looks for the cheapest path to each possible destination around the world. Obviously, the agent cannot take every flight themselves (that is, cannot travel the entire graph). However, they can easily scan through the list of flights and their prices to update their spreadsheet of estimates.

After a single scan through the flight list, the agent knows the best direct flights from Chicago to every other city. They scan through the list again, asking if they can build better paths to each possible destination using what they know about the best trips so far. They repeatedly scan the list, updating their estimates, until they have the best path to every destination.

Like the travel agent, with each iteration of Bellman-Ford's outer loop, we are effectively building better paths. This progressive construction of paths is shown in Figure 7-6. The bold lines represent the best-known path from node 2 to node 0 after each iteration.

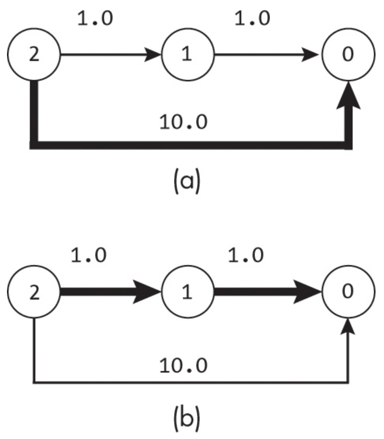


Figure 7-6: The Bellman-Ford algorithm finding progressively better paths from node 2 to node 0

Figure 7-6(a) shows the best path from node 2 to node 0 after one iteration through each edge. Since the algorithm has looked at each edge only once, it sees a direct path only from node 2 to node 0 with cost `10.0`. It did not have a chance to realize it could build a better path using the edges (2, 1) and (1, 0). During the second iteration, the algorithm uses the knowledge of a path with cost `1.0` from node 2 to node 1 to build a path to node 0. The best path to node 0 is updated to go through node 1 and costs `2.0`, as shown in Figure 7-6(b).

We can cap the total number of iterations at $|V| - 1$, where $|V|$ is the number of nodes in the graph. Because negative cycles are not allowed, a least-cost path can never return to the same node, as doing so would strictly increase the cost of the path. This is the reason real-world travel itineraries between different cities do not include cycles—that is, multiple layovers at the same airport.

Since the least-cost path can never repeat nodes, it can touch at most all $|V|$ nodes and use $|V| - 1$ edges. For the six-node graph in Figure 7-7, for example, the lowest-cost path from node 0 to node 1 is [0, 3, 4, 5, 2, 1]. Although there are alternate paths with fewer steps, the lowest-cost path from node 0 to node 1 uses five edges and visits all the nodes in the graph.

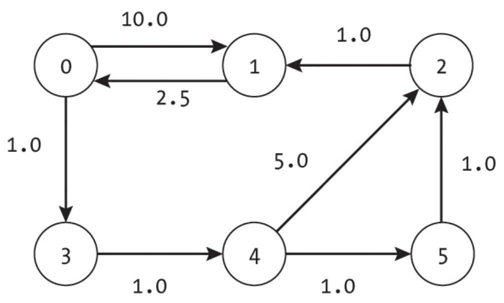


Figure 7-7: An example graph with a five-step path from node 0 to node 1

Bellman-Ford uses this constraint to both stop the algorithm and test for cycles. After $|V| - 1$ iterations of the outer loop, it has found all possible lowest-cost paths. Because this algorithm uses two nested `for` loops (one over the number of nodes and the other over each edge), its cost scales as the product $|E|\,|V|$.

Additional iterations through the edges will not help unless there is a negative cycle. Armed with this knowledge, the algorithm tries one additional iteration and tests whether any costs decrease. If so, it knows the graph has a negative cycle.

We can picture this last test as our travel agent performing one last check over their list of least-cost flights. They notice that adding another leg between Pittsburgh and Boston further lowers the price. Confused, they look back over the flight data and see that a trip from Chicago to Boston to Pittsburgh to Boston to Seattle is the cheapest option so far. The loop from Boston to Pittsburgh and back provides a negative cycle. Obviously, something has gone wrong with the flight pricing, creating a loop that effectively costs negative dollars. The travel agent hurries to call their client about a potential free 10-stop trip before the airline fixes the problem.

The Code

The Bellman-Ford iterates over each edge $|V| - 1$ times. Every iteration, it asks the following simple question: "Does the current edge provide a better path to its destination node?" The code uses a pair of `for` loops to drive this search:

```
def BellmanFord(g: Graph, start_index: int) -> Union[list,
None]:
    cost: list = [math.inf] * g.num_nodes
    last: list = [-1] * g.num_nodes
    all_edges: list = g.make_edge_list()
    cost[start_index] = 0.0

    for itr in range(g.num_nodes - 1):
        for edge in all_edges:
          ❶ cost_thr_node: float = cost[edge.from_node] +
 edge.weight
          ❷ if cost_thr_node < cost[edge.to_node]:
                cost[edge.to_node] = cost_thr_node
                last[edge.to_node] = edge.from_node

    for edge in all_edges:
      ❸ if cost[edge.to_node] > cost[edge.from_node] + edg
e.weight:
            return None
    return last
```

The `BellmanFord()` function takes a `Graph` and starting index and returns either the best path to each destination (using a `last` array) or `None` if the graph has a negative loop. We need to import `Union` from Python's `typing` library to allow type hints for the multiple return values.

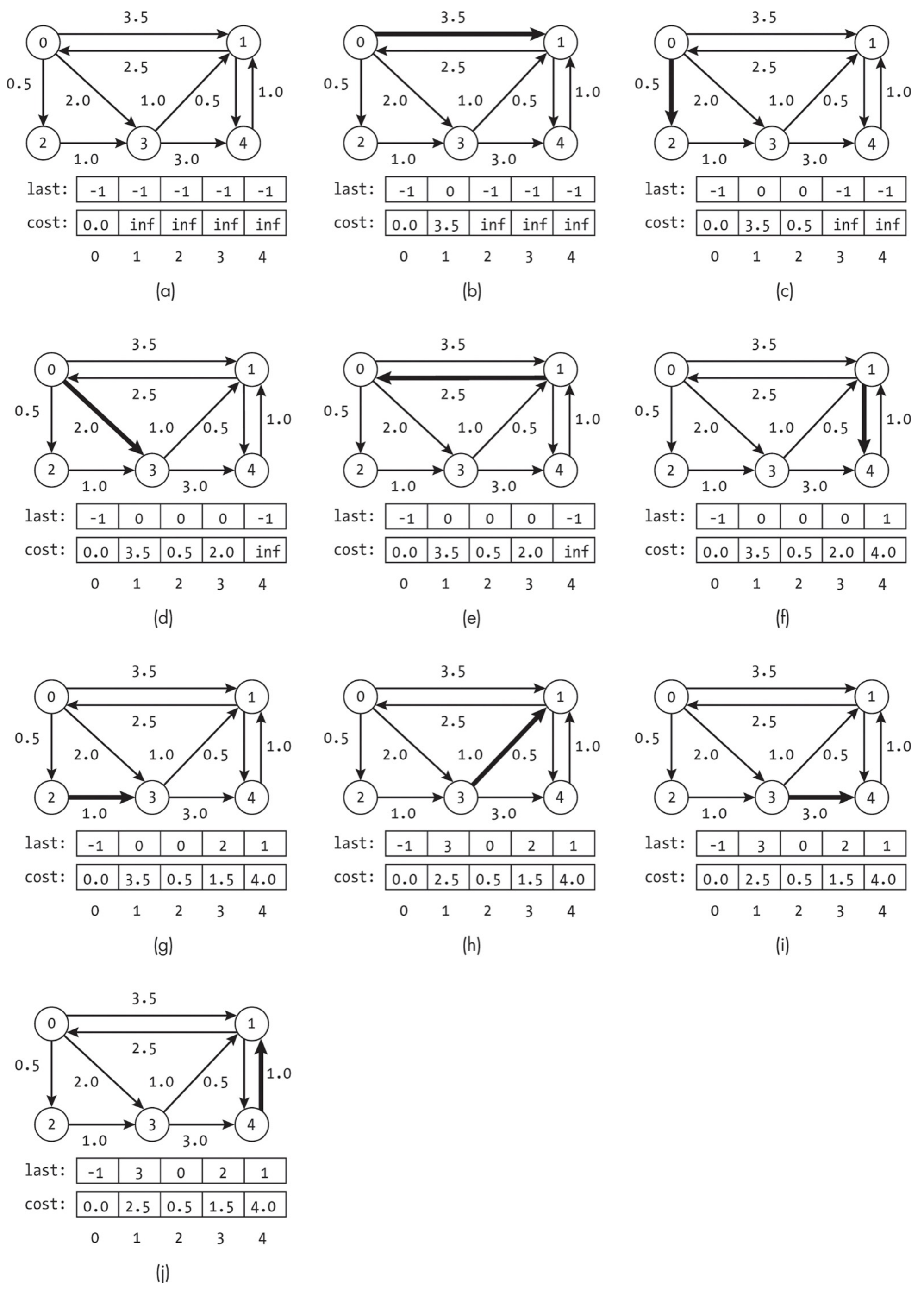
This code starts by creating the tracking data structures, including the best costs so far (`cost`) and the previous node on the current path (`last`). It also extracts a full list of edges for the algorithm using the `make_edge_list()` function, which iterates through each node and assembles a list of every edge in the graph. Finally, it sets the cost of the starting node to 0.0.

A pair of nested `for` loops drives $|V| - 1$ iterations over the list of edges. For each edge, the code assesses the cost to the destination using that edge ❶. If this cost is less than the current best cost ❷, the code updates both the best-cost estimate and the path to the node. Note that the reduced cost may not result from changing the previous node (`last`); rather, the cost to the previous node could have decreased due to a better path to that previous node.

When the code has completed $|V| - 1$ iterations of the outer loop, it has finished the optimization. Before concluding, it checks whether the solution is valid. If any cost could still be improved by taking another step ❸, the graph must have a negative cost cycle, in which case the algorithm returns `None`. Otherwise, it returns the `last` list.

An Example

Figure 7-8 shows the first iteration of the outer loop of the Bellman-Ford algorithm. Since the algorithm takes $(|V| - 1)\ |E|$ steps, where $|E|$ is the number of edges, it is not feasible to show all 36 steps. Instead, we consider the first iteration of the outer loop to review how the paths (represented by `last`) and estimated costs change. Each subfigure represents the state of the algorithm after examining the bolded edge.



(a)

	0	1	2	3	4
last:	-1	-1	-1	-1	-1
cost:	0.0	inf	inf	inf	inf

(b)

	0	1	2	3	4
last:	-1	0	-1	-1	-1
cost:	0.0	3.5	inf	inf	inf

(c)

	0	1	2	3	4
last:	-1	0	0	-1	-1
cost:	0.0	3.5	0.5	inf	inf

(d)

	0	1	2	3	4
last:	-1	0	0	0	-1
cost:	0.0	3.5	0.5	2.0	inf

(e)

	0	1	2	3	4
last:	-1	0	0	0	-1
cost:	0.0	3.5	0.5	2.0	inf

(f)

	0	1	2	3	4
last:	-1	0	0	0	1
cost:	0.0	3.5	0.5	2.0	4.0

(g)

	0	1	2	3	4
last:	-1	0	0	2	1
cost:	0.0	3.5	0.5	1.5	4.0

(h)

	0	1	2	3	4
last:	-1	3	0	2	1
cost:	0.0	2.5	0.5	1.5	4.0

(i)

	0	1	2	3	4
last:	-1	3	0	2	1
cost:	0.0	2.5	0.5	1.5	4.0

(j)

	0	1	2	3	4
last:	-1	3	0	2	1
cost:	0.0	2.5	0.5	1.5	4.0

Figure 7-8: The starting state and nine steps performed during the first iteration of Bellman-Ford's outer loop

Figure 7-8(a) shows the state of the algorithm before it examines any edges. All the nodes have an estimated infinite cost except the starting node. In Figure 7-8(b), the algorithm tests the first edge and sees that it provides a better path to its destination, node 1. It updates the path to node 1 to be the path from node 0 and the best cost to be `3.5`.

The search continues through each of the graph's edges, considering the cost to a single node—the current edge's destination—with each iteration. In Figure 7-8(c), it finds a better path to node 2; in Figure 7-8(d), it finds a better path to node 3. It doesn't update anything in Figure 7-8(e) because the best cost from node 0 to node 0 is `0.0`, and we don't need an unnecessary loop through node 1 to get back to the starting point.

At the end of the first iteration of the outer loop, shown as Figure 7-8(j), the search has examined each edge and made updates to the best paths and cost estimations. However, the algorithm is not complete. The true best path to node 4 is [0, 2, 3, 1, 4] and has a cost of `3.0`. It will not find this final cost until it reconsiders the edge (1, 4). When it considered this edge in the first round, it had yet to find the best path to node 1, so its cost estimate is too large.

As the algorithm continues, it revisits edges and continually updates the best path and its cost. Figure 7-9 shows the final step of the algorithm.

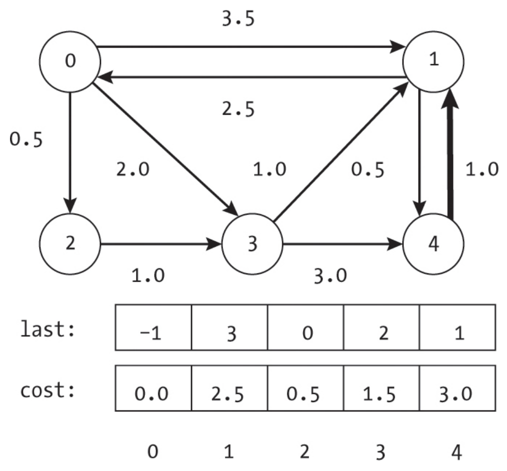


Figure 7-9: The final state of the Bellman- Ford algorithm

After examining the edge (4, 1) for the fourth time, the algorithm has completed both loops. The costs and paths have converged to their true values.

All-Pairs Shortest Paths

Both our explorer and travel agent examples were satisfied with an algorithm that finds the lowest-cost path from a given start node to all other nodes in the graph. However, what if we want to find the shortest path between *any* pair of nodes in the graph? Even within the context of our previous two analogies, we can see the appeal of such an approach. Once the adventurer has mapped out the entire magical labyrinth, they may want to move back and forth between arbitrary rooms to help other adventurers who are in trouble. Likewise, our travel agent may want to go global, planning trips from any starting location to any destination in the world. In both cases, we need to find the least-cost path between two arbitrary nodes.

The *all-pairs shortest-path problem* aims to find the shortest path between every pair of nodes in a graph. Phrased another way, we now want to build a *matrix* `last` such that each row `last[i]` contains the previous-node list for paths starting from node *i*. In this formulation, the entry of the matrix `last[i][j]` is the node immediately before *j* on the path from *i* to node *j*. As with our previous `last` array formulation in the various search algorithms and other shortest-path algorithms, for a fixed starting point, we can trace the previous nodes backward from our destination to the origin.

We can solve the problem of finding all pairs of shortest paths by adding a loop around either of the algorithms discussed so far in this chapter. For example, we could fill in the `last` matrix using a single `for` loop and the Bellman-Ford algorithm:

```
last: list = []
for n in range(g.num_nodes):
    last.append(BellmanFord(g, n))
return last
```

Since the cost of the Bellman-Ford algorithm scales as $|E|\,|V|$, the total cost of this approach scales as $|E|\,|V|^2$. Similarly, we could wrap Dijkstra's algorithm (as implemented in this chapter) with a cost that scales as $|V|\,(|V| + |E|)\log(|V|)$. This is the computational equivalent of calling a travel agent in each city and asking for the lowest-cost trips out of that city. Using the combined information of the shortest path from each starting node to all possible destinations, we can assemble the costs for traveling between any pair of cities.

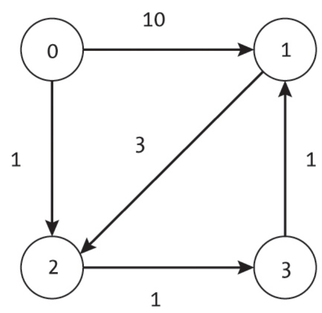
The following section introduces an alternate algorithm for finding all least-cost paths: Floyd-Warshall. This algorithm scales well to dense graphs where $|E|$ is much greater than $|V|$. Instead of iterating over unvisited nodes or all edges, the Floyd-Warshall algorithm considers each node that could be on the intermediate path and decides whether to use it.

The Floyd-Warshall Algorithm

The *Floyd-Warshall algorithm* solves the all-pairs shortest-path problem by iteratively considering and optimizing the nodes between each origin and destination. An *intermediate path* consists of those nodes after the origin and before the destination. This algorithm effectively builds up better paths by considering nodes for inclusion in the intermediate path. An outer `for` loop iterates over each node *u* and asks, "Would any path be better if we included a stop at node *u*?" For each intermediate node *u*, the algorithm tests every path under consideration to see if it helps. If so, it adds it.

Throughout the process, we maintain matrix versions of the `last` and `cost` arrays we used in both Dijkstra's and Bellman-Ford. Each row of these matrices corresponds to the arrays for a single starting node, and each entry indicates the value (cost or previous node on the path) for a specific destination node. We initialize both matrices to represent best paths *without* any intermediate nodes. The initial value of `cost[u][v]` is the edge weight of (u, v) if the edge exists, 0 if $u = v$, and infinite otherwise. Similarly, the value of `last[u][v]` is u if the edge (u, v) exists and $u \neq v$. Otherwise, the value is `-1` to indicate the lack of a path.

Figure 7-10 shows an example of the Floyd-Warshall algorithm's state. The graph on the left is for reference, while the two matrices show the current estimated costs and best paths. This initial state is the computational equivalent of the travel agent planning for a customer who will take only direct flights. A pair of cities (u, v) is considered only if there is a direct flight from u to v. All other cities might as well have infinite cost.

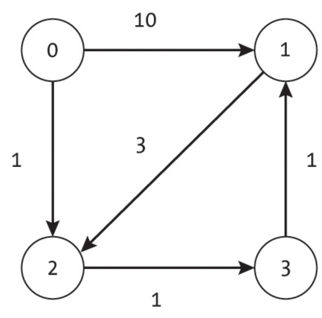


$$\underset{\text{cost}}{\begin{bmatrix} 0.0 & 10.0 & 1.0 & \text{inf} \\ \text{inf} & 0.0 & 3.0 & \text{inf} \\ \text{inf} & \text{inf} & 0.0 & 1.0 \\ \text{inf} & 1.0 & \text{inf} & 0.0 \end{bmatrix}} \quad \underset{\text{last}}{\begin{bmatrix} -1 & 0 & 0 & -1 \\ -1 & -1 & 1 & -1 \\ -1 & -1 & -1 & 2 \\ -1 & 3 & -1 & -1 \end{bmatrix}}$$

Figure 7-10: The state of the data structures at the start of the Floyd-Warshall algorithm

The Floyd-Warshall algorithm, using a computational technique called *dynamic programming*, effectively builds the best paths using intermediate nodes in $\{0, 1, . . . , k\}$ from paths that can contain only intermediate nodes in $\{0, 1, . . . , k - 1\}$. Since negative loops are not allowed, each node can be used at most once in the path. For each origin-destination pair (u, v), the algorithm checks whether there is a better path through node k that uses only intermediate nodes $\{0, 1, . . . , k\}$. We can program this by reusing the `cost` and `last` matrices from the previous iteration ($k - 1$). If there is a better path through k, the combined cost of the best paths from u to k and from k to v must be less than the cost of the current path from u to v. We can directly read these paths and costs from the last iteration's `cost` and `last` matrices.

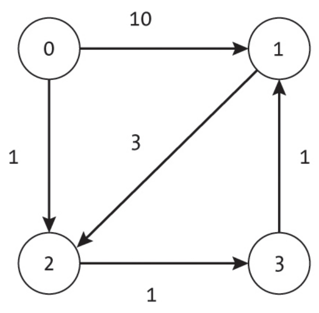
To see how this works, consider the graph and algorithm state in Figure 7-11, which takes place after paths with potential intermediate nodes 0, 1, and 2 in the figure's graph have been tested. The `cost` and `last` matrices represent the best paths that can have intermediate nodes in $\{0, 1, 2\}$. The best path from node 0 to node 1 is still the direct step [0, 1] because we cannot use node 3 yet.



$$\underset{\text{cost}}{\begin{bmatrix} 0.0 & 10.0 & 1.0 & 2.0 \\ \text{inf} & 0.0 & 3.0 & 4.0 \\ \text{inf} & \text{inf} & 0.0 & 1.0 \\ \text{inf} & 1.0 & 4.0 & 0.0 \end{bmatrix}} \quad \underset{\text{last}}{\begin{bmatrix} -1 & 0 & 0 & 2 \\ -1 & -1 & 1 & 2 \\ -1 & -1 & -1 & 2 \\ -1 & 3 & 1 & -1 \end{bmatrix}}$$

Figure 7-11: The state of the Floyd-Warshall algorithm's data structures after testing nodes 0, 1, and 2

When we consider the paths that could use node 3 as an intermediate node, we find several better paths, as shown in Figure 7-12. Let's again consider the path from node 0 to node 1. When we ask whether there is a better path from node 0 to node 1 through node 3, we find that there is. The path through node 3 has a cost of 3.0, since the cost of the path from 0 to 3 (through node 2) is `2.0` and the cost of the path from 3 to 1 is `1.0`.



$$\begin{bmatrix} 0.0 & 3.0 & 1.0 & 2.0 \\ \text{inf} & 0.0 & 3.0 & 4.0 \\ \text{inf} & 2.0 & 0.0 & 1.0 \\ \text{inf} & 1.0 & 4.0 & 0.0 \end{bmatrix} \qquad \begin{bmatrix} -1 & 3 & 0 & 2 \\ -1 & -1 & 1 & 2 \\ -1 & 3 & -1 & 2 \\ -1 & 3 & 1 & -1 \end{bmatrix}$$

cost last

Figure 7-12: The state of the Floyd-Warshall algorithm's data structures after testing nodes 0, 1, 2, and 3

By adding node 3 as an intermediate node along the path from node 0 to node 1, we also add node 2. The best path is now [0, 2, 3, 1]. This illustrates the power of Floyd-Warshall's iterative approach. We are not just considering the intermediate node in isolation but also the best paths to and from that node.

Since the algorithm tests for an improved path between each pair of nodes for every possible intermediate node, it requires a triply nested `for` loop over the nodes. The cost of the algorithm therefore scales as $|V|^3$. While this might seem expensive, the relative running time of the previous approaches depends on the relative number of edges and nodes.

The Code

The core of the Floyd-Warshall algorithm is a triply nested set of `for` loops that iterate first over each intermediate node to add (k) and then through each pair of nodes (i, j) that need a path, as shown in the following code:

```
def FloydWarshall(g: Graph) -> list:
    N: int = g.num_nodes
    cost: list = [[math.inf] * N for _ in range(N)]
    last: list = [[-1] * N for _ in range(N)]
```

```
❶ for i in range(N):
      for j in range(N):
          if i == j:
              cost[i][j] = 0.0
          else:
              edge: Union[Edge, None] = g.get_edge(i, j)
              if edge is not None:
                  cost[i][j] = edge.weight
                ❷ last[i][j] = i

  for k in range(N):
      for i in range(N):
          for j in range(N):
            ❸ if cost[i][j] > cost[i][k] + cost[k][j]:
                  cost[i][j] = cost[i][k] + cost[k][j]
                ❹ last[i][j] = last[k][j]
  return last
```

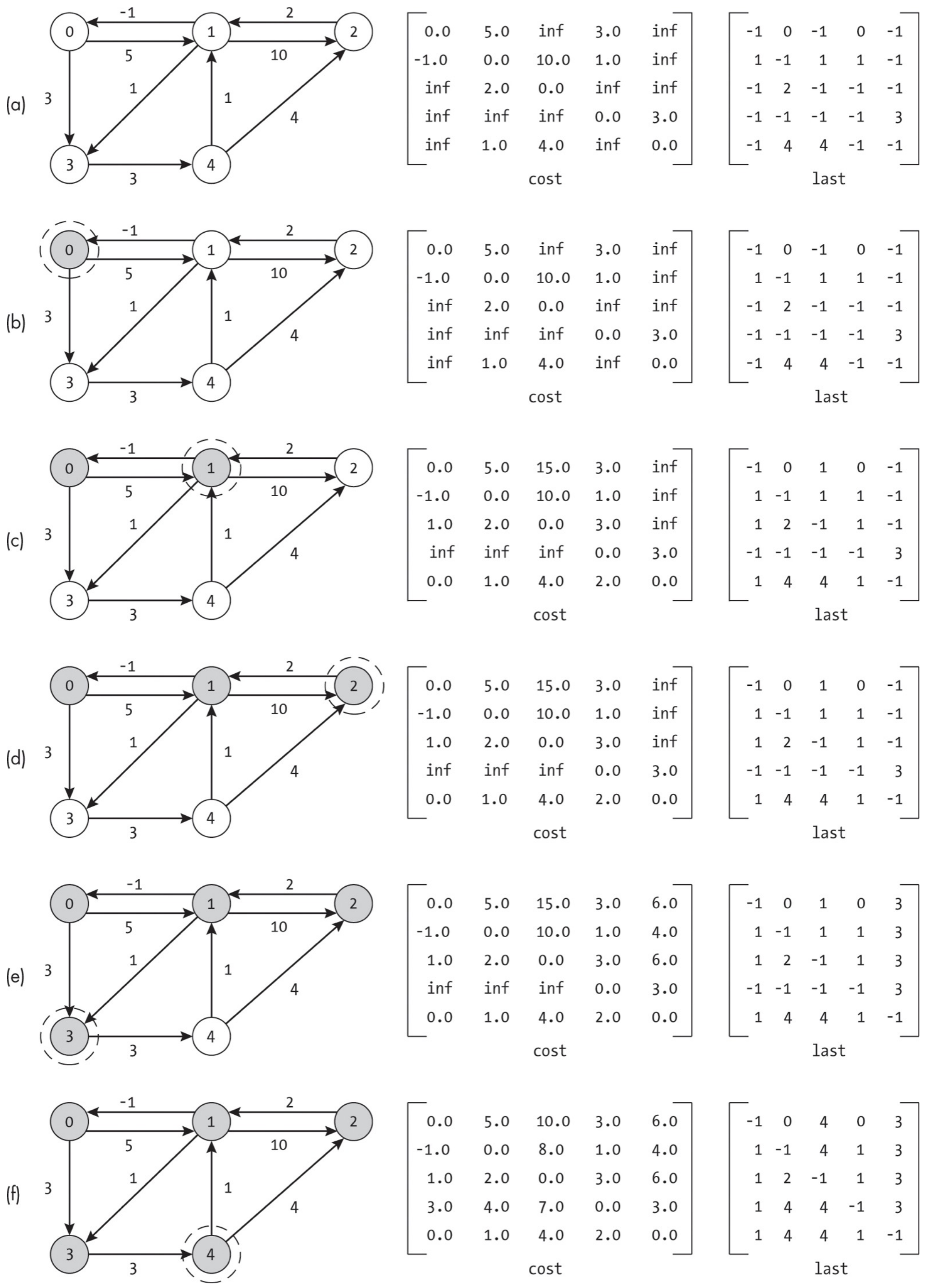
The code starts by setting up the initial `cost` and `last` matrices. A pair of nested `for` loops is used to iterate over each entry ❶. Best costs are set to 0.0 for the diagonals (`i == j`), the edge weights for nodes linked by edges, and infinity otherwise. The code uses the `Graph` class's `get_edge()` function to check for and retrieve an edge, requiring an additional import of `Union` from Python's `typing` library. Similarly, the previous nodes are set to the origin for any pair linked by an edge and `-1` otherwise (including the diagonals) ❷.

The code performs the majority of the processing via the triply nested `for` loops. The outer loop iterates over the intermediate node `k` under consideration. The two inner loops iterate through each pair of nodes `i` and `j`. For each pair, the code checks whether it can achieve a better path through node `k` ❸. If so, it updates both the `cost` and `last` arrays to reflect this. Unlike previous algorithms in the book, the `last` array's entry is updated to match the last step on the *path* from `k` to `j` ❹.

When the code finishes checking all intermediate nodes for all possible pairs of origin and destination, it returns the `last` matrix of paths.

An Example

Figure 7-13 shows an example of the Floyd-Warshall algorithm operating on a graph with five nodes. Each of the first five subfigures shows the state of the data structures *after* the iteration that adds the dashed node to the set of possible intermediate nodes. Shaded nodes have already been added. Thus, Figure 7-13(a) shows the state before the first iteration and Figure 7-13(b) shows the state after the conclusion of the first iteration, when the node 0 has been considered as an intermediate node.



(a)

```
 0.0   5.0   inf   3.0   inf      -1  0 -1  0 -1
-1.0   0.0  10.0   1.0   inf       1 -1  1  1 -1
 inf   2.0   0.0   inf   inf      -1  2 -1 -1 -1
 inf   inf   inf   0.0   3.0      -1 -1 -1 -1  3
 inf   1.0   4.0   inf   0.0      -1  4  4 -1 -1
            cost                       last
```

(b)

```
 0.0   5.0   inf   3.0   inf      -1  0 -1  0 -1
-1.0   0.0  10.0   1.0   inf       1 -1  1  1 -1
 inf   2.0   0.0   inf   inf      -1  2 -1 -1 -1
 inf   inf   inf   0.0   3.0      -1 -1 -1 -1  3
 inf   1.0   4.0   inf   0.0      -1  4  4 -1 -1
            cost                       last
```

(c)

```
 0.0   5.0  15.0   3.0   inf      -1  0  1  0 -1
-1.0   0.0  10.0   1.0   inf       1 -1  1  1 -1
 1.0   2.0   0.0   3.0   inf       1  2 -1  1 -1
 inf   inf   inf   0.0   3.0      -1 -1 -1 -1  3
 0.0   1.0   4.0   2.0   0.0       1  4  4  1 -1
            cost                       last
```

(d)

```
 0.0   5.0  15.0   3.0   inf      -1  0  1  0 -1
-1.0   0.0  10.0   1.0   inf       1 -1  1  1 -1
 1.0   2.0   0.0   3.0   inf       1  2 -1  1 -1
 inf   inf   inf   0.0   3.0      -1 -1 -1 -1  3
 0.0   1.0   4.0   2.0   0.0       1  4  4  1 -1
            cost                       last
```

(e)

```
 0.0   5.0  15.0   3.0   6.0      -1  0  1  0  3
-1.0   0.0  10.0   1.0   4.0       1 -1  1  1  3
 1.0   2.0   0.0   3.0   6.0       1  2 -1  1  3
 inf   inf   inf   0.0   3.0      -1 -1 -1 -1  3
 0.0   1.0   4.0   2.0   0.0       1  4  4  1 -1
            cost                       last
```

(f)

```
 0.0   5.0  10.0   3.0   6.0      -1  0  4  0  3
-1.0   0.0   8.0   1.0   4.0       1 -1  4  1  3
 1.0   2.0   0.0   3.0   6.0       1  2 -1  1  3
 3.0   4.0   7.0   0.0   3.0       1  4  4 -1  3
 0.0   1.0   4.0   2.0   0.0       1  4  4  1 -1
            cost                       last
```

Figure 7-13: Iterations of the Floyd-Warshall algorithm on a graph with five nodes

Consider this example in the context of our travel agent's plan to conquer the global market. They create two spreadsheets, where the first (`cost`) maps the origin and destination pair to the total cost. The second (`last`) maps the same pair to the city on the itinerary immediately before the destination. At a loss for how to start, the agent fills in the direct flights as shown in Figure 7-13(a). The `cost` matrix contains the cost of the flight between any pair of cities (with `0.0` for a "flight" to the same city) and infinity if there is no direct path. The `last` matrix shows the city from which the originating flight came, or `-1` if there is no previous city. This is the starting state for the algorithm.

Next, the travel agent considers Chicago (node 0) and asks, "What if I route them through that city? Of course, I'll use the best possible path I'm currently aware of to get my customers from the origin to Chicago and from Chicago to their destination. I'm only talking about adding a single intermediate stop." As shown in Figure 7-13(b), this does not help any of the paths, and the travel agent does not change their matrices.

The agent then moves on to New York (node 1) and asks the same question. As shown in Figure 7-13(c), the possibilities expand. By considering a layover in New York, travelers from Chicago (node 0) can now reach Portland, Maine (node 2). Similarly, travelers from Portland (node 2) and Charlotte (node 4) can reach Chicago (node 0) and Pittsburgh (node 3).

Buoyed by their success, the travel agent immediately considers layovers in Portland (node 2). This does not provide much help, as using Portland as an intermediate stop does not reduce the cost of any of the paths. The agent sighs, wonders if their success with New York was a fluke, and continues with their search.

Their persistence is rewarded after considering Pittsburgh (node 3), as shown by the matrices in Figure 7-13(e). The agent discovers new paths to Charlotte (node 4) from Chicago (node 0), New York (node 1), and Portland (node 2).

So far, the agent has found only paths to new cities, however. None of the layovers has reduced the cost between cities that already had a path. Therefore, considering Charlotte (node 4) is an eye-opening experience, as it offers an improved layover for multiple trips. Before considering Charlotte as a stop, the trip from Chicago to Portland took the path [0, 1, 2] with a cost of `15`. Now travelers can make the same trip using the path [0, 3, 4, 2] with a cost of only `10`. Even the trip from New York to Portland is cheaper via the path [1, 3, 4, 2] than it was with a direct flight.

Computing Graph Diameter

The *diameter* of a graph is a measure that characterizes the maximum distance nodes in a graph. We define the diameter as the maximum distance between any two nodes in the graph

$$diameter = \text{argmax}_{u \in E, v \in E}\ dist(u, v)$$

where, as noted earlier in the chapter, $dist(u, v)$ is the cost of the shortest path between u and v. We can use the algorithms in this chapter to construct this measure that helps analyze or compare graphs.

For example, consider our labyrinth adventurer. After years of questing and victory over a hundred underground labyrinths and too many monsters, they decide to retire and start a labyrinth aid operation. They want to pick one labyrinth, spend their days there, and help other struggling adventurers (for a reasonable fee). Their key consideration is how quickly they can jump in and help their clients. After all, it's no good if the monsters overwhelm the clients before the adventurer can provide aid (or the client can pay). This is complicated by the fact that both the rescuer and their clients could be at any node in the labyrinth when they run into trouble. If the adventurer has multiple simultaneous clients, they might even find themselves dashing between rooms.

The adventurer decides to find a labyrinth with a diameter between 5 and 10 rooms. Any larger and they will not be able to get to their client in time. Any smaller and there probably is not enough of a challenge for the other adventurers to need help. Satisfied with their reasoning, they compute the diameter of all the dungeons in their vicinity and pick one in the correct range.

We can extract the diameter of a graph directly from the `cost` matrix in the Floyd-Warshall algorithm by iterating over entries and finding the maximum. Alternately, we can reconstruct it from the `last` matrix by walking each of the paths backward and summing up the path cost. Here is code for the second approach in order to illustrate the use of the `last` matrix:

```
def GraphDiameter(g: Graph) -> float:
 ❶ last: list = FloydWarshall(g)
    max_cost: float = -math.inf

    for i in range(g.num_nodes):
        for j in range(g.num_nodes):
            cost: float = 0.0
            current: int = j

          ❷ while current != i:
                prev: int = last[i][current]
              ❸ if prev == -1:
                    return math.inf

                edge: Union[Edge, None] = g.get_edge(prev,
current)
                cost = cost + edge.weight
                current = prev

          ❹ if cost > max_cost:
                max_cost = cost

    return max_cost
```

This code uses the Floyd-Warshall algorithm to compute the `last` matrix of paths ❶. It then iterates over all pairings of origin and destination (`i`, `j`) using a pair of nested `for` loops. For each such pairing, the code starts at the destination and walks backward through the last matrix until it hits the origin ❷. Along the way, it extracts each edge and adds its weight to the current sum. If the path dead-ends (that is, if `last[i][current] == -1` when `current != i`), the function will immediately return an infinite distance to indicate there is no path between a pair of nodes ❸. If all pairs have a valid path, the code tracks the costliest one seen ❹ and returns this maximum as the graph's diameter.

Why This Matters

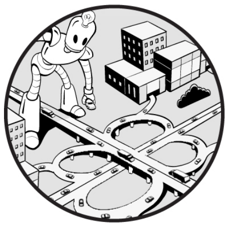
Lowest-cost algorithms are directly applicable to a range of real-world problems, from path planning to optimization. The algorithms in this chapter provide practical methods for efficiently finding such paths. Both Dijkstra's algorithm and the Bellman-Ford algorithm return solutions to all possible destinations in the graph. The Floyd-Warshall algorithm extends this even further and returns the shortest path between all possible origins and destinations.

The three algorithms presented in this chapter also illustrated general techniques that can be adapted to solve other graph problems. Dijkstra's algorithm maintains a priority queue of unvisited nodes that represent an unexplored frontier of possibilities. In Chapter 10, we will see how another algorithm uses this same approach to solve a different optimization problem. The Bellman-Ford algorithm provides a glimpse into algorithms that operate over the set of edges. The Floyd-Warshall algorithm demonstrates a more complex dynamic programming approach. It constructs the best path with a subset of possible intermediate nodes from the simpler problem of best paths constructed from a smaller such subset.

The next chapter introduces algorithms that can incorporate additional heuristic information to limit the number of nodes they must visit when finding the lowest-cost path from a given origin to a given destination. While these algorithms do not produce shorter paths than the ones in this chapter, they run faster by focusing the search on the most promising nodes.

8

HEURISTIC-GUIDED SEARCHES



This chapter introduces *heuristic-guided search* algorithms, sometimes called *best-first searches*. These algorithms incorporate heuristic information about nodes' estimated distance from the goal to prioritize the order in which to explore them. By focusing on the most promising paths toward the goal, they can achieve significant computational savings.

As we saw in the last chapter, finding the shortest (or lowest-cost) path from a specified start node to a specified goal node mirrors the daily task of navigating through the world. When planning a route to work or the store, we might measure cost by the time taken, the distance traveled, or the aggravation incurred due to the number of bad intersections.

After explaining what constitutes a heuristic, this chapter introduces two canonical heuristic search algorithms. *Greedy best-first search* prioritizes nodes solely by their estimated cost to the goal, while *A* search* (pronounced "A-star") combines the cost to reach an intermediary node with the estimated cost from that node to the goal. This combination makes A* search a much more powerful tool for efficiently finding good paths.

Choosing Appropriate Heuristics

The algorithms in this chapter rely on estimated costs to guide their searches. To add such heuristic information to an algorithm, we must pick a method of estimating costs from what we know about each node. While the difficulty of defining a good heuristic varies widely across problems, the approaches for many real-world scenarios are simple and intuitive. After introducing Euclidean distance as a common heuristic used in path planning, we discuss the constraints and challenges involved in choosing heuristics.

Euclidean Distance

Euclidean distance is a common, powerful, and intuitive heuristic used in many real-world path-planning problems that estimates the cost to a node at a given location by the straight-line distance to get there. For example, suppose we are navigating a cross-country road trip from Boston to Seattle. If the starting city is located at (x_1, y_1) and the destination is at (x_2, y_2), then the Euclidean distance between the two is as follows:

$$dist = \sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2)}$$

We can code this equation as follows:

```
def euclidean_dist(x1: float, y1: float, x2: float, y2: fl
oat) -> float:
    return math.sqrt((x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 -
y2))
```

Unless you are a bird flying directly to your destination, Euclidean distance provides, at best, a lower bound of the true cost. Roads don't exist between every pair of cities on the map. Even highways that stretch directly from one point to another are unlikely to follow a straight line, since they must curve around geographic features like mountains and lakes, lengthening your journey.

Despite its optimistic nature, Euclidean distance can provide vital heuristic information. For example, these estimates can help us choose appropriate waypoints as we plan a road trip. On a trip from Boston to Seattle, Cleveland would obviously make a better rest stop than Miami, because Miami is farther from both Seattle and Cleveland than our starting point.

Figure 8-1(a) shows an example graph on a two-dimensional plane. Figure 8-1(b) shows the corresponding edge weights, which largely correspond to the straight-line distance between nodes. However, the cost between two nodes can also be *greater* than the Euclidean distance. The edge between nodes 2 and 4 has a weight of 3.5, which could indicate additional cost, such as traversing a steep hill or using a dirt road. The cost of traversing between nodes (the edge weight) must simply be greater than or equal to the estimated distance.

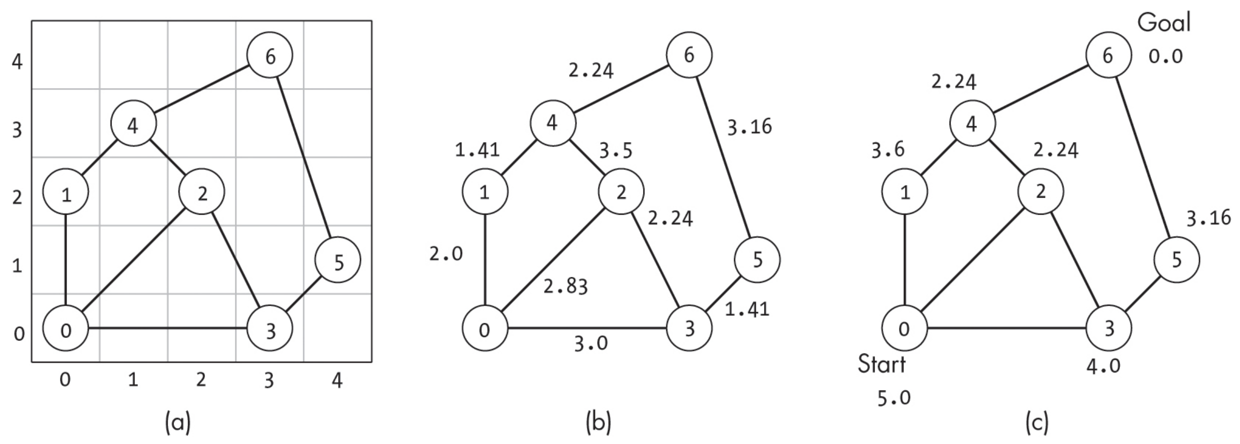


Figure 8-1: The weights and Euclidean distances between nodes on a 2D plane

Given this arrangement, we can provide lower-bound estimates of the potential cost from each node to the goal by taking the Euclidean distance from that node to the goal, as shown in Figure 8-1(c), for a goal node of 6. The estimated cost from node 0 to node 6 is 5.0, reflecting their Euclidean distance. As noted previously, these estimates do not always capture the full cost. For example, the estimated distance from node 0 to node 6 is too optimistic, as the node lacks a direct path to node 6.

Admissible Heuristics

We define a heuristic to be *admissible* if the estimated cost from the start node to the goal is always less than or equal to the true cost, or, in other words, if the heuristic does not overestimate the true cost. Euclidean distance, for example, is a common, effective, and admissible heuristic for real-world path planning, since the straight-line distance to our goal provides an optimistic estimate of the cost to get there. The admissibility requirement is essential for the correct operation of some searches, such as A* search, and will be one of our major constraints when choosing a heuristic for a given purpose.

Heuristic Design Challenges

While it's relatively easy to define a heuristic for estimating distances, let's consider a more difficult case. Imagine you want to solicit an introduction to a new contact through your professional network. Each person (node) can reach out only to their present or past coworkers to pass along the request. To facilitate your introduction, you must find a sequence of weighted edges that indirectly connect you to the person you ultimately want to meet, where each edge weight represents the cost of passing along the request. The cost of passing the request to a friend with whom you converse daily would be low, while the cost of connecting with an annoying former coworker with whom you never want to speak again would be very high.

Unfortunately, it's difficult to capture all these factors with a single admissible heuristic. You might be able to glean some estimate of distance to the goal from each person's job history. The estimated cost of communication through a person in a different industry from the goal node should generally be higher; for example, professional interactions between a professional baseball player and a computer scientist are rarer than those between two baseball players. Similarly, the estimated cost would be lower if the two individuals worked at the same company. However, piecing together positive and negative indicators like this makes it difficult to form a good quantitative estimate, as the indicators are too noisy. The fact that two people once worked together does not help you if they have never met or are nemeses.

Worse, it is much more difficult to ensure your heuristics for such problems are admissible. If you place a high cost on different industries, you will occasionally overestimate the cost of passing your message between them—perhaps you know a programmer who still talks regularly to the movie star who was their childhood best friend. Similarly, this metric would not capture the family members who work for different companies but communicate often.

Choosing a good heuristic involves maximizing information while maintaining admissibility and minimizing computation cost. It is trivial to design an admissible heuristic by assigning each node a cost of negative infinity, but this strategy is obviously useless in guiding a search. Similarly, we can design a perfectly informative and admissible heuristic by using the algorithms in the previous chapter: we just solve the all-pairs shortest-path problem and compute the heuristic from the true lowest-cost path between each node and the goal. Yet this does not help either, as the computational cost of the search is too high. The point of a heuristic is to reduce the computational cost of the search itself. As we consider new problems and heuristics, it is always important to examine the trade-offs between information, computational cost, and admissibility.

The following sections introduce two canonical heuristic searches, starting with the simplest approach: greedy best-first search.

Greedy Best-First Search

A *greedy best-first search* always chooses the option that looks best at a given point in the search, exploring the next unvisited node with the lowest estimated cost based on the best heuristic value. The algorithm maintains a minimum priority queue of nodes to test. As it progresses toward the goal, at each step it chooses the lowest-cost node from the priority queue and explores that node next. Each time the algorithm sees new neighbors, it adds them to the queue with a priority equal to their heuristic value. The algorithm proceeds node by node until it finds a path to the goal.

We can view greedy best-first search as a modification of breadth-first search. Whereas the latter search prioritizes nodes by the order in which they were seen, using a queue to visit the earliest-seen node, best-first search orders the nodes with a heuristic.

Greedy best-first search takes the approach we might expect of an eager but clever squirrel navigating a maze, as shown in Figure 8-2. The squirrel (S) can smell the delicious pile of acorns that is his goal destination (G). Using its nose, the squirrel can deduce the straight-line path he could take directly to the nuts, if there were no walls (Figure 8-2(a)).

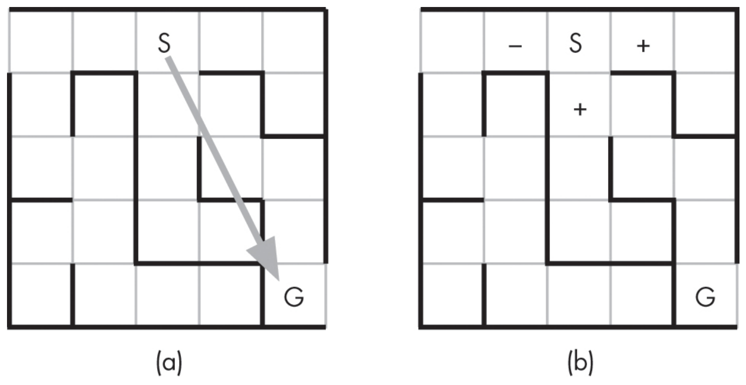


Figure 8-2: A maze with a heuristic estimate of the goal's direction

At any given location (node), the squirrel can also determine adjacent locations to which it can move and estimate which one will take it closer to the destination. The squirrel uses a smell-based heuristic—nodes with a stronger smell of acorns are closer to the goal. Figure 8-2(b) shows that two of the neighboring squares will move the squirrel closer to the acorns (+), while one will move it farther away (–). The squirrel always chooses the path with the strongest smell, following the scent toward the food, sometimes backtracking to points where the smell was stronger than the current location. Along the way, it mentally notes alternative paths and adds them to the list of options to try.

Although greedy best-first search might quickly find a path to the goal if it is working with a good heuristic, the final path is not guaranteed to be optimal. We might take a path that looks good due to an optimistic estimate at an early node and skip a better path with a more realistic estimate. The squirrel might take a longer path that temporarily curves away from the food simply because the smell was stronger in that direction. We will see an example of this scenario later in this section.

The Code

To implement the greedy search algorithm, we need to provide an additional piece of information beyond what we included in earlier searches: the nodes' heuristic values. There are multiple approaches to providing this information. For clarity of illustration, we'll start by passing a precomputed list `h` that maps the nodes' indices to their heuristic values, and we'll introduce an alternative method later in the chapter.

The code for greedy best-first search is similar to that of breadth-first search. Instead of using a queue to store the nodes in the order they were seen, we use a custom *min-heap-based priority queue* (described in Appendix B) to retrieve nodes in order of decreasing estimated cost:

```
def greedy_search(g: Graph, h: list, start: int, goal: in
t) -> list:
    visited: list = [False] * g.num_nodes
    last: list = [-1] * g.num_nodes
    pq: PriorityQueue = PriorityQueue(min_heap=True)

 ❶ pq.enqueue(start, h[start])
 ❷ while not pq.is_empty() and not visited[goal]:
        ind: int = pq.dequeue()
        current: Node = g.nodes[ind]
        visited[ind] = True

        for edge in current.get_edge_list():
            neighbor: int = edge.to_node
         ❸ if not visited[neighbor] and not pq.in_queue(n
eighbor):
                pq.enqueue(neighbor, h[neighbor])
                last[neighbor] = ind

    return last
```

The code starts by setting up the internal data structures, including a list indicating whether we have visited each node (`visited`), a list mapping each node to the one that preceded it on the search path (`last`), and a minimum priority queue (`pq`). It then inserts the starting node into the priority queue, with its heuristic cost as the priority ❶. In the squirrel maze analogy, this marks the point right before the squirrel's search begins. Standing ready outside the maze, it smells the acorns and has one available option on its mental priority queue: the starting node.

The exploration of greedy best-first search takes place in a `while` loop that continues exploring until the code has either exhausted the priority queue or visited the goal node ❷. At each iteration, the code retrieves the next node on the priority queue—the one with the best heuristic value—and visits that node. You can picture this as the squirrel running to the location of the next best option.

The code uses a `for` loop to iterate over each of the current node's neighbors. If a neighbor has not been visited and is not in the priority queue, then it has not been seen before ❸. The code therefore adds it to the priority queue (with the estimated distance as the priority) and marks the current node as the step before it in the path.

When the `while` loop completes, the greedy search will have either found a path to the goal node or discovered that no such path exists. In the former case, unlike other algorithms we have seen previously, there may still be unexplored nodes on the priority queue. In the latter case, the priority queue will be empty; there are no more nodes to explore. The goal node's entry in `last` will be `-1`, reflecting the lack of any path back to the starting node. The code concludes by returning the `last` list.

An Example

Figure 8-3 shows an example greedy best-first search on the graph from Figure 8-1. In each subfigure, the current node being explored is enclosed in a dashed circle, while nodes that have been visited are shaded. The edge weights are shown alongside each edge.

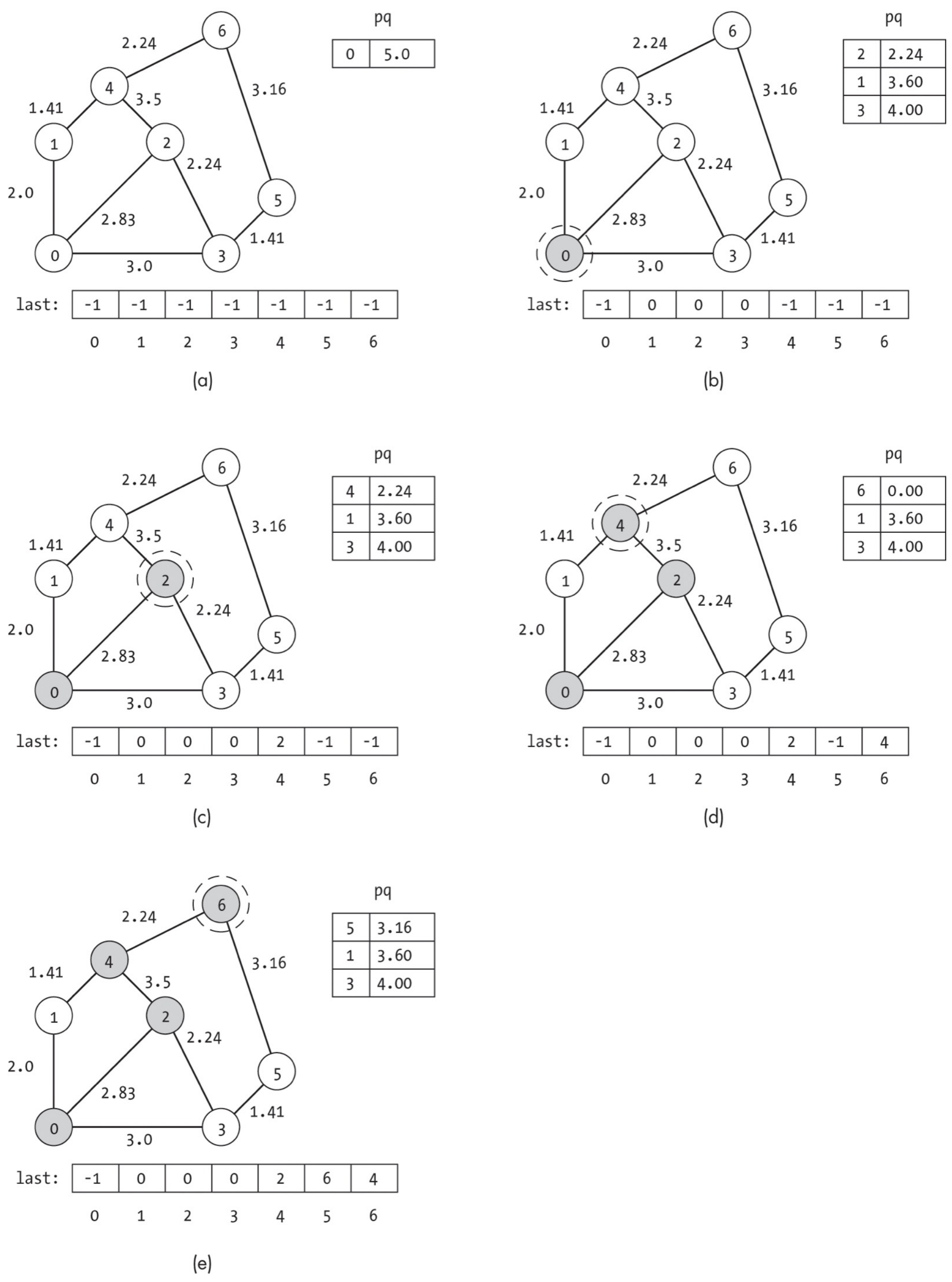


Figure 8-3: The steps of a greedy best-first search

To avoid cluttering the illustrations, the heuristic values for each node are not shown directly in Figure 8-3. However, these heuristic values are the same Euclidean distances as shown in Figure 8-1(c). We provide them to the algorithm as a list `h`:

```
h: list = [5.0, 3.6, 2.24, 4.0, 2.24, 3.16, 0.0]
```

Each subfigure also shows the current priority queue (in sorted order for illustration) and the `last` list. Although the code maintains the priority queue in heap ordering, we show the priority queue in sorted order to make the relative orderings clearer.

The search begins by putting the start node into the priority queue with its corresponding cost estimate 5.0, as shown in Figure 8-3(a). During the first iteration of the `while` loop, it dequeues node 0 from the priority queue, visits it, and adds each previously unseen neighbor to the priority queue, as shown in Figure 8-3(b). The priorities for the nodes are equal to their heuristic costs (that is, the Euclidean distances to the goal node) as given in Figure 8-1(c): node 1 = 3.6, node 2 = 2.24, and node 3 = 4.0. The search sets the `last` values to 0 for each of these neighbors to indicate that the path toward them comes from node 0.

At each step in the search, the algorithm chooses the node that looks most promising as it progresses toward the goal. Checking the priority queue, it moves on to node 2. As shown in Figure 8-3(c), it then adds node 2's unvisited neighbors to the priority queue. With priority 2.24, node 4 now sits at the top of the queue. The search progresses through node 4 in Figure 8-3(d) to the goal node in Figure 8-3(e).

As you can see from this example, greedy best-first search does not produce an optimal path to the goal. The search is lured to node 2 with the promise of its proximity to the goal node, but is forced to detour through node 4 and over a costly 3.5 weighted edge. Greedy best-first search cannot tell that it would have been better to progress through node 1, because it doesn't consider the cost of the paths to get to a node. It looks only at the estimated cost from a given node to the goal and uses that for prioritization. By the time the search has finished visiting node 2, it has already seen that node 4 has a better heuristic value than node 1.

We can picture this suboptimality in the context of a frustrating bicycle trip. Suppose that after a morning of cycling with no destination in mind and without paying attention to the path, you and your friend are exhausted and want to find your way home. You stop at a fork in the road and consider the options. You know the left path ends at an intersection adjacent to your house but traverses a small mountain to get there. The right path is flat but terminates at an intersection a few blocks away from your house. Both paths get you closer to home, but with radically different costs (edge weights). Unfortunately, a greedy algorithm doesn't take that into consideration. Before you can open your mouth, your overeager friend lets out a cheer and pedals up the left path. When you try to protest, they just call out, "Who cares about a little hill? This path gets us closer."

A* Search

A* search combines the heuristic estimates of greedy best-first search with a more comprehensive accounting for the observed edge costs, providing an efficient mechanism for finding the shortest path between two nodes. Whereas greedy best-first search completely ignores the edge cost, A* balances the promise of the heuristic estimates with the cold, hard facts about the best paths we have seen to each node. This combination results in an accurate and computationally efficient algorithm.

The key intuition behind the A* algorithm is that we want to rank potential nodes in our path by their estimated `total` cost. It is not sufficient to focus on the cost from the current node to the goal; we also must ask how expensive it was to get to that node in the first place. To answer this question, A* tracks an additional piece of information: the cost of the best path found so far to each node. As shown in Figure 8-4, the priority used for unvisited nodes is then just the sum of the cost of the best path to the node so far plus the estimated cost of traveling from that node to the goal.

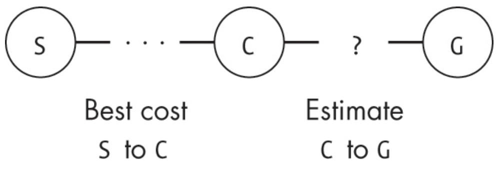


Figure 8-4: The combination of true cost to a node and estimated cost to the goal

In contrast to the eager squirrel's demonstration of greedy best-first search, we can visualize A* search as a more meticulous explorer with an advanced degree in cartography searching for a path to a proposed archeological dig site. In addition to the standard compass, canteen, and obligatory exploring hat, our protagonist carries a clipboard to track information about the area. They represent each node as a row with three columns of information: the best cost to the node (titled Best Cost), the best path to the node (Best Path), and the estimated total cost to the goal through that node (Heuristic). Throughout their journey, the explorer continually updates these three columns of information.

The explorer travels from village to village (node to node). GPS coordinates provide an estimate of the shortest possible distance. Signposts, trail markers, and in-person interviews reveal the actual distance to neighboring nodes. Each time the explorer discovers a new node, they compute its estimated cost to the goal and write that down in the Heuristic column.

As their journey continues, the explorer always moves to the next location (node) with the lowest estimated total cost. Each time they consider a neighboring location, whether new or previously seen, they ask themselves whether they've found a better path to that node than they had previously discovered. If so, they document this discovery in the Best Cost and Best Path columns. Perhaps earlier in their journey they discovered a 10-mile trail through a dense, spider-infested jungle to reach the archeology site. Their notes detail this path and its tremendous cost. However, they later discover a new, three-mile-long paved highway to the same site via a small village to the east. They eagerly erase their old values and update both the Best Cost and the Best Path columns to reflect this new find.

The Code

The code for A* search in Listing 8-1 orders the potential nodes by the estimated total cost through that node and onto the goal. Again, it uses a precomputed list `h` of heuristic values for each node.

```
def astar_search(g: Graph, h: list, start: int, goal: int)
-> list:
    visited: list = [False] * g.num_nodes
    last: list = [-1] * g.num_nodes
    cost: list = [math.inf] * g.num_nodes
    pq: PriorityQueue = PriorityQueue(min_heap=True)

  ❶ pq.enqueue(start, h[start])
    cost[start] = 0.0
  ❷ while not pq.is_empty() and not visited[goal]:
        ind: int = pq.dequeue()
        current: Node = g.nodes[ind]
        visited[ind] = True

        for edge in current.get_edge_list():
            neighbor: int = edge.to_node
          ❸ if cost[neighbor] > cost[ind] + edge.weight:
                cost[neighbor] = cost[ind] + edge.weight
                last[neighbor] = ind

              ❹ est_value: float = cost[neighbor] + h[neig
hbor]
                if pq.in_queue(neighbor):
                    pq.update_priority(neighbor, est_valu
e)
                else:
                    pq.enqueue(neighbor, est_value)
    return last
```

Listing 8-1: The code for A search*

The code starts by setting up the internal data structures, including a list indicating whether it has visited each node (`visited`), a list mapping each node to the one that preceded it on the search path (`last`), a list storing the cost of the best path found from the starting node to each subsequent node (`cost`), and a minimum priority queue (`pq`). It inserts the starting node into the priority queue, with its estimated cost as the priority, and sets the cost of the starting node to 0 to reflect the fact that the search is already at that node ❶.

The search is now ready to begin. A `while` loop continues exploring nodes until the code has either exhausted the priority queue or visited the goal node ❷. At each iteration, the code retrieves the next node on the priority queue—the node with the lowest estimated total cost to the goal—and visits it next.

The algorithm uses a `for` loop to iterate over each of the current node's neighbors. It checks whether the current node provides a better path to the neighbor, computing the full cost by combining the best cost to the current node with the edge weight to the neighbor ❸. If the code finds a better path to a node, it updates both the `cost` and `last` lists. It then updates the neighboring node's estimated total cost using the new cost to `neighbor` plus the estimated cost from `neighbor` to the goal ❹. If the neighbor is in the priority queue already, the code updates its priority with the `update_priority()` function to take the new estimated total cost into account. Otherwise, it adds the node to the priority queue.

As in greedy best-first search, the `while` loop in A* search completes when it has either found a path to the goal node or concluded that no such path exists—that is, if the search exhausts the priority queue before visiting the goal node. The code finishes by returning the `last` list.

An Example

Figure 8-5 shows an example A* search. As in our example for greedy best-first search, we show the edge weights and again use the heuristics from Figure 8-1(c):

```
h: list = [5.0, 3.6, 2.24, 4.0, 2.24, 3.16, 0.0]
```

Each subfigure also shows the `last` array, `cost` array, and priority queue. The current node being explored is marked with a dashed circle and the visited nodes are shaded. Again, the priority queue is shown in sorted order for clarity.

Figure 8-5(a) shows the initial state of the search before it visits the first node. The priority queue initially contains only the start node. Since the distance from the start node to itself is 0.0, the estimated total cost of the starting node is just the estimated distance to the goal. The `cost` array reflects the known costs to get to each node: 0.0 for the starting node and infinite for everything else, because the search has not observed a path to those nodes yet.

Figure 8-5(a) represents our hypothetical explorer's state before the start of an expedition. They have been hired to find the shortest path from a city (node 0) to a proposed archeological dig site (node 6). Before they land at the starting city, they have only a rough (and optimistic) estimate of the distance to the dig site given the geographical coordinates of its location. The explorer checks their lists, dons their helmet, and says, "I know the dig site is at least five miles from the city. It's time to get started."

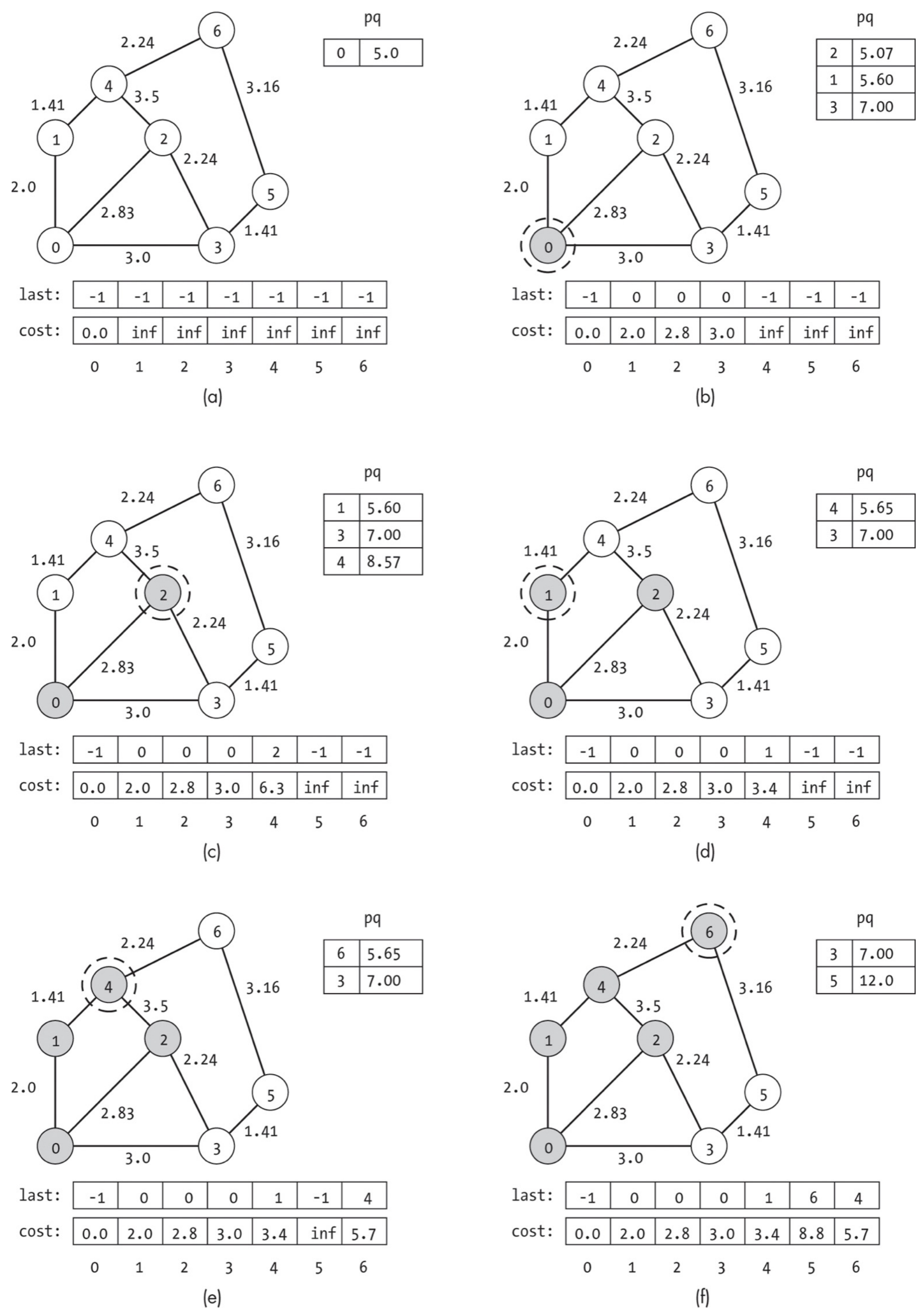


Figure 8-5: The steps of the A algorithm on an example graph*

In Figure 8-5(b), the search dequeues the top node from the priority queue and explores it. This is equivalent to our intrepid explorer arriving at the starting city and looking around. The search finds three neighbor nodes and computes their expected total cost as the sum of the distance to the current node, the edge weight to the neighbor, and the estimated cost from the neighbor to the goal, resulting in the following total estimated costs:

Node 1: 0.0 + 2.0 + 3.6 = 5.6

Node 2: 0.0 + 2.83 + 2.24 = 5.07

Node 3: 0.0 + 3.0 + 4.0 = 7.0

This corresponds to the explorer updating their lists after finding three roads out of the initial city. From a helpful signpost, they know the distance to the villages and their locations. Each road presents a potential path with a different estimated cost.

Since the estimated total cost of node 2 looks the best, the search explores it next, as shown in Figure 8-5(c). From there, it considers two neighbors, nodes 3 and 4. Node 3 already has a lower cost from the starting node (3.0 versus 2.83 + 2.24 = 5.07), so the search does not update its path or priority. The search has not seen node 4 before, so it provides an initial cost value of 2.83 + 3.5 = 6.33 and a total cost estimate of 2.83 + 3.5 + 2.24 = 8.57. This cost reflects the impact of both the path to node 2 and the extreme cost of traversing from node 2 to node 4.

Through the eyes of the explorer, these decisions look similar. They see a signpost indicating two new villages. Village 3 is an additional 2.24 miles away. Compared to the direct path from city 0 to village 3, the detour through village 2 to village 3 is much longer. They immediately realize there is no need to add an unnecessary stop and therefore refrain from updating the row for village 3. In contrast, while the path to village 4 from village 2 is exceptionally difficult, it offers the prospect of getting closer to their goal, so they update the row for village 4.

The search continues by taking the unvisited node with the best estimated total cost. Unlike greedy search, it does not jump to the node estimated to be closest to the goal, in this case node 4. Though this node has the best estimated cost to the goal (2.24), the cost of getting there using the current path is high (6.3 through node 2). Instead, the search explores node 1, as shown in Figure 8-5(d), and finds a better path to node 4, updating the estimated total cost to 2.0 + 1.41 + 2.24 = 5.65. It also updates the `last` array to indicate that the path to node 4 goes through node 1 instead of node 2.

This step mirrors the explorer thinking about the total cost of the route. The archologists who hired the explorer want a low-cost route to reach the site repeatedly. Knowing this, the explorer tries village 1 before crossing the mountain from village 2 to village 4.

The search continues to node 4 in Figure 8-5(e), then node 6 in Figure 8-5(f). At each stop, it considers the unvisited neighbors and checks whether it has found a better path. It stops after reaching node 6 because it knows it has found the best path to the goal, even without having visited nodes 3 and 5.

Why A Finds the Optimal Path*

The skeptical reader might wonder how we can be sure A* search has found the best path, since it explores only a portion of the graph without visiting every node. However, as long as its heuristic is admissible, A* will always find the optimal path. To see why, let's examine the state after A* search has reached the goal node through some path and consider an alternate path to the goal node through an unvisited node *v*. Because of our admissible heuristic and priority queue ordering, any path through node *v* must be longer than the one we have already found.

Since the search did not visit node *v* before the goal node, node *v*'s priority value (estimated total cost) must have been greater than the priority value of the goal node. At the point that the search visited the goal node, the goal node's priority value equals the actual cost of the path found. The estimated cost to the goal is always 0 for the goal itself, so the goal's priority value equals the cost to the node preceding it plus the corresponding edge weight. We are no longer relying on heuristics. We have an actual path cost.

In contrast, the priority value of the unvisited node *v* is a lower bound of the true distance due to use of an admissible heuristic. It can never be less than the true distance. Our heuristic is optimistic. Any path to the goal through node *v* must cost at least as much as node *v*'s priority value, which was greater than that of the goal node. Thus, the cost of a path to the goal through node *v* must be higher than the one already found.

Applying A* to Puzzles

As long as we can generate a useful and admissible heuristic, we can apply heuristic-based searches to efficiently find solutions for the puzzle graphs from Chapter 6, such as the prisoners-and-guards puzzle. As a reminder, Figure 8-6 shows the state graph for that puzzle (originally introduced in Figure 6-8).

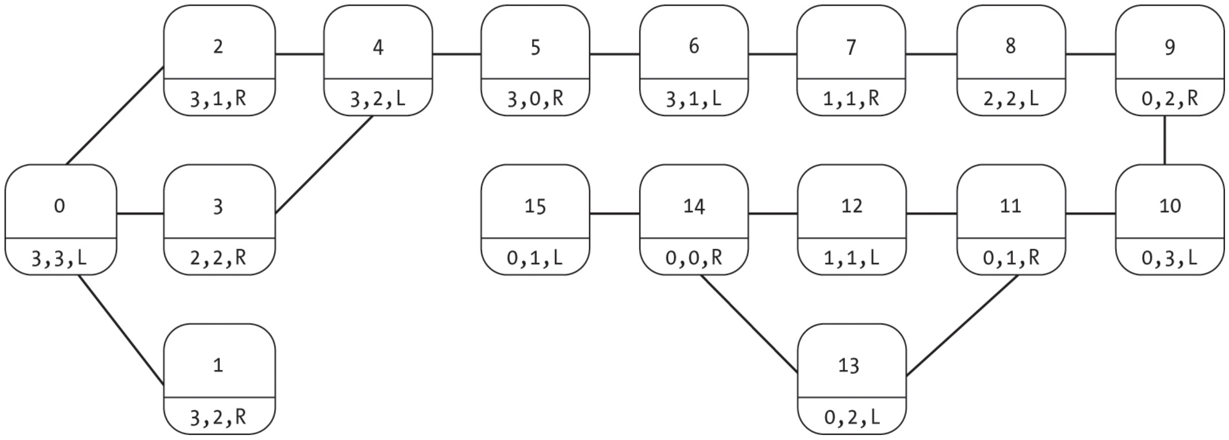


Figure 8-6: The state graph for the prisoners-and-guards river-crossing puzzle

We can use two basic facts about the physical properties of the boat to derive an admissible metric indicating the distance to the goal state:

- The boat can carry at most two people. If there are *k* people on the left shore, we need to least *ceil*(*k* / 2) trips to get them all onto the right shore.
- If the boat is on the right shore, it must return to the left shore before picking up more people.

Using those conditions, we can define a function to generate the heuristic from the `PGState` stored in each node:

```
def pg_generate_heuristic(g: Graph) -> list:
    heuristic = [0.0] * g.num_nodes
    for node in g.nodes:
        state: PGState = node.label
      ❶ num_left: int = state.guards_left + state.prisoner
s_left
      ❷ min_trips_l_to_r: int = math.ceil(num_left / 2.0)
      ❸ min_trips_r_to_l: int = max(0, min_trips_l_to_r -
 1)
        if not state.boat_side == "L" and min_trips_l_to_r
```

```
> 0:
                min_trips_r_to_l += 1
        heuristic[node.index] = min_trips_l_to_r + min_tri
ps_r_to_l

    return heuristic
```

The code loops over the graph's nodes with a `for` loop, checking the puzzle state of each node to determine the number of people on the left shore ❶. It then computes the minimum number of trips the boat needs to make from left to right by considering how many people still need to be moved over and noting that at most two people can come over each time ❷. It also computes the minimum number of trips the boat needs to make from right to left by noting that, while there are more passengers to transport, the boat needs to return to the left shore to pick them up ❸. The heuristic is the sum of these two sets of trips.

Table 8-1 compares the values of this heuristic function for each state with the true distance to the goal node, which we calculate by counting the steps from each state to the goal in Figure 8-6. As you can see, the heuristic is admissible and never overestimates the true distance.

Table 8-1: Values from the `pg_generate_heuristic()` Function vs. the True Distance to the Goal Node

State	**3 3 L**	**3 2 R**	**3 1 R**	**2 2 R**	**3 2 L**	**3 0 R**	**3 1 L**	**1 1 R**	**2 2 L**	**0 2 R**	**0 3 L**	**0 1 R**	**1 1 L**	**0 2 L**	**0 0 R**	**0 1 L**
Heuristic distance	5	6	4	4	5	4	3	2	3	2	3	2	1	1	0	1
True distance	11	12	10	10	9	8	7	6	5	4	3	2	1	1	0	1

Given this heuristic function, we can run A* search on the river-crossing puzzle:

```
g: Graph = create_prisoners_and_guards()
h: list = pg_generate_heuristic(g)
last: list = astar_search(g, h, 0, 14)
```

The prisoners-and-guards problem provides a demonstrative example of applying A* search to puzzles because we can enumerate the states and compare the heuristic values to the true best path. However, the structure of the graph, with a single long sequence of states without branches, means that A* search does not significantly outperform breadth-first search on this puzzle. In contrast, A* search could provide significant advantages on puzzles with much larger state spaces, since the power of A* search is its ability to focus on exploring only promising paths toward the goal.

Searching Unknown Graphs

While the algorithms introduced in this chapter so far have treated both the graph and heuristic values as known items passed to the search, these approaches work equally well on problems for which the algorithm needs to dynamically construct an unknown graph. Consider the puzzle-building examples from Chapter 6. There, we used breadth-first search to quite literally explore the state space, building up a graph as we encountered new nodes and edges. We can do the same thing with heuristically guided searches.

Instead of passing a list of heuristic values to each node, we could pass a function that dynamically evaluates the heuristic based on the information in the node. For example, if a node has auxiliary data *x* and *y*, indicating its spatial position, we could define the heuristic function to be the Euclidean distance from that node to the known goal position. In the case of a real-world explorer, this might correlate to their using a GPS to estimate the distance to the objective as they trek through the jungle.

We can visualize this dynamic construction and evaluation using the "clouded out" mechanism from video games. For example, Figure 8-7 shows a grid as a 5×5 maze. Explored areas like the long dead end at the bottom of the graph are shown as boxes, while unexplored areas are grayed out. Anything could be in the gray zone: a direct path to the goal, numerous dead ends, or a giant monster. We won't know until we explore it.

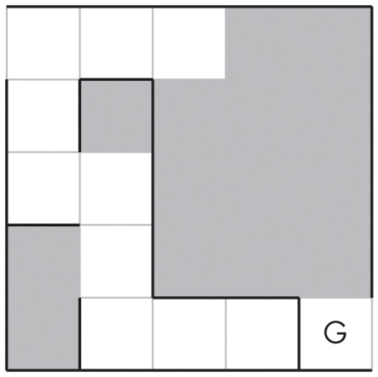


Figure 8-7: A maze with the unseen areas grayed out

In this section, we modify the code from Listing 8-1 to conduct dynamic exploration by constructing the graph as new nodes are discovered. The auxiliary data structures such as `last` and distance must also be dynamically extended to account for new states.

The Code

For our example code, we generalize the algorithm by using a `World` class. This class provides basic information about the puzzle, including the following:

- The starting state's index
- A given state's neighbors
- The cost of a transition between any two neighboring states
- The heuristic value at a state
- Whether the current state is the goal

Given this interface, we do not need to know anything about the state space ahead of time. Just like a real-life explorer consulting GPS and road signs, we use the `World` interface throughout the algorithm to examine the local state of the world.

Note that the `World` class does not need to enumerate the state space. It also does not need to construct and store the full graph. We can use functions like the ones in Chapter 6 to dynamically determine a state's neighbors given information about the state. This dynamic evaluation allows us to explore massive state spaces without a huge memory overhead.

We can define a simple `World` class for the distance-based example search in this chapter as follows:

```
class World:
    def __init__(self, g: Graph, start_ind: int, goal_ind:
int):
        self.g = g
        self.start_ind = start_ind
        self.goal_ind = goal_ind

    def get_num_states(self) -> int:
        return self.g.num_nodes

    def is_goal(self, state: int) -> bool:
        return state == self.goal_ind

    def get_start_index(self) -> int:
        return self.start_ind

    def get_neighbors(self, state: int) -> set:
        return self.g.nodes[state].get_neighbors()

    def get_cost(self, from_state: int, to_state: int) ->
 float:
        if not self.g.is_edge(from_state, to_state):
            return math.inf
        return self.g.get_edge(from_state, to_state).weigh
t

    def get_heuristic(self, state: int):
      ❶ pos1 = self.g.nodes[state].label
        pos2 = self.g.nodes[self.goal_ind].label
      ❷ return math.sqrt((pos1[0]-pos2[0])**2 + (pos1[1]-p
os2[1])**2)
```

The `World` class stores the underlying graph (`g`), the starting index (`start_ind`), and the goal index (`goal_ind`). In this example, they are all provided by the user. The class then uses basic getter functions to provide the information needed. For example, `get_start_index()` returns the index of the starting state, while `get_neighbors()` lists the neighboring states. The `get_cost()` function uses the actual edge cost between two nodes, or infinity if there is no such edge.

The `get_heuristic()` function assumes that the node's coordinates are stored in its label as an (x, y) tuple or list ❶ and uses the Euclidean distance to the goal as the heuristic ❷ (which requires the code to include `import math`). When creating the graph, we will also need to set the labels of the nodes to contain these coordinates.

Using the `World` class, we create a modified version of A* search that dynamically allocates and fills the data structures. For simplicity's sake, we use dictionaries to map each state's index (or string) directly to the corresponding information, as shown in Listing 8-2.

```
def astar_dynamic(w: World):
    visited: dict = {}
    last: dict = {}
    cost: dict = {}
    pq: PriorityQueue = PriorityQueue(min_heap=True)
    visited_goal: bool = False

  ❶ start: int = w.get_start_index()
    visited[start] = False
    last[start] = -1
    pq.enqueue(start, w.get_heuristic(start))
    cost[start] = 0.0

    while not pq.is_empty() and not visited_goal:
        index: int = pq.dequeue()
        visited[index] = True
        visited_goal = w.is_goal(index)

      ❷ for other in w.get_neighbors(index):
            c: float = w.get_cost(index, other)
            h: float = w.get_heuristic(other)

          ❸ if other not in visited:
                visited[other] = False
                cost[other] = cost[index] + c
                last[other] = index
                pq.enqueue(other, cost[other] + h)
          ❹ elif cost[other] > cost[index] + c:
                cost[other] = cost[index] + c
                last[other] = index
                pq.update_priority(other, cost[other] + h)

    return last
```

Listing 8-2: An A search for an unknown graph*

The code in Listing 8-2 defines a modified version of the `astar_search()` function defined in Listing 8-1, `astar_dynamic()`. This function creates empty helper data structures and inserts the starting state into each one ❶. The use of dictionaries means we do not need to know anything about the number of total states or their underlying indices. At this point, the data structure has information only about that single state because the code has not explored any others. It gets the index of the starting state from the `get_start_index()` function and its estimated cost (priority) via the `get_heuristic()` function.

The Listing 8-2 algorithm uses a `while` loop to explore states on the priority queue until it has either run out of states to explore or found the goal. During each iteration, the algorithm dequeues the most promising state (`index`), marks it visited, and checks whether it is the goal by using the `is_goal()` function. In the real world, this might consist of entering a new city and looking around for familiar landmarks.

For each state it explores, the code checks all its neighbors using the `get_neighbors()` function to return the state's local neighborhood ❷. The code then computes the cost from the current node to that neighbor (`c`), using the `get_cost()` function to return the edge weight. Similarly, it dynamically computes the heuristic value of that neighbor (`h`) using the `get_heuristic()` function.

Once it has the distance to the neighbor and that neighbor's heuristic value, the code checks whether it has encountered the neighboring state before. It checks whether the state has been seen by testing whether it has an entry with any value in the `visited` dictionary ❸. If the neighboring state is not in the dictionary, it has never been seen and is added to each data structure. The neighboring state's cost is the sum of the cost to get to the current state (`cost[index]`) plus the cost of the next state transition (`c`). The neighboring state's priority is this best cost to that state (`cost[other]`) plus the heuristic estimate at that state (`h`).

If the neighbor has been seen before ❹, the code checks whether the new path is better by comparing the neighbor's cost (`cost[other]`) with the cost through the new path (`cost[index] + c`). If the code finds a better path, it updates the path to the state (`last`), the cost to reach the state (`cost`), and the state's priority. Again, the code updates the priority in the queue directly.

An Example

Let's apply the `astar_dynamic()` function to the graph from Figure 8-1. Of course, the algorithm does not know the graph ahead of time. It does not even know how many nodes the graph has. All the code sees is what the `World` class provides.

We can extend our graph from the greedy and A* examples shown in Figures 8-3 and 8-5 by adding the node's spatial position in its label, as in the following code:

```
g.nodes[1].label = [0, 2]
```

Admittedly, while this manual assignment approach works for the purpose of setting up this demonstrative example, it leads to a long problem and tedious setup for a large number of points. We discuss programmatic approaches to reading in graph data in Appendix A.

Figure 8-8 shows the search. In Figure 8-8(a), the algorithm knows the starting state with index 0 and the existence of a goal state. However, it knows nothing about the rest of the graph, including the edges out of node 0. The goal node doesn't even have a number yet because the search has not seen it.

When the search visits node 0 in Figure 8-8(b), it discovers three edges to three neighbors. Each of these edges has a weight provided by the `World` class's `get_cost()` function and a heuristic value provided by the class's `get_heuristic()` function. While this is not much information, it is enough for us to build up a picture of the neighborhood around the starting state. The search augments its auxiliary data structures (`visited`, `last`, `cost`, and `pq`) to account for this by adding new entries for the corresponding nodes. It does not explicitly create a graph or store the edges.

This search follows the same sequence as the A* search example in Figure 8-5. The main difference in Figure 8-8 lies in what the search knows about the graph at each step. It is only guaranteed to have seen all the node's edges once it visits that node. For example, while the algorithm knows about both nodes 1 and 4 for several iterations, it does not learn about the edge between them until Figure 8-8(d), when it has visited node 1.

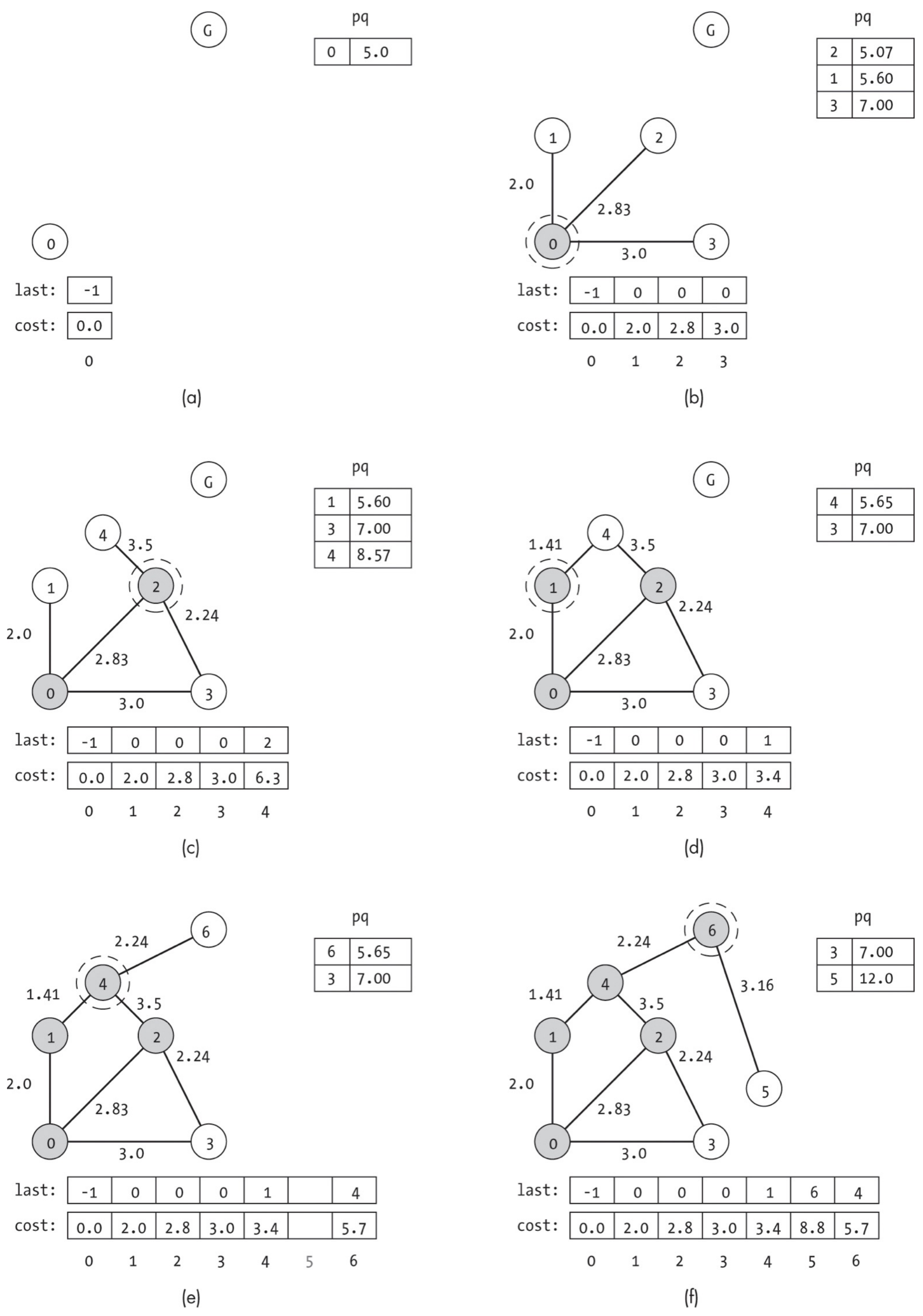


Figure 8-8: The steps of an A search algorithm on an unknown graph*

As in the previous A* search example, the code continues until it visits the goal node. At this point the `World` class's `is_goal()` function returns `True`, and we know that we have found the shortest path. However, as shown by Figure 8-8(f), this does not mean the search has explored the entire graph.

Not only did it skip visiting nodes 3 and 5, but it also never learned about the edge between them. There could be an entire world branching out beyond those nodes.

Why This Matters

Greedy best-first search and A* search provide mechanisms for incorporating heuristic estimates into our search algorithms, helping us find the best paths between two nodes. Greedy best-first search is simple and needs to track very little information but may produce nonoptimal paths. The combination of an admissible (optimistic) heuristic and good bookkeeping on the cost so far allows A* search to efficiently choose which nodes to explore while guaranteeing it finds the lowest-cost path.

The main advantage of these algorithms, particularly A*, is that the use of heuristic information focuses the search. Just as GPS coordinates can help us determine which of two possible roads will bring us to our destination more quickly, heuristics allow us to prioritize which nodes to explore next. As a result, A* search is a practical algorithm that has become a staple of artificial intelligence and video game path planning.

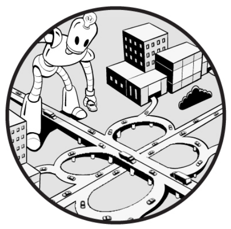
In the next part of the book, we step away from search algorithms and examine tasks related to the connectivity of graphs. We review how to sort nodes in a directed graph, consider how to test the connectivity of graphs, and examine random behavior on graphs. Many of these algorithms use searches as a core component.

PART III

CONNECTIVITY AND ORDERING

9

TOPOLOGICAL SORT



In addition to their physical corollaries, such as one-way streets, we can use directed edges in a graph to specify a *dependency* or *ordering* between nodes. Incoming edges to a node represent links from actions that must be completed before reaching the current node. Meanwhile, outgoing edges point to actions enabled by the completion of the current node.

Consider the example of a recipe for chocolate chip cookies. Each node represents a step in the recipe, including instructions such as “Add the flour” and “Stir the mixture.” Some of the steps have a clear and non-negotiable ordering. We wouldn’t want to stir the mixture before adding anything to the bowl or add ingredients before we measure them.

This chapter introduces algorithms for *topological sort*, which orders a graph’s nodes in the order specified by directed edges. After discussing the concepts behind topological ordering and presenting several real-world use cases and algorithms for sorting, we’ll explore the importance of graphs without loops for topological sort and consider why instruction manuals are not drawn as graphs.

How Topological Sort Algorithms Work

A topological sort algorithm finds an ordering of nodes such that if there exists a directed edge from node *u* to node *v*, then node *u* must precede node *v* in the ordering. In other words, each directed edge represents a *dependency*. Some graphs have multiple valid orderings. For example, Figure 9-1 shows a graph with valid topological orderings [0, 1, 2, 4, 3] and [0, 2, 1, 4, 3].

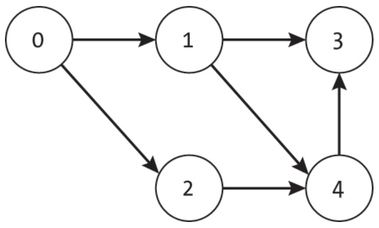


Figure 9-1: A directed graph with two valid topological orderings

A key constraint for topological sort is that the graph must be *acyclic*, meaning it does not contain any *cycles*. A directed graph has a cycle if it is possible to define a path from a node such that the path returns to that same node, as shown in Figure 9-2 (a slight modification of the graph in Figure 9-1). Starting from node 1, we can travel back to node 1 through nodes 4 and 3 via the path [1, 4, 3, 1].

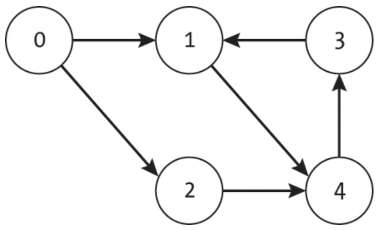


Figure 9-2: A directed graph with a cycle

A graph with cycles will not have a valid topological ordering because it contains at least one node with a path that leads back to itself. That means this node must precede itself in the sorted list, which is obviously impossible. No matter how we order nodes 1, 3, and 4 in Figure 9-2, we'll always have an edge pointing from a later to an earlier node. We call a directed graph with no cycles a *directed acyclic graph* or *DAG*.

We can test whether a topological ordering is valid by checking the relative ordering of each pair of nodes with a directed edge:

```
def is_topo_ordered(g: Graph, ordering: list) -> bool:
  ❶ if len(ordering) != g.num_nodes:
        return False

  ❷ index_to_pos: list = [-1] * g.num_nodes
    for pos in range(g.num_nodes):
        current: int = ordering[pos]
        if index_to_pos[current] != -1:
            return False
        index_to_pos[current] = pos

  ❸ for n in g.nodes:
        for edge in n.get_edge_list():
            if index_to_pos[edge.to_node] <= index_to_pos
[n.index]:
                return False
    return True
```

The code starts by checking that the ordering contains the same number of nodes as the graph ❶. If not, the ordering cannot be valid, and the function returns `False`. Next, a `for` loop builds an inverted index mapping each node to its location in the sorted array ❷. This will allow it to easily look up the relative ordering of any two nodes given their indices. During this loop, the code also checks for duplicate nodes in the ordering by testing if `index_to_pos` is already set. If it finds a node listed twice, the function returns `False` to indicate an invalid ordering.

The code then iterates through each node and each of its outgoing edges using a nested pair of `for` loops ❸. For each of these directed edges, the code checks that the nodes are ordered correctly by comparing their positions in `ordering` (using `index_to_pos`). The function returns `False` as soon as it finds a single pairing out of order. Finally, if the code has made it through this gauntlet of checks, it returns `True`.

Use Cases

Topological sort has a variety of real-world use cases. This section describes how to represent several of these cases as graphs to which we could apply the topological sort.

Code Dependencies

Programmers often break large programs into a series of modules or libraries to facilitate understandability, maintainability, and extensibility. Instead of a single file with a million lines of code, a programmer might create three modules: one for representing and processing graphs, another for handling the user interface, and a third for reading and writing files. Ideally, they would design the modules to build off one other, allowing them to reuse core library functions throughout the code. For example, the graph library and user interface code might rely on the file module to read and write configuration files.

Such code dependencies mean that the code is processed in a particular order. We can represent these dependencies as a directed graph where each module is a node in the graph and each `import` or `include` statement represents an incoming edge to that node. The topological sort of this graph tells the computer the order in which it needs to process the files.

Task Lists

We can extend the importance of following a recipe in a particular order to a wide range of tasks, from writing to cleaning to assembling furniture. In each of these cases, nodes represent steps on our task list and edges represent dependencies between them. For example, we need to get out the mop and bucket before we can wash the floor. As another example, Figure 9-3 shows a graph representing the steps for making a peanut-butter-and-jelly sandwich.

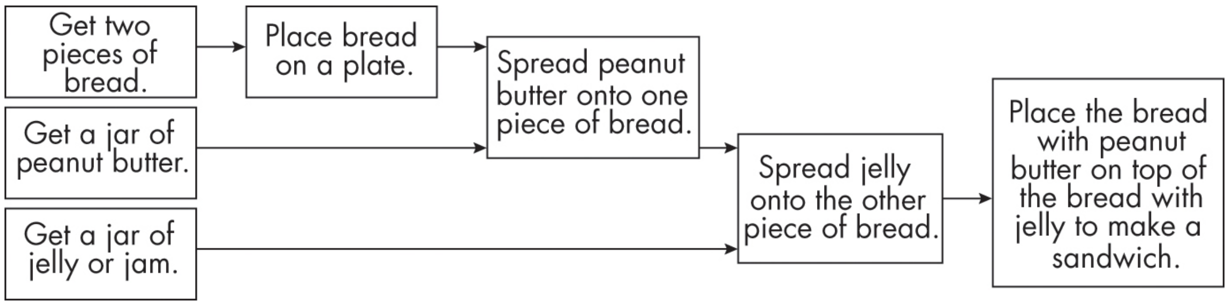

Get two pieces of bread.
Place bread on a plate.
Get a jar of peanut butter.
Spread peanut butter onto one piece of bread.
Get a jar of jelly or jam.
Spread jelly onto the other piece of bread.
Place the bread with peanut butter on top of the bread with jelly to make a sandwich.

Figure 9-3: The task graph for making a peanut-butter-and-jelly sandwich

Although most instruction manuals are written as a linear series of instructions, there is often some flexibility in the order, meaning the graph is not necessarily a straight line of nodes. For example, consider the process of assembling a prefabricated chair. It might not matter whether you attach the left or right arm first. However, it could be vital that you finish the body of the chair before attaching the seat cushion.

Representing instructions as a graph clearly offers opportunities for parallelism. Two people working on a single recipe could theoretically complete it more quickly than one alone; one baker can measure out the flour while another measures out the sugar, and so on. Unfortunately, this parallel representation would likely cause more rather than fewer problems. I personally struggle to remember which step I'm on when using linear instructions, spending minutes trying to remember whether I've added the salt to the bowl. Tracking which steps have been completed through a branching graph of tasks is almost guaranteed to result in me missing steps.

Fortunately, while humans struggle to track this type of state, computers excel at it, using graph structures to find opportunities for parallelism. In fact, there are entire systems that structure workflows as graphs. Distributed workflow systems are often built around the concept of task graphs, where multiple tasks are executed in an order determined by their dependency. The design and optimization of such workflows is an active area of development, and the topic could fill its own book.

Teaching and Learning

Many academic subjects consist of a large set of interrelated concepts, some of which must be understood before moving on to the next. A math teacher would find it difficult to explain the concept of exponents before their students have learned how to multiply. Other concepts can be covered in parallel, however. When learning a foreign language, the student might be able to learn vocabulary related to coffee shops regardless of whether they know computer science terminology.

We can specify the recommended order for teaching or learning concepts as a graph. Nodes represent concepts, such as the French word for coffee or the idea of recursion, while edges represent dependencies between the concepts. In the case of computer science, we might put an edge from the concept of functions to that of recursion to indicate that a student should learn about functions before venturing into the world of recursion.

Relying on this graph representation, we can use topological sort to determine which courses in a college curriculum we need to take first. Consider the graph in Figure 9-4 built from hypothetical course prerequisites.

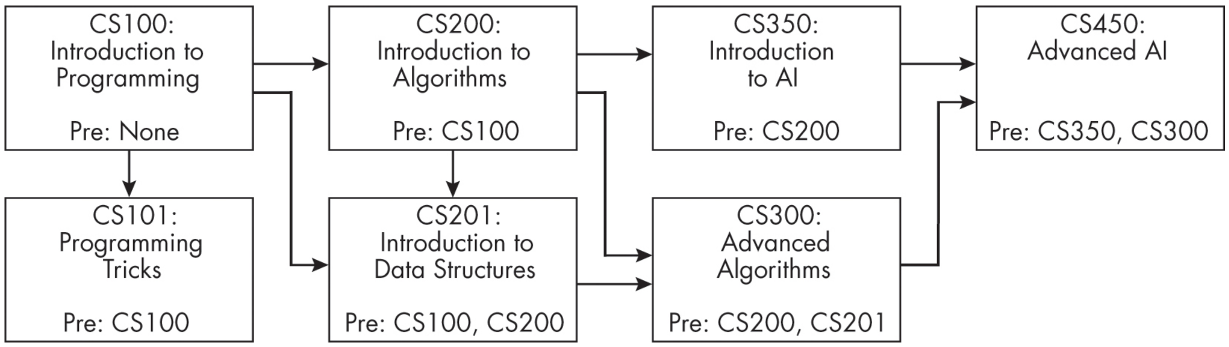

CS100: Introduction to Programming
Pre: None
CS200: Introduction to Algorithms
Pre: CS100
CS350: Introduction to AI
Pre: CS200
CS450: Advanced AI
Pre: CS350, CS300
CS101: Programming Tricks
Pre: CS100
CS201: Introduction to Data Structures
Pre: CS100, CS200
CS300: Advanced Algorithms
Pre: CS200, CS201

Figure 9-4: A graph representing the prerequisites of computer science courses

To take CS450: Advanced AI, a student would need to complete the chain of prerequisites for this course, all the way back to CS100: Introduction to Programming.

Kahn's Algorithm

Computer scientist Arthur B. Kahn developed one approach, now called *Kahn's algorithm*, for performing topological sort on a directed acyclic graph. This algorithm operates by finding nodes with no incoming edges, removing them from the list of pending nodes, adding them to a sorted list, and then removing the outbound edges from that node. The algorithm repeats until it has added every node to the sorted list.

Intuitively, the Kahn's algorithm sort mirrors how we might perform a complex task in the real world. We perform a subtask that we can accomplish without completing any other steps first, then move on to another subtask. Any action that requires us to have performed an as-yet-uncompleted task must wait until we have finished all its dependencies.

The Code

When implementing Kahn's algorithm, we don't need to modify the graph we're working with by removing nodes or edges. Instead, we can use an additional array `count` to track the number of incoming edges to each node and decrease those counts whenever we "remove" a node. Since we don't have to modify the graph data structure, we can avoid making an initial copy, re-adding the removed edges, or leaving the structure altered.

The code for Kahn's algorithm uses a stack and multiple loops, as shown in Listing 9-1.

```
def Kahns(g: Graph) -> list:
    count: list = [0] * g.num_nodes
    s: list = []
    result: list = []

  ❶ for current in g.nodes:
        for edge in current.get_edge_list():
            count[edge.to_node] = count[edge.to_node] + 1
    for current in g.nodes:
        if count[current.index] == 0:
            s.append(current.index)

  ❷ while len(s) > 0:
        current_index: int = s.pop()
        result.append(current_index)
      ❸ for edge in g.nodes[current_index].get_edge_list
():
            count[edge.to_node] = count[edge.to_node] - 1
            if count[edge.to_node] == 0:
                s.append(edge.to_node)
    return result
```

Listing 9-1: Kahn's algorithm for topological sorting

The code starts by creating the helper data structures used by the algorithm. The array `count` stores the count of *incoming* nodes for each edge and will be used to check for nodes without any incoming edges. The stack `s` (implemented as a list) stores the indices of unprocessed nodes without incoming edges. These will be the nodes that the code can remove from the graph. Finally, the list `result` will hold the topologically sorted list of node indices.

The code uses a pair of nested `for` loops to count the number of incoming edges for each node ❶. In the first loop, since the algorithm is working on a directed graph, the code must iterate over all nodes (outer loop) and their outgoing edges (inner loop) and increment the count of incoming edges for the edge's destination (`to_node`). The next `for` loop iterates over the `count` array, finds nodes that have no incoming edges (`count[current.index] == 0`), and inserts their index into the stack `s`. The function has now set up all the initial information it needs to perform the topological sort.

The main body of the code is a `while` loop that processes each item in the stack ❷. At each iteration, the code pops a node index from `s`. It retrieves the corresponding node and adds this index to the end of the `result` array. The function then virtually removes the node and its outgoing edges by iterating over each outgoing edge and decreasing the count of edges going to the destination ❸. In the process, it checks whether any node falls to zero incoming edges and, if so, adds that node's index to the stack. The code completes by returning the array of sorted node indices with `return result`.

Each iteration focuses on a single node and its outgoing edges, meaning the running time of the algorithm scales linearly with the number of nodes plus the number of outgoing edges.

An Example

Figure 9-5 shows how to run Kahn's algorithm on an example graph, where each subfigure represents a step in the algorithm's progression. The figure displays the count of the incoming edges (the values in `count`) above each node and grays out removed nodes and edges.

The algorithm initializes the counts of incoming edges based on the input graph and stores the two nodes without any incoming edges (nodes 0 and 1) in the stack, as shown in Figure 9-5(a). During the first step of the sort in Figure 9-5(b), the algorithm takes the top element (node 1) off the next stack and "removes" it and its outgoing edges from the graph, resulting in decreased counts for nodes 3 and 4. Since neither destination node's count decreases to zero, the code adds neither to the stack.

In Figure 9-5(c), the sort continues by popping node 0 from the stack and removing its edges to nodes 2 and 3. This reduces the count in incoming edges to both nodes 2 and 3 down to zero, allowing the sort to add them to the stack. The process continues, node by node, until every node in the stack has been exhausted.

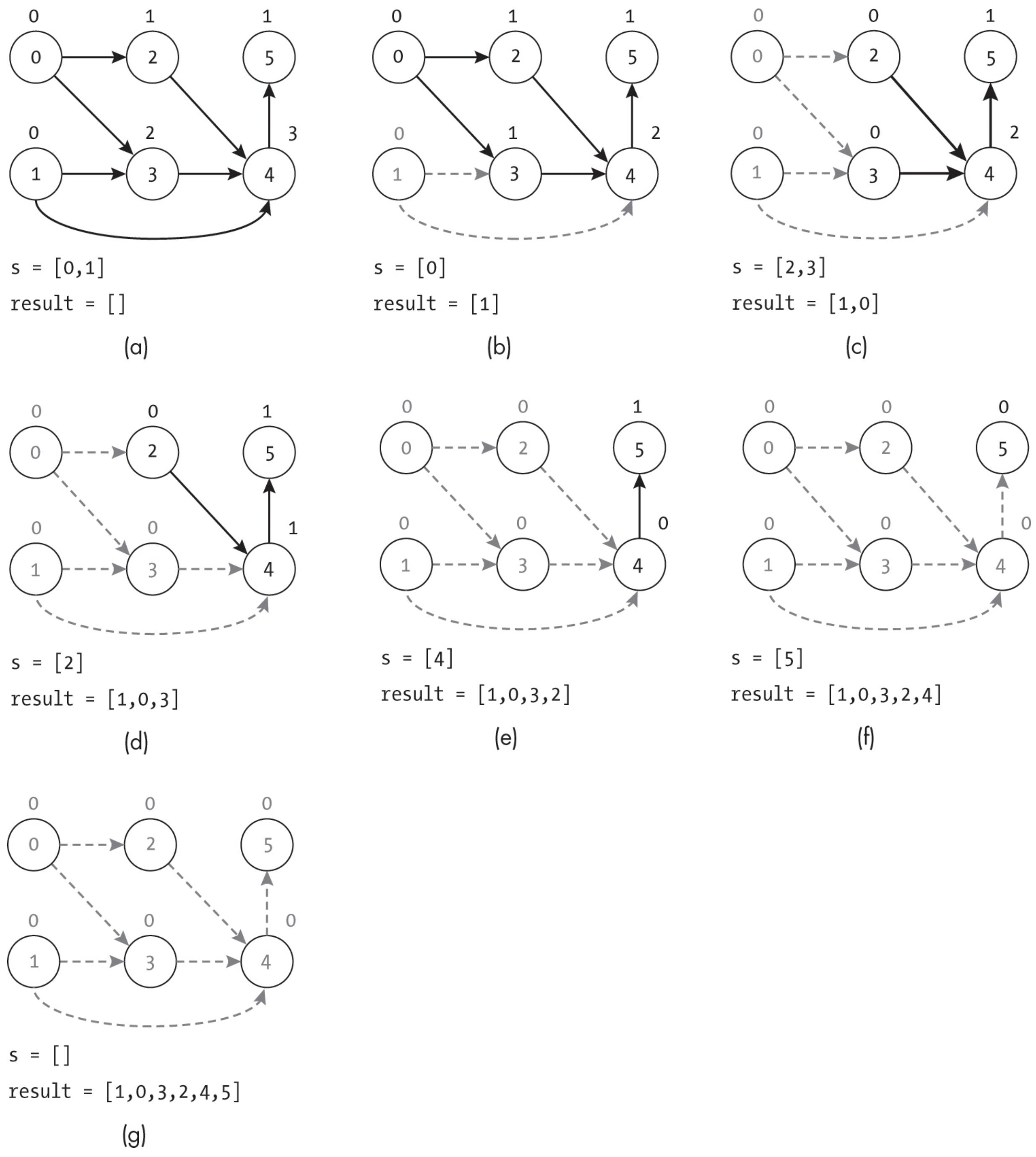


Figure 9-5: Running Kahn's algorithm on an example graph

We can picture Kahn's algorithm in the context of a careful baker following a complex recipe. Before they start, the baker writes down all the tasks and which ones are the necessary prerequisites. Task 5a, "Add two cups of flour," must come after Task 4, "Mix the wet ingredients in a large bowl," and Task 1a, "Measure two cups of flour." However, the baker can complete Task 5a either before or after Task 5b, "Add a tablespoon of baking soda," because the order in which the flour and baking soda are added does not matter. The baker counts the number of prerequisites for each task and writes those numbers next to their respective tasks.

The baker begins by choosing one task with no outstanding prerequisites, performing that task, and checking it off the list. They then go through the list of tasks and update the count of unsatisfied prerequisites for all future tasks that depended on the one they just completed. After measuring out two cups of flour, for example, they can update the dependency count of Task 5a, "Add two cups of flour," from 2 to 1—they've measured the flour but still need to mix the wet ingredients before proceeding. Once they've completed Task 4, "Mix the wet ingredients in a large bowl," they can put Task 5a on their list of next steps.

Depth-First Search

Beyond Kahn's algorithm, an alternative approach to sorting the nodes in a directed acyclic graph is to use the trusty multipurpose tool of depth-first search. Depth-first search starts at a given node *u*, then explores the nodes after *u* in topological order. We can modify depth-first search to track the order in which we complete processing of each node. By saving the order in which the search finishes exploring each node, depth-first search can reconstruct the *inverse* ordering of nodes. The last nodes in topological ordering will be the first to finish in a depth-first search and thus will appear at the start of the list.

The Code

The code for topological sort based on depth-first search largely mirrors the other depth-first search implementations that we have used throughout the book, as shown in Listing 9-2. However, we maintain one additional piece of information: a list `s` to track the order in which the search completes its visits to each node.

```
def topological_dfs(g: Graph) -> list:
    seen: list = [False] * g.num_nodes
    s: list = []
 ❶ for ind in range(g.num_nodes):
        if not seen[ind]:
            topological_dfs_recursive(g, ind, seen, s)
 ❷ s.reverse()
    return s

def topological_dfs_recursive(g: Graph, index: int, seen:
 list, s: list):
    seen[index] = True
    current: Node = g.nodes[index]
    for edge in current.get_edge_list():
        neighbor: int = edge.to_node
        if not seen[neighbor]:
            topological_dfs_recursive(g, neighbor, seen,
 s)
 ❸ s.append(index)
```

Listing 9-2: The depth-first search algorithm for topological sorting

The code in Listing 9-2 consists of two functions. The outer `topological _dfs()` function sets up the data structures, calls depth-first search on different starting nodes, and processes the results. It starts by creating an empty list `s` and a list `seen` with all entries set to `False`. The function then loops through each node. If it finds a node that has not yet been visited, the function starts a depth-first search from that node ❶. Finally, the function takes the list of node indices and outputs them to the result list in reverse order ❷.

The inner `topological_dfs_recursive()` function that follows is a recursive implementation of depth-first search with one modification: it appends each completed node onto a list. This function starts by marking the current node as seen, then iterates through the neighbors via the list of edges and recursively explores any unseen neighbors. Finally, it inserts the current node index at the end of the list `s` to track the order in which it finished visiting the nodes ❸.

An Example

Figure 9-6 shows an example topological sort by depth-first search, where the current node being visited is circled by a dashed line and the seen nodes are shaded.

The search begins in Figure 9-6(a) on node 0 and explores down the path of node 2 in Figure 9-6(b), node 4 in Figure 9-6(c), and node 5 in Figure 9-6(d). At each node, the algorithm marks the node as seen and recursively explores its unseen neighbors. It isn't until the search finishes processing a node and backtracks to its predecessor that it inserts the node into the sorted list, as shown in Figure 9-6(d), where the search hits a dead end at node 5 and is forced to backtrack. Inserting a node into the inversely sorted list means that node must come after all nodes that preceded it in the search.

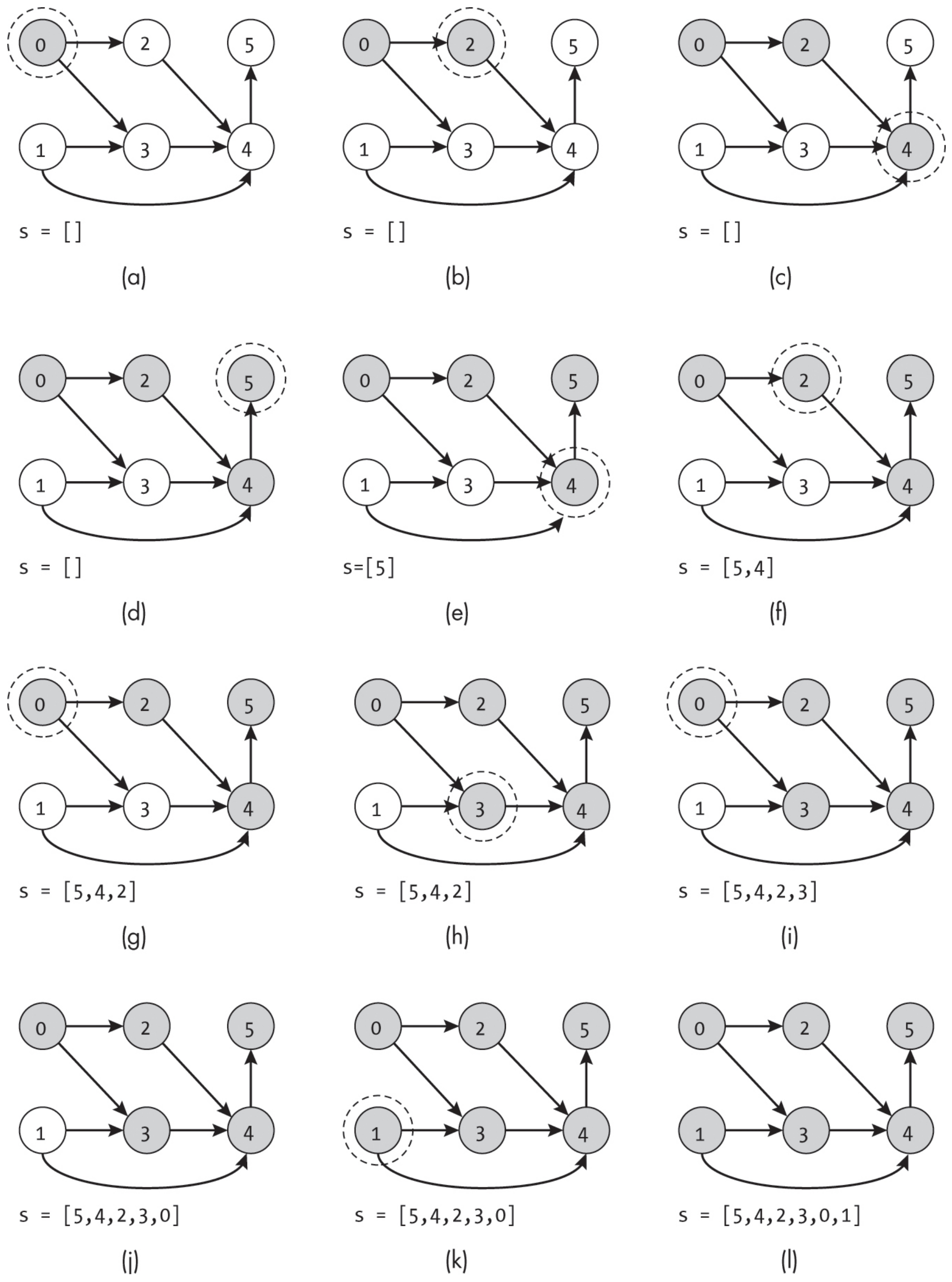


Figure 9-6: Running a depth-first search for topological sort on an example graph

As the search backtracks through node 4 in Figure 9-6(e) and node 2 in Figure 9-6(f), it checks for other outgoing edges. Finding none, it adds the current node to the list and backtracks. When it returns to node 0 in Figure 9-6(g), the depth-first search can continue before backtracking once more. It recursively explores node 3 in Figure 9-6(h).

As shown in Figure 9-6(j), we might not be done after our initial depth-first search completes. Node 1 did not lie on any paths out of node 0 and thus has not been explored. The search continues by checking whether all the nodes have been seen and, if not, starting a depth-first search from that unseen node. At the end of the algorithm, the search has visited all nodes and assembled a list of nodes in inverse topological order, as shown in Figure 9-6(l).

We can picture this search in the context of a college student planning out a series of courses to take. They list the courses they want to take as nodes in a graph and use directed edges to represent the prerequisites. The depth-first search starts at a node with the question, "What courses will this course allow me to take?" When the search reaches a dead end, the student knows they have found a course that is not a prerequisite for anything else in their curriculum

Returning to the course list in Figure 9-4, consider what happens when the student starts with CS200 and follows the path through CS350 to CS450. CS450 is fully explored, so they turn to their list of unexplored courses. The student returns to CS350, which is not a prerequisite for any other classes, and adds it to the list. Ultimately, they build up the list [CS450, CS350, CS300, CS201, CS200]. They continue to the next unvisited course (perhaps CS100 in this case) and continue building out their study plan.

Order of Starting Nodes

One counterintuitive aspect of the depth-first search approach to topological sort is that the `topological_dfs()` base function in Listing 9-2 starts the recursive searches on each node according to that node's index in the graph. It does not bother to sort the nodes according to number of incoming edges or any other aspect of its location.

This leads to cases where the search might begin at a node that has incoming edges, as with the graph in Figure 9-7. After all, we naturally think of depth-first search as starting at the beginning of a chain of nodes.

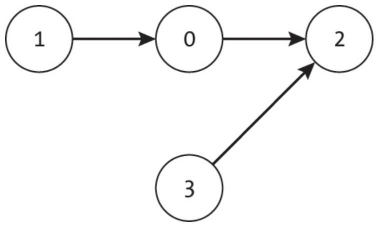


Figure 9-7: A graph where node 0 has an incoming connection

Fortunately, depth-first search works perfectly well in situations where node 0 is not the start of the topological ordering. In Figure 9-7, the depth-first search algorithm will start at node 0 and find nodes 0 and 2 in the initial recursive depth-first search. At the end of the first recursive depth-first search, the list containing the reversed ordering is `[2, 0]`. While it may seem like we are making an error in skipping node 1, we will add it to the correct location during the next search.

The algorithm next starts a search at node 1 and adds that to the end of the list. Since the list is in *reverse* topological ordering, node 1 is in the correct location *after* node 0. After the search starts at node 1, the list is `[2, 0, 1]`. After starting at node 3, the list becomes `[2, 0, 1, 3]`. When the function concludes, it reverses the list using Python's `reverse()` function and returns the correct ordering `[3, 1, 0, 2]`.

Detecting Cycles

As noted earlier, a key constraint of topological sort is that the graph must be acyclic. Consider the hypothetical course sequence shown in Figure 9-8. All students must begin by taking CS100: Introduction to Programming, which has no prerequisites. However, things get more complex in the next semester. In an attempt to cover more material, the instructor of CS200: Introduction to Algorithms wants their students to know both programming and the basics of graphs. They require both CS100 and CS202: Introduction to Graphs as prerequisites. At the same time, the instructor for CS202 wants their students to know data structures, so they require CS201: Introduction to Data Structures as a prerequisite. Unfortunately, the instructor for CS201 wants their students to already know basic algorithms, so they require CS200 as a prerequisite.

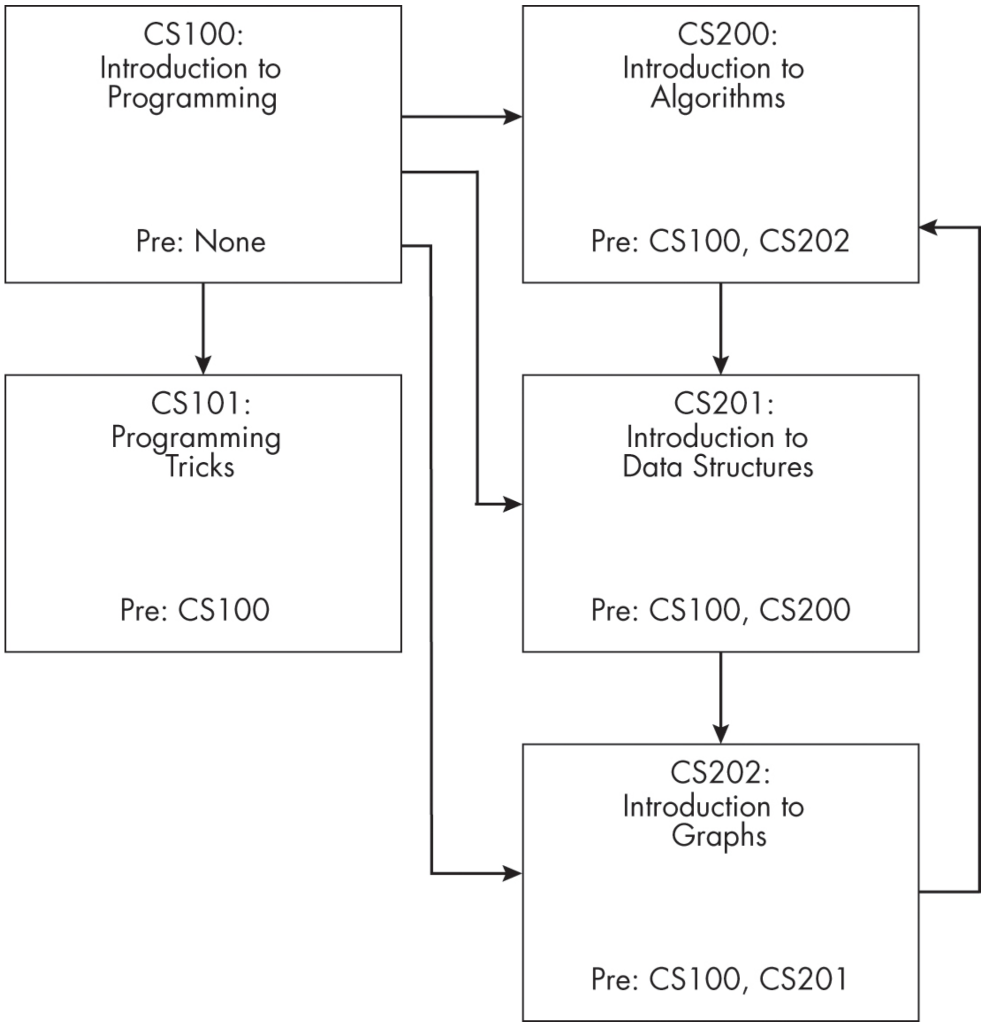

CS100: Introduction to Programming
Pre: None
CS200: Introduction to Algorithms
Pre: CS100, CS202
CS101: Programming Tricks
Pre: CS100
CS201: Introduction to Data Structures
Pre: CS100, CS200
CS202: Introduction to Graphs
Pre: CS100, CS201

Figure 9-8: A set of course prerequisites with a cycle

When a student completes CS100, they look at the courses they can now take and run into a problem. Each of the 200 level courses requires a different 200 level course as a prerequisite. There are no 200 level courses the student can take without missing a prerequisite.

We can adapt the algorithms presented in this chapter to detect whether a graph has cycles, providing a useful tool for vetting school requirements, instruction manuals, recipes, or any other graph representation of a sequence of events. One easy approach to doing this is to look at what happens when we run Kahn's algorithm from Listing 9-1 on a graph with cycles. This algorithm relies on incoming connections dropping to zero when it has removed all the preceding nodes. To remove a node that is part of a cycle, the algorithm would first need to remove its predecessors, including the node itself. Thus, the count for a node in a cycle will never drop to zero, and the node will never be added to the sorted list. We therefore know that if the algorithm's returned list does not contain all the nodes from the graph, at least one node must be part of a cycle and thus be unremovable.

We wrap Kahn's algorithm in a function called `check_cycle_kahns()`:

```
def check_cycle_kahns(g: Graph) -> bool:
    result: list = Kahns(g)
    if len(result) == g.num_nodes:
        return False
    return True
```

The code for this check requires one additional `if` statement to test the length of the returned list. If the list is the same size as the graph, the code returns `False` to indicate no cycles. Otherwise, it returns `True`.

Reordering Lists

Let's consider a task that uses *topological sort*: reordering a list of items with forward dependencies. As an example, we'll use topological sort to carry out the practical task of sorting the pages of a choose-your-own-adventure book so that you never need to flip backward.

As shown in Figure 9-9, we can visualize a choose-your-own-adventure book with *N* pages as a graph with *N* nodes. Because the reader must turn most pages consecutively to follow individual storylines, the corresponding nodes for most pages have one incoming edge from the previous page and one outgoing edge to the next page. However, what makes these books exciting are the decision points. Figure 9-9 shows transitions around page *k*. The reader has two options: they can either turn to page *i* to explore the haunted house or turn to page *j* to climb the rain-soaked cliffs.

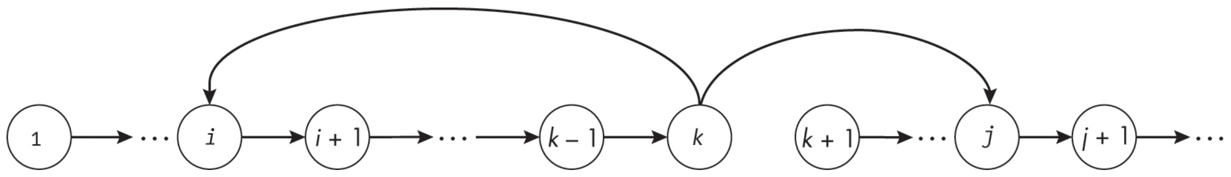


Figure 9-9: A graph representing a choose-your-own-adventure book in page order

Topological sort allows us to rearrange the pages into story order, as shown in Figure 9-10. The story starts on page 1. The narrative paths progress from left to right. They branch off at decision points, with some decisions leading to an unfortunate early end (represented by shaded nodes).

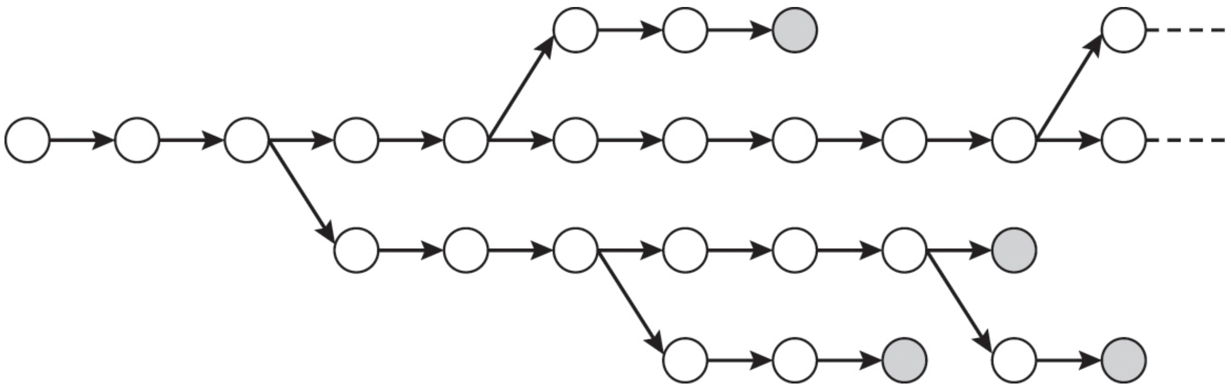


Figure 9-10: Reorganizing the book into story order

We can use the two algorithms in this chapter to do the sorting for us. As input, we take a list of lists that maps each page to its corresponding options. The list `[[1], [3, 4], [-1], [-1], [2]]` represents a five-page story with the index in the list corresponding to the current page. Page 0 leads deterministically to page 1. Page 1 has an option to proceed to page 3 or 4. Both pages 2 and 3 represent the end of the story (as represented by `-1`). Finally, page 4 leads deterministically back to page 2.

Listing 9-3 shows the code to sort the story by transforming the list into a graph and, for the purposes of this example, calling Kahn's algorithm.

```
def sort_forward_pointers(options: list) -> list:
    num_nodes: int = len(options)
    g: Graph = Graph(num_nodes)
    for current in range(num_nodes):
        for next_index in options[current]:
            if next_index != -1:
                g.insert_edge(current, next_index, 1.0)
❶ return Kahns(g)
```

Listing 9-3: Sorting forward pointers

The code creates a graph with one node for each page. It then uses one `for` loop to iterate through each page and a second `for` loop to iterate through the outgoing options for that page. It checks whether the page represents a terminal state (`next_index == -1`); if not, the code adds an edge to the following page in story order. Finally, the code calls Kahn's algorithm to perform the topological sort and returns the result ❶. (Alternatively, the code could use `topological_dfs()` from Listing 9-2.)

As an example implementation of this code, let's apply it to the 10-page adventure shown in Figure 9-11.

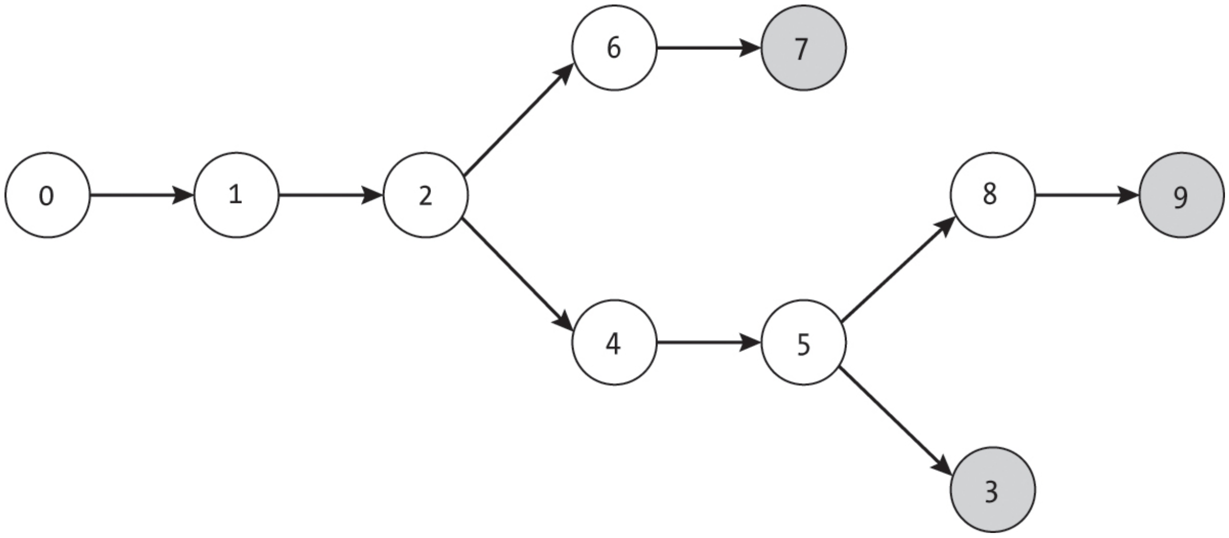


Figure 9-11: A story graph with 10 pages

We represent the options as a list of lists, with option `-1` indicating the end of a narrative line, whether it's a positive or negative conclusion:

```
[[1], [2], [4, 6], [-1], [5], [3, 8], [7], [-1], [9], [-1]]
```

When we run the input through the `sort_forward_pointers()` function from Listing 9-2, the code returns the following ordering:

```
[0, 1, 2, 6, 7, 4, 5, 8, 9, 3]
```

Comparing this result to Figure 9-11, we can see that if we were to reorder the pages to begin with page 0, then turn to pages 1, 2, 6, and so on, we would never need to flip backward while following a narrative line.

While sorting choose-your-own-adventure books might not be a typical problem you need to handle in your everyday workflow, it's easy to extrapolate from this example and apply the same techniques to other use cases. You can simply construct dependency graphs from either forward pointers (for choose-your-own-adventure books or recipes) or backward pointers (course prerequisites or code dependencies).

Why This Matters

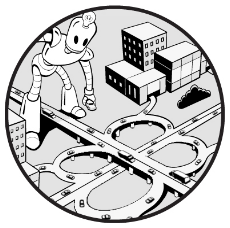
Topological sort demonstrates how to use directed edges in graphs to enforce more abstract constraints like the ordering of items. We can transform a range of dependency and ordering problems into graphs by modeling the items as nodes and the dependency between them as directed edges.

As shown throughout the chapter, topological sort has a range of real-world use cases. We often perform topological sort in our day-to-day lives without even realizing it. Before we brew coffee, we fill the kettle with water. We know the correct series of steps for this particular operation without needing to represent it as a graph, of course. However, transforming topological sort into a graph problem greatly scales up the types of problems we can solve using this algorithm. Compilers can use topological sort to determine the order in which to compile thousands of source files in a project, for example. Cloud-based workflow systems can likewise use it to determine which computation to perform next. Once you start looking for it, topological sort arises throughout both the computational and everyday domains. Knowing how to model such problems and sort the tasks is the first step in implementing efficient solutions.

The next chapter considers connectivity within graphs and how to choose a subset of edges that makes the graph fully connected. Specifically, we examine the problem of finding the minimum cost set of edges that connect all of the nodes.

10

MINIMUM SPANNING TREES



The *minimum spanning tree* of a weighted, undirected graph is the set of edges with the smallest total weight that connects all the nodes. We can use this concept to model and optimize a variety of real-world problems, from designing power grids to hypothesizing how chipmunks should be constructing their burrows.

This chapter introduces two classical algorithms for constructing minimum spanning trees. Prim's algorithm is a nodewise agglomerative algorithm that builds a bigger and bigger set of connected nodes. Kruskal's algorithm constructs a minimum spanning tree from a sorted list of edges by adding one edge at a time.

After discussing how minimum spanning trees can be applied to several real-world problems, we consider two additional algorithms closely related to minimum spanning trees: grid-based maze generation and single- linkage clustering. We show how these tasks can be mapped into graph problems and solved using variations of the algorithms from this chapter.

The Structure of Minimum Spanning Trees

A *spanning tree* of a graph is a set of edges that connects all the nodes in the graph without forming any cycles. We can visualize spanning trees as the backbone of a real-world infrastructure network—the minimum connections needed to make every node reachable from any other node. These might be power lines, roads, links in a computer network, or the tunnels between holes in a chipmunk burrow. The *minimum spanning tree* is the set of edges that connect all the nodes while minimizing the sum of the edge weights.

We can picture these requirements in terms of an especially well-organized chipmunk's burrow, as shown in Figure 10-1. The chipmunk constructs their domain as a series of holes (nodes) linked by tunnels (edges). As in a graph, each tunnel directly links exactly two holes in a straight line. The chipmunk imposes two additional requirements. First, each hole to the surface needs to be reachable through its tunnels from any other hole. After all, what good are multiple entrances if they don't let you vanish into one and pop out of another? Second, the total distances of tunnels must be minimized. The chipmunk is lazy and would prefer to expend its energy randomly popping out of the ground at various points rather than digging new tunnels.

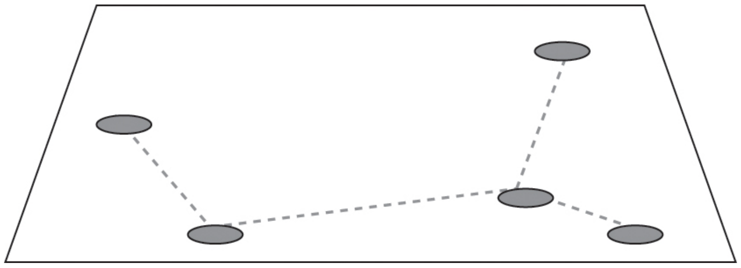


Figure 10-1: Five chipmunk holes connected as a minimum spanning tree

Formally we define the problem of finding the minimum spanning tree in a weighted, undirected graph as follows:

Given a graph with a set of nodes V and edges E, find the set of edges $E' \subseteq E$ that connects every node in V while minimizing the sum of edge weights $\sum_{e \in E'} weight(e)$.

By definition, the minimum spanning tree will have $|V| - 1$ edges, the minimum number needed to connect $|V|$ nodes. Any more edges would add cycles and unnecessary weight.

Use Cases

This section introduces a few real-world examples of using the minimum spanning tree concept to design cost-efficient physical networks or optimize communications in a social network.

Physical Networks

Minimum spanning trees are useful in determining the minimum cost set of links that we need to build to fully connect a physical network. Imagine that the Algorithmic Coffee Shop Company is looking to build a state-of-the-art pneumatic tube system for delivering beans between its locations. After promising to serve over 10,000 varieties of coffee, the company quickly realizes that it lacks the storage space in some locations to keep such a vast variety on hand. Instead, it decides to build a central warehouse and ship small packets of beans to each store as needed. Every store will now boast an unparalleled selection.

The planners quickly realize that it is prohibitively expensive to build pneumatic tubes from every store to the warehouse. The two stores in Javaville are each located over 10 miles from the distribution center, but only two blocks from each other. It is much cheaper to build a single tube from the distribution center to the Main Street location and then a second tube from Main Street to the Coffee Boulevard location. A request for the Coffee Boulevard location can be satisfied by first sending the beans to the Main Street location and then forwarding them to Coffee Boulevard.

This multistep routing turns the design of the pneumatic delivery system into a minimum spanning tree problem, as shown in Figure 10-2. Each of the Algorithmic Coffee Shop Company's buildings is a node and each potential tube between any pair is an edge.

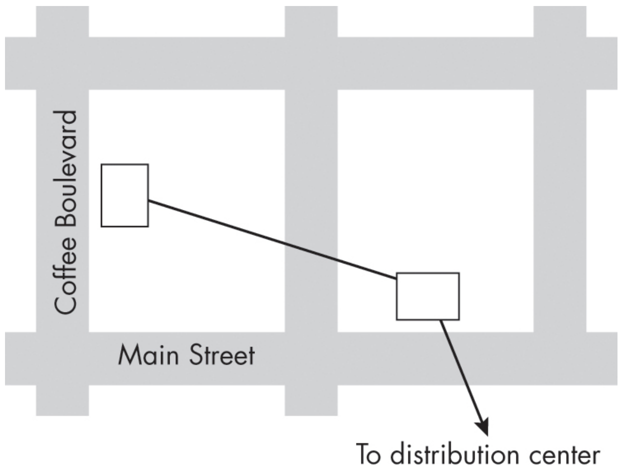


Figure 10-2: Two coffee shops on a minimum spanning tree delivery network

In Figure 10-2, the weight of an edge is the cost it would take to build the pneumatic tube between the two buildings. While often a factor of distance, the cost can also increase due to environmental factors. For example, building a tube that cuts through the center of a city is much more expensive than the same length tube under a farm. The planners need to find the set of edges (tubes to construct) that connects all the buildings while minimizing the cost.

Aside from pneumatic coffee tubes, more typical applications of minimum-cost spanning trees to physical networks include the following:

Constructing highways Nodes are cities, edges are highways, and edge weight is the cost to construct a highway between two points.

Power grids Nodes are cities, edges are transmission lines, and edge weight is the cost to construct the transmission lines between two points.

Bridging an archipelago Nodes are islands in the archipelago, edges are physical bridges between two islands, and edge weight is the cost to construct a bridge between two islands.

Design of airline networks Nodes are airports, edges are flights, and edge weight is the cost of flying between two airports.

Social Networks

Minimum spanning trees also apply to non-physical networks. For example, imagine a Society for Personal Communication Between Data Structure Experts that does not believe in bulk emails. Such announcement methods are much too impersonal. Instead, the organizers insist that each message be passed by a personal call from member to member. However, like in any organization consisting of experts, there exists a range of old friendships and feuds. Last year, Alice Hash Table had a falling out with Bob Binary Search Tree, and they no longer talk.

Every year, the organization develops an elaborate phone tree allowing the organization to spread the news of its upcoming conference while minimizing the discomfort of its members. Each member is represented as a node with edges to each other member. The cost of an edge is the level of discomfort two members have with talking to one another. In the best case, a chat among friends, the weight is minimal to represent the time cost of the phone call. However, in the worst case, the cost between two feuding members results in days of lost productivity and angry muttering. The organization needs to find the set of pairwise communications that informs every member about the conference details while minimizing overall angst. This requires all nodes to be connected using the minimum number and cost of edges.

Prim’s Algorithm

Constructing a minimum spanning tree requires an algorithm to select a minimum cost subset of the edges from the full graph such that the resulting graph is fully connected. One method of finding a graph’s minimum spanning tree is *Prim’s algorithm*, which was independently proposed by multiple people including computer scientist R.C. Prim and mathematician Vojteˇch Jarník. The algorithm operates very similarly to Dijkstra’s algorithm in Chapter 7, working through an unvisited set and building up a minimum spanning tree one node at a time.

Prim’s algorithm starts with an unvisited set of all nodes and arbitrarily chooses one to visit. This visited node forms the start of the minimum spanning tree. On each iteration, the algorithm finds the unvisited node with the minimum edge weight to *any* of the nodes that it has previously visited, asking, “Which node is closest to our set’s periphery and thus can be added with the least cost?” The algorithm removes this new node from the unvisited set and adds the corresponding edge to the minimum-cost spanning tree. It keeps adding nodes and edges, one per iteration, until it has visited every node.

Prim's algorithm will visit each node at most once and consider each edge at most twice (once from each end). Additionally, for each node, we may see a cost proportional to the logarithm of $|V|$ to insert or update a node in the priority queue implemented as a standard heap. The total cost of the algorithm therefore scales as $(|V| + |E|) \times \log(|V|)$.

We can picture Prim's algorithm as a construction company hired to upgrade bridges between islands in an archipelago. The company plans to replace the rotting wooden bridges connecting the archipelago with fully modern versions. Because the old wooden bridges will not support the weight of the construction equipment, from the company's point of view, only islands joined by a new bridge are truly connected. Their contract specifies that, in the end, any pair of islands must be reachable with a new modern bridge.

The builders start at a single island and work outward, connecting more and more islands with new bridges. At each step, they choose to upgrade the shortest wooden bridge that joins an island in the current connected set to an island outside that set. By always starting new bridges from an island in the connected set, the builders can move their equipment to the new edge's origin using modern bridges. By always ending bridges on islands outside the connected set, the builders increase the coverage of the connected set at every stage.

The Code

At each step of Prim's algorithm, we track the unconnected nodes along with the best edge weight seen that would connect them. We maintain this data using a custom `PriorityQueue` implementation that provides an efficient mechanism for looking up values in the queue and modifying priorities. For the purposes of this code, you need to understand only the basics of inserting items into the priority queue, removing items from the priority queue, and modifying priorities. However, if you're curious, you can review the details in Appendix B.

The code itself loops over the nodes in the priority queue until it is empty. Every time it removes a new node from the priority queue (the unvisited set), it examines that node's unvisited neighbors and checks whether the current node provides better (that is, lower cost) edges to any of its unconnected neighbors. If so, it updates the neighbor’s information with the new edge and weight:

```
def prims(g: Graph) -> Union[list, None]:
    pq: PriorityQueue = PriorityQueue(min_heap=True)
    last: list = [-1] * g.num_nodes
    mst_edges: list = []

  ❶ pq.enqueue(0, 0.0)
    for i in range(1, g.num_nodes):
        pq.enqueue(i, float('inf'))

  ❷ while not pq.is_empty():
        index: int = pq.dequeue()
        current: Node = g.nodes[index]

      ❸ if last[index] != -1:
            mst_edges.append(current.get_edge(last[inde
x]))
        elif index != 0:
            return None

      ❹ for edge in current.get_edge_list():
            neighbor: int = edge.to_node
            if pq.in_queue(neighbor):

                if edge.weight < pq.get_priority(neighbo
r):
                    pq.update_priority(neighbor, edge.weig
ht)
                    last[neighbor] = index

    return mst_edges
```

The code starts by creating a trio of helper data structures, including a min-heap-based priority queue of unconnected nodes (`pq`), an array indicating the last node visited before a given node (`last`), and the final set of edges for the minimum spanning tree (`mst_edges`). The code requires importing the custom `PriorityQueue` class defined in Appendix B, as well as importing `Union` from Python's `typing` library.

All nodes are inserted into the priority queue at the start of the algorithm ❶. The starting node (0) is given priority 0.0 and the rest are given infinite priority. The code then proceeds like Dijkstra's algorithm, processing the unvisited nodes one at a time. A `while` loop iterates until the priority queue of unvisited nodes is empty ❷. During each iteration, the node with the minimum distance to any of the visited nodes is chosen and dequeued from the priority queue. As we will see, this effectively removes the node from the unvisited set.

The code next checks whether there exists an edge back to one of the nodes in the connected set ❸. There are two cases in which the node's `last` entry might be `-1`. The first is node 0, which does not have a predecessor by virtue of being explored first. The second case is in a disconnected component where `index` is not reachable from node 0. In this latter case, because all the nodes cannot be connected, the graph does not have a minimum spanning tree and the function returns `None`.

After adding the new node to the visited set (by dequeuing it), a `for` loop iterates over each of the node's neighbors ❹, checking whether the neighbor is unvisited (still in the priority queue). If so, the code checks whether it has found a better edge to the node by comparing the previous best edge weight with that of the new edge. The code finishes by returning the set of edges making up the minimum spanning tree.

Note that if a graph is disconnected, each connected component has its own minimum spanning tree. An alternative approach to the code presented here is to return the list of edges that create the minimum spanning trees for each connected component. We can implement this by removing the `elif` check ❸ and its corresponding return. The code will then move on to the next component by selecting a node from the priority queue and continue selecting edges.

An Example

Figure 10-3 shows an illustration of Prim's algorithm on a graph with eight nodes. The table to the right of each subfigure shows the information tracked for each node, including the node's ID, the distance to that node from the connected set of nodes as stored by the node's priority, and the closest member of the current connected subset as stored in the `last` list. All nodes except the first one start with an infinite distance and a `last` node pointer of `-1` to indicate that we have yet to find a path that leads to that node. After removing a node from the priority queue, we gray out its row to indicate it is no longer under consideration.

The search starts at node 0 in Figure 10-3(a). This corresponds to our island bridge building company setting up operations at its headquarters on its home island. The search removes this node from the priority queue, checks each of node 0's neighbors, and updates the information accordingly. Node 1 is assigned a distance of 1.0 and node 3 a distance of 0.6. Both neighbors' `last` values now point back to node 0 as the closest node in the connected subset.

In Figure 10-3(b), the search progresses to the closest node that is not in the connected subset. This corresponds to building the first bridge between islands. The algorithm dequeues node 3 with a distance (priority) of 0.6, adds it to the connected subset, and checks its neighbors 4 and 6. These are both newly reachable via an edge from node 3. The search updates both nodes' priorities and `last` values.

The search next explores node 1 in Figure 10-3(c). While checking the neighbors of node 1, it finds a shorter edge connecting to node 4. This is equivalent to the workers noticing the old wooden bridge (1, 4) is shorter and thus cheaper to upgrade than the other wooden bridge (3, 4) that is currently slated for an upgrade. The search thus updates the distance from node 4 to 0.5 and updates its `last` pointer to node 1 to reflect the origin of the connecting edge. The search is now scheduled to use the edge from (1, 4) to join node 4 to our connected set instead of the previous edge (3, 4).

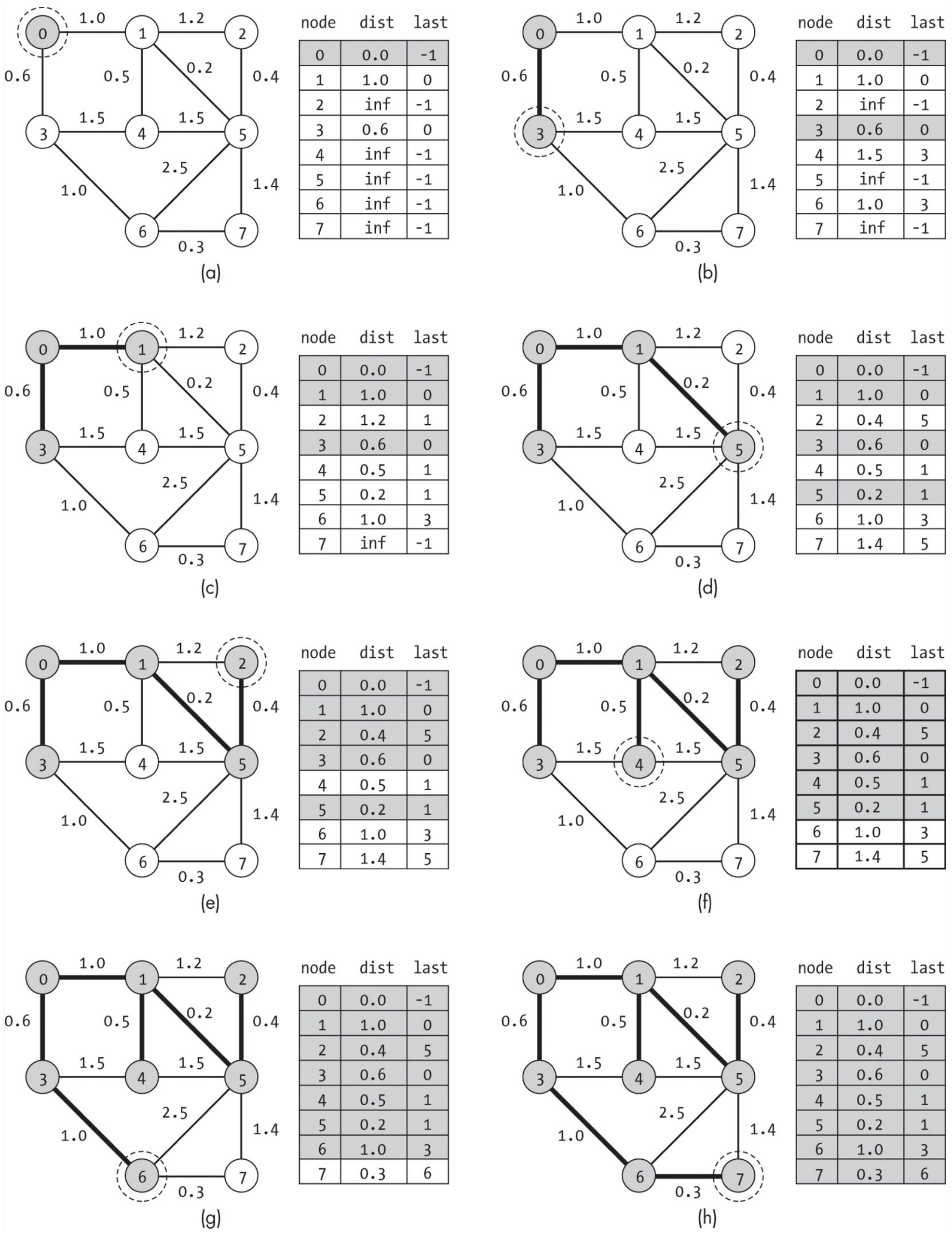


Figure 10-3: An illustration of Prim's algorithm

In the next five subfigures, the search progresses to node 5, node 2, node 4, node 6, and node 7, respectively, checking each node's unvisited neighbors and updating any for which it finds shorter edges. The size of the connected subgraph grows by one each step until all nodes are connected.

Kruskal's Algorithm

An alternative to the node-by-node approach of Prim's algorithm is to take an edge-centric approach to constructing minimum spanning trees. Kruskal's algorithm, invented by multidisciplinary scholar Joseph B. Kruskal, works by looping over a sorted list of edge weights and progressively adding edges to build the minimum spanning tree. Intuitively, we want to add the graph's smaller edges, since they are the least expensive connections between nodes. If we maintain a list of edges sorted by weight, we can proceed through it, adding the next edge that would help build the minimum spanning tree. This loop over a sorted list forms the core of Kruskal's algorithm.

Kruskal's algorithm's cost scales proportional to $|E| \log (|E|)$. The algorithm starts by extracting and sorting each edge, requiring time proportional to $|E| \log (|E|)$. Using an efficient implementation of the union-find algorithm, we can combine the sets in $|E| \log (|V|)$ time. As long as $|E| \geq |V|$, the algorithm will scale as $|E| \log (|E|)$.

We can visualize Kruskal's algorithm in the context of a pet owner building a complex living space for their beloved hamster. The hamster already has several large habitats that the owner decides to connect using clear tubes, giving their pet free range to roam between cages. The habitats' arrangement within the room is fixed. The owner, looking to minimize the total tubing needed, measures each pairwise distance between habitats, sorts the list, and determines which tube to add next. Unlike the island building example, the pet owner does not need to worry about transporting construction equipment from node to node. They can easily move between any pair of nodes to build the connection.

Union-Find

Beyond finding the next lowest-cost edge, we need to answer one additional question when considering each new edge: does this edge join nodes from currently disconnected clusters? If not, the edge is redundant. Remember that the key word here is *minimum*. If we already have edges (A, B) and (B, C), the edge (A, C) doesn't help, as node C was already reachable from node A through node B.

To efficiently implement Kruskal's algorithm, we make use of a new helper data structure, `UnionFind`. This data structure allows us to represent a collection of different sets, which we will use to track the connected components of the graph. The data structure facilitates a few efficient, set-based operations, including the following:

`are_disjoint(i, j)` Determines whether two elements **`i`** and **`j`** are in different sets. We use this to test whether two nodes are part of the same connected set.

`union_sets(i, j)` Merges the set with element **`i`** and the set with element **`j`** into a single set. We use this to connect two sets of nodes when adding an edge.

The data structure also tracks a count of the disjoint sets that is updated with each operation (`num_disjoint_sets`).

For the purposes of the algorithms in this book, it is not necessary to dive into the details of `UnionFind`. It is sufficient to treat it as a module that facilitates the operations described. Interested readers can find a basic description and the code sufficient to implement the algorithms in this book in Appendix C.

The Code

Given the helper data structure, the code for Kruskal's algorithm consists of two main steps. First, we create a list of all the graph's edges and sort it. Second, we iterate through that list by checking whether the current edge joins disconnected components and, if so, adding it to our minimum spanning tree:

```
def kruskals(g: Graph) -> Union[list, None]:
    djs: UnionFind = UnionFind(g.num_nodes)
    all_edges: list = []
    mst_edges: list = []

  ❶ for idx in range(g.num_nodes):
        for edge in g.nodes[idx].get_edge_list():
          ❷ if edge.to_node > edge.from_node:
                all_edges.append(edge)
  ❸ all_edges.sort(key=lambda edge: edge.weight)

    for edge in all_edges:
      ❹ if djs.are_disjoint(edge.to_node, edge.from_node):
            mst_edges.append(edge)
            djs.union_sets(edge.to_node, edge.from_node)

❺ if djs.num_disjoint_sets == 1:
        return mst_edges
    else:
        return None
```

The code starts by creating a series of helper data structures, including a `UnionFind` data structure representing the current disjoint sets (`djs`) to determine which points already belong to the same cluster, a list (`all_edges`) that will store the *sorted* list of edges, and an empty list (`mst_edges`) to hold the resulting edges for the minimum spanning tree. The code then loops over every node in the graph to fill these helper data structures ❶. For each node, it inserts each of the node's edges into the list of all edges.

Since our representation of an undirected graph includes the edge (A, B) in the adjacency lists for both node A and node B, the code uses a simple check to avoid adding the same edge twice ❷. (Note that this check is only needed to improve the efficiency when using this representation of an undirected graph. The code would still work correctly without the check but would include twice the number of edges in `all_edges`.)

After the full list of edges is assembled, the code sorts the edges in order of increasing weight ❸. The code iterates over each edge in the sorted list with a single `for` loop, then uses the `UnionFind` data structure to check whether the edge connects two currently unconnected components ❹. If so, the edge is useful. The code adds it to the set of edges from the minimum spanning tree (`mst_edges`) and merges the disjoint sets for the edge's nodes.

Finally, the code checks whether it was able to connect all the nodes into a single connected component ❺. If so, it returns the list of edges for the minimum spanning tree. Otherwise, it returns `None`. If we remove this final check, the code will instead return the edges from the individual minimum spanning trees for graphs that are not a single connected component.

An Example

Figure 10-4 shows an example of Kruskal’s algorithm running on a graph with 8 nodes and 12 edges.

The search begins with an empty set of edges and thus a disconnected set of nodes. In Figure 10-4(a), the search selects the edge with the lowest weight from our graph. This corresponds to the edge (1, 5) with a weight of 0.2. The edge in the figure is marked in bold to indicate it is part of the minimum-cost spanning tree. Nodes 1 and 5 are now part of the same connected subset, and the search has reduced the number of disjoint sets from eight to seven.

The search continues in Figure 10-4(b) by choosing the edge with the next lowest weight. This time it connects nodes 6 and 7 through an edge with weight 0.3. It has reduced the number of disjoint sets to six.

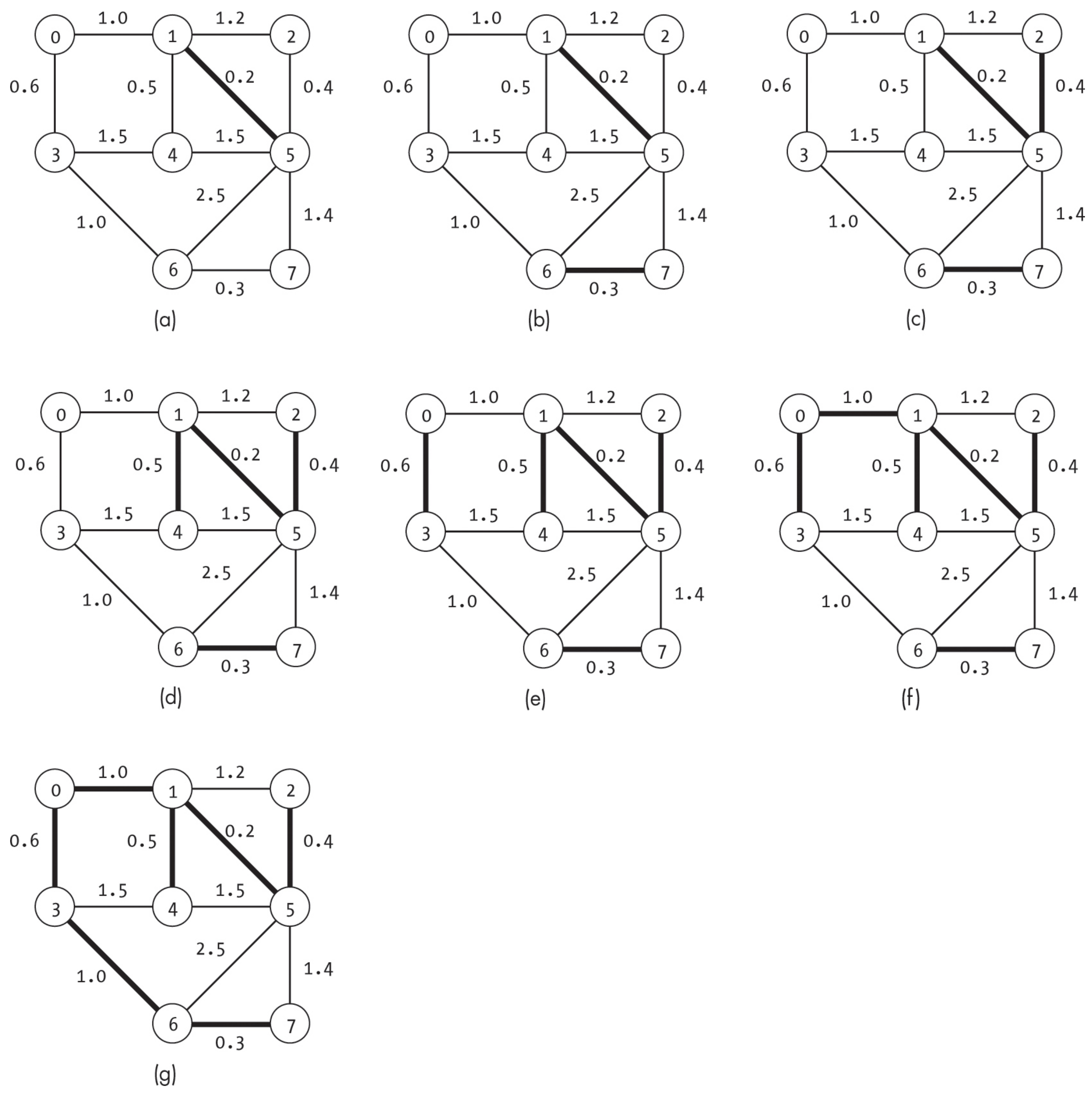


Figure 10-4: An illustration of Kruskal's algorithm

In the next two subfigures, the search adds nodes 2 and 4 to the first connected subset {1, 5}, resulting in a connected set consisting of {1, 2, 4, 5}. In Figure 10-4(e), the algorithm merges another two singleton nodes by joining nodes 0 and 3 via the edge with weight 0.6. It then joins up the remaining three disjoint sets by adding the edges (0, 1) and (3, 6) in the following two subfigures. At this point, we are down to a single set, which means our minimum-cost spanning tree edges connect all the nodes in the graph.

Maze Generation

While the graph searches presented in preceding chapters allow us to algorithmically solve mazes, they cannot help us generate mazes in the first place. In this section, we take a detour from the more canonical uses of minimum spanning tree algorithms, such as building transportation networks, to show how we can extend Kruskal's algorithm to create random but always solvable mazes. To make the mazes sufficiently fun, we ensure that each has exactly one valid solution.

Imagine we are given the task of generating a maze for the children's place mat at a local family restaurant. Our design can be simple but must be solvable, with only one path through the maze. The restaurant owners wisely do not want to challenge young patrons with impossible mazes, lest this results in screaming and thrown food.

Representing Grid-Based Mazes

For simplicity of the code in this section, we represent our mazes using a regular grid of squares like the ones on graph paper. After hours of careful consideration about how to draw our mazes, we decide to shade individual edges to represent the maze's walls. The player can move between any two adjacent squares that do not have a wall between them. As we draw each line, we eliminate a potential option for leaving that square and perhaps chuckle at the difficult task we are creating.

Figure 10-5(a) shows an example grid-based maze. We can equivalently represent this grid structure using a graph, as shown in Figure 10-5(b).

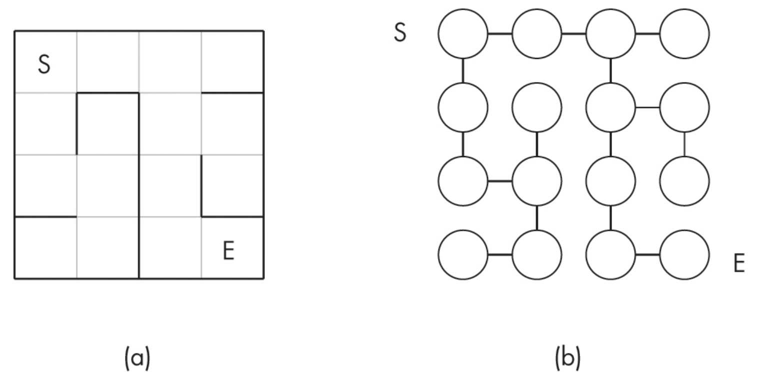


Figure 10-5: A grid-based maze and its graph representation

In Figure 10-5(b), each square in the maze corresponds to a single graph node. We add undirected edges between any two adjacent nodes without a wall so that an edge indicates the ability to travel from one node to another.

Generating Mazes

We construct our maze by starting with a grid-based graph and building a randomized spanning tree algorithm based on Kruskal's algorithm to connect all the nodes. The grid-based initial structure gives us connections based on adjacency. Each node has up to four connections to the nodes above, below, left, and right of it. Generating a spanning tree allows us to ensure that each node is reachable from any other node and that we can reach the ending node from the starting one.

We define the valid edges using a connected grid-based graph, as shown in Figure 10-6. Like the grids we generated in Chapter 5, this graph represents all the nodes we need to connect and the set of potential edges we can use to connect them. If our grid has a width of w and a height of h, it contains $h \times w$ nodes and undirected edges (with equal weights of 1) connecting neighboring nodes.

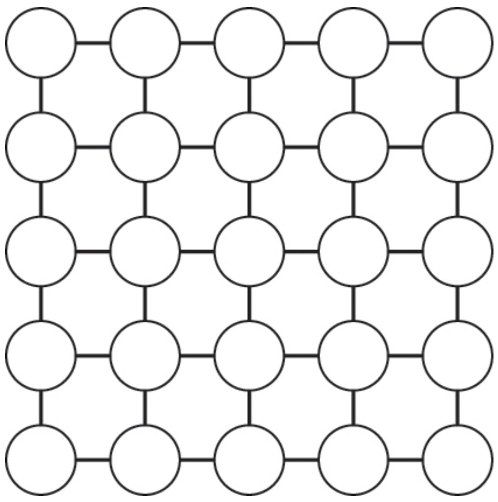


Figure 10-6: A grid-based graph

If we used the graph in Figure 10-6 for our final maze, there would be a huge number of potential paths between any two locations. In other words, the graph does not make for a particularly fun or challenging maze. Beginning at the start node, we could traverse directly to the end node by moving the minimum number of steps in one horizontal and one vertical direction. Real-world equivalents would be a hedge maze implemented as an empty lawn or a blank maze on our place mat. To construct an interesting maze, we need to use a minimum subset of these edges.

As in Kruskal's algorithm, we start with an empty spanning tree where none of the nodes are connected. In the case of our grid-based place mat, we start with a grid of boxes. One by one, we add edges to our spanning tree and erase the lines between adjacent boxes. We can alternatively visualize the connection of two components as a cartoon figure removing a physical wall between two adjacent rooms by using an oversized sledgehammer or simply bursting through the wall. As our cartoon character gleefully opens up passageways (or we carefully erase grid lines), the disparate components connect and a path through the maze forms.

The key to generating a random maze is, intuitively, to choose the next edge randomly. Both Kruskal's and Prim's algorithms rely on some method to break ties among equal-weight edges. In this case, however, all edges have the same edge weight (1.0), so we can just pick one at random. If the chosen edge connects two disjoint components, we keep it. This edge opens a path between two previously unreachable components. Otherwise, in the case where the chosen edge joins two already connected components, we discard it, since adding multiple paths between components would result in loops and break the maze convention of having a single path.

The Code

The following code allows us to randomly create a set of maze edges from a grid-based graph:

```
def randomized_kruskals(g: Graph) -> list:
  ❶ djs: UnionFind = UnionFind(g.num_nodes)
    all_edges: list = []
    maze_edges: list = []

  ❷ for idx in range(g.num_nodes):
        for edge in g.nodes[idx].get_edge_list():
            if edge.to_node > edge.from_node:
                all_edges.append(edge)
  ❸ while djs.num_disjoint_sets > 1:
        num_edges: int = len(all_edges)
      ❹ edge_ind: int = random.randint(0, num_edges - 1)
        new_edge: Edge = all_edges.pop(edge_ind)

      ❺ if djs.are_disjoint(new_edge.to_node, new_edge.fro
m_node):
            maze_edges.append(new_edge)
            djs.union_sets(new_edge.to_node, new_edge.from
_node)

    return maze_edges
```

The function takes a full grid-based graph (`g`) to define the list of edges. The code starts by setting up helper data structures, including a `UnionFind` data structure representing the disjoint sets (`djs`), a list of all edges (`all_edges`), and a list of the maze or spanning tree edges (`maze_edges`) ❶. As in Kruskal's algorithm, the code extracts the comprehensive list of edges from the graph ❷.

The algorithm iterates through a single `while` loop until all nodes are the same set (and thus reachable) ❸. During each iteration of the loop, the algorithm selects an edge randomly ❹, using Python's `random` library's `randint()` function (which requires us to include `import random` at the start of the file). It then removes the selected edge from the list of all edges and checks whether it joins two previously disjoint sets ❺. If so, the edge is added to the list of maze edges and the corresponding sets are merged. Otherwise, the edge is ignored. The algorithm completes after all the nodes are merged into a single set, returning the list of edges that defines the maze: the minimum spanning tree.

An Example

Figure 10-7 shows an example of the first few steps of this algorithm. The left diagram of each subfigure shows the current maze as defined by the walls that have been removed, while the right diagram shows the maze as defined by edges that have been added to a graph. During each step (each iteration of the `while` loop), one edge at most is added.

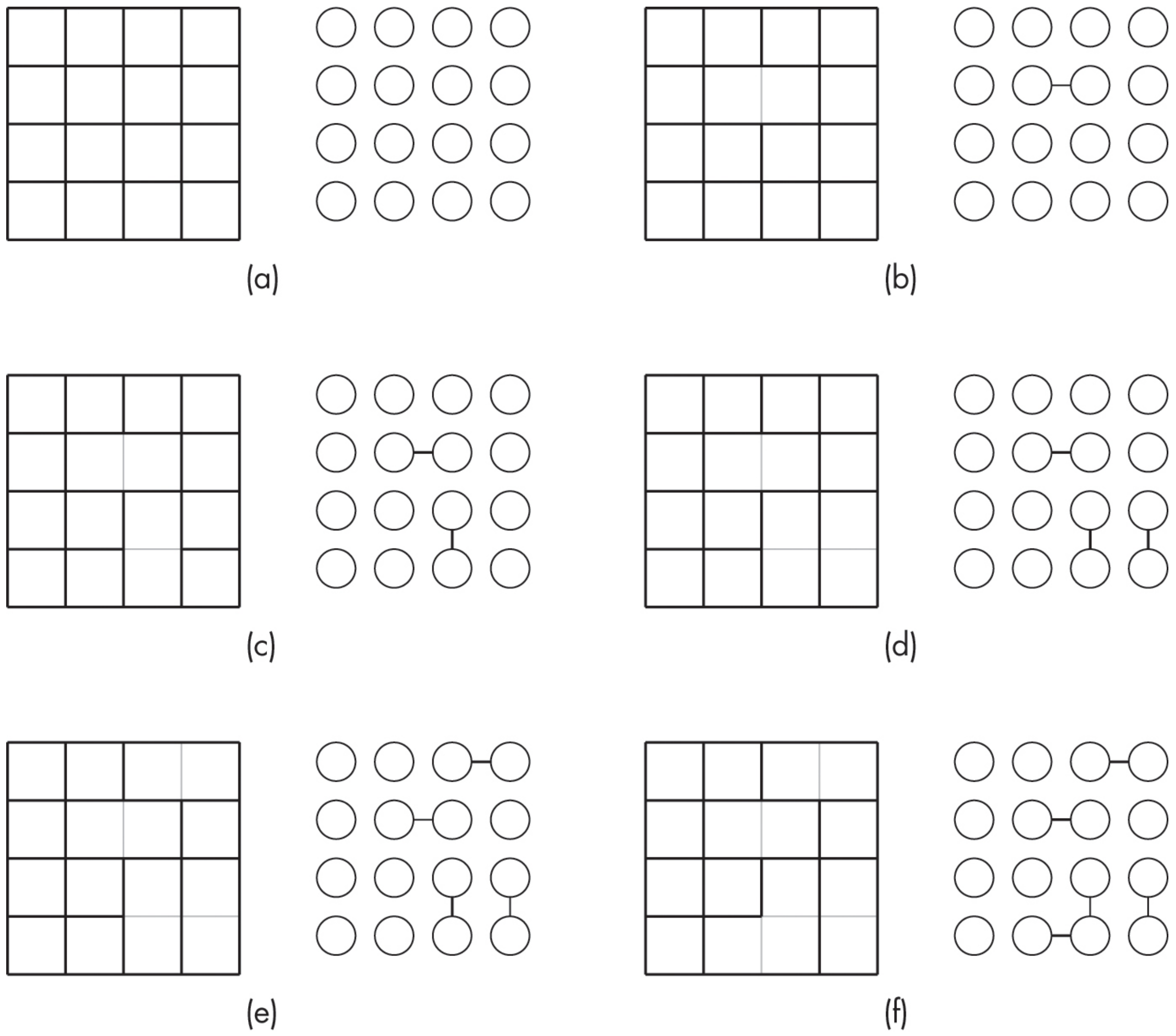


Figure 10-7: Six steps of the maze construction algorithm

It is not strictly necessary to construct the full grid-based graph (`g`) ahead of time. Instead, we could just programmatically fill the `all_edges` list based on computed adjacencies, as we did when constructing grids in Chapter 5, for example. However, for the purposes of this chapter, starting with the full grid-based graph makes the code's connection to Kruskal's algorithm more apparent and keeps the function simpler.

The randomized Kruskal's algorithm is a simplistic approach to generating mazes that makes no guarantee that the ending node is at the end of a deep path with a bunch of turns. It may result in quite boring mazes such as the ones shown in Figures 10-8(a), 10-8(b), and 10-8(c). We can only be sure that the algorithm will *not* produce a maze where the end is unreachable, such as the one shown in Figure 10-8(d).

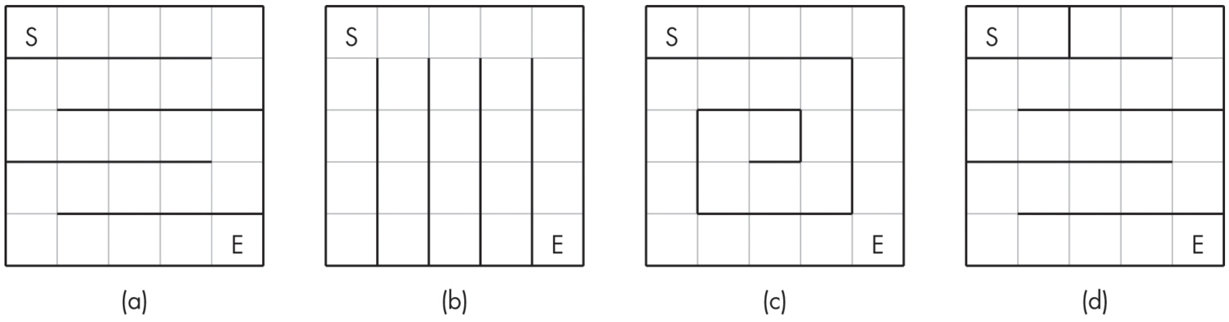


Figure 10-8: Three overly simple mazes and one unsolvable maze

Beyond the exciting commercial opportunities involved in designing children's place mats, the maze-generation algorithm in this section shows how we can extend the basic components of minimum spanning trees and Kruskal's algorithm. Further, it demonstrates how randomization can be used within an algorithm to create different spanning trees.

Single-Linkage Hierarchical Clustering

We can also adapt Kruskal's algorithm to handle the seemingly different problem of clustering spatial points. *Clustering* is a common unsupervised data-mining and machine-learning approach that assigns data points to clusters such that the points within each cluster are similar (for some given definition of similar). For example, we might cluster cafés based on geographic proximity so that all the coffee shops in Anchorage are placed together in one cluster, while cafés in Honolulu are placed in another. The resulting clusters provide a partitioning of data points that can help us discover structure in the data or classify similar data points.

There is a wide range of clustering techniques that vary in how they define similar points and how points are assigned to clusters. As its name implies, hierarchical clustering is an approach that creates a hierarchy of clusters by merging two "nearby" clusters at each level of the hierarchy. Each data point initially defines its own cluster; these clusters are iteratively joined until all the points are part of the same cluster. Even within hierarchical clustering, there are various approaches to determining which clusters to merge, including the following:

- Computing the mean position over each cluster's points and merging the clusters with the closest centers
- Finding the farthest of any pair of points from two clusters and merging the clusters whose maximum distance is the smallest
- Finding the closest pair of points from two clusters and merging the clusters whose minimum distance is the smallest

This section focuses on the last approach, called *single-linkage clustering*, which joins the two clusters with the closest pair of individual points. We present an algorithm to implement it that is nearly identical to Kruskal's algorithm on graphs.

Figure 10-9 shows an example of single-linkage clustering. The left-hand figure shows the five two-dimensional points (0, 0), (1, 0), (1.2, 1), (1.8, 1), and (0.5, 1.5). The right-hand figure shows the hierarchical clustering.

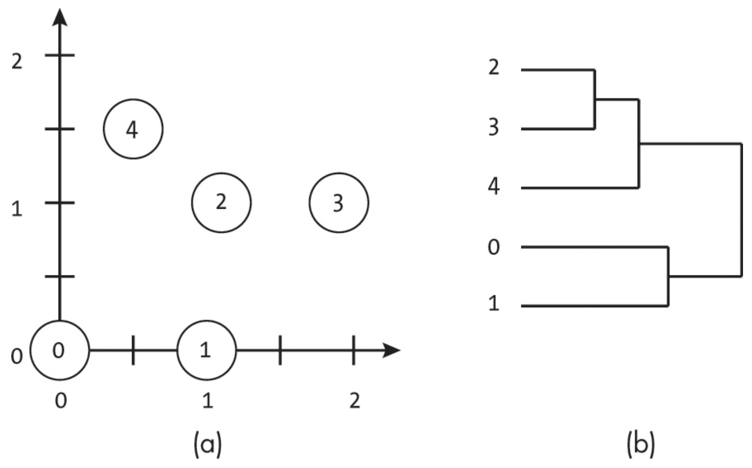


Figure 10-9: A set of two-dimensional points (left) and the corresponding single-linkage clustering (right)

We start with each point in its own cluster and create a merged cluster from the two individual points with the closest distance (points 2 and 3). Next, we merge the cluster {2, 3} with {4} because points 2 and 4 have the smallest distance of any pair of points in different clusters. This process continues as shown in the right-hand side of Figure 10-9.

The advantage of hierarchical clustering is that it provides an easily visualized and interpretable structure. We can use this structure to dynamically change the number of clusters (level of partitioning) by walking up the hierarchy until we hit a given distance threshold. Points that have been joined together before we hit the threshold are clustered together, while clusters that have not been merged remain disjoint.

The Code

To simplify the logic of the clustering code, we define two small helper classes that store information about the points and the resulting clustering links. First, to represent the two-dimensional points we are clustering, we define a `Point` class to store the coordinates and compute pairwise distances:

```
class Point:
    def __init__(self, x: float, y: float):
        self.x: float = x
        self.y: float = y

    def distance(self, b) -> float:
        diff_x: float = (self.x - b.x)
        diff_y: float = (self.y - b.y)
        dist: float = math.sqrt(diff_x*diff_x + diff_y*dif
f_y)
        return dist
```

The `distance()` function computes the Euclidean distance in two-dimensional space and requires us to include `import math` at the start of the file in order to use the `math` library's square root function. (Appendix A further discusses creating graphs from spatial points, including the use of alternative distance functions.)

Second, since we are not using an explicit graph, we also define a `Link` data structure to hold the connection between points in the same cluster:

```
class Link:
    def __init__(self, dist: float, id1: int, id2: int):
        self.dist: float = dist
        self.id1: int = id1
        self.id2: int = id2
```

This data structure is effectively identical to an undirected graph edge. It stores a pair of identifiers for the points and the distance (weight) between them. We define it here as an independent data structure to highlight the fact that we do not need to explicitly build a graph for single-linkage clustering.

Using these two helper data structures, we can then implement the single-linkage hierarchical clustering algorithm using an approach based on Kruskal's algorithm:

```
def single_linkage_clustering(points: list) -> list:
    num_pts: int = len(points)
    djs: UnionFind = UnionFind(num_pts)
    all_links: list = []
    cluster_links: list = []

  ❶ for id1 in range(num_pts):
        for id2 in range(id1 + 1, num_pts):
            dist = points[id1].distance(points[id2])
            all_links.append(Link(dist, id1, id2))

  ❷ all_links.sort(key=lambda link: link.dist)

    for x in all_links:
      ❸ if djs.are_disjoint(x.id1, x.id2):
            cluster_links.append(x)
            djs.union_sets(x.id1, x.id2)

    return cluster_links
```

The code takes a list of `Point` objects (`points`) to cluster. The function starts by creating a series of helper data structures, including a `UnionFind` data structure representing the disjoint sets (`djs`) to determine which points already belong to the same cluster, an empty list (`all_links`) to hold all pairwise distances, and an empty list (`cluster_links`) to hold the `Link` objects representing each merge. The code then uses a nested pair of `for` loops to iterate through all pairs of points ❶. For each pair, the code computes the distance using the points' `distance` function and creates a `Link` data structure to hold this distance information. After all the pairwise distances are computed, the code sorts the links in order of increasing distance ❷.

Next, another `for` loop iterates over each edge in the sorted list, using the `UnionFind` data structure to check whether the next pair of points is already in the same cluster ❸. If not, the program adds the link to `cluster_links`, joining the two clusters that contain those points, and merges the disjoint sets for the points.

Finally, the code returns the list of `Link` objects representing the clustering. Each `Link` represents a connection between two previously disjoint clusters. The links in `cluster_links` will be ordered by increasing distance, so the first element represents the first two points merged.

An Example

Figure 10-10 shows the steps of our clustering algorithm on the points from Figure 10-9. The left column of the figure shows the current clusters as connected graph components of the two-dimensional points. The right column of the figure shows the same clusters as merged points in the hierarchy with each cluster represented as a circle.

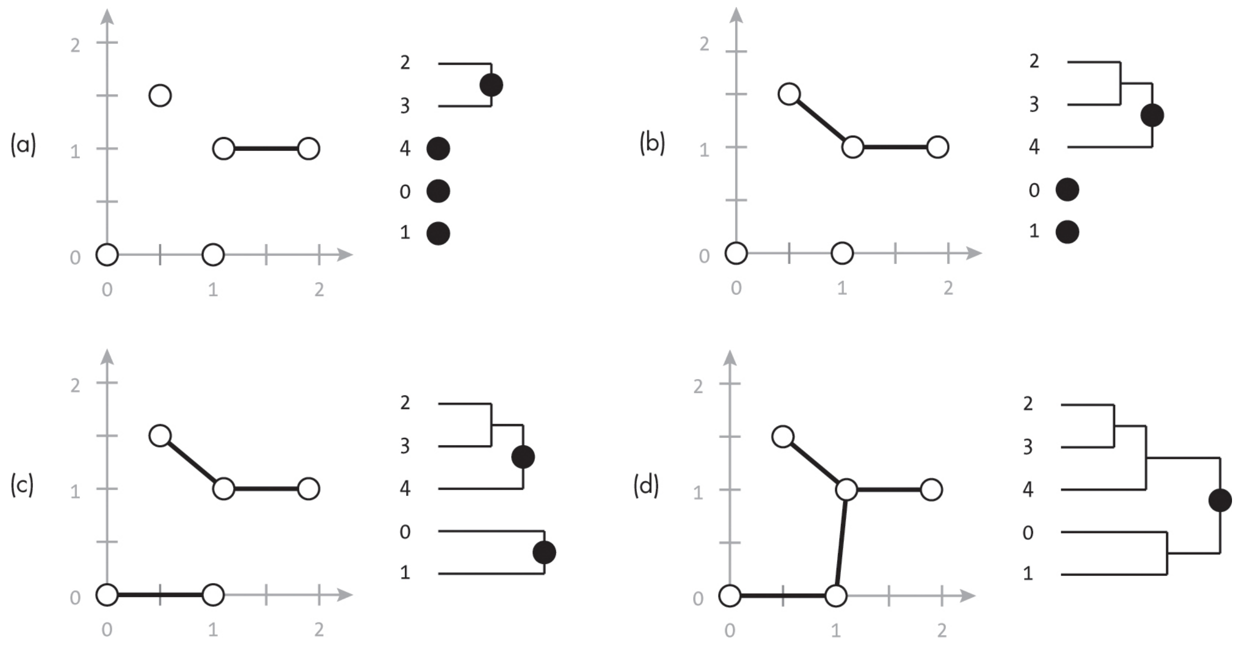


Figure 10-10: Single-linkage clustering

In Figure 10-10(a), the algorithm has joined the closest two points—those at (1.2, 1) and (1.8, 1)—into a single cluster. Using the points' labels from Figure 10-9, we call these points 2 and 3, respectively.

In the next step, in Figure 10-10(b), the algorithm joins the two clusters with the closest pair of points. At this stage, the closest points are (1.2, 1) and (0.5, 1.5) with a distance of approximately 0.86. Since (1.2, 1) is already part of a cluster, the algorithm merges the entire cluster with the one containing the single point (0.5, 1.5). The resulting cluster contains three points {2, 3, 4}.

The algorithm continues in Figure 10-10(c) by creating a new merged cluster from the two remaining individual points (0, 0) and (1, 0). The algorithm has now created two separate clusters with three and two points, respectively. During the final step, in Figure 10-10(d), these two clusters are merged by adding a link between the closest pair of points from each cluster (1.2, 1) and (1, 0).

Since single-linkage clustering grows the clusters by linking increasingly distant pairs of points, we can use this distance as a stopping threshold for the algorithm. For example, if we set the maximum distance to 0.95, we would produce the three distinct clusters shown in Figure 10-10(b).

Why This Matters

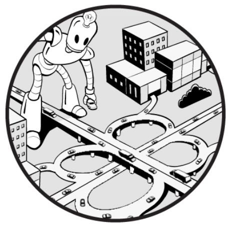
The minimum spanning tree problem allows us to solve a range of real-world optimization problems, from building roads to designing communication networks. Within the field of computer science, we can use minimum spanning trees to help solve a range of problems in networking, clustering, and analysis of biological data. For example, we can represent a communication network as a graph and find the minimum spanning tree to inform which links need to be upgraded to ensure that all nodes are reachable through the new technology.

We can also apply the same basic approach to problems we might not normally think of as graph based. Using a variation of Kruskal's algorithm, we can search for structure in real-valued datasets by building clusters of similar data points or design solvable mazes by introducing randomization into the algorithm to create novel solutions. In single-linkage clustering, we use the distances to determine which points are similar.

The next chapter expands on this discussion, introducing algorithms that help us identify the nodes and edges that are essential to maintaining connectivity.

11

BRIDGES AND ARTICULATION POINTS



In this chapter we consider another aspect of connectivity: nodes and edges that are essential to maintaining the integrity of a connected component in an undirected graph. These are known as *articulation points* and *bridges*, respectively. Understanding which nodes or edges are essential to maintaining connectivity is important in a range of real-world problems. Any time we must ensure that there is no single point of failure in a network, we need to find its bridges and articulation points.

After formally defining bridges and articulation points, this chapter provides a few demonstrative real-world use cases where these concepts apply, such as developing a robust transportation network for a set of islands and building the best secret labyrinth for an evil wizard. We then present two algorithms to efficiently search for these elements in undirected graphs, building on the depth-first search algorithm introduced in Chapter 4.

Defining Bridges and Articulation Points

For every pair of nodes in an undirected graph to be mutually reachable, they must be part of the same connected component. In Chapter 3, we learned that a connected component in an undirected graph is a subset of nodes $V' \subseteq V$ such that u is reachable from v for all pairs $u \in V'$ and $v \in V'$. As a concrete example, consider a series of islands joined by ferries. Nodes represent islands and edges represent the ferry routes between them. To provide full travel options, the transportation planners need the resulting graph to consist of a single connected component. That is, a person must be able to travel between any two islands on the ferry network, whether by a direct connection or by a series of trips.

Figure 11-1 shows an example graph with two separate connected components {0, 1, 2, 4, 5} and {3, 6, 7}.

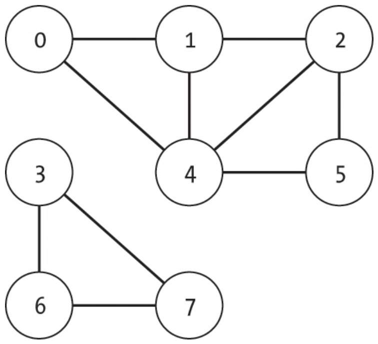


Figure 11-1: A graph with two separate connected components

A *bridge* is an edge whose removal splits a single connected component into two disjoint components. Figure 11-2(a) shows an example graph with two bridges (1, 2) and (4, 5). Removing either edge would split the single connected component into two. Removing both would split the graph into three separate connected components, as shown in Figure 11-2(b).

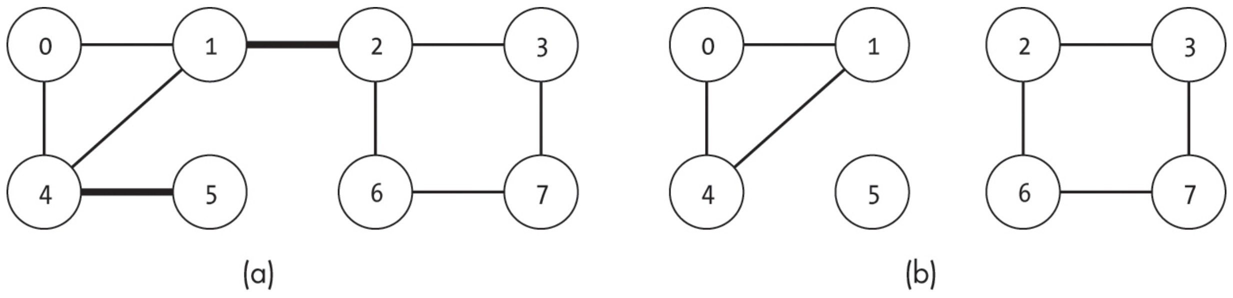


Figure 11-2: A graph with two bridges (a) and the three separate components that arise from removing the bridges (b)

Similarly, an *articulation point* (or *cut vertex*) is a node whose removal splits a connected component into two or more disjoint components. For example, the graph in Figure 11-3 has three articulation points: the shaded nodes 1, 2, and 4.

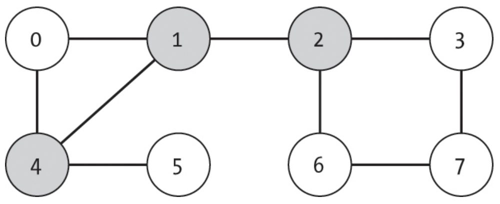


Figure 11-3: A graph with three articulation points

Figure 11-4 shows the impact of individually removing each of the articulation points in Figure 11-3.

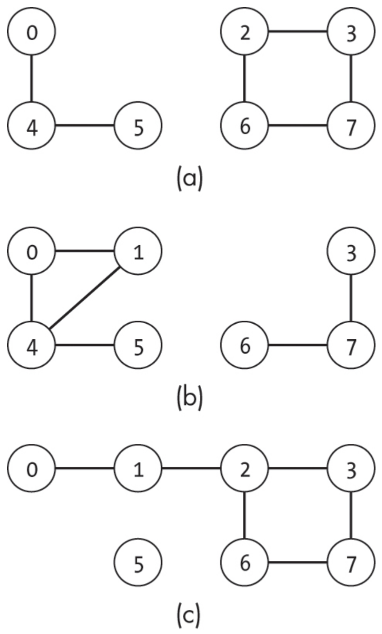


Figure 11-4: The results of removing different articulation points

In Figure 11-4(a), removing node 1 creates the components {0, 4, 5} and {2, 3, 6, 7}. Figure 11-4(b) shows that removing node 2 creates the components {0, 1, 4, 5} and {3, 6, 7}, while removing node 4 would create components {0, 1, 2, 3, 6, 7} and {5}, as shown in Figure 11-4(c).

Use Cases

Identifying the bridges and articulation points in a graph is essential for understanding single points of failure in a network. This section provides some real-world applications for finding bridges and articulation points. We first show how to apply these concepts to create a resilient ferry network, then examine how to extend the same techniques to prevent the spread of disease or construct optimal magical labyrinths.

Designing Resilient Networks

A *resilient network* needs to be able to gracefully handle the loss of an individual edge or node without losing connectivity. To expand the island example from the previous section, let's consider two hypothetical ferry networks among eight of the Hawaiian islands, as shown in Figures 11-5 and 11-6.

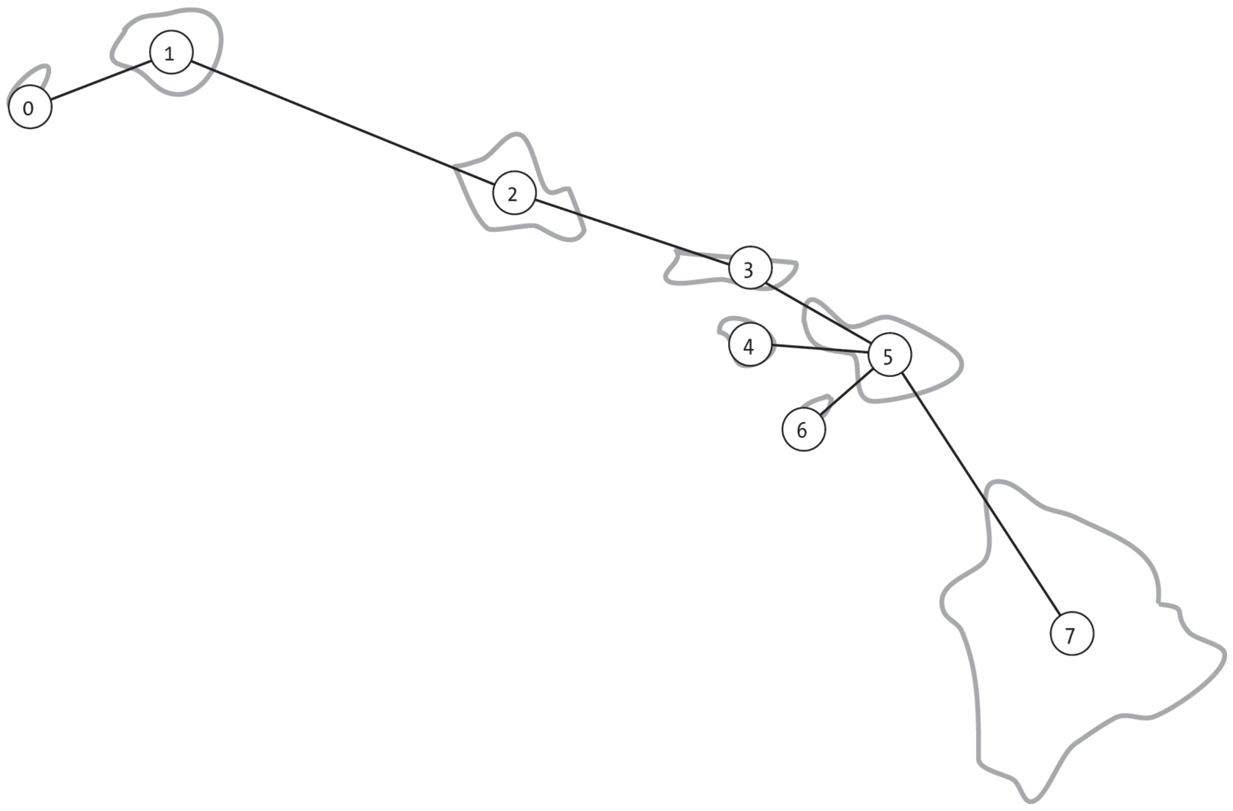


Figure 11-5: A map of hypothetical ferry routes among the Hawaiian islands

Figure 11-5 shows a minimal ferry network needed to connect the eight islands. If all ferries are running without problems, it is possible to travel between any two islands. It might take multiple hops for someone to reach their destination, but there will be a path.

However, the network is fragile. If the ferry between O‘ahu (node 2) and Moloka‘i (node 3) breaks down, it splits the network in two. People can no longer travel from Maui (node 5) and Ni‘ihau (node 0). Each ferry route in the graph is a bridge. The loss of any single route would disconnect at least one island. Similarly, many of the nodes in Figure 11-5 are articulation points. If the ferry terminal in O‘ahu (node 2) is closed due to weather, it will disconnect Kaua‘i (node 1) from Maui (node 5).

By understanding their network's bridges and articulation points, planners could design a more robust network with no bridges, as shown in Figure 11-6. A single broken ferry (removal of an edge) won't cut off travel between any two islands. If the ferry between O‘ahu (node 2) and Moloka‘i (node 3) breaks down, for example, a traveler could still make their way from Maui (node 5) to Ni‘ihau (node 0) via other routes. The network also lacks articulation points. If the ferry terminal in Maui closes, for instance, it cuts off only that island.

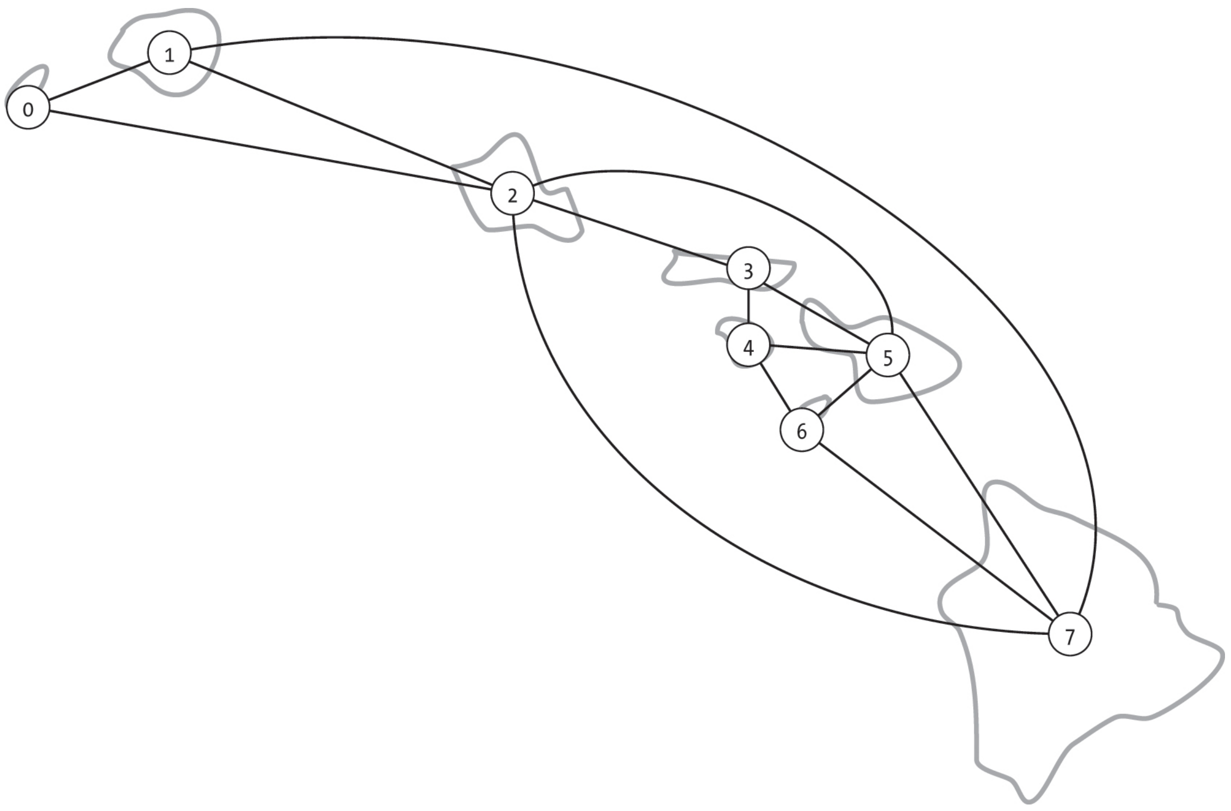


Figure 11-6: A second map of hypothetical ferry routes among the Hawaiian islands

We can extend these concepts beyond transportation systems to computer networks, power grids, communication networks, or wastewater systems. While it's often preferable to construct graphs without bridges or articulation points, it's not always feasible. However, understanding a network's weaknesses may still be helpful for planning purposes.

Preventing the Spread of Diseases

Consider how a common cold migrates through a social network. For simplicity's sake, let's assume you need to be in proximity to a sick person to catch the cold. You cannot catch the cold from someone you never see. If the edges represent real-world interactions between people, the virus can pass only between neighboring nodes.

We can use the concept of bridges and articulation points to model or stop the spread of disease. That coffee meeting with a former workmate acts as a bridge that allows the cold to jump between two otherwise disjoint sets of people: your previous and your current coworkers. A person who self-isolates and cuts off the spread of the virus between groups is an articulation point. By not going to any events for a few weeks, you can help prevent a cold from passing between your different social circles. Your running friends, the members of your data structures reading group, and your coworkers will each be limited to their own colds without sharing any through you.

Designing Magical Labyrinths

In contrast to the previous two cases, where we want to minimize the bridges and articulation points, imagine an evil wizard deciding where to place the most effective traps in their labyrinth. A tunnel that serves as the only connection between two sections of the labyrinth is a bridge. If one section contains the labyrinth entrance and the other contains the goal, the wizard knows that thorough adventurers must pass through the tunnel, making it a great place for the best trap. Similarly, a room that must be traversed to move between two parts of the labyrinth is an articulation point—an ideal place to deploy high-level monsters.

In more common settings, we can use these same techniques to place tollbooths on critical highways (bridges) or information booths at the intersection of airport terminals (articulation points). In these cases, we use the fact that people traveling from one part of the graph to another must pass through this single node or edge. Understanding the connectivity of the graph thus lets us optimize potentially scarce or expensive resources.

A Bridge-Finding Algorithm

The computer scientist Robert Tarjan proposed a range of useful algorithms for understanding graphs using the properties of their depth-first search trees. This section introduces a *bridge-finding algorithm* on undirected graphs that uses this approach. The algorithm starts a depth-first search from an arbitrary node and tracks both the edges used and the order in which the nodes are first visited (the *order index* or *preorder index*, denoted *order*(u)). We can use this information to look for bridges by asking whether an edge in the depth-first search tree provides the only path to reach the nodes in its subtree.

Edges that do not appear in the depth-first search tree *T* can be immediately ruled out as bridges because we were already able to reach the nodes without using them. This means we need to consider only the edges in *T*.

Figure 11-7 shows an example of a graph and two representations of its corresponding depth-first search tree rooted at node 0. Figure 11-7(a) shows the initial graph. Figure 11-7(b) shows the corresponding depth-first search tree when starting from node 0; the number outside each node indicates the order index. Figure 11-7(c) shows the same tree with the *untraversed edges* as dashed lines. These untraversed edges are called *back edges* and lead back to a node that has already been visited during the depth-first search.

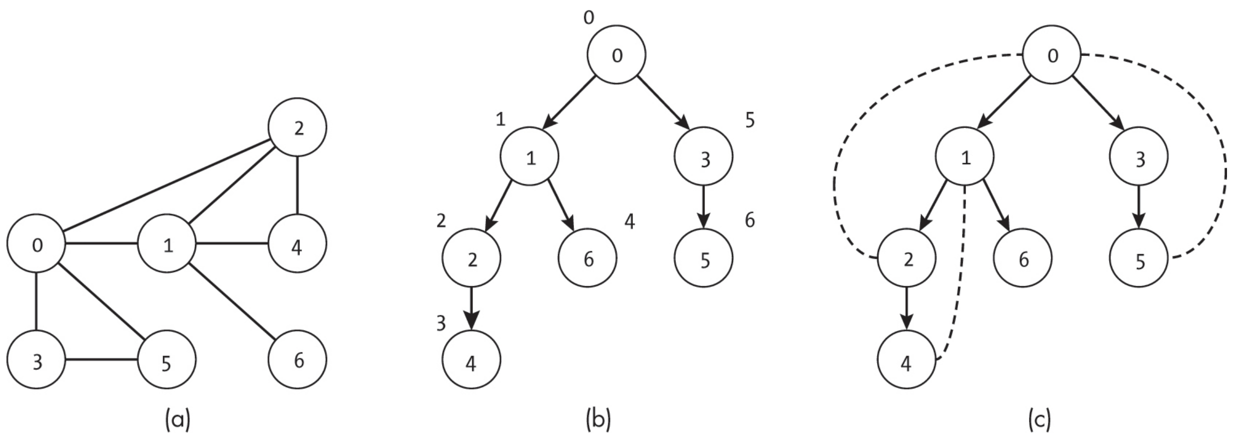


Figure 11-7: An undirected graph (a), a depth-first search tree (b), and the depth-first search tree with untraversed edges (c)

We can identify bridges by looking for edges leading into a subtree of *T*, where that subtree's nodes only have neighbors in the same subtree. In other words, if the edge (*v*, *u*) is a bridge, there is no way to get into or out of the subtree of node *u* except through the edge into (*v*, *u*). Edge (1, 6) in Figure 11-7(a) presents one such example, providing the only path into or out of the subtree rooted at node 6. In contrast, the edge (0, 3) is not a bridge, because node 5 has an edge back to node 0.

The key to this algorithm is that we can look at the minimum and maximum order index in the neighborhoods of *u* and its descendants. By the properties of depth-first search, all the nodes in *u*'s subtree must have an order index in the range [*order*(*u*), *order*(*u*) + *K* − 1], where *K* is the number of nodes in the subtree (including *u*). This is because the search travels to those nodes after visiting *u* and before visiting nodes in other subtrees. If the nodes in *u*’s subtree have any neighbor with an order index outside that range, the edge to that neighbor would provide an alternate path into *u*’s subtree.

We can use a common simplification by observing that any unvisited nodes reachable from the subtree would be explored by the depth-first search and thus would appear in the subtree. Therefore, we need to check only for back edges to neighbors with a lower order index. We can test whether edge (*v*, *u*), where *v* is the parent of *u*, is a bridge by checking if any node in *u*’s subtree has a neighbor *w* such that *order*(*w*) < *order*(*u*), excluding the connection (*v*, *u*) itself. If there is such a neighbor, we have found a back edge that bypasses (*v*, *u*) and know that (*v*, *u*) is not a bridge. Conversely, if *order*(*w*) ≥ *order*(*u*) for all the subtree’s neighbors *w* when excluding the connection (*v*, *u*), then (*v*, *u*) is a bridge.

Node 2 in Figure 11-7 provides an example of this case. The search reached node 2 via the edge (1, 2) and assigned it an order index of 2, as shown in Figure 11-7(b). For edge (1, 2) to be a bridge, there must be no alternate path out of that subtree. However, node 2 itself has an edge to node 0 (with order = 0), providing such an alternate route.

The opposite case is shown in Figure 11-8 with edge (0, 1). The graph in Figure 11-8(a) has a slight modification from the one in Figure 11-7(a), the exclusion of edge (0, 2), which results in edge (0, 1) now being a bridge. Figure 11-8(b) shows the corresponding depth-first search subtree rooted at node 0, with the untraversed edges in gray. The dotted ovals in both figures indicate the subtree of node 1. As shown in Figure 11-8(b), the only connection from the subtree of node 1 to a node with order less than 1 is the edge (0, 1) itself.

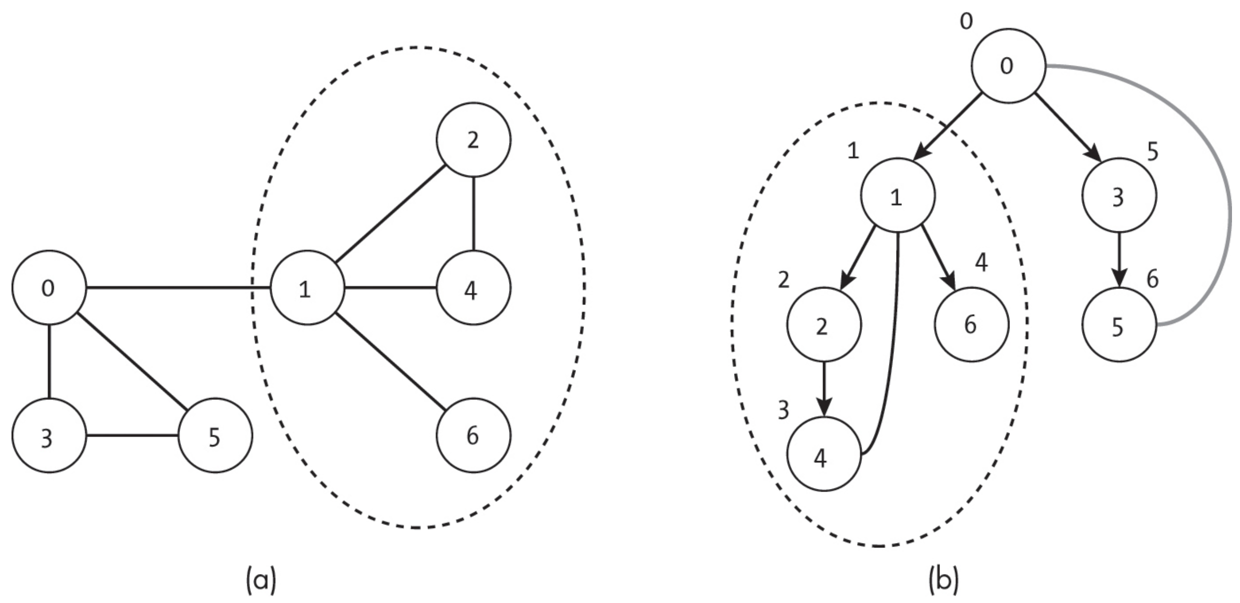


Figure 11-8: An undirected graph (a) and a depth-first search tree (b)

The bridge-finding algorithm checks each subtree in the depth-first search tree by recording the lowest-order indices that neighbor any node in the subtree. The only adjacent edge we exclude is the link between the subtree root *u* and its parent, as this is the edge we are testing.

The Code

We can implement the bridge-finding algorithm using a single pass of depth-first search. To simplify the code, we'll use the helper data structure `DFSTreeStats` to track information about the order in which the depth-first search reaches various nodes, including:

`parent` (`list`) Maps each node's index to that of its parent in the depth-first search tree

`next_order_index` (`int`) Stores the next order index to assign

`order` (`list`) Maps each node's index to its order index

`lowest` (`list`) Maps each node to the *lowest* order index of any nodes in its depth-first search subtree or their immediate neighbors (excluding the node's parent)

The data structure `DFSTreeStats` provides a wrapper for this information and saves us from having to pass many parameters to the search function. We can also use the object to perform basic assignments and updates. We define `DFSTreeStats` in the following code:

```
class DFSTreeStats:
    def __init__(self, num_nodes: int):
      ❶ self.parent: list = [-1] * num_nodes
        self.next_order_index: int = 0
        self.order: list = [-1] * num_nodes
        self.lowest: list = [-1] * num_nodes

    def set_order_index(self, node_index: int):
        self.order[node_index] = self.next_order_index
        self.next_order_index += 1
      ❷ self.lowest[node_index] = self.order[node_index]
```

The constructor sets all the information to its initial values ❶. The code initializes all entries of the lists `parent`, `order`, and `lowest` to `-1` in order to indicate that these values are unset for each node. It sets `next_order_index` to `0` in preparation for the first node.

The helper method `set_order_index()` records the current node's order index and increments the next one to assign. It also initializes the lowest order index seen for this node, which is initially the order index of the node itself ❷.

We use a depth-first search adapted from those in Chapter 4 to fill in the entries of `DFSTreeStats` and find the bridges:

```
def bridge_finding_dfs(g: Graph, index: int, stats: DFSTre
eStats, results: list):
  ❶ stats.set_order_index(index)

    for edge in g.nodes[index].get_sorted_edge_list():
        neighbor: int = edge.to_node
      ❷ if stats.order[neighbor] == -1:
            stats.parent[neighbor] = index
            bridge_finding_dfs(g, neighbor, stats, result
s)
          ❸ stats.lowest[index] = min(stats.lowest[index],
                                        stats.lowest[neighbo
r])
            ❹ if stats.lowest[neighbor] >= stats.order[neigh
bor]:
                    results.append(edge)
        elif neighbor != stats.parent[index]:
            ❺ stats.lowest[index] = min(stats.lowest[index],
                                        stats.order[neighbo
r])

def find_bridges(g: Graph) -> list:
    results: list = []
    stats: DFSTreeStats = DFSTreeStats(g.num_nodes)
    for index in range(g.num_nodes):
        if stats.order[index] == -1:
            bridge_finding_dfs(g, index, stats, results)
    return results
```

The recursive helper function `bridge_finding_dfs()` starts by setting the current node's order index and initial value for the lowest order index reachable in the subtree using the `set_order_index()` helper method ❶.

The code then uses a `for` loop to check each of the node's neighbors. For consistency of ordering with other examples, we use the function `get_sorted_edge_list()` to traverse the neighbors in order of their index, though traversing them in sorted order is not necessary for the correctness of the algorithm. If a neighbor has not been visited (its `order` value is unset) ❷, the code sets its parent and recursively explores it. After returning from the recursive call, the code checks whether it has found a smaller order index neighboring the subtree by comparing the child's `lowest` entry with its own `lowest` entry ❸.

At this point, the search has finished exploring the depth-first search subtree rooted at `neighbor`. It can check whether `edge` is a bridge by comparing the lowest-order index of any node in the subtree or its immediate neighbor with the order index of the subtree's root ❹. The code appends new bridges to `result`.

If the neighbor has been seen (its `order` value is set), then the code first checks whether the neighbor is the parent node itself. If it is, then the edge under consideration was just traversed and the search ignores it. Otherwise, the code checks whether this neighbor represents a node outside the subtree by checking that neighbor's order index ❺.

The `find_bridges()` function provides a wrapper that sets up the statistics and `results` data structures, then starts the search(es). The code finds all bridges in each connected component by performing a single depth-first search from that component, using the approach adapted from Listing 4-2. Since each node is visited only once and each edge is examined at most twice (once in each direction), the cost of the full algorithm scales as $|V| + |E|$.

An Example

Figure 11-9 shows an example run of the bridge-finding algorithm on a graph with eight nodes. Each subfigure shows the state of the search after *completing* the visit to the circled node. The `DFSTreeStats`'s lists `order` and `low` are shown. The arrows indicate edges traversed so far and the dashed edges indicate ones that were seen by search but not traversed, while the bolded gray arrows are the bridges.

Figure 11-9(a) shows the state of the algorithm after the search completes node 6. At this point, it has initially visited and set a preorder index for nodes 0, 1, 2, 3, 7, and 6. Nodes 4 and 5 are unvisited and thus do not have a preorder index. Similarly, the lower bounds correspond to the algorithm's state after visiting only some of the visited node's children. Node 6 has the final state of `lowest` since the search has finished processing it. In contrast, the `lowest` value of node 3 is not finalized because the algorithm has not finished searching its subtree.

In Figure 11-9(b), the search backtracks to node 7 and completes that node. During this process, the algorithm checks whether the edge (7, 6) could be a bridge. Since `lowest[6]` is less than `order[6]`, we know there is an alternate path out of the subtree (through node 2) and the edge is not a bridge.

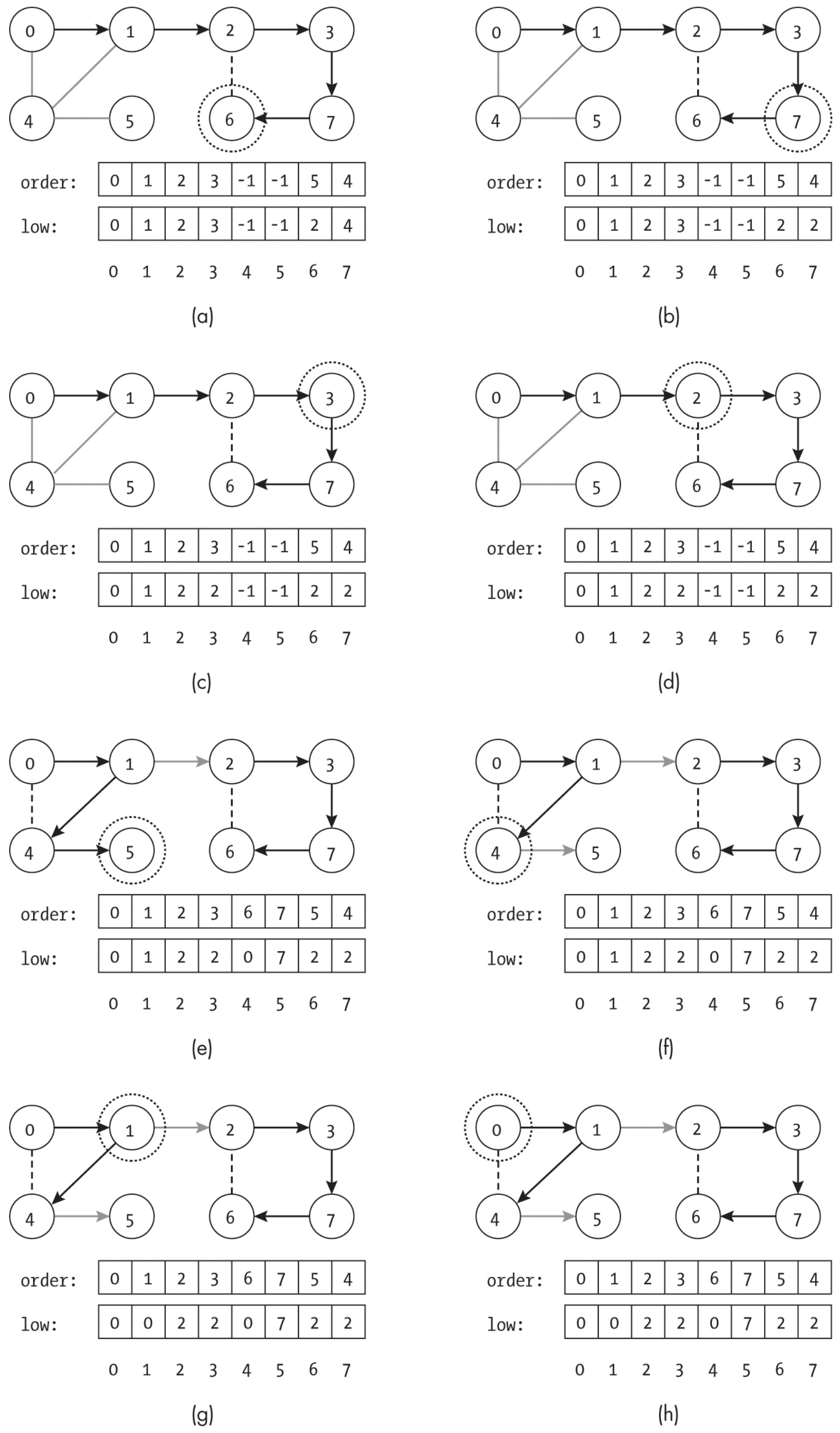


Figure 11-9: The stages of the bridge-finding algorithm

By Figure 11-9(e), the search has found the first bridge. While it has not finished processing node 1, it has fully searched the subtree rooted at node 2.

After returning from node 2, the algorithm finds that `lowest[2]` is equal to `order[2]`, indicating that the edge (1, 2) is the only path into or out of the subtree rooted at node 2. It adds (1, 2) to the list of bridges before moving on to the other children of node 1. In Figure 11-9(f), after finishing the subtree rooted at node 5, the search finds that the edge (4, 5) must be another bridge, as the removal of that edge disconnects node 5.

To visualize this, imagine our evil wizard inspecting their newly created magical labyrinth. They start by walking the labyrinth, building a depth-first tree, and recording the preorder index of each room with a chalk marking on the wall. Each time they enter a new room, they recursively explore any unvisited neighboring rooms and poke their head into previously visited neighboring rooms to check the marks on the wall. When visiting the Room of Loose Ceiling Tiles they might see a new neighbor, the Room with the Ugly Carpet, and also find a connection back to the previously visited Room That Is Always Uncomfortably Warm. Throughout the process, they track the lowest number they have seen since entering each room.

After backtracking through each hallway, the wizard checks their notes to determine whether any of the rooms they just visited have a neighboring room with a preorder index less than the room at the far end of the hallway (the room that they just left when backtracking). After backtracking through their personal favorite, the Hallway of Excessive Chandeliers, the wizard effectively asks, "Is there another passage through which the adventurers could reach one of the rooms up ahead? Or do they have to go through the Hallway of Excessive Chandeliers?" If there is no alternate path, they can mark the Hallway of Excessive Chandeliers as a bridge, be happy in the knowledge that the adventurers will always get to see this opulently decorated passage, and also plan to deploy a good trap.

An Algorithm for Finding Articulation Points

We can adapt the bridge-finding algorithm to identify articulation points by considering the roots of each subtree instead of the edges directly connected to them, using very similar logic. We identify articulation points by looking for a node u whose descendants in the depth-first search tree do not have a neighbor above u in that tree. An edge from a node outside the subtree of u to one of the descendants of *u* would provide the critical alternate path around *u*.

To understand how to use the node's subtrees to identify articulation points, consider the two cases shown in Figure 11-10. We map the depth-first search subtrees onto the original undirected graph with arrows and label each node with its order index. The current node under consideration is shaded and a dashed boundary marks the node's descendants.

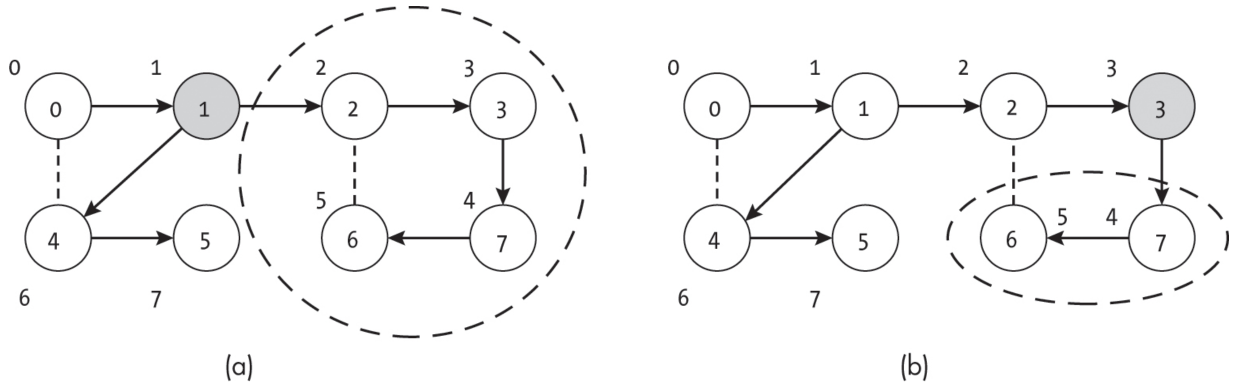


Figure 11-10: Two nodes and their subtrees in a graph's depth-first search tree

In Figure 11-10(a), the algorithm is considering node 1 and descendants {2, 3, 6, 7}. Removing node 1 would cut its descendants off from the rest of the graph. In contrast, node 3 is not an articulation point, as shown in Figure 11-10(b). That node's descendants include node 6, which has a link back to node 2 (outside the subtree of node 3). The edge (2, 6), although not included in the depth-first search tree, provides an alternate path to nodes 6 and 7 in the event node 3 is removed.

This logic works for every node except the root node. Since the root node has no ancestors, we cannot use the same approach of checking the subtrees for back edges. Instead, we must look for cases where the root node has more than one subtree. As shown in the example graph in Figure 11-11, the root node will have multiple subtrees only if the graph has components that would be disconnected by the removal of the root node. If there were an edge joining the subtrees, the depth-first search would traverse that edge before returning to the root.

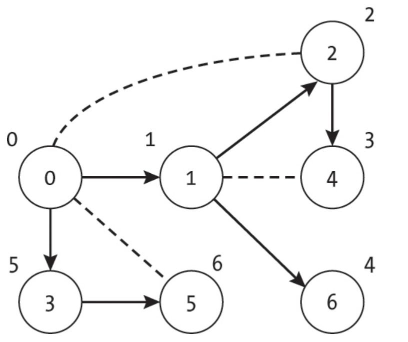


Figure 11-11: A depth-first subtree where the root node is an articulation point

We can combine the specialized root test with the lower bound tracking from the bridge-detection algorithm to identify the articulation points in a graph, as shown in the following code.

The Code

As with the bridge-finding algorithm, we implement the *articulation-point-finding algorithm* with a single pass of depth-first search that completes both the search and the identification. We reuse the `DFSTreeStats` data structure track and update information about each node's parent, order index, and lowest reachable order index.

To simplify the code, we break the search into two functions. The first function handles the non-root nodes and performs the recursive exploration:

```
def articulation_point_dfs(g: Graph, index: int, stats: DF
STreeStats,
                           results: set):
 ❶ stats.set_order_index(index)
    for edge in g.nodes[index].get_edge_list():
        neighbor: int = edge.to_node
        if stats.order[neighbor] == -1:
            stats.parent[neighbor] = index
            articulation_point_dfs(g, neighbor, stats, res
ults)
         ❷ stats.lowest[index] = min(stats.lowest[index],
                                         stats.lowest[neighbo
r])

         ❸ if stats.lowest[neighbor] >= stats.order[inde
x]:
                    results.add(index)

            elif neighbor != stats.parent[index]:
              ❹ stats.lowest[index] = min(stats.lowest[index],
                                            stats.order[neighbo
r])
```

The recursive function `articulation_point_dfs()` performs the majority of the work for this algorithm. It starts by setting the current node's order index and tentative lower bound ❶, then performs the depth-first search by iterating over each neighbor, checking whether it has been visited, and, if not, recursively exploring it.

The code tracks the lower bounds for the neighbors of any node in the subtree. For subtrees in the depth-first search tree (previously unexplored nodes), the code updates the lower bound based on the lowest neighbor of that entire subtree ❷. The logic for identifying articulation points takes place after this recursive exploration of each child. The code determines whether the subtree it just visited would be cut off by the removal of the *current* node by checking whether any node in that subtree contains a neighbor above the current node in the depth-first search tree ❸.

For neighbors that are not part of the depth-first search subtree (previously explored nodes) and are not the current node's parent, the code compares the node's lower bound to the neighbor's order index ❹.

For the root node, we add some additional logic to track the number of subtrees:

```
def articulation_point_root(g: Graph, root: int,
                            stats: DFSTreeStats, results:
 set):
    stats.set_order_index(root)
    num_subtrees: int = 0

    for edge in g.nodes[root].get_edge_list():
        neighbor: int = edge.to_node
      ❶ if stats.order[neighbor] == -1:
            stats.parent[neighbor] = root
            articulation_point_dfs(g, neighbor, stats, res
ults)
            num_subtrees += 1

  ❷ if num_subtrees >= 2:
        results.add(root)
```

The `articulation_point_root()` function starts by setting the order index of the root and initializing the `num_subtrees` counter. It then starts the depth-first search by iterating over each neighbor, checking if it has been visited ❶, and, if not, recursively exploring it using the `articulation_point_dfs()` function. Instead of using the lower bound logic in deciding whether the root is an articulation point, the code simply checks whether the root has two or more subtrees ❷. If so, it appends the root to the results.

The function for finding all articulation points consists of using the `articulation_point_root()` function to run this search on each connected component in the graph:

```
def find_articulation_points(g: Graph) -> set:
    stats: DFSTreeStats = DFSTreeStats(g.num_nodes)
    results: set = set()
    for index in range(g.num_nodes):
      ❶ if stats.order[index] == -1:
            articulation_point_root(g, index, stats, resul
ts)
    return results
```

The `find_articulation_points()` function starts by creating and initializing the data structures needed for the algorithm. Since the data structures are indexed by node and the different connected components are disjoint, the code can use a single `stats` and `results` object for all connected components. The code then iterates over each node, checks whether it has been visited by a search ❶, and, if not, starts a new depth-first search from that node. It finishes by returning the list of all articulation points.

An Example

Figure 11-12 shows an illustration of the algorithm for finding articulation points. Each subfigure shows the state of the algorithm after *completing* the visit to the circled node. The edges tested are represented by an arrow if they are part of the depth-first search tree or a dashed line if they are not. Unexplored edges are solid gray lines, and the discovered articulation points are shaded.

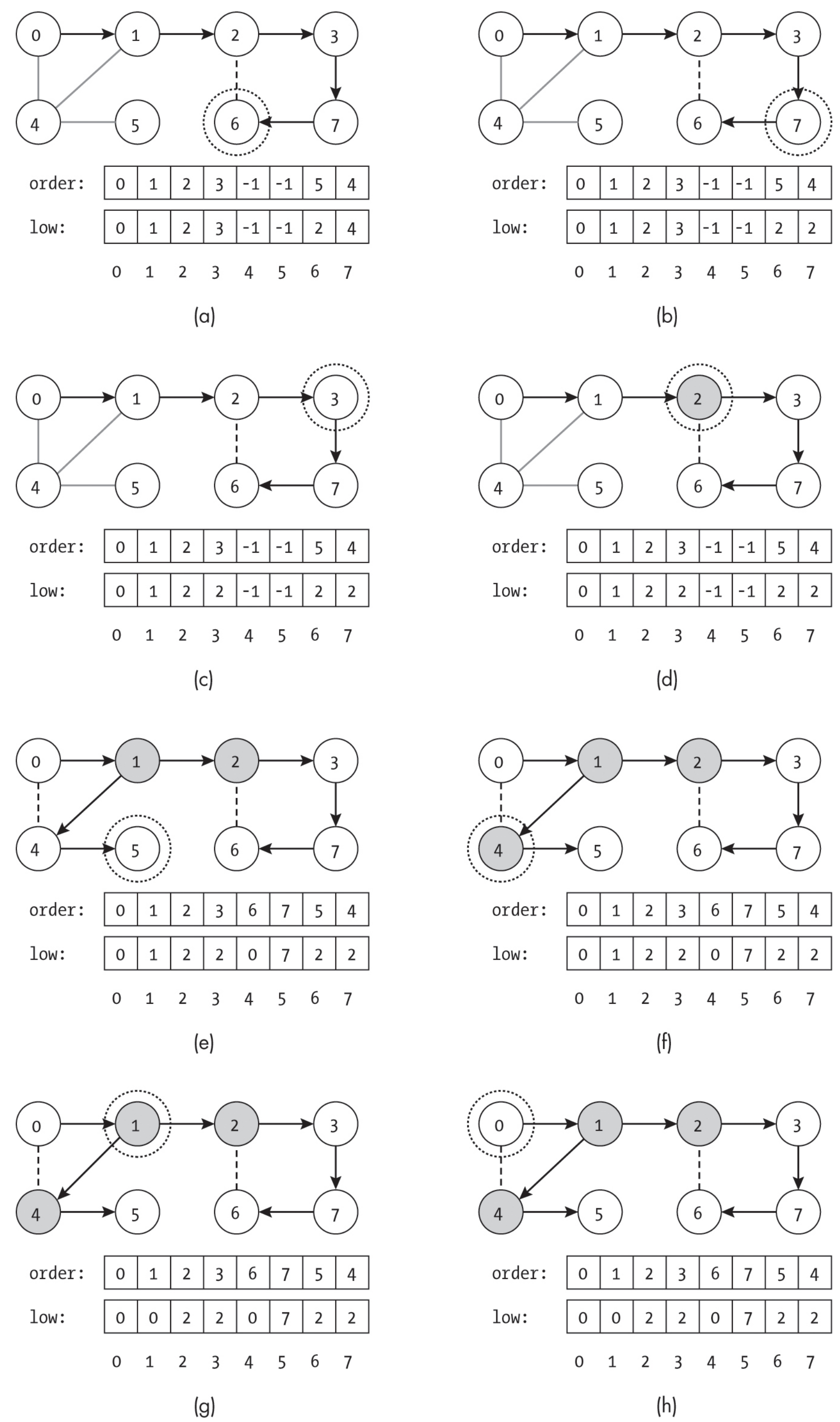


Figure 11-12: The stages of the articulation-point-finding algorithm

The majority of the algorithm behavior represented in Figure 11-12 is the same as that in Figure 11-9. The order in which the nodes are explored and the values of `DFSTreeStats` at each step are identical. The difference in behavior arises where articulation points are detected in Figure 11-12(d). The lowest-order index for any neighbor of the subtree rooted at node 3 is 2, the order index of the current node. We know that node 2 has at least one subtree without connections back to any of its ancestors, meaning that removing node 2 would disconnect that subtree.

Figure 11-12(e) is interesting because, although it shows the state after finishing node 5, the algorithm has already marked (the unfinished) node 1 as an articulation point due to performing the articulation point test after checking each subtree. Regardless of what happens while exploring the other descendants of node 1, we know that removing that node would disconnect the subtree rooted at node 2.

Figure 11-12(h) shows the final step of the algorithm. At this point, the search has returned from the call to the `articulation_point_dfs()` function and is testing the root. Instead of using the low boundary, it checks how many subtrees the root has, revealing that node 0 has a single depth-first search subtree. All nodes in the graph are reached through node 1 before the search returns to node 0. Therefore, node 0 is not an articulation point.

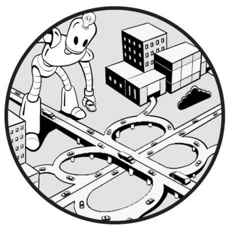
Why This Matters

Bridges and articulation points are critical for understanding the structure of graphs, including their points of failure and bottlenecks. As we saw in the example use cases, these features apply to a variety of real-world problems, from incorporating redundant routes into airline networks to designing the ultimate magical labyrinth.

The algorithms introduced in this chapter provide practical methods for identifying these structural elements and using depth-first search trees and order indexes to determine which nodes are reachable via alternate paths. This again highlights the power and versatility of a simple depth-first search and shows how augmenting information like order indexes can provide deep insights into the overall structure of the graph.

The next chapter further extends our discussion of connectivity, this time considering directed graphs and the related concept of strongly connected components. We introduce an algorithm that builds off the ideas presented in this chapter of collecting statistics with depth-first search to understand the graph’s structure.

12
STRONGLY CONNECTED COMPONENTS



Previous chapters used connected components on undirected graphs to answer questions like "Can we get to a given location from here?" or "Would removing this edge break the graph's connectivity?" Such questions and the algorithms that answer them become more complex once we start thinking about *directionality*. When examining *reachability* on a directed graph, it is no longer enough to say, "We can get from A to B." We need to understand whether we can get back to A and, if not, how that impacts travel through the graph.

This chapter explores the concept of *strongly connected components*, sets of nodes in a directed graph such that any node in the set is reachable from any other node in the set. These components help us understand the structure of the graph and how it can be traversed. We start by formally introducing the concept of strongly connected components and providing example code for checking whether a set of nodes is strongly connected. We describe a few real-world applications of strongly connected components, including modeling how computer programs get stuck on certain states and how information would flow through a social network, then examine an example algorithm for identifying a graph's strongly connected components.

Defining Strongly Connected Components

The formal definition of a strongly connected component in a directed graph is a maximal set of nodes $V' \subseteq V$ such that for any two nodes $u \in V'$ and $v \in V'$, there exists a path of directed edges from u to v. In other words, you can reach any node in the strongly connected component when starting from any other node in the same component. If every node in a directed graph is part of the same strongly connected component, we call the graph *strongly connected*.

We can visualize the importance of strongly connected components in the context of transportation networks. Let's return to the magical labyrinth designed by an evil wizard, as introduced in Chapter 11. To thwart wandering adventurers, the wizard uses one-way doors to connect each of their labyrinth's rooms, as shown in Figure 12-1. They thus give each path a predefined flow. For example, adventurers can use the door between rooms A and B to travel from A to B, but not the other way around.

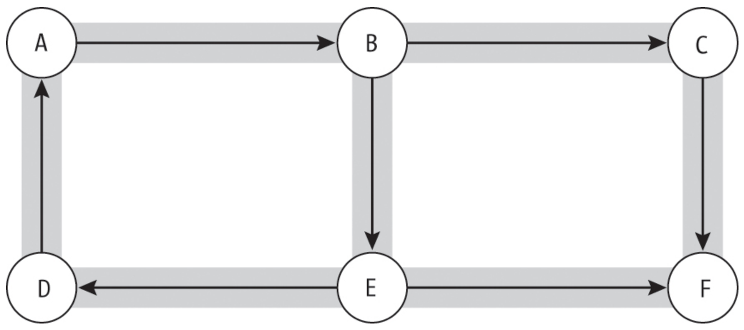


Figure 12-1: A graph modeling the one-way doors between six rooms

This approach succeeds beyond the wizard's expectations. They had hoped simply to prevent adventurers from backtracking and surprising their minions from behind. Instead, they find that certain rooms become unreachable from other rooms. The labyrinth contains multiple strongly connected components: {A, B, D, E}, {C}, and {F}. Adventurers can wander from room A to B to E to D and back to A without running afoul of the doors. However, disaster strikes as soon as they need to leave that component. Adventurers who have ventured into room C soon find there is no path for them to return to rooms A, B, D, or E. As time goes on, adventurers are effectively funneled into room F, which allows the wizard to trap the adventurers in that room with a boss-level monster.

Determining Which Nodes Are Mutually Reachable

The key aspect to understanding and building strongly connected components is determining which nodes are *mutually reachable*. Let's start by reviewing what it means for node *v* to be reachable from node *u*. As noted in "Reachability" in Chapter 3, node *v* is reachable from node *u* only if there exists a sequence of (directed) edges forming a continuous path starting at node *u* and terminating at node *v*. Under this definition, every node is reachable from itself by using the empty set of edges; we can always get to where we already are by not moving.

We can define a helper function `get_reachable()` that uses a breadth-first search to retrieve the set of all other nodes that are reachable in a directed graph `g` from a given starting index (`index`). We use a `set` data structure to track both the reachable nodes and the nodes seen during the current search, as shown in Listing 12-1.

```
def get_reachable(g: Graph, index: int) -> set:
    seen: set = set()
    pending: queue.Queue = queue.Queue()

  ❶ seen.add(index)
    pending.put(index)

  ❷ while not pending.empty():
        current_index: int = pending.get()
        current: Node = g.nodes[current_index]
        for edge in current.get_edge_list():
            neighbor: int = edge.to_node
          ❸ if neighbor not in seen:
                pending.put(neighbor)
                seen.add(neighbor)

    return seen
```

Listing 12-1: Retrieving a set of node indices reachable from the given node

The code starts by setting up the data structures: a set of seen and thus reachable nodes (`seen`) and a queue of future node indices to explore (`pending`). Note that the use of the queue data structure requires us to include `import queue` in the file. The code adds the initial node (`index`) to both its `seen` set and its queue of node indices to explore ❶.

The code uses a breadth-first search to discover all other reachable nodes in the graph. While there are nodes to explore (and thus `pending` is not empty) ❷, the code dequeues the next index, retrieves the node, and checks each of its neighbors using a `for` loop. If the code has not previously encountered a neighbor, that neighbor's index is added to both the queue and the `seen` set ❸. When the code runs out of nodes to explore, it returns the set of seen indices. This set includes every node that can be reached from `index`.

The algorithm in Listing 12-1 operates like a methodical adventurer planning their trip through a magical labyrinth to which they have a map. The adventurer maintains a list (queue) of rooms to evaluate, starting with just the entrance on the list. At each step, they take the top room from their list, cross it out, carefully locate it on their map, and check for adjacent rooms. They add any unexplored neighbors to the bottom of their list before drawing a tidy checkmark next to the room on the map. They then continue evaluating rooms, taking the top (or oldest) item on their list, until their list runs out.

Determining Whether Nodes Are Strongly Connected

We can use the reachability function from Listing 12-1 to define a brute-force check as to whether a set of nodes, given by a list of indices (`inds`), are strongly connected, as shown in the following code:

```
def check_strongly_connected(g: Graph, inds: list) -> boo
l:
    for i in inds:
        reachable = get_reachable(g, i)
        for other in inds:
            if other not in reachable:
                return False
    return True
```

This code uses a pair of nested `for` loops to check whether every node in the component is reachable from every other node. It starts at each node index in `inds` and uses the `get_reachable()` helper function from Listing 12-1 to generate a set of reachable nodes. It then checks whether each index in `inds` occurs in this reachable set. If any node is not reachable from any other, the function returns `False`. Otherwise, it returns `True`.

Although the `check_strongly_connected()` function is simple, consisting of just two loops and a helper search function, it is not an efficient approach. We introduce it here because it provides an easily understandable and intuitive overview of what is required for a set of nodes to be strongly connected. It is the computational equivalent of a hapless adventurer who begins a new exploration of the labyrinth from every possible room and records which destinations they reach. For a set of $|V|$ nodes, the function needs to run $|V|$ searches and then check the other $|V| - 1$ nodes against the resulting reachable set.

Worse, the `check_strongly_connected()` function does not tell us whether nodes are missing from the strongly connected component. Remember that strongly connected components are *maximal* sets of nodes that are mutually reachable. The function tells us only whether each pair of nodes in the list is mutually reachable. It does not tell us what other nodes could be part of the set.

Use Cases

Identifying strongly connected components in a graph is essential for understanding possible movement throughout the graph. This section provides some real-world applications for identifying strongly connected components: analyzing the flow of operations through a program, the flow of gossip through a network, and the ability to traverse a transportation network.

Modeling Computer Program States

We can model the states of a computer program as a directed graph. The startup state might be a single node with edges to states for loading initial data, initializing variables, and checking the network connection. For example, Figure 12-2 shows a diagram of states in a video game. Individual states of the program represent processing user input and rendering the screen. The dotted line indicates the core game loop, which forms a strongly connected component where each state is reachable from each other state. However, states such as loading the initial data files or exiting the game are not part of this component; once you load the initial data files, the program never returns to that state.

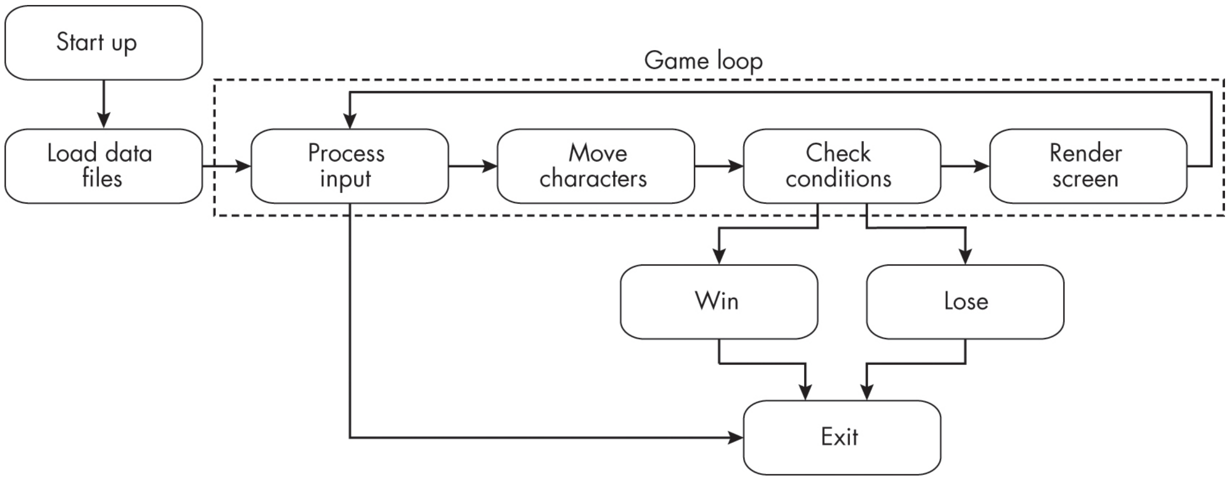


Figure 12-2: A toy video game modeled as a set of program states

Computer programs may have multiple strongly connected components encapsulating different actions or logic. For example, a data analysis program might have one strongly connected component for batch processing data from a file and another to allow user interaction.

Understanding a Gossip Network

The ability to find a graph's strongly connected components can help us determine how far information can spread in directed communication networks. For such a network, a strongly connected component would be a group of people such that if one person knows something, the whole group knows it. That is, information passes from any node in the group to every other node in that same group.

Consider the diagram of a social network in Figure 12-3, where each node represents an individual. An edge from node *u* to node *v* indicates that person *u* will tell person *v* an exciting rumor about the release of a new graph algorithms book. The lack of an edge indicates the lack of such communication. If any node shares a rumor, this information travels to all nodes that are reachable from the starting node.

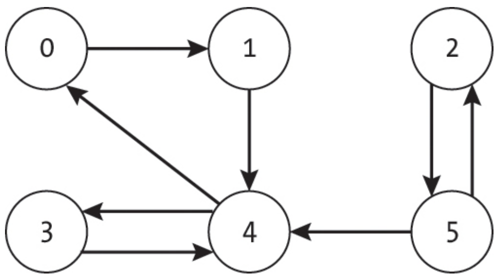


Figure 12-3: A graph modeling a communication network

Strongly connected components provide an insight into groups that fully share information. In Figure 12-3, nodes 0, 1, 3, and 4 form a strongly connected component. Any information that person 4 shares will eventually be learned by those other nodes in the group.

Remember that information can still flow between two nodes in different strongly connected components. In Figure 12-3, any information shared by node 5 will eventually make its way to node 0 by way of node 4. However, the inverse is not true. Since node 4 does not share information with node 5, both nodes 5 and 2 are cut off from the secrets originating from the graph's left side. They will not learn about the upcoming graph algorithms book until it is officially announced.

Planning a Travel Network

When planning out a real-world travel network, it's essential to understand the strongly connected components involved. In the travel context, a strongly connected component is necessary for a traveler to make a round trip. If two locations are not in the same strongly connected component, such as the wizard's dungeon with one-way doors, anyone traversing the network could get stuck in a subset of locations.

For example, in an airline network, any city *v* reachable from city *u* must be part of the same strongly connected component. Otherwise, both planes and passengers from the first component will get stuck in the second component. If an airline offers a set of flights from Toronto to Perth, they will need another set of flights that can return the planes and passengers to Toronto.

Note that it is not strictly necessary for the airline network to form a single connected component, as the airline can serve two disjoint markets with two different strongly connected components. Perhaps it operates a commuter network serving cities in Florida and a separate network for New England. However, each subnetwork must be strongly connected.

The same considerations apply when designing any travel network. It would be a disaster to have a section of the city with only incoming one-way roads. Commuters would drive into the section and be unable to leave. The area would soon fill with cars and persistent honking as desperate drivers tried to find any way out.

Kosaraju-Sharir's Algorithm

Kosaraju-Sharir's algorithm (or just *Kosaraju's algorithm*) is a practical, understandable, and visualizable algorithm for finding strongly connected components. In their book *Data Structures and Algorithms* (Addison-Wesley, 1983), Aho, Hopcroft, and Ullman describe the approach as being independently invented by computer scientists S. Rao Kosaraju and M. Sharir. This algorithm identifies strongly connected components by using a pair of depth-first searches along with an inverted graph.

Kosaraju-Sharir's algorithm begins by performing a depth-first search of the graph that effectively asks, "What nodes can I reach from this starting node?" Throughout the search, it records the *finishing time* of each node. The finishing time, also called *post-order index*, is a counter that records the order in which a search finishes processing a node. The first node finished is given a time of 0, the second a time of 1, and so on. In our labyrinth example, this corresponds to the wizard walking the labyrinth's rooms with a depth-first search—only proceeding through doors in the correct direction—and recording the final time they left each room.

Let's return to the wizard inspecting their upgraded labyrinth shown in Figure 12-1. During the initial inspection, they disable the one-way door spells so they can roam their own dungeon freely. The wizard then starts their inspection at room A. They walk to B, then C, then F before hitting their first dead end. Since there is nowhere to go, F gets a finishing order of 0. The wizard backtracks to room C (whose door they kept open because they control the labyrinth), realizes there is nowhere new to go, and gives room C a finishing order of 1. It isn't until they backtrack to B that they find a new path forward. This time they go to room E, then room D, as they need to explore that path before they can call room B finished. This search progresses by assigning finishing times as follows: F = 0, C = 1, D = 2, E = 3, B = 4, and A = 5. The ordering from this first search depends not only on the structure of the graph but also on the starting node chosen. If the wizard started their inspection at room D, they would finish that room last and get a finishing order of F = 0, C = 1, E = 2, B = 3, A = 4, and D = 5.

The second phase of the algorithm reverses the direction of the graph's edges and runs another set of depth-first searches. By doing so, it effectively pivots from asking, "What nodes can I reach from this starting node?" to "Where can I start if I want to reach this node?" This corresponds to the wizard inspecting their labyrinth in the reverse direction of the one-way doors. At each room, they use a depth-first search that travels in the opposite direction of the one-way doors, allowing them to see which rooms lead to the current one.

To see this, consider the simple graph in Figure 12-4(a) and its reverse in Figure 12-4(b). A depth-first search that starts from node 0 will find nodes reachable from node 0 (in this case, node 1). However, the same search on the reversed graph in Figure 12-4(b) will find nodes that could reach node 0 in the original graph (that is, node 2).

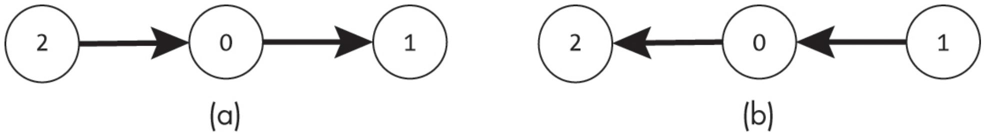


Figure 12-4: A simple graph (a) and its reverse (b)

By running the second search on the reversed graph and choosing our starting nodes by *decreasing* finishing order, Kosaraju-Sharir's algorithm combines the two questions: "What nodes can I reach from this starting node?" and "Where can I start if I want to reach this node?" During the second phase, the algorithm starts a new search (on the reversed graph) from each previously unvisited node and records which nodes are newly visited. The set of nodes visited during each search makes up a strongly connected component in the graph.

Although Kosaraju-Sharir's algorithm performs multiple depth-first searches, each node is visited at most two times: once in the original graph and once in the reversed graph. During each visit, the algorithm checks the current node's outgoing edges exactly once, requiring time proportional to $|V|$ + $|E|$ for each search. Reversing the graph requires another iteration over all nodes and their outgoing edges, scaling proportional to $|V| + |E|$. The overall running time of the algorithm therefore scales as $|V| + |E|$.

The reasoning behind why this algorithm works is a bit complex, and a full proof is outside the scope of the book. The interested reader can find a good discussion of this in algorithmic texts such as *Data Structures and Algorithms* and Sedgewick and Wayne's *Algorithms*, 4th edition (Addison-Wesley, 2011). For now, let's examine how this algorithm builds off those in Chapters 4 and 11 to solve a novel problem.

Transposed Graphs

A core step of Kosaraju-Sharir's algorithm is performing the depth-first search on a version of the graph with its edges reversed, known as a *transposed graph*. This term comes from the *matrix transpose operation*, which reverses each edge's direction; we'll discuss this in greater detail shortly.

For our adjacency list representation of a graph, we define a `make_transpose _graph()` function that creates a transposed graph by iterating through each of the graph's edges and adding the inverse edge to a new graph:

```
def make_transpose_graph(g: Graph) -> Graph:
  ❶ g2: Graph = Graph(g.num_nodes, undirected=g.undirecte
d)
    for node in g.nodes:
        for edge in node.get_edge_list():
            g2.insert_edge(edge.to_node, edge.from_node, e
dge.weight)
    return g2
```

The `make_transpose_graph()` code starts by creating an empty graph (`g2`) with the correct number of nodes and copying the original graph's `undirected` setting ❶. It then iterates through each node in the original graph and each of its edges. For each edge, the code adds an edge in the opposite direction to the new graph. The code finishes by returning the new graph.

We can picture this function in the context of our earlier labyrinth example by considering a change to the underlying environment. An apprentice evil wizard, looking for an accomplishment on which to base their reputation, decides to reverse the direction of *every* door in the labyrinth. They execute this audacious plan by starting with a fresh map, considering each door on the old map, and adding it to the new map with the inverse direction. Once every door has been processed, they submit it to their supervisor and await the inevitable praise.

Note that while the `make_transpose_graph()` function technically supports undirected graphs by copying the original graph's `undirected` setting, it is useful only for directed graphs. The transpose of an undirected graph will be equivalent to the original graph.

The Code

The code for Kosaraju-Sharir's algorithm uses a helper function that implements the individual searches needed during each phase of the algorithm, performing a modified depth-first search over unseen nodes and adding each newly visited node to a given list in order of increasing finish time:

```
def add_reachable(g: Graph, index: int, seen: list, reacha
ble: list):
    seen[index] = True
    current = g.nodes[index]

    for edge in current.get_edge_list():
        if not seen[edge.to_node]:
            add_reachable(g, edge.to_node, seen, reachabl
e)
    reachable.append(index)
```

In addition to the graph (`g`) and the current index (`index`), the function takes in a Boolean list (`seen`) indicating which nodes have been seen and a list of node indices representing the finish order (`reachable`). The code starts by marking the current node index as seen and retrieving the node data structure. A single `for` loop iterates over the unseen neighbors and recursively explores them. When the search has finished processing a node, it adds that node to the end of the `reachable` list. Since the node index is added at the end of the function, the final list will be sorted in order of increasing finish times once recursive exploration completes.

The code for the full Kosaraju-Sharir's algorithm consists of iterating through the list of nodes twice and calling the helper functions on currently unvisited nodes:

```
def kosaraju_sharir(g: Graph) -> list:
    seen1: list = [False] * g.num_nodes
    finish_ordered: list = []
  ❶ for ind in range(g.num_nodes):
        if not seen1[ind]:
            add_reachable(g, ind, seen1, finish_ordered)

  ❷ gT: Graph = make_transpose_graph(g)

    seen2: list = [False] * g.num_nodes
    components: list = []
  ❸ while finish_ordered:
        start: int = finish_ordered.pop()
        if not seen2[start]:
            new_component: list = []
          ❹ add_reachable(gT, start, seen2, new_component)
            components.append(new_component)

    return components
```

The code starts by setting up the data structures for the first search: a Boolean list indicating whether each node has been seen (`seen1`) and a single list of the node indices ordered by *increasing* finish time (`finish_ordered`). The `finish_ordered` list is initially empty because no nodes have been visited. A `for` loop then performs the first phase of the algorithm by iterating through each node index and starting a depth-first search from any unvisited nodes ❶. Since the code uses the same `finish_ordered` list for each call, all the nodes are included in a single list.

The second phase starts by reversing the edges of the graph using the `make_transpose_graph()` function ❷. The code creates a new Boolean list to indicate which nodes it has seen during the second round of searches (`seen2`), then creates an empty list of components (`components`). The `components` list will be a list of lists where each entry contains a list of all node indices in that strongly connected component. A `while` loop iterates over the nodes in order of *decreasing* finish order ❸, achieving this ordering by treating the `finish_ordered` list as a stack and popping off the last element during each iteration. The stack size shrinks by one and the code checks a new node index (`start`) to see if it needs to start a search from that node.

Using the `add_reachable()` function, the code starts a new search from each unvisited node ❹. Each time, it passes in a new empty result list (`new_component`) to represent the current strongly connected component. The function fills the list with the node indices of the new strongly connected component. The code then appends the `new_component` list to `components`.

An Example

Let's examine how Kosaraju-Sharir's algorithm behaves on the example graph shown in Figure 12-5.

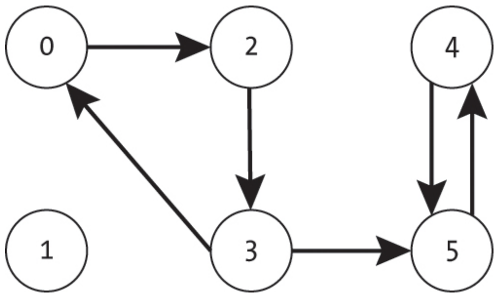


Figure 12-5: A directed graph with six nodes

The first stage of the algorithm is shown in Figure 12-6 and considers the nodes in order of increasing node index. The algorithm runs a depth-first search starting from node 0 and computes the order in which each node finishes. Node 4 is at the end of a long dead end and thus finishes first. In contrast, node 0 does not finish until its depth-first search finds four other nodes. After the first depth-first search completes, node 1 is still unvisited, so the algorithm starts a new search there. The final ordering is 4, 5, 3, 2, 0, 1, as shown by the finish order indicated outside each node.

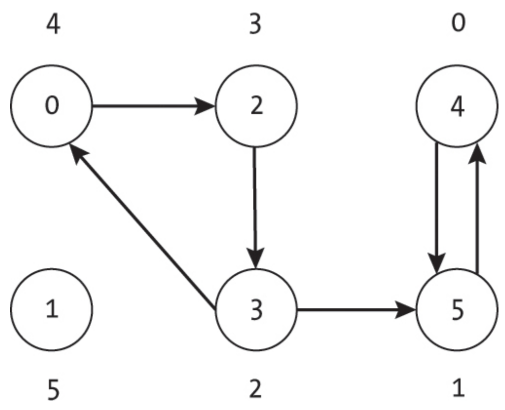


Figure 12-6: The first iteration of Kosaraju-Sharir's algorithm produces an ordering of nodes.

In this phase of the algorithm, we start a new depth-first search from every unvisited node, as in the case of node 1. Nodes that are not reachable from node 0 are thus included in later depth-first searches and added to the end of the ordering.

In its second stage, Kosaraju-Sharir's algorithm transposes the graph, uses a new array of unseen markers, and repeats this sequence of up to $|V|$ depth-first searches. Instead of searching the nodes in arbitrary order, such as using increasing node index, the algorithm chooses the starting nodes using the inverse of the finish ordering from the first step: 1, 0, 2, 3, 5, 4. The last node to finish becomes the starting point for the first search. Each time the algorithm hits an unseen node during our outer loop, it starts a new depth-first search from that node. It adds all unseen nodes encountered during this depth-first search to the current component and marks them as seen.

Figure 12-7 shows the three searches performed during the second stage on our example graph, starting from node 1 in Figure 12-7(a), node 0 in Figure 12-7(b), and finally node 5 in Figure 12-7(c). Nodes circled with dashed lines indicate the nodes visited during each depth-first search, while gray nodes indicate the nodes with a `seen` value set to `True` by either the current search or a previous one.

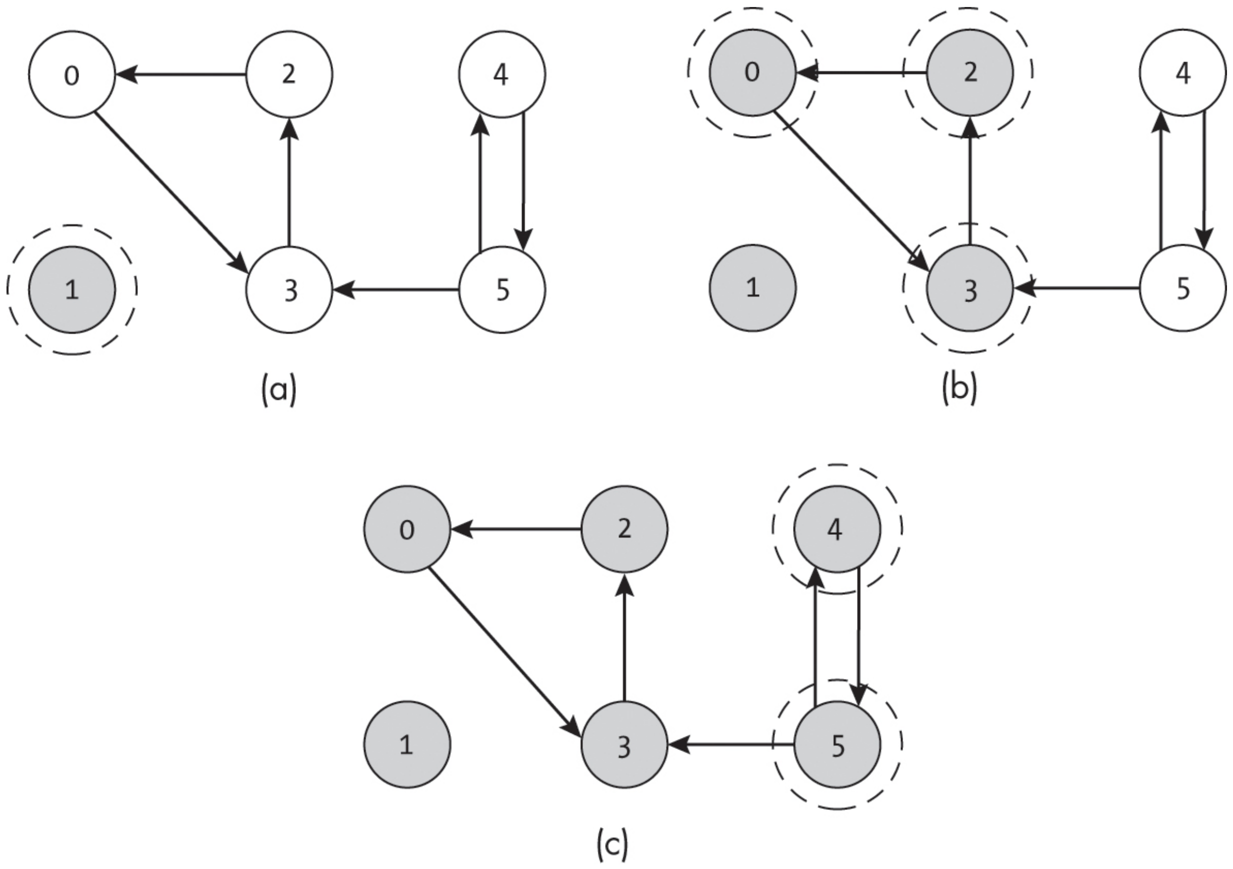


Figure 12-7: The second stage of Kosaraju-Sharir's algorithm

As shown in Figure 12-6, we only start searches at unseen nodes, and individual searches do not visit already seen nodes from previous searches.

The first depth-first search, shown in Figure 12-7(a), starts at node 1. In Figure 12-7(b), the depth-first search of the reversed graph starts at node 0 and progresses to node 3 and then node 2. At this point, it hits a dead end and backtracks. When the depth-first search returns to node 0, we know that we have visited all nodes that both could reach node 0 and are reachable from node 0 in the original graph. The third and final depth-first search is shown in Figure 12-7(c) and explores nodes 4 and 5.

Why This Matters

Strongly connected components provide insight into the subsets of nodes that are mutually reachable. The concepts in this chapter provide both a practical tool for thinking about real-world problems, such as transportation networks or gossip networks, and a foundation for thinking about the fundamental structure of a graph itself. For example, identifying the strongly connected components provides one mechanism for partitioning a large graph into meaningful subgraphs.

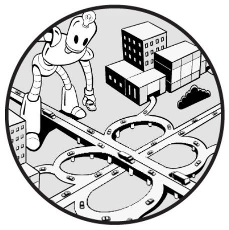
The algorithms presented in this chapter build off the basic depth-first search from Chapter 4 to analyze reachability within graphs and construct strongly connected components. Kosaraju-Sharir's algorithm provides both a practical and visualizable approach to finding connected components and shows how we can continue to adapt search algorithms to more complex problems.

Beyond the algorithm covered in this chapter, a variety of other approaches have been developed to find strongly connected components. For example, Robert Tarjan proposed an algorithm for this purpose that relies on just a single depth-first search, using the same principles as his algorithms discussed in Chapter 11. As with all topics in this book, a rich variety of approaches with different trade-offs exist. The goal of this chapter is to provide a foundation for understanding and comparing these different approaches.

The next chapter discusses random walks through graphs, building on the concepts behind strongly connected components to examine how walks can get stuck in absorption states or wander around forever.

13

RANDOM WALKS



So far, this book has introduced a variety of algorithms focused on achieving specific goals. This chapter considers algorithms that seek to do something a bit different: inducing *random behavior* on graphs. Analyzing random movement through a graph allows us to model and study systems with nondeterministic behavior such as randomized network routing or real-world social interactions.

Random walks on graphs have a rich mathematical history that extends well beyond the scope of this book. This chapter provides an overview of random walks, an introduction of how to analyze them with Markov chains, and code for implementing a random walk over a graph. We consider the types of questions we can investigate and systems we can model with random walks, such as gambling and luck-based board games. Finally, we consider how we can reconstruct underlying graphs from sample observations.

Introducing Random Walks

A *random walk* on a graph is a sequence of nodes where the next node in the sequence is chosen randomly based on some probability distribution. We denote the *transition probability* from node *u* to node *v* as:

$$p(u \rightarrow v) \text{ where } 0 \leq p(u \rightarrow v) \leq 1$$

This means that whenever we are at a node u, we select the next node from u's neighbors using the given probability distribution.

We can visualize a random walk as a tourist who absolutely refuses to plan ahead. Convinced that serendipity produces the best vacations, they set out to explore the city with neither a map nor a clue as to where they are going. When they reach an intersection, they consider possible routes and choose one at random. The tourist makes each decision in isolation, without thought to past or future transitions.

We use the graph structure to restrict probabilities in a few ways. First, we limit movement to nodes that are connected to the current node by an edge. In the case of directed graphs, this must be an edge in the correct direction:

$$p(u \rightarrow v) = 0 \text{ if } (u, v) \notin E$$

In other words, our tourist has no chance of traversing from point u to point v unless there is a road connecting them. For the sake of clarity, in this chapter we also require the probability to be greater than zero if the edge exists:

$$p(u \rightarrow v) > 0 \text{ if } (u, v) \in E$$

This means that our tourist can, in theory, traverse all the roads in the city.

Second, transition probabilities must sum to 1.0 over all outgoing adjacent edges:

$$\textstyle\sum_v p(u \rightarrow v) = 1 \text{ for every } u \in V$$

This restricts the probabilities to form a valid distribution. Every node must include at least one outgoing edge. In directed graphs, we can use self-loops $p(u \rightarrow u) > 0$ to model cases where the walk does not proceed to a new node. This constraint corresponds to the tourist always having at least one path along which to proceed, even if that path loops back to the current location.

Probabilities in Random Walks

The simplest random walk is one where we choose among the outgoing edges with equal probability. In this case, our hypothetical tourist chooses from the current intersection's streets completely at random. If the intersection has two outgoing edges, the tourist chooses either with a 50 percent probability. If the intersection has four outgoing edges, they each get a probability of 25 percent.

Figure 13-1 shows an undirected and unweighted graph (a) and the corresponding transition probabilities out of each node (b).

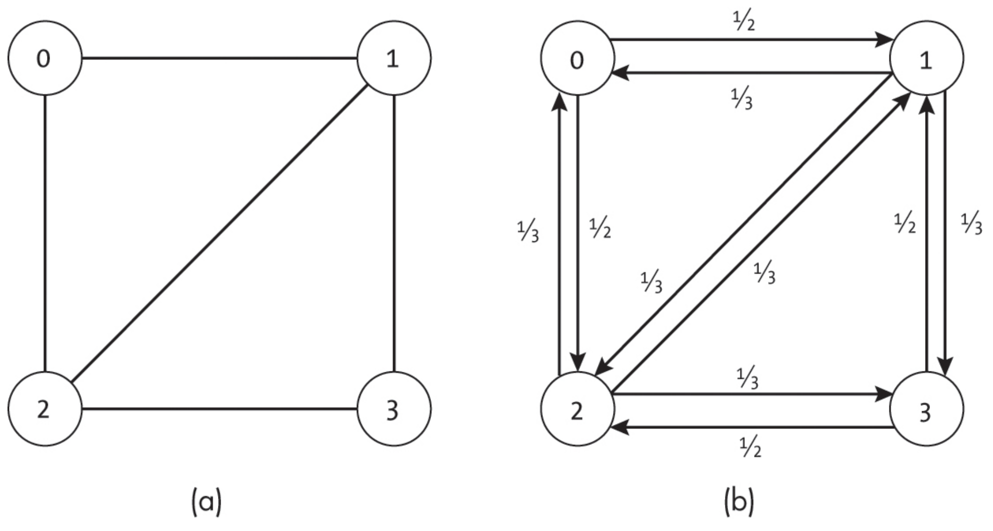


Figure 13-1: An undirected graph (a) and its random walk probabilities (b)

While we discuss random walks on both directed and undirected graphs, we always model these systems as directed graphs because, in many cases, the transition probabilities between two nodes will not be symmetric. In formal terms:

$$p(u \rightarrow v) \neq p(v \rightarrow u)$$

In Figure 13-1(b), for example, the probability of moving from node 0 to node 1 is 1/2, while the probability of moving in the reverse direction is only 1/3. For our wandering tourist, the probability of traveling between two intersections depends on how many roads branch out from the current intersection.

We can use weighted graphs to model more realistic scenarios by assigning different probabilities to each edge and storing them in the edge weights. We constrain these probabilities (the edge weights) such that the sum of probabilities over all outgoing edges equals 1.0. Figure 13-2 shows an example as a directed, weighted graph. A random walk at node 3 has three possible next states: it can move to node 1 with a probability of 0.2 or to node 2 with a probability of 0.6, or it can stay at node 3 (via a self-loop) with a probability of 0.2.

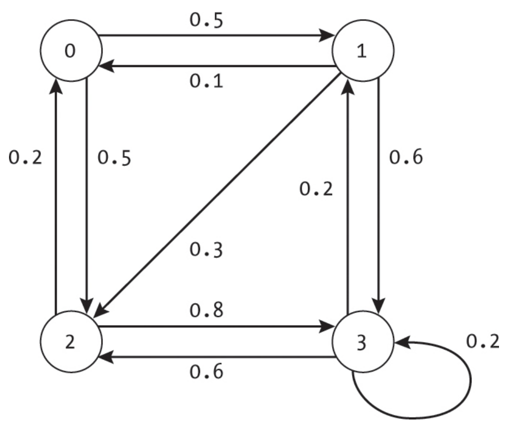


Figure 13-2: A directed graph with transition probabilities

These more general graphs correspond to a tourist who is probabilistically influenced by factors beyond the number of roads. They tend to head toward areas with more interesting architecture or follow the smell of coffee. When arriving at a particular four-way intersection, they have a surprising 90 percent chance of taking a left toward a street of cafés.

Random Walks as Markov Chains

Random walks over a graph's edges are one example of a *Markov chain* or *Markov model*, a system for which the probability of the next state depends solely on the current state. Each step is taken without consideration to the previous path, a property known as *time invariance*. We draw the connection to Markov chains to allow us to tap into the vast amount of related analysis. The full breadth of research into analyzing different types of Markov chains far exceeds the scope of this book. In this chapter, we will only briefly touch upon a few concepts and terminology that help analyze random walks and demonstrate their modeling power.

The time invariance property corresponds to our wandering tourist's habit of considering only the paths open in front of them. They do not consider pesky details such as where they have already been, how many steps they have taken, or even what time of day it is. While this is not optimal for conventionally regular activities such as eating and sleeping, the tourist resolutely adheres to their randomized vacationing principles.

In probability and statistics references, these transition probabilities are often written as $p(X_t \mid X_{t-1})$ to indicate the probability of being in state X_t at timestep t given that the system was in state X_{t-1} at time $t - 1$. Combining this shorthand with the graph-based notation, we get the following equation:

$$p(u \rightarrow v) = p(X_t = v \mid X_{t-1} = u)$$

Given the independence of each transition, we can compute the probability of an entire path $[v_0, v_1, v_2, \ldots, v_{k]}$ from a fixed starting node v_0 by multiplying the probability of each transition:

$$p([v_0, v_1, v_2, \ldots, v_k]) = \prod_{i=1 \text{ to } k} p(X_t = v_i \mid X_{t-1} = v_{i-1})$$

Markov chains are useful for a wide variety of tasks for which transitions are independent of previous paths. Artificial intelligence uses a variety of (more powerful) Markov models to simulate or reason about a range of real phenomena, from understanding speech to decision-making with autonomous agents. For example, the *hidden Markov model* is a staple of machine learning that uses random transitions over unseen states, where each state produces noisy output. Efficient algorithms exist for estimating the underlying states from the output or even learning both the transition probabilities and output distributions from sample data.

In contrast, the random walks we consider in this chapter represent a particularly simple Markov model. The current state (node) is visible at each state and the decisions are fully random. As we will see, however, even these seemingly simple models can provide a wealth of simulative and analytical power.

Transition Probabilities

When modeling a random walk on a graph using the edge weights, we require the weights out of each node to form a valid probability distribution. We can test whether the edge weights of our `Graph` data structure form a valid probability distribution by iterating over each node and checking that the sum of outgoing edge weights is 1.0, as shown in Listing 13-1.

```
def is_valid_probability_graph(g: Graph) -> bool:
    for node in g.nodes:
      ❶ edge_list: list = node.get_edge_list()
        if len(edge_list) == 0:
            return False
        total: float = 0.0
        for edge in edge_list:
          ❷ if edge.weight < 0.0 or edge.weight > 1.0:
                return False
            total += edge.weight
      ❸ if abs(total - 1.0) > 1e-10:
            return False

    return True
```

Listing 13-1: Checking the validity of probabilities stored in the edge weights

The code uses a `for` loop to iterate over each node and check that the weights of its outgoing edges form a valid probability distribution. First, it extracts the node's edge list and checks whether it is empty ❶. If so, there is nowhere to go from that node and the code returns `False`. The constraint that the sum probabilities out of a node equal 1 requires that every node must have at least one outgoing edge with a nonzero weight, even if it is a self-loop.

The code uses a second `for` loop to iterate over the outgoing edges. It checks that each edge has a valid probability between 0.0 and 1.0, immediately returning `False` if not ❷, then adds the current the edge weight to the total. After reviewing all edges, the code checks if the total weight is 1.0, allowing for a small accumulation of floating-point errors ❸. It returns `True` only if all these conditions pass for all nodes and edges.

Matrix Formulation

The matrix representation of a graph is useful when analyzing properties of a random walk on the graph and is commonly used in statistics and machine learning texts to describe random walks. In the matrix representation, transition probabilities are often specified using a *transition matrix* (*M*) where the value in row *i* and column *j* of the matrix corresponds to the probability of moving to node *j* given that the walk is at node *i*:

$$\mathrm{M}[i][j] = p(i \rightarrow j)$$

We can even reuse the `GraphMatrix` data structure from Chapter 1 to store these values. Because we restrict the entries to be probabilities, we impose additional restrictions to the values in `GraphMatrix`'s `connection` list:

- $0 \leq$ `connections[i][j]` ≤ 1 for all `i` and `j`
- $\sum_j$ `connections[i][j]` $= 1$ for all `i`

These constraints mirror the restrictions placed on edge weights earlier.

We can use matrix math to model the effect of taking a random step. We let V_t be a vector of probabilities such that $V_t[u]$ is the probability that our random walk is at node u (so $0 \leq V_t[u] \leq 1$ for each u and $\sum_u V_t[u] = 1$) at time step t. For example, we would use $V_t = [0.5, 0.4, 0.0, 0.1]$ to represent the probability that our walk is at each of the four nodes in Figure 13-2. The vector indicates there is a 50 percent chance of being at node 0, a 40 percent chance of being at node 1, a 0 percent chance of being at node 2, and a 10 percent chance of being at node 4.

The vector V_0 gives the probability of starting the walk at each of the nodes. For example, $V_0 = [1.0, 0.0, 0.0, 0.0]$ indicates a deterministic start at node 0, while $V_0 = [0.5, 0.5, 0.0, 0.0]$ indicates an equal chance of starting from either node 0 or node 1. The vector V_1 then gives the probability of being at each node after randomly starting the walk according to V_0 and taking a single additional random step. V_2 indicates the probability after two steps of the random walk and so forth.

We can use matrix algebra with the transition matrix M to compute subsequent probability distributions:

$$V_{t+1} = V_t M$$

Each entry $V_{t+1}[u]$ gives us the probability that our random walk is at node u at the following time step $t + 1$. We can even add a method to the `GraphMatrix` class that performs this computation, as shown in Listing 13-2.

```
def simulate_random_step(self, Vt: list) -> list:
    if len(Vt) != self.num_nodes:
        raise ValueError("Incorrect length of probability d
ist")

    Vnext: list = [0.0] * self.num_nodes
    for i in range(self.num_nodes):
        for j in range(self.num_nodes):
            Vnext[j] += Vt[i] * self.connections[i][j]
    return Vnext
```

Listing 13-2: Simulating a single step of a random walk on a graph

The code starts by checking that the incoming vector `Vt` has the correct length and, if not, raising an error. It then creates a result vector `Vnext` and uses a pair of nested `for` loops to perform the computation. It returns the new vector of probabilities.

NOTE

As mentioned in Chapter 1, the code in this book uses a list of lists to represent matrices for the purposes of illustration. For the sake of efficiency, production code for this computation should use a library that supports efficient matrix operations, such as `numpy`.

The computation for simulating a random step also works from a deterministic state: $V[u] = 1$ for exactly one node u. By then multiplying $V_{t+1} = V_t M$, we get the probability distribution of where our random walk will be after exactly one step away from u. We can repeat this process by multiplying by M again as follows:

$$V_{t+2} = V_t M M$$

This gives us the probability distribution of nodes that are reached in exactly two steps.

Alternatively, we can extend our matrix notation such that M_t is the transition matrix for a random walk of exactly t steps, so $M_t[u][v]$ is the probability of transitioning from node u to node v in exactly t steps. We can compute this matrix directly using matrix multiplication:

$$M_t = \prod_{I = 1 \text{ to } t} M$$

While the matrix formulation is useful for describing and analyzing the properties of a random walk, the code in the remainder of the chapter uses the adjacency list representation of our `Graph` class for consistency with the other chapters. All the functions can be adapted to work with the `GraphMatrix` class.

Use Cases

Random walks are used to model and analyze problems that involve non-deterministic behavior. Random behavior shows up in a wide range of real-world systems, from human interaction to the explicit randomness in some computer algorithms. In this section, we look at three example use cases: social networks, randomized explorations, and games of chance.

Information Chains in Social Networks

We can use random walks to model how a rumor spreads through a social network where the interactions have a random component. Determined not to gossip excessively, each person in the network resolves to pass information to only a single other person when they hear a rumor. However, they are bursting to share the latest gossip as soon as they hear it, so they share the news with the first person they come across. This is inherently a probabilistic selection, as they don't know which friend they'll run into first. After sharing their news, they are temporarily satisfied and hold off on discussing the rumor until it is shared with them again.

We could model the social network as a graph where the edges represent the probability that each neighbor is the person with whom the rumor is shared. The rumor itself randomly walks the graph, getting passed from person to person.

Exploration

Previous chapters discussed numerous algorithms for deterministically exploring graphs, such as depth-first search or A* search. However, many real-world explorations involve a randomized element, such as weather-based path closures. Random walks allow us to model systems with such constraints.

Consider the explorer from Chapter 8 searching for a best path to an archeological site. The current conditions might add a random element to their exploration. When faced with a fork in the path, the northern route might have a 50 percent chance of being flooded out, while the southern route has a 10 percent chance of being blocked by an angry swarm of hornets. Taking these probabilities into account, we could model their unpleasant journey through the jungle as a random walk.

We can use a similar approach to analyze robot path planning in dynamic environments. A search-and-rescue robot exploring a damaged building may run into different obstacles at different times, such as fires or flooded passages, and need to reroute.

Games of Chance

We can also use a random walk to simulate the outcomes of games of chance. Graph nodes represent different game states, and the edges indicate the possible (probabilistic) changes in those states. For example, we can use the Markov chain shown in Figure 13-3 to represent a gambler playing a one-dollar slot machine.

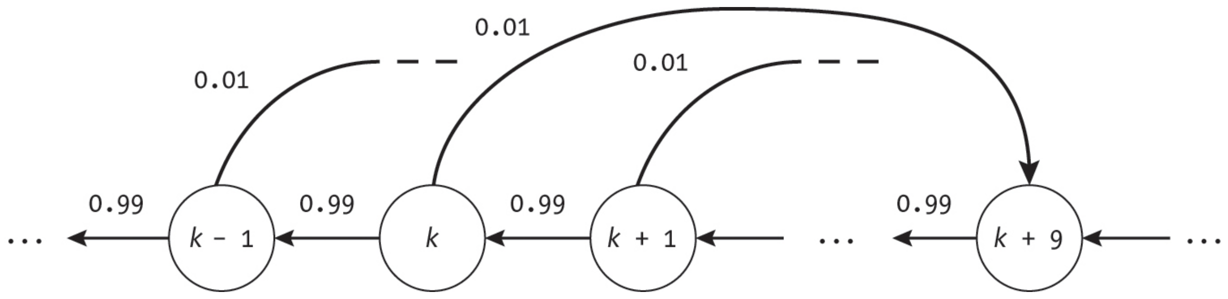


Figure 13-3: A subset of nodes for a gambler's graph

Each state represents the number of dollars in the gambler's possession. Each pull of the slot machine decides the next state without regard to previous pulls or the gambler's current financial situation. Maybe the machine has a 1 in 100 chance of paying out 10 dollars, moving the gambler from state *k* to state *k* + 9, and a 99 in 100 of chance of paying out nothing, moving the gambler from state *k* to *k* – 1.

As we model more complex games of chance, we need correspondingly more complex graphs. Later in this chapter we show how to use graphs to model luck-based board games. We discuss nodes that simultaneously represent multiple players' states and the transitions among them.

Simulating Random Walks

One powerful approach to understanding random walks and their underlying graph is to *repeatedly simulate* a random walk over the graph and analyze the paths taken. We simulate a random walk on a graph by repeatedly selecting the next state based on the probability distribution over the neighbors of the current state. As a prerequisite, we need a function to sample from a finite set of options with pre-specified probabilities.

We provide a simple algorithm for illustration purposes that draws a random number uniformly within [0, 1) and checks to which neighbor it corresponds by iterating over each outgoing edge and accumulating the cumulative probability of the previous nodes. We are effectively breaking the range [0, 1) into regions for each option where the size of the region corresponds to the probability of that option. Figure 13-4 shows an example where 50 percent of the range leads to node 0, 20 percent leads to node 1, and 30 percent leads to node 3.

By tracking the cumulative probability seen so far while iterating over the options, we are tracking the start and end of each region and comparing it to the selected value. We want to find the bin whose region brackets our randomly selected value. As soon as we cross the bin edge where the cumulative value exceeds the randomly selected value, we know we have gone too far.

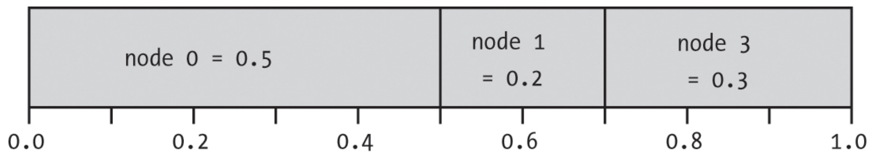


Figure 13-4: The transition probability to three nodes and the corresponding cumulative probability

The code for a random walk on a graph consists of a random number generation followed by a single `for` loop that iterates over the outgoing edges, as shown in Listing 13-3.

```
def choose_next_node(current: Node) -> int:
  ❶ prob: float = random.random()
    cumulative: float = 0.0
    edge_list: list = current.get_edge_list()

    for edge in edge_list:
        cumulative += edge.weight
      ❷ if cumulative >= prob:
            return edge.to_node
  ❸ return edge_list[-1].to_node
```

Listing 13-3: Choosing the next node in a random walk

The code starts by drawing a random number from [0, 1) using Python's `random` library ❶. It then uses a `for` loop over each outgoing edge to compute the total (cumulative) probability seen so far. The selected edge is the first one whose weight causes this cumulative probability to exceed the selected random number ❷. If the code makes it to the end of the list (due to imprecisions in how floating-point numbers are stored), the code simply returns the last edge ❸.

The `choose_next_node()` function in Listing 13-3 does not perform any validity checks on the probability distribution leaving each node. This is intentional so as not to pay the cost of performing the check each time the function is called. Instead, I recommend checking the distributions for all nodes once using the `is_valid_probability_graph()` function from Listing 13-1.

Given this helper function, the code to perform the random walk from a given starting node (`start`) is relatively short:

```
def random_walk(g: Graph, start: int, steps: int) -> list:
  ❶ if not is_valid_probability_graph(g):
        raise ValueError("Graph weights are not probabiliti
es.")

    walk: list = [-1] * steps
    current: int = start
    walk[0] = current
    for i in range(1, steps):
      ❷ current = choose_next_node(g.nodes[current])
        walk[i] = current

    return walk
```

The function starts by confirming that the graph's weights represent a valid probability distribution and raising an error if not ❶. It then allocates a list `walk` to store the results, sets the current node to the start node, and sets the first step in the walk to the start node. The code uses a `for` loop to iterate through each step, using the `choose_next_node()` function from Listing 13-3 to continually select the next node from the current one ❷. The code adds the new node to the `walk` list, which it returns after taking all the steps.

Statistical Measures

Random walks are a powerful tool for understanding randomized systems and computing a variety of practical statistics measures. For example, we might want to find the probability of reaching a particular node or determine how many steps it will take to get there. These measures are useful in answering questions such as "What is the probability a gambler will lose all their money?" or "How long (on average) will it take a rumor to reach me?" or "If a tourist randomly wanders the city for years, how long will they spend at each location?"

In this section, we briefly consider how such questions apply to the case of our wandering tourist and outline how to compute the answers. After considering the probability of reaching specific nodes and the average number of steps to do so, we analyze the long-term behavior of random walks.

Hitting and Absorption Time

The *hitting time* of a set of nodes $A \subset V$ is the average number of steps a random walk must take before it first hits a node in A from a given starting node. For example, if A is the set of all intersections with cafés in the town our tourist is exploring, they might want to calculate the hitting time of that set to find out when they are likely to get their next cup of coffee.

If a random walk cannot leave the nodes in A, we refer to these hitting times as *absorption times* because the walk is absorbed into A. For example, in Figure 13-5, node k forms an absorbing set. Once a walk arrives at node k, it stays there forever.

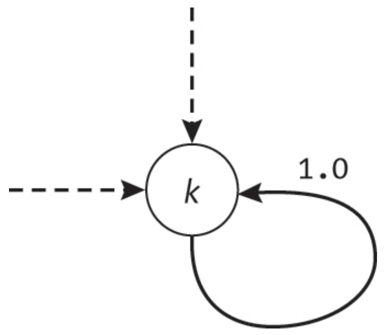


Figure 13-5: An absorbing set consisting of a single node in a graph

Absorbing nodes can also be used to represent the termination of a random walk. As a concrete example, the tourist could decide to stop their random walk when they hit their hotel.

When analyzing random walks on graphs, we often consider two statistical quantities related to hitting times: the probability the walk will reach the subset and the expected time to do so.

Hitting Probability and Absorption Probability

The *hitting probability* for a subset of nodes $A \subseteq V$ is the probability that a random walk from node u will hit a node $v \in A$ in that subset. We can use this measure to answer questions like "What is the probability that our tourist will encounter a coffee shop?" Similarly, the *absorption probability* for a subset of nodes A is the probability that a random walk from node u will be absorbed by that subset. This allows us to ask questions like "What is the probability the tourist will return to the hotel and stop their random walk?"

Absorbing subsets of nodes can impact the hitting probability of non-absorbing nodes. For example, consider the weighted graph in Figure 13-6, which shows a graph with three nodes and an absorbing subset of $\{0, 1\}$.

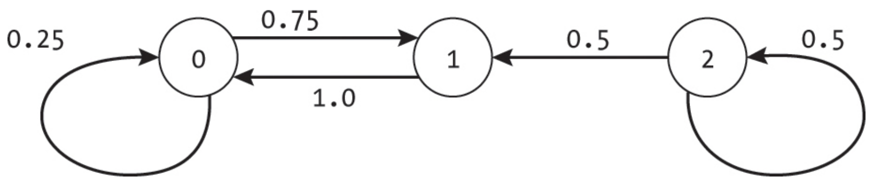


Figure 13-6: A graph with three nodes and transition probabilities

The probability that a walk starting from node 2 will hit node 0 is 1.0 given a potentially infinite number of steps. In contrast, once the walk hits the nodes $A = \{0, 1\}$, it can never travel to node 2, as it will be stuck in an endless set of steps connected to nodes 0 and 1.

Expected Hitting Time

The *expected hitting time* is the expected number of steps a random walk takes (on average) before it first hits a node in A. This allows us to determine how long on average it will take our tourist to get their next coffee. The *absorption time* is similar, quantifying the expected time until the random walk reaches the absorbing set. For our tourist, this would be how long their walk will be on average (before they reach their hotel and stop for the day).

For example, the expected hitting time of node 1 from node 0 (denoted h_{01}) in Figure 13-6 is as follows:

$h_{01} = 1 \times p(\text{first hit node 1 after 1 step})$

$+ 2 \times p(\text{first hit node 1 after 2 steps})$

$+ 3 \times p(\text{first hit node 1 after 3 steps}) + \ldots$

$h_{01} = 1 \times \frac{3}{4} + 2 \times \frac{1}{4} \times \frac{3}{4} + 3 \times (\frac{1}{4})^2 \times \frac{3}{4} + 4 \times (\frac{1}{4})^3 \times \frac{3}{4} + \ldots$

$h_{01} = 4/3$

This is because the possible walks from node 0 include walks that start [0, 1], [0, 0, 1], [0, 0, 0, 1], and so forth. If we have the full transition matrix, we can use a set of equations to solve for the expected hitting times.

The expected hitting time may be infinite, like that from node 0 to node 2 (h_{02}) in Figure 13-6. No matter how long of a walk we consider, we can never hit node 2 from node 0.

Stationary Distribution

Stationary distribution represents the distribution of visits to each state if we keep wandering a strongly connected graph forever. This distribution provides insight into how likely we are to spend time at each node. For example, we could ask, “After our tourist has been wandering for days, what is the probability that they are currently at the café on Fifth Street?” We can also use stationary distribution to predict the probable locations of millions of wandering tourists all following the same randomized rules as they traverse a city.

Returning to the matrix notation introduced earlier in the chapter (where M is the transition matrix and V_t is a vector of probabilities such that $V_t[u]$ is the probability that our random walk is at node u at time step t), the stationary distribution is a vector V^* where:

$$V^* = V^* M$$

In other words, adding one more step to the random walk will not change the distribution of the possible locations.

We can derive stationary distributions from the structure of the graph. Consider the two-node graph in Figure 13-7, where M = [[0.25, 0.75], [0.5, 0.5]].

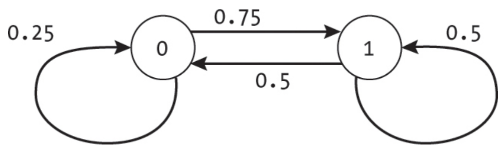


Figure 13-7: A graph with two nodes and four transition probabilities

The structure of the graph in Figure 13-7 indicates that over the long term, random walks will tend to spend more time at node 1 than at node 0. The probability of staying at node 0, due to a self-loop, is only 0.25, while the probability of staying at node 1 is 0.5. We can quantify that difference in time spent at each node by using the stationary distribution, which for this graph is $V^* = [0.4, 0.6]$.

Luck-Based Board Games

We can bring together the topics in this section to consider one of the most entertaining applications of random walks: analyzing luck-based board games for children. These games involve no actual choices, but instead rely on random numbers generated by spinners or dice to decide the moves. After hours spent spinning dials that determine whether a plastic piece moves ahead one, two, or three spaces, even non-statistically minded people might inquire about the expected absorption time of the goal state by asking, "How much longer do I have to play this game?" We can answer such questions by modeling the game as a random walk on a graph.

Consider the simple example of a game where the goal is to be the first player to complete a circuit of the board. During their turn, each player uses a small spinner labeled with 1, 2, and 3 that indicates how many steps to take. Figure 13-8 shows a graphical representation of several states in this game. The nodes' numbers correspond to the squares on the board, where the player is currently at square k. Based on the random spinner, they could move to squares $k + 1$, $k + 2$, or $k + 3$, each with a 1/3 probability.

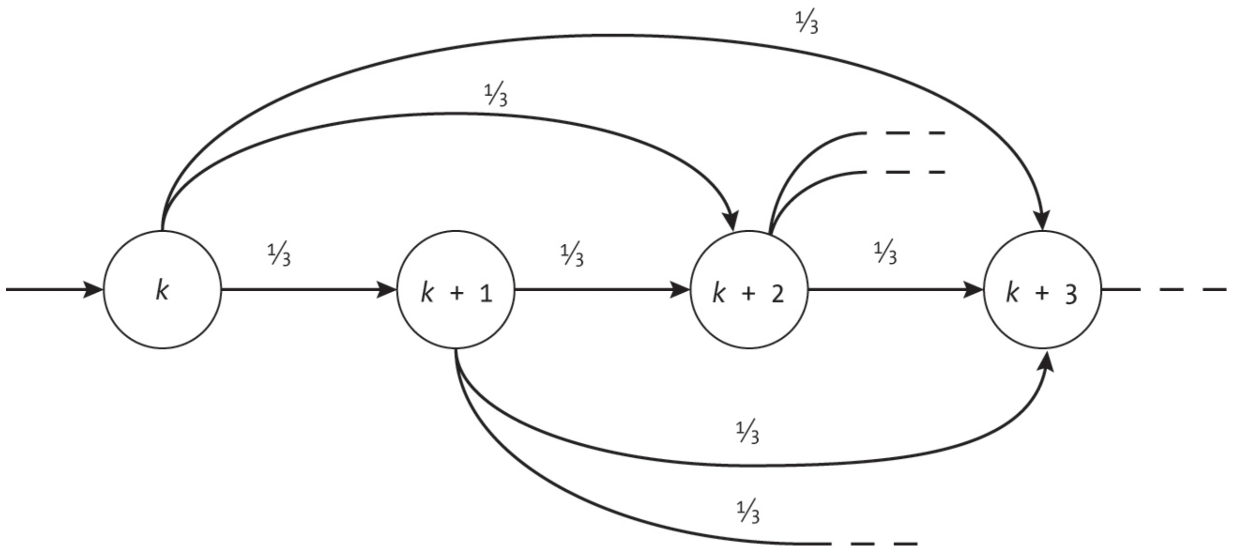


Figure 13-8: A graph representing sample states of a spinner-based game

In an attempt to make the game more exciting (or possibly to cause more children to cry in frustration), the creators label some of the squares "Go back X spaces." These represent traps that should be avoided at all costs. We can incorporate this behavior directly into our graph.

Figure 13-9 shows the state graph where square *k* + 2 contains the instructions "Move back one space." This effectively removes node *k* + 2 from the graph (represented in the figure by graying out the node). It is not possible to finish a turn in that state. The transition probabilities of the neighboring nodes change correspondingly. The probability of going from node *k* to *k* + 1 increases from 1/3 to 2/3, because now spins of both 1 and 2 will end on square *k* + 1. Similarly, a spin of 1 from node *k* + 1 will land the player back on node *k* + 1. We model this as a self-loop with probability 1/3.

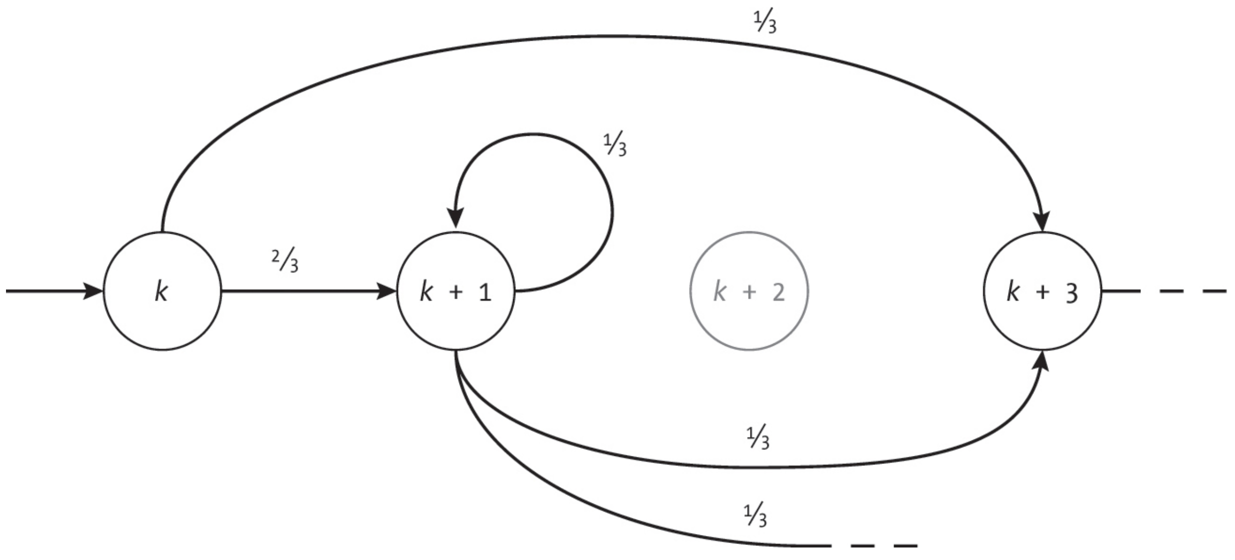


Figure 13-9: A graph representing the movements from state k *in a spinner-based game*

While we can add more edges to capture square-based and spinner-based transitions, significantly more complexity is required to capture aspects such as player-player interactions. The graph model presented so far in this section captures only the dynamics of a single random walk through the board game. This works fine if all players are independent. However, if the game provides the ability to knock a player back two squares when someone lands on their square, we need a model that incorporates both players' positions.

We can create such models by increasing the number of nodes to match the available states of the board game. Instead of N nodes for N squares, we could use $2N^2$ nodes by modeling the game state as a tuple of Alice's location (player 1's state), Bob's location (player 2's state), and a Boolean indicating whether it is Alice's turn. For example, the tuple `(5, 4, False)` indicates that Alice is on square 5, Bob is on square 4, and it is Bob's turn. If we combine the three-option spinner with the new "knock-back" rule, there is a 1/3 probability that Bob will spin a 1, move forward a square, and knock Alice back two squares. More formally, $p($`(5, 4, False)` $\rightarrow$ `(3, 5, True)`$)$ = 1/3.

Transition Probabilities

Beyond analyzing given graphs, we can extend the concepts in this chapter to *estimating* the graphs themselves from observed data. Imagine that we've found the logbook of a tourist who spent the last year randomly wandering the streets of an unknown city. Each entry lists at least a location and a time. Eager to understand their journey, we consider their long sequence of visited locations.

We can easily reconstruct individual nodes from the location names, such as Integer Square and Floating Point Harbor. With a little reasoning, we can also identify the existence of edges; for example, the transition from Integer Square to If-Then Intersection implies an edge connects them. After hours of studying the tourist's agonizingly looping path, we step back and try to reconstruct their transition matrix.

Maximum Likelihood Estimations

We can estimate a transition matrix using a sequence of observations that result from one or more walks to statistically estimate transition probabilities. A *maximum likelihood estimation* allows us to find the model parameters (transition probabilities) that maximize the chance that we will see the sampled data given those parameters. We won't discuss the mathematical details of this approach here, but in short, a maximum likelihood estimation uses the independence of each step to compute the transition probability from node u to node v by counting the following two quantities:

$\boldsymbol{N_u}$ The number of times node u appears in the data at the start of a move (not the last node in the path)

$\boldsymbol{N_{u \to v}}$ The number of times node v appears immediately after node u in the data

Given this information, we compute the probability of a transition as follows:

$$p(u \to v) \approx N_{u \to v} / N_u$$

If we have departed 100 times from node 1, and 30 of those times we moved directly to node 3, we estimate $p(1 \to 3) = 30 / 100 = 0.3$.

A Transition Matrix Estimation Algorithm

The algorithm to estimate an underlying graph from observational data consists of three phrases. In the first phase, we use the nodes visited along the walks to compute the number of nodes for the graph. This corresponds to scanning through the tourist's logbook and determining how many intersections we will need to track. Second, we construct the count array for the number of times the tourist visited each node (N_u) and the count matrix for the node-to-node transitions ($N_{u \to v}$). We compute the counts by iterating over the steps in the walk, effectively retracing the tourist's journey. Finally, we build the graph by inserting edges for all nonzero node-to-node transitions.

Because we are constructing the graph from nodes in the walk, we include only those nodes that the algorithm visits at least once. This means that, given a disconnected graph, we would capture only the graph that is reachable from the initial starting node(s). If the tourist is frightened of water and refuses to cross bridges, we might miss all of the locations on the other side of the river.

Listing 13-4 reflects these three phases.

```
def estimate_graph_from_random_walks(walks: list) -> Graph:
    num_nodes: int = 0
  ❶ for path in walks:
        for node in path:
            if node >= num_nodes:
                num_nodes = node + 1

    counts: list = [0.0] * num_nodes
    move_counts: list = [[0.0] * num_nodes for _ in range(n
um_nodes)]
  ❷ for path in walks:
        for i in range(0, len(path) - 1):
            counts[path[i]] += 1.0
            move_counts[path[i]][path[i + 1]] += 1.0

    g: Graph = Graph(num_nodes)
  ❸ for i in range(num_nodes):
        if counts[i] > 0.0:
            for j in range(num_nodes):
              ❹ if move_counts[i][j] > 0.0:
                    g.insert_edge(i, j, move_counts[i][j] /
counts[i])
    return g
```

Listing 13-4: Estimating a transition matrix from observed data

The function takes in a list of lists (`walks`) that contain the nodes visited on multiple random walks. The code starts by using a pair of nested `for` loops to iterate over each walk and each node in that walk and records the highest index seen ❶. It then allocates data structures for the two sets of counts (`counts` and `move_counts`) and fills these using a `for` loop to iterate over each step in the path ❷. Once it has completed the counts, the code creates a graph (`g`) and iterates over each pair of nodes with two nested `for` loops ❸. If the transition count between the two nodes is nonzero ❹, it computes the probability and inserts the corresponding edge into the graph.

Keep in mind that using the highest index seen has the downside that if the underlying graph consists of multiple disconnected components, the re-created graph will include indices that we will never visit. Since these indices have a count of zero, they will not be given any outgoing edges, and the resulting graph will fail the `is_valid_probability_graph()` check. We use the maximum index approach in Listing 13-4 for simplicity and consistency with previous chapters, but a more robust approach is to use a *node name* to index mapping, as described in Appendix A.

As an example of estimating the graph, consider a travel journal that consists of two paths and contains visits to numbered buildings:

```
path1 = [0, 1, 0, 0, 1, 0, 0]
path2 = [0, 1, 0, 1, 0, 0, 0]
```

We can feed these paths into our estimation function to learn about our tourist's vacation behavior:

```
g = estimate_graph_from_random_walks([path1, path2])
```

A few points immediately jump out when looking at the paths. First, the tourist always started from node 0. Second, they visited only two buildings, 0 and 1. In this case, building 0 is the hotel with a café in the lobby, while building 1 is the coffee shop across the street.

As we compute the maximum likelihood estimate, we accumulate the following values:

$$N_0 = 8, N_1 = 4$$

$$N_{0 \to 0} = 4, N_{0 \to 1} = 4, N_{1 \to 0} = 4, N_{1 \to 1} = 0$$

The tourist started a move from node 0 eight times ($N_0 = 8$). Four of those times they stayed at node 0 ($N_{0 \to 0} = 4$), and four times they went to node 1 ($N_{0 \to 1} = 4$). They started moves from node 1 four times, always going to node 0 ($N_{1 \to 0} = 4$) and never to node 1 ($N_{1 \to 1} = 0$).

We use these statistics to estimate the pairwise transition probabilities:

$p(0 \to 0) = 4 / 8 = 0.5$

$p(0 \to 1) = 4 / 8 = 0.5$

$p(1 \to 0) = 4 / 4 = 1.0$

$p(1 \to 1) = 0 / 4 = 0.0$

These probabilities provide the edge weights for the estimated graph, as shown in Figure 13-10. Note that we do not include the edge (1, 1) because it has a weight of 0.

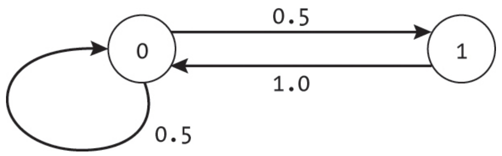


Figure 13-10: A graph with edge transition probabilities estimated from data

When at the hotel, the tourist has a 50 percent chance of staying at the hotel and a 50 percent chance of going across the street to the coffee shop. However, when at the coffee shop, they always return directly to the hotel.

Limitations of Working with Finite Data

Of course, our estimates of transition probability are likely to be very noisy until we accumulate significant observations. If we have a big enough graph, we might never have a chance to observe certain nodes or low-probability edges. This is a fundamental problem of working with random data. If our tourist has a 5 percent chance of turning down a narrow alley, they might reasonably skip it each of the 10 times we see them at its entrance.

Using statistics, we can analyze not only the maximum likelihood values but also their error bars and our confidence levels. These analyses are outside the scope of this book. For now, we just give a word of caution against trying to draw rigorous conclusions from a small amount of random data.

Random Starting Nodes

We can extend both our model and its estimation to account for cases where the random walk starts at different nodes. The idea of a *random starting node* may not seem intuitive since physical walks start at a single location such as the tourist's hotel. However, randomized starts can represent a variety of real-world phenomena, such as the tourist randomly leaving one of many hotels or a rumor starting at a random point in a social network.

Let's label the vector of starting probabilities S where $S[u]$ indicates the probability that we start our random walk at node u. Since S contains probabilities, we restrict $0 \leq S[u] \leq 1$ and $\sum_u S[u] = 1$.

Choosing a Random Starting Node

We can directly sample the starting node with the same approach we used for `choose_next_node()` in Listing 13-3:

```
def choose_start(S: list) -> int:
 ❶ prob: float = random.random()
    cumulative: float = 0.0

 ❷ for i in range(len(S)):
        cumulative += S[i]
        if cumulative >= prob:
            return i
    return len(S) - 1
```

The `choose_start()` function draws a random number from [0, 1) using Python's `random` library ❶. Instead of iterating over a node's edges as in `choose_next_node()`, the function iterates over the values in S until it finds the correct one ❷.

As a more detailed example, imagine an evil wizard who creates dungeons that use randomized entryways to prevent former adventurers from writing strategy guides (thereby messing with the adventurers' retirement plans). The wizard teleports each new arrival to a random location using a carefully chosen distribution S. They restrict adventurers from ever starting their quest in the treasure room by setting that room's starting probability to zero, $S[treasure] = 0$. To reduce the ability for adventurers to share information, the wizard also enforces $S[room] < 0.1$ for all rooms, meaning that there is less than a 10 percent chance the adventurers will start in any particular room.

Estimating the Probability Distribution for Starting Nodes

If we run many random walks, we might be interested in estimating the distribution over starting states. We can use a similar approach to that of the transition probabilities to estimate the starting probabilities. If $N_0[u]$ is the number of walks that start at node u, then we define the probability of starting at node u as the fraction of times a walk started from that node:

$$S[u] = N_0[u] \ / \sum_v N_0[v]$$

We can implement this estimation in the following code:

```
def estimate_start_from_random_walks(walks: list) -> list:
    num_nodes: int = 0
  ❶ for path in walks:
        for node in path:
            if node >= num_nodes:
                num_nodes = node + 1
    counts: list = [0.0] * num_nodes

  ❷ for path in walks:
        counts[path[0]] += 1.0

    for i in range(num_nodes):
        counts[i] = counts[i] / len(walks)
    return counts
```

Like the code for estimating the transition probabilities, the code to estimate the starting probabilities starts by computing the maximum node index and creating a data structure to track occurrences ❶. In this case, however, the code looks only at the first node in the walk. It uses a single `for` loop over the different walks to count how many of the walks start at each node ❷. It then computes the empirical probability of starting at each node.

Our adventurer could theoretically use this approach to gather information for their upcoming strategy guide. They enter the dungeon a few hundred times, carefully tracking where they land each time. It is up to them to decide whether the payoff of writing a comprehensive guide to this dungeon is worth the hassle of navigating through the same dungeon over and over again.

Why This Matters

Analyzing random walks on graphs is useful for both understanding the structure of a graph and analyzing its underlying system. More importantly, however, the concept of random walks extends the range of real-world problems we can model with graph algorithms. We can move beyond deterministic questions, such as how to find the shortest path between two nodes, to account for more realistic behaviors. For example, to model path planning with occasional wrong turns, we could use a random walk that takes the optimal path most of the time but makes a random error at some intersections.

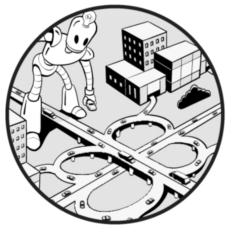
In the next chapter, we switch topics and consider the graphs from a perspective of overall capacity, finding the maximum flow through a network to model systems from plumbing to transportation.

PART IV

MAX FLOW AND BIPARTITE MATCHING

14

MAX-FLOW ALGORITHMS



Earlier chapters have demonstrated various ways to use graphs to model connectivity and transportation problems. This chapter considers the *overall capacity* of networks and how things can flow through them. Imagine we want to model the amount of water that can flow through a network of pipes. We can use edge weights to represent how much water can flow between any two nodes, allowing us to determine the maximum capacity of the entire network.

The *maximum-flow problem* seeks to determine how much flow a graph can support when given edges with limited capacity. This phrasing is intentionally general. We could be modeling the flow of water through a pipe, the flow of people through a transportation network, or the flow of information through a social network. Each application brings its own terminologies, measurements, and units. However, they all boil down to the same fundamental question.

In this chapter, we consider the task of computing the maximum flow on a directed, weighted graph, using the *Ford-Fulkerson* and *Edmonds-Karp* algorithms. Along the way, we show how to extend the `Graph` and `Edge` data structures on which we've relied in previous chapters to account for dynamic usage of capacity through the edges.

The Maximum-Flow Problem

Given a graph with weighted edges that represent the directionality and capacity of flow between two adjacent nodes, how do we determine the maximal flow through the network? We call the node from which the flow originates the *source node* and label it *s*. We call the destination node of the flow the *sink node* and label it *t*. We use *capacity*(*u*, *v*) to indicate the capacity of an edge from *u* to *v*—that is, the maximum amount of flow an edge can support. We indicate the flow through that edge with *flow*(*u*, *v*). The total flow through the network is the amount of flow leaving the source (or, equivalently, the amount entering the sink).

We can visualize the max-flow problem in the context of a wastewater processing system outside of a city. Imagine that wastewater flows from the city via a single source pipe and into the sewage treatment plant via a single sink pipe. In between the source and sink, the wastewater travels through pipes of various sizes, with its flow dividing and recombining at individual nodes. The capacity of the pipes dictates how much wastewater can flow through them.

To model realistic behavior, the maximum-flow problem imposes several constraints. The first is that the source node (the city) only has flow out and the sink node (the sewage treatment plant) only has flow in. In mathematical terms:

$$\mathit{capacity}(u, s) = 0 \text{ for every node } u$$

$$\mathit{capacity}(t, v) = 0 \text{ for every node } v$$

This corresponds to the very reasonable constraints that no wastewater may flow back to the city through the source pipe, and nothing may flow out from the sewage treatment system.

The second constraint is that the flow through an edge (pipe) cannot be less than zero nor can it be more than the edge's capacity. In mathematical terms:

$$0 \leq \mathit{flow}(u, v) \leq \mathit{capacity}(u, v) \text{ for any pair of nodes } u \text{ and } v$$

The upper bound translates to the physical constraints of the pipe. If we try to push too much water through a pipe, it will burst. Nobody wants that. The lower bound of zero indicates a directionality of the pipe, such as one-way valves to prevent flow back through the pipe.

The final constraint is that for all nodes except the source and the sink, the amount of flow into the node must equal the amount of flow out. Mathematically, this means that for every node *u*:

$$\sum_v flow(v, u) = \sum_v flow(u, v)$$

This constraint prevents invalid situations at the node, where water is magically appearing or disappearing at the pipes' junctions.

Until the last few sections of this chapter, we impose an additional constraint that will help us reason about maximum-flow algorithms. We disallow *anti-parallel edges*, pairs of directed edges between the same nodes in opposite directions. In practical terms, this means that if there is an edge from node *u* to node *v*, we do not allow an edge from node *v* to node *u*. This restriction helps simplify the definition of a *residual network,* discussed in a later section. We will relax this restriction toward the end of the chapter.

Figure 14-1 is an example of the maximum-flow problem on a small graph. The edge weights in Figure 14-1(a) represent capacity. To compute the total flow from a source of node 0 to a sink of node 3, we can add up the flow along each path. Figure 14-1(b) shows a configuration with maximum flow. Along the top path, we can send 5 units of flow from node 0 to node 1. The edge from node 1 to node 3 can take even more, but that does not help us. We cannot get more than 5 units of flow to node 1, so we cannot have more than 5 units of flow out. Therefore, the maximal flow along the top path is 5.

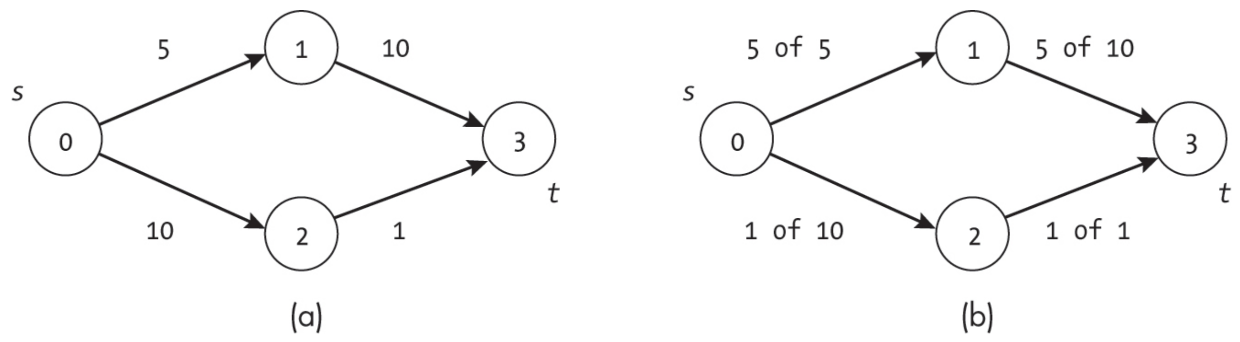


Figure 14-1: A graph with capacities (a) and the maximum flow along the graph's edges (b)

Similarly, Figure 14-1(a) shows a pair of restrictions along the graph's bottom path. While the edge from node 0 to node 2 looks promising with its capacity of 10, we will not be able to push that amount *out* of node 2. The edge (2, 3) presents a severe bottleneck with a capacity of 1. Like a large water pipe that transitions to a small one, the combination of edges limits the overall capacity of the bottom path to 1 and the total flow in the network to 6.

The maximum-flow problem gets significantly more complicated for larger graphs. Consider what happens in Figure 14-2 when we add a single new edge from node 2 to node 1 with capacity 7. Perhaps upset by constant sewer backups, the government builds a new pipe from node 2 to node 1. The edge (2, 1) presents an alternative path for the flow out of node 2. Up to 7 units of flow can split off and take edge (2, 1), while 1 unit continues to use (2, 3).

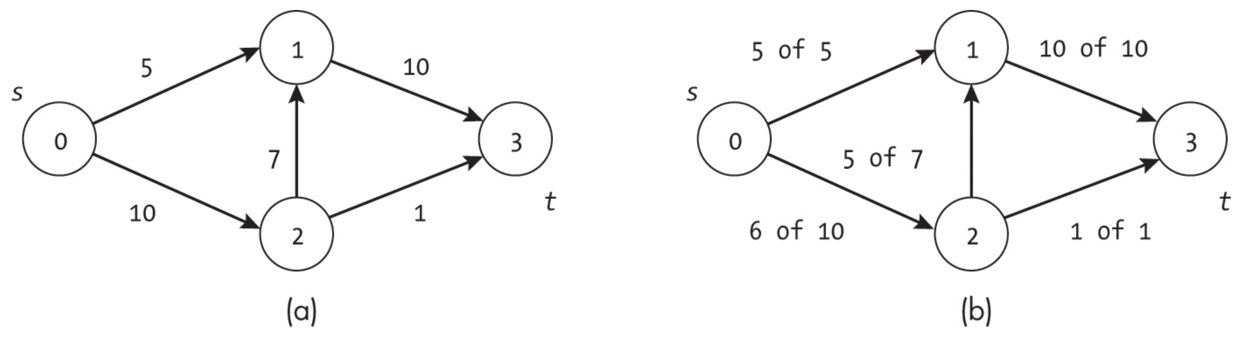


Figure 14-2: A second example graph with capacities (a) and the maximum flow along those edges (b)

However, we need to ensure that the new path through node 1 can handle this additional flow. We already have 5 units of flow from node 0 to node 1. Since the edge (1, 3) has capacity 10 and we are using 5, it has only 5 units of capacity remaining. Despite building a shiny new edge of capacity 7, we can send only 5 more units of flow through our network.

Use Cases

The maximum-flow problem naturally mirrors a range of real-world phenomena, including the flow of liquid through pipes, people through a transportation network, or information through a social network.

Physical Pipelines

Much of the terminology of the maximum-flow problem stems from the physical phenomenon of substances flowing through pipes. The terms *source*, *sink*, *capacity*, and even *flow* mirror their physical counterparts. We can easily map these types of physical problems to their computational equivalent.

While this chapter's primary running example is the flow of water through a wastewater system, the pipeline analogy goes well beyond sewage or interior plumbing, allowing us to ask additional questions. Perhaps we are interested in the flow of maple syrup through a processing plant. Given a complex series of pipes and nodes, what is the capacity of the overall system? How much liquid can we send through it before risking catastrophic maple syrup processing failure? These initial questions provide the foundation for further analysis and optimization, including answering such follow-up questions as "Where are the bottlenecks in the current system?" or "Where should we expand capacity by adding another pipe?"

Transportation Networks

Transportation networks are also fertile ground for maximum-flow analysis. Imagine that your favorite sporting team has a championship game in a faraway city. Many thousands of local fans want to fly there and attend what can only be called the most important game of the century. The airline can model this demand as a maximum-flow problem to determine how many fans can currently travel between the two cities. Edges are routes between pairs of cities with limited numbers of airline seats that constitute their capacity. The flow through an edge is the number of occupied seats. The local city is the source node from which fans are traveling for this occasion, while the host city is the sink node.

The airline can use this analysis to determine if they should add another flight. If the number of interested fans far exceeds the capacity of their flight schedule, there is more money to be made.

Communication Networks

We can also use the maximum-flow problem to model information passing through a communication or social network. For example, imagine you want to influence another person's decision by strategically passing information through your social network. Perhaps you are trying to convince the hiring manager at your favorite company that you would be an ideal successor to the previous CEO. In hopes of swaying their decision, you start to share stories of your amazing achievements, making you the source node and the hiring manager the sink node.

Unfortunately, the members of your network have limited time and interest in passing such messages. This capacity varies between any two nodes. Maybe two friends meet for coffee each morning and one can pass volumes of information to the other. However, a strained relationship might have limited information transfer capacity. Modeling this situation as a maximum-flow problem can help you determine how much information you can realistically get to the sink node.

Extending the Data Structures

Before introducing our first algorithm, we need to extend the `Edge` and `Graph` data structures to fully model capacities and flows. While the edge weights of the `Graph` data structure cannot model both the capacity limit and how much is used, the max-flow problem requires graphs that capture both a *fixed total capacity* and a *dynamic flow amount*.

In this section, we define two new data structures. The `CapacityEdge` class is based on our `Edge` class with additional support for representing the amount of capacity that is in use. The `ResidualGraph` class is similarly based on the formulation of the `Graph` class, but with additional functionality to track dynamically changing flows through the graph.

Edges with Capacity

To model the max-flow problem, the graph's edges need to be able to store two pieces of information: a fixed total capacity and a dynamic flow amount. We define a `CapacityEdge` class that stores the following information:

`from_node (int)` Stores the node index of the edge's origin

`to_node (int)` Stores the node index of the edge's destination

`capacity (float)` Stores the edge's total capacity

`used (float)` Stores the amount of the edge's capacity that is being used

We replace the single weight value in an edge with the combination of `capacity` and `used`. Figure 14-3 shows a visualization of these attributes in the context of flow, where `capacity` is the width of the pipe and `used` is the amount occupied.

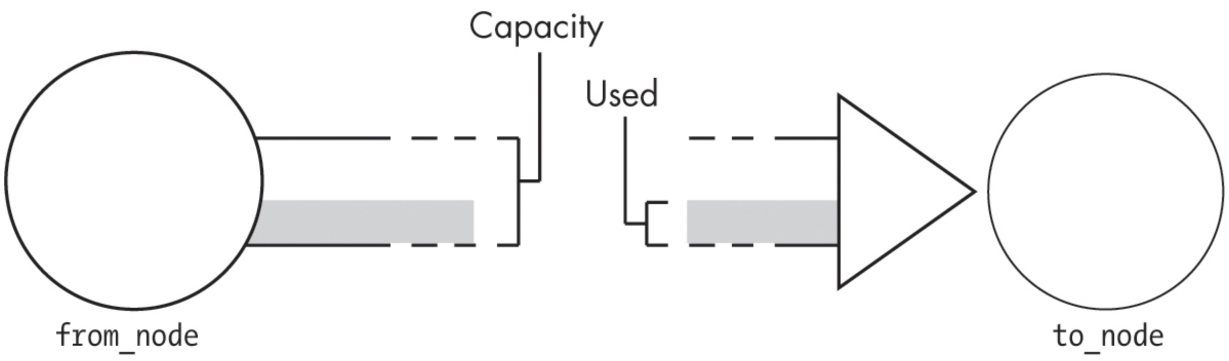


Figure 14-3: The attributes of a `CapacityEdge` *object*

Unlike the `Edge` data structures we defined in Chapter 1 and have used throughout the book, `CapacityEdge` objects provide both storage and functions to operate on that storage, as shown here:

```
class CapacityEdge:
    def __init__(self, from_node: int, to_node: int, capaci
ty: float):
        self.from_node: int = from_node
        self.to_node: int = to_node
        self.capacity: float = capacity
      ❶ self.used: float = 0.0

    def adjust_used(self, amount: float):
      ❷ if self.used + amount < 0.0 or self.used + amount >
self.capacity:
            raise Exception("Capacity Error")
        self.used += amount

    def capacity_left(self) -> float:
        return self.capacity - self.used

    def flow_used(self) -> float:
        return self.used
```

The constructor initializes the object's variables, setting `used` to `0` to indicate that the edges start without any flow ❶. Next, the `adjust_used()` function allows algorithms to modify the flow through the edge. It takes an adjustment amount and adds it to the amount being used. We can visualize the function as a faucet knob. If we turn it one way, by passing in a positive amount, the flow increases. If we turn it the other way, by passing in a negative amount, the flow decreases. Unlike a faucet, however, the function does not automatically "stop turning" when it has reached its limit. The code includes an additional check to ensure the used capacity falls within the limits specified by the edge ❷. Specifically, the flow through an edge can never be less than 0 nor more than the edge's total capacity.

Pushing the faucet analogy further, we may wish for indicators on how much we can turn the faucet in each direction. We provide the function `capacity_left()` to indicate the unused capacity remaining on the edge (also called the *forward residual)*. This is the amount of additional flow we can add to an edge. Similarly, we provide the function `flow_used()` for indicating the current capacity used (also called the *backward residual)*.

Residual Graphs

Just as we needed to add functionality to track how much capacity is used within an edge, we must augment our graph representation to support storage of and computation on these dynamic edges. We also add auxiliary tracking information specific to the max-flow problem, namely the indices of the source and sink nodes. We call this augmented graph a *residual graph* because it tracks the residual (or remaining) capacity between pairs of nodes.

We define a `ResidualGraph` class that uses a more minimal adjacency list representation and contains the following:

`num_nodes (int)` Stores the total number of nodes in the graph.

`source_index (int)` Stores the index of the source node.

`sink_index (int)` Stores the index of the sink node.

`edges (list)` Stores a dictionary for each node containing the adjacent edge objects out of that node keyed by their destination node. To access the `CapacityEdge` from node `j` to node `k`, we use `edges[j][k]`.

`all_neighbors (list)` Stores a set of all in-neighbor and out-neighbor indices for each node.

The `ResidualGraph` representation differs from the `Graph` class in that we are no longer storing `Node` objects. Instead, the same adjacency list information, including the use of a dictionary, is incorporated into the `edges` list. While this presents a more compact representation that is sufficient for max-flow algorithms, we lose the ability to easily store auxiliary data within the nodes that we used for other algorithms.

Although we are working with directed graphs, the algorithms we introduce will need to scan over all neighboring nodes, including in-neighbors that are not included in a traditional adjacency list. To facilitate these computations, we store the additional list `all_neighbors`. Restricting the connection between any two nodes to a single directed `CapacityEdge` (not allowing anti-parallel edges) makes reasoning about forward and backward flows easier. As we will see, this restriction does not diminish the representational power of the graph because we can transform a graph with anti-parallel edges into one without them.

To demonstrate how the `edges` and `all_neighbors` lists capture the structure of the graph, consider the example `ResidualGraph` shown in Figure 14-4, along with its two list data structures. The four-node graph is shown on the left, the corresponding `edges` list in the middle, and the `all_neighbors` list on the right. Node 1 has two outgoing edges (nodes 2 and 3) and thus two entries in its adjacency dictionary `edges[1]`. Each entry in the dictionary `edges[1]` maps the neighbor's index to the corresponding `CapacityEdge` out of node 1. Since node 1 also has an incoming edge from node 0, the set `all_neighbors[1]` contains three indices: 0, 2, and 3.

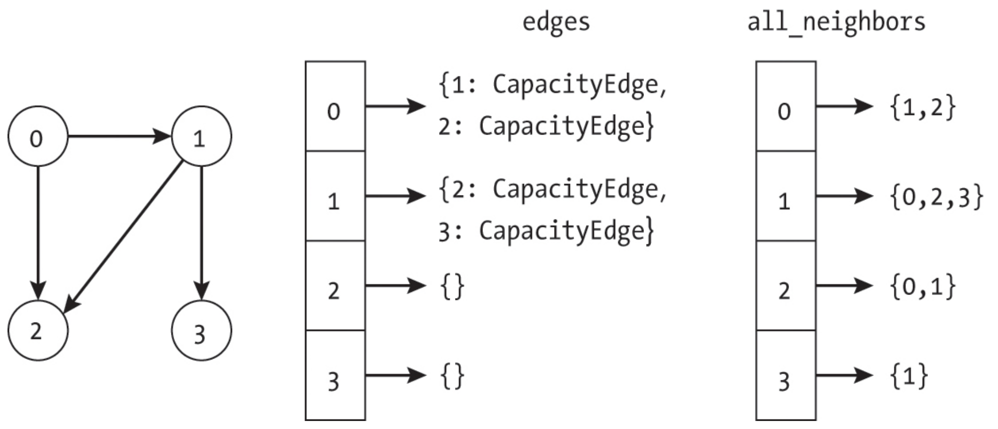


Figure 14-4: A residual graph and its internal list data structures

The `ResidualGraph` class provides functions for creating and operating on this type of graph:

```
class ResidualGraph:
    def __init__(self, num_nodes: int, source_index: int, s
ink_index: int):
        self.num_nodes: int = num_nodes
        self.source_index: int = source_index
        self.sink_index: int = sink_index
        self.edges: list = [{} for _ in range(num_nodes)]
        self.all_neighbors: list = [set() for _ in range(nu
m_nodes)]

    def get_edge(self, from_node: int, to_node: int) -> Uni
on[CapacityEdge,

None]:
        if from_node < 0 or from_node >= self.num_nodes:
            raise IndexError
        if to_node < 0 or to_node >= self.num_nodes:
            raise IndexError
        if to_node in self.edges[from_node]:
            return self.edges[from_node][to_node]
        return None

    def insert_edge(self, from_node: int, to_node: int, cap
acity: float):
      ❶ if from_node < 0 or from_node >= self.num_nodes:
            raise IndexError
        if to_node < 0 or to_node >= self.num_nodes:
            raise IndexError

      ❷ if from_node == self.sink_index:
            raise ValueError("Tried to insert edge FROM sin
k node.")
        if to_node == self.source_index:
            raise ValueError("Tried to insert edge TO sourc
e node.")
        if from_node in self.edges[to_node]:
            raise ValueError(f"Tried to insert edge {from_n
ode}->{to_node}, "
                             f"edge {to_node}->{from_node}
 already exists.")
        if capacity <= 0:
            raise ValueError(f"Tried to insert capacity {ca
pacity}")

     ❸ self.edges[from_node][to_node] = CapacityEdge(from_
node, to_node,
                                                     capac
ity)
     ❹ self.all_neighbors[from_node].add(to_node)
        self.all_neighbors[to_node].add(from_node)

    def compute_total_flow(self) -> float:
        total_flow: float = 0.0
        for to_node in self.edges[self.source_index]:
            total_flow += self.edges[self.source_index][to_
node].flow_used()
        return total_flow
```

The constructor creates an empty graph by creating empty adjacency dictionaries (`edges`) and neighbor sets for all nodes (`all_neighbors`). Next, the `get_edge()` function mirrors the version from the `Graph` class and allows us to access each edge. Much of the code for this function consists of bounds checking: the function raises an `IndexError` if `from_node` or `to_node` is not in the graph. If the edge is not in the graph, the function returns `None`. If the nodes are valid and the edge is in the graph, the code returns the corresponding `CapacityEdge`. The code relies on importing `Union` from the `typing` library to support the type hints for multiple return types.

Because the `ResidualGraph` is both using different structures to store the edges with capacities and adding more neighbor information to track incoming edges, the `insert_edge()` function needs to track this information accordingly. The code starts with the same index validity checking we used in the `Graph` class ❶ and adds checks to ensure the structural constraints we put on the graphs in the max-flow problem ❷. These include checking that (1) there are no edges out of the sink, (2) there are no edges into the source, (3) the newly inserted edge is not the reverse of an existing edge, and (4) the capacity is greater than zero.

If all the checks pass, the code creates a new `CapacityEdge` and adds it to the dictionary in the corresponding entry of the `edges` list ❸. If an edge has already been inserted between these two nodes in the same direction, the code overwrites it. Finally, the code adds `to_node` to `from_node`'s list of all neighbors and adds `from_node` to `to_node`'s list of all neighbors ❹.

The `compute_total_flow()` function demonstrates how to use the values within the `ResidualGraph` to reason about its properties, computing the total flow from source to sink by summing up the flow along each edge leaving the source. Since all flow originates from a single source node, this is the total flow through the graph.

The remaining functions in the `ResidualGraph` class are closely tied to the operation of the Ford-Fulkerson algorithm; we'll present them in context as we introduce the algorithm.

The Ford-Fulkerson Algorithm

Mathematicians L.R. Ford Jr. and D.R. Fulkerson developed a general approach for finding the maximum flow through a graph by repeatedly finding underutilized paths from the source to the sink and increasing the flow along those paths. This approach relies upon the idea of an *augmenting path*, a route from the source node to the sink node along which the flow can be increased. Ford-Fulkerson is technically a general approach that encompasses a range of specific algorithms because the original paper does not specify which search algorithm to use to find the augmenting path. This section introduces an example implementation using depth-first search.

The general Ford-Fulkerson approach may fail to terminate in pathological cases where irrational numbers are used for the capacities. These cases can be avoided by limiting the precision of the capacities or, as we will see later in this chapter, by selecting augmenting paths with the fewest edges.

Defining Augmenting Paths

The simplest form of an augmenting path is a series of directed edges from source to sink whose current flow is less than the edges' capacities. In this case, as illustrated in Figure 14-5(a), we can just add flow along the path [0, 2, 3] to increase the total flow by 2 units. Figure 14-5(b) shows the resulting total of 7 units of flow leaving the source and entering the sink.

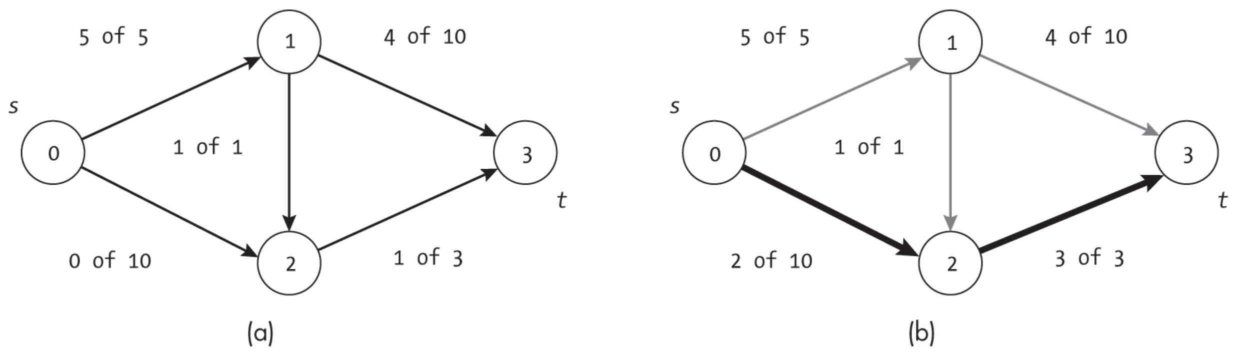


Figure 14-5: A capacity graph before (a) and after (b) adding flow

Adding forward flow gets us only part of the way, however. Figure 14-6 shows a situation where there is no path from source to sink that has unused capacity. This graph is not at maximum flow, since the flow from node 1 to node 2 is siphoning off potential flow from node 1 to node 3. Simultaneously, this flow is contributing to the edge from node 2 to node 3 being fully utilized and thus unable to accept any more flow from edge (0, 2).

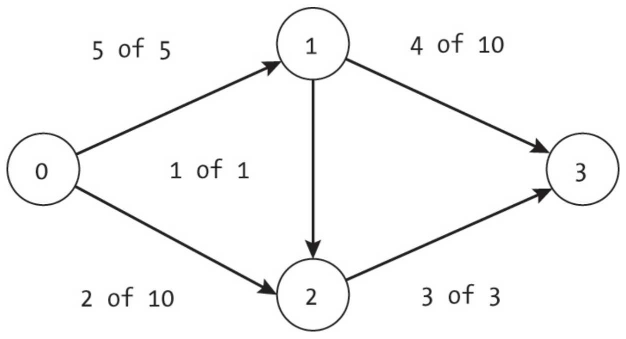


Figure 14-6: A graph with no underutilized forward paths

We can increase the flow of the graph using the two steps shown in Figure 14-7. The 5 units of flow entering node 1 are initially partitioned into two streams, with 1 unit of flow going to node 2 and 4 units going to node 3. We change this allocation to divert 1 additional unit of flow to node 3 as shown in Figure 14-7(a). The total flow through the graph remains constant, but the flow through edge (2, 3) now drops below capacity. Second, we increase the flow from node 0 through node 2 to node 3 as shown in Figure 14-7(b), increasing the overall flow through the graph to 8 units.

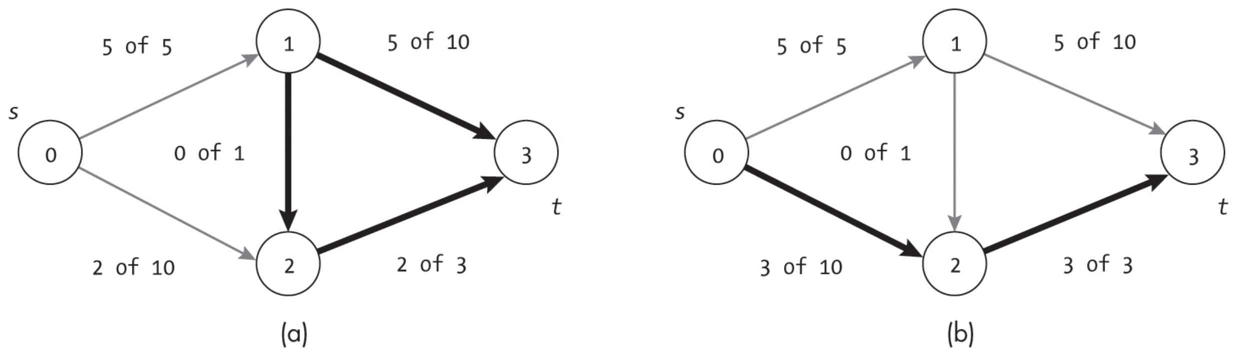


Figure 14-7: Two steps for adding flow to the graph in Figure 14-6

To reroute flow through the network, the algorithm also needs the ability to reduce flow through an edge by diverting it along another edge. We therefore define the *residual* along a directed edge (u, v) as follows. The forward residual is the unused capacity $capacity(u, v) - flow(u, v)$ in the direction from node u to node v. This aligns how we normally think of additional capacity. The backward residual is the used capacity $flow(v, u)$ in the direction opposite the edge—that is, from node v to node u. This corresponds to capacity that can be removed from the input of node u, allowing us to accept input from somewhere else.

We can push more flow through an underutilized directed edge or push flow back in the opposite direction of a directed edge. Figure 14-8 shows an example case of combining forward and backward residuals.

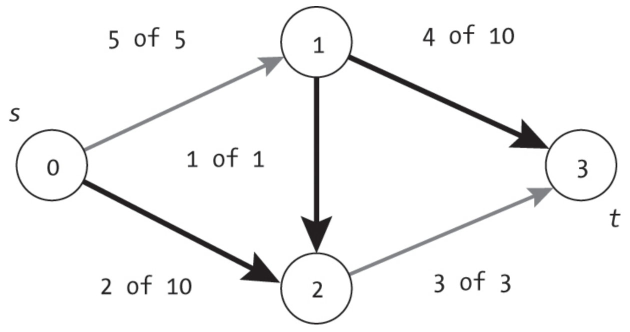


Figure 14-8: An augmenting path that includes reducing flow along an edge

The bolded edges indicate an undirected path from source to sink where we can modify the flows as follows:

- Edge (0, 2) has a forward residual of 8, so we can add more units of flow down that edge to node 2.
- Edge (1, 2) has a backward residual of 1, so we can turn down that flow by 1 unit to allow node 2 to take more flow from another source (in this case, node 0). Since node 2's outgoing flow is capped at 3 units and we need to keep the flow out equal to the flow in, we need to reduce the incoming flow from node 1 into node 2 in order to increase the flow from node 0 into node 2.
- Edge (1, 3) has a forward residual of 6, so it can take extra output of node 1 that is no longer traveling to node 2. Again, we need to keep the flow into node 1 balanced with the flow out of node 1.

The key to understanding the Ford-Fulkerson algorithm is that pushing flow through a backward edge is just reducing the flow leaving the origin node so that it can travel to a new destination. As we will see in the next section, our need to push flow in either direction means it no longer suffices to explore the edges out of each node's (directed) adjacency list. We need to consider edges into and out of the node.

We can visualize this algorithm in the context of a sewage engineer managing the wastewater system described previously. The engineer maximizes the total flow of wastewater by routing the flow through the optimal sets of pipes. The main constraint is the capacity of the pipes (edges) and the junction boxes (nodes). The last engineer, trying to show off, ignored the total capacity and pushed more flow than was viable. The overloaded pipe promptly burst, resulting in a geyser of wastewater that was discussed in the newspapers for weeks.

The new engineer tackles this problem by continuously finding a path from source to sink that can take more wastewater and sending as much wastewater through that path as possible (but no more). Sometimes this means pushing back against an existing flow, which is fine as long as that flow can be pushed through another junction (node) to the sink. Any wastewater flowing into a junction must also flow out. Otherwise, it risks bursting the junction box. The engineer keeps increasing the flow again and again until all the paths are fully saturated.

Finding an Augmenting Path

Before we can define a search algorithm, we need to formalize the computation of the residual along a path. Remember from the previous description that an augmenting path can contain a combination of forward residuals and backward residuals. We define a helper method within the `ResidualGraph` class to simplify the logic of computing the residual (either forward or backward) between any two nodes on the path:

```
def get_residual(self, from_node: int, to_node: int) -> flo
at:
 ❶ if to_node not in self.all_neighbors[from_node]:
        return 0

 ❷ if to_node in self.edges[from_node]:
        return self.edges[from_node][to_node].capacity_left
()
    else:
        return self.edges[to_node][from_node].flow_used()
```

The `get_residual()` function first checks whether the two nodes are connected at all ❶. If not, the edge is neither a forward nor a backward edge and has zero residual. If the edge (`from_node`, `to_node`) is in the adjacency list of directed edges, then it is a forward edge ❷, and the function returns the capacity remaining (forward residual). Otherwise, the edge must exist in the opposite direction, so the code returns the flow used (backward residual).

In this section, we use a modified depth-first search to check the graph for an augmenting path:

```
def find_augmenting_path_dfs(g: ResidualGraph) -> list:
    seen: list = [False] * g.num_nodes
    last: list = [-1] * g.num_nodes
    augmenting_path_dfs_recursive(g, g.source_index, seen,
 last)
    return last

def augmenting_path_dfs_recursive(g: ResidualGraph, curren
t: int,
                                  seen: list, last: list):
```

```
    seen[current] = True
    for n in g.all_neighbors[current]:
     ❶ if not seen[n] and g.get_residual(current, n) > 0:
            last[n] = current
          ❷ if last[g.sink_index] != -1:
                return
            augmenting_path_dfs_recursive(g, n, seen, last)
```

This code consists of a pair of functions. First, the `find_augmenting_path_dfs()` function sets up the lists `seen` and `last` for a depth-first search, as described in Chapter 4. It then calls the recursive depth-first search function and returns the `last` list that represents the augmenting path.

The `augmenting_path_dfs_recursive()` function performs the recursive depth-first exploration. As with a standard depth-first search, it marks the current node as seen, then loops through the node's neighbors. The code uses a `for` loop over the residual graph's `all_neighbors` list to explore along both directions of the directed edges. When exploring the neighbors of the current node, the code checks both that the node has not been seen (as in standard depth-first search) and that the residual is nonzero ❶. This latter condition prevents the search from using edges that are already saturated. If the edge is viable and the node has not been seen, the search updates the tracking information and recursively explores that node.

The code incorporates an optional early termination check ❷. It stops exploring new neighbors as soon as any path has been found from the source to the sink. By checking whether `last[g.sink_index]` has been assigned, the code can skip the recursive exploration at both the node before the sink and earlier nodes along the path.

Figure 14-9 shows the iterations of `find_augmenting_path_dfs()` on the graph from Figure 14-6. Each edge is labeled with the *X* of *Y*, where *X* is the flow used and *Y* is the edge's total capacity. The shaded nodes have been seen, and the node enclosed in the dashed circle is the one on which the recursive function has just been called.

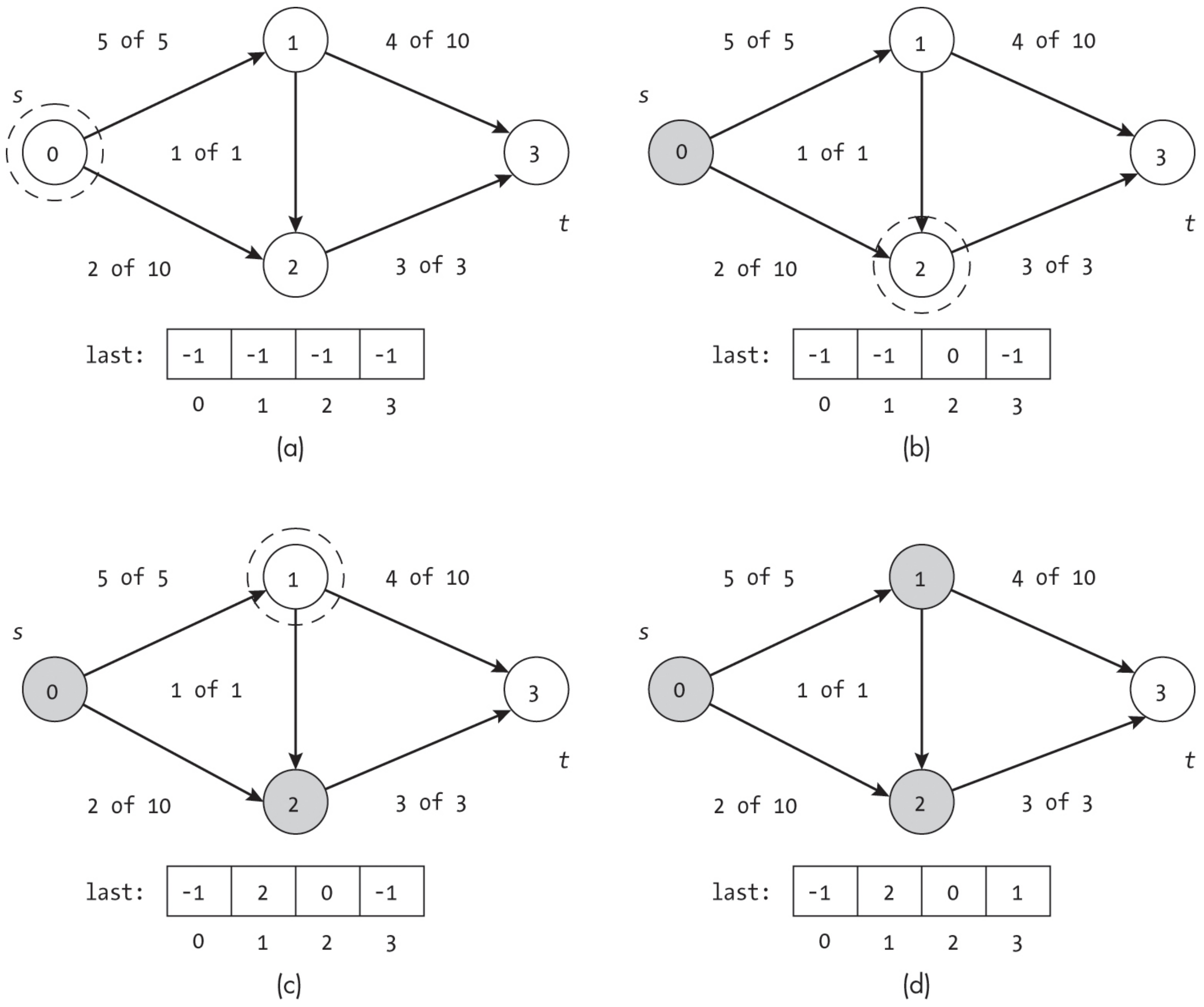


Figure 14-9: The steps of the search to find an augmenting path

Figure 14-9(a) shows the state of the algorithm before the source node is visited. Figure 14-9(b) shows the second step of the search: after visiting node 0, the algorithm finds two neighbors, nodes 1 and 2. Only edge (0, 2) has unused capacity, so the search continues down that branch.

Since the algorithm considers both outgoing and incoming edges, it finds two options at node 2. Both edges (1, 2) and (2, 3) are at capacity in their respective directions. However, edge (1, 2) is incoming to node 2 and thus has a backward residual of 1. This edge provides us an opportunity to decrease the flow into node 2. As shown in Figure 14-9(c), the search follows this edge to explore node 1.

While exploring node 1, the algorithm finds a path to the sink node with unused capacity. The code never visits the sink node, but rather returns as soon as any path is found. In this case, as shown in Figure 14-9(d), the algorithm returns a `last` array of `[-1, 2, 0, 1]`, indicating the augmenting path.

Updating a Path's Capacity

After finding an augmenting path, the Ford-Fulkerson algorithm must determine how much additional flow it can push through the path, then update the path's capacities to indicate the increased flow. To enable this, we add two functions to the `ResidualGraph` class. The `min_residual_on_path()` function traverses a path using the last pointers and computes the minimum residual of any edge along the path:

```
def min_residual_on_path(self, last: list) -> float:
    min_val: float = math.inf

    current: int = self.sink_index
  ❶ while current != self.source_index:
        prev: int = last[current]
        if prev == -1:
            raise ValueError
        min_val = min(min_val, self.get_residual(prev, curr
ent))
      ❷ current = prev
    return min_val
```

The code starts by setting the `min_val` to infinity (requiring the file to include `import math`) as an indicator that there is no minimum yet. It then uses a `while` loop to walk the chain of pointers backward from the sink until it reaches the source ❶. At each step, it considers the node preceding the current one and checks that it is not `-1`, which would indicate a broken path. The code updates the minimum value using `get_residual()` on the current edge and moves on to the previous node ❷. After examining all the edges along the path, the code returns the smallest residual it encountered.

If we apply `min_residual_on_path()` to the result shown in Figure 14-6, with a `last` array of `[-1, 2, 0, 1]`, we traverse the path shown in Figure 14-8. The minimum residual along this path is 1 along edge (1, 2).

Once we have determined how much additional flow we can push through a path, we update the path using the `update_along_path()` function:

```
def update_along_path(self, last: list, amount: float):
    current: int = self.sink_index
  ❶ while current != self.source_index:
        prev: int = last[current]
        if prev == -1:
            raise ValueError

      ❷ if current in self.edges[prev]:
            self.edges[prev][current].adjust_used(amount)
        else:
            self.edges[current][prev].adjust_used(-amount)
        current = prev
```

Like the `min_residual_on_path()` function, the code for `update_along_path()` walks the last pointers backward from the sink node to the source node using a `while` loop ❶. Again, it checks that the previous node indicates a valid path. If so, it checks the direction of the edge along the path before updating the amount used ❷. Forward edges appear in the adjacency list, and the code adds the new flow to the amount of capacity used. Backward edges are ones where the algorithm is pushing flow back. The edge direction is the opposite way, so the edge itself is not in the adjacency list. The code subtracts the new flow from the amount already in use.

Putting It All Together

The Ford-Fulkerson algorithm using depth-first search consists of putting together the pieces we have introduced throughout the chapter. As shown in Listing 14-1, the algorithm repeatedly searches for an augmenting path. When it finds one, it computes the minimum residual along that path and increases the flow accordingly.

```
def ford_fulkerson(g: Graph, source: int, sink: int) -> Res
idualGraph:
  ❶ residual: ResidualGraph = ResidualGraph(g.num_nodes, so
urce, sink)
    for node in g.nodes:
        for edge in node.edges.values():
            residual.insert_edge(edge.from_node, edge.to_no
de, edge.weight)

  ❷ done = False
    while not done:
      ❸ last: list = find_augmenting_path_dfs(residual)
      ❹ if last[sink] > -1:
            min_value: float = residual.min_residual_on_pat
h(last)
            residual.update_along_path(last, min_value)
        else:
            done = True

    return residual
```

Listing 14-1: The Ford-Fulkerson algorithm using depth-first search

The code starts by creating a `ResidualGraph` where the capacities are equal to the weights of the original graph ❶. This effectively copies the graph while also transforming the representation.

The main loop of the algorithm is relatively small and begins by using a Boolean `done` to track whether it found an augmenting path on the last iteration ❷. If so, `done` will be `False` and the code searches for a new augmenting path ❸. The code checks that the returned path is valid ❹ and, if so, computes the minimum residual along the path using the `min_residual_on_path()` function and updates the flow along the path using the `update_along_path()` function. If the code finds a `last` value of –1 for the sink node, it knows there is no path from the source to the sink and can set `done` to `True`. The function finishes by returning the residual graph.

An Example

Figure 14-10 shows the Ford-Fulkerson algorithm running on an example graph, where each subfigure represents the state of `ResidualGraph` after one iteration of the algorithm. Bolded arrows indicate the augmenting path found during each iteration, and the used portions of the capacities have been updated to fully use that augmenting path, as shown by the *X* of *Y* notation along each edge.

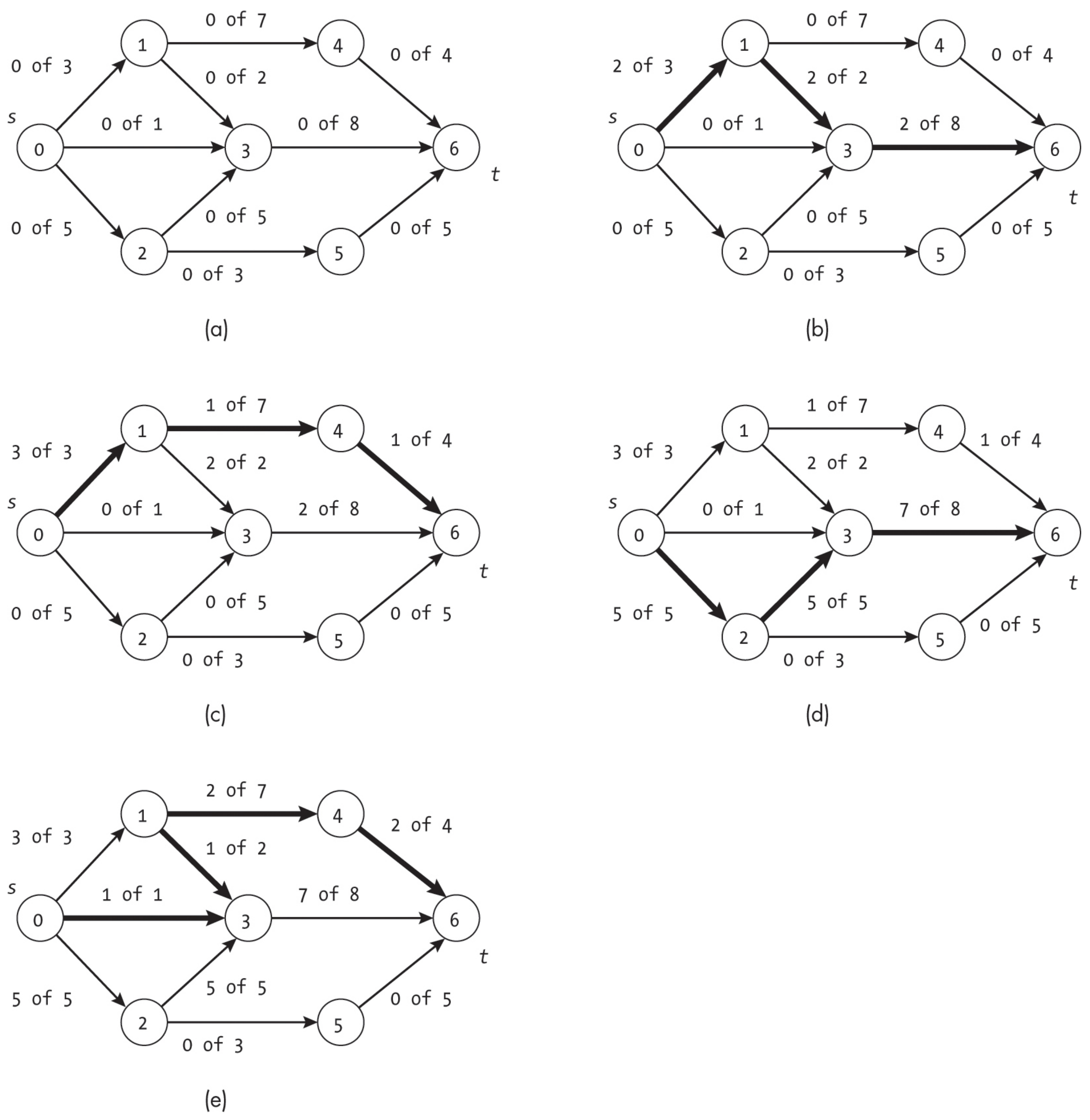


Figure 14-10: The steps of the Ford-Fulkerson algorithm with depth-first search on a graph with seven nodes

Figure 14-10 demonstrates how using depth-first search impacts the order in which the Ford-Fulkerson algorithm retrieves augmenting paths. For example, although there is a capacity-3 path along the bottom of the example graph via edges (0, 2), (2, 5), and (5, 6), the algorithm first fills in some smaller flows, such as the capacity-1 path in Figure 14-10(c).

Figure 14-10(e) shows an augmenting path that uses both forward and backward residuals. To increase the flow through edge (4, 6) into the sink, the algorithm redirects flow out of node 1 from edge (1, 3) to edge (1, 4). This gives node 4 an input of 2 units that the algorithm can pass along to the sink. However, it leaves node 3 short by 1 unit. The search offsets this loss of input at node 3 with an additional flow from the source through edge (0, 3).

Once the algorithm has computed the `ResidualGraph`, we can use that data structure to answer other questions. For example, we can use the `compute _total_flow()` function to compute the graph's maximum flow.

The Edmonds-Karp Algorithm

The computer scientist Yefim Dinitz (under the name E.A. Dinic) and the pair of computer scientists Jack Edmonds and Richard M. Karp independently published analyses of the Ford-Fulkerson algorithm that selected augmenting paths with the fewest number of edges. This approach, now called either the *Dinitz algorithm* or the *Edmonds-Karp algorithm*, makes use of this path selection to avoid the problematic behavior when using irrational edge capacities and thus bounds the number of iterations of the algorithm in all cases. This section shows how we can use breadth-first search to find such augmenting paths.

The Code

The majority of the Edmonds-Karp algorithm uses the functions introduced earlier for the Ford-Fulkerson algorithm, such as `update_along_path()` and `min_residual_on_path()`. All we need to change is the function for finding the augmenting paths and the outer function that calls it.

We use a modified version of breadth-first search to find the augmenting paths:

```
def find_augmenting_path_bfs(g: ResidualGraph) -> list:
    seen: list = [False] * g.num_nodes
    last: list = [-1] * g.num_nodes
    pending: queue.Queue = queue.Queue()

  ❶ seen[g.source_index] = True
    pending.put(g.source_index)
  ❷ while not pending.empty() and not seen[g.sink_index]:
        current: int = pending.get()
        for n in g.all_neighbors[current]:
         ❸ if not seen[n] and g.get_residual(current, n) >
0:
             ❹ pending.put(n)
                seen[n] = True
                last[n] = current

    return last
```

The `find_augmenting_path_bfs()` function starts by setting up the standard breadth-first search data structures, including the list of whether each node has been seen (`seen`), the list of previous nodes on the path (`last`), and the queue of nodes to explore (`pending`). The use of the `Queue` data structure requires an additional `import queue` at the top of the file. The function then inserts the source node as the starting point ❶. The main `while` loop continues until either the pending queue is empty or the sink node has been seen ❷. As in the depth-first search code, this second check allows the search to terminate as soon as it finds *any* path from the source to the sink.

When exploring the neighbors of the current node, the code checks both that the node has not been seen (as in standard breadth-first search) and that the residual is nonzero ❸. If the edge is viable and the node has not been seen, the search updates the tracking information and adds it to the queue ❹.

Figure 14-11 shows the iterations of `find_augmenting_path_bfs()` on a graph with some of its capacity used. The shaded nodes have been seen, and the node enclosed in the dashed circle is the one the search has just *finished* processing.

Figure 14-11(a) shows the state of the algorithm before the `while` loop starts, while Figure 14-11(b) shows the first step of the search. After visiting node 0, we find edges with unused capacity to two unvisited neighbors (nodes 1 and 3). Both neighbors are added to the `pending` queue.

The algorithm diverges from a standard breadth-first search in Figure 14-11(c). Although node 2 is a neighbor of node 1, the edge (1, 2) is already full. We cannot send any more flow through that edge, so the algorithm rules out paths using that edge and keeps node 2 as unseen. It is not until Figure 14-11(d) that it finds a viable route to node 2 (from node 3).

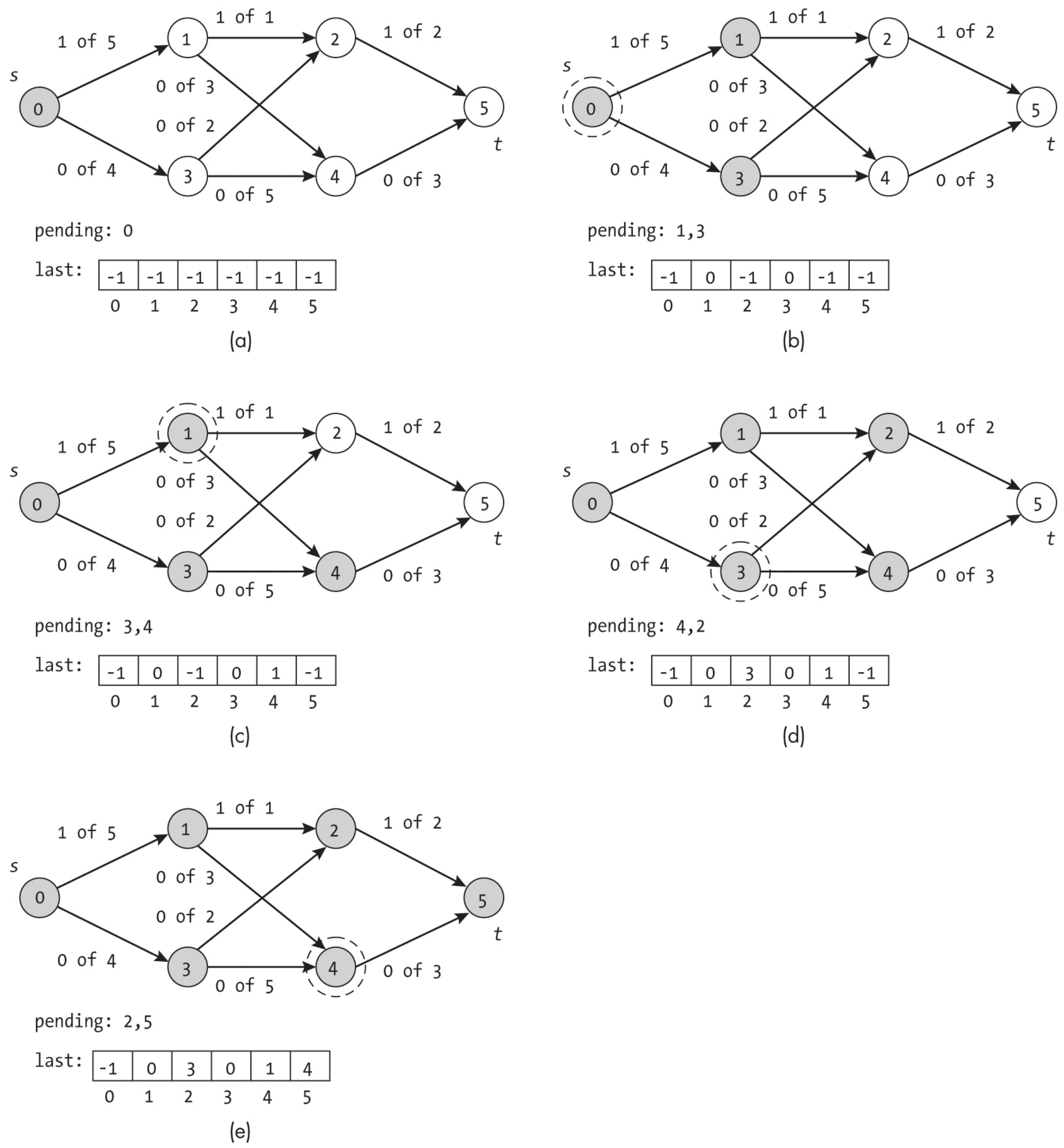


Figure 14-11: The steps of the search to find an augmenting path

The search completes in Figure 14-11(e) after finding a path to the sink node. At this point, it has found a viable path [0, 1, 4, 5] from the source to the sink and need not consider any other nodes.

The code for the top-level Edmonds-Karp algorithm is nearly identical to the depth-first search version of Ford-Fulkerson from Listing 14-1:

```
def edmonds_karp(g: Graph, source: int, sink: int) -> Resid
ualGraph:
    residual: ResidualGraph = ResidualGraph(g.num_nodes, so
urce, sink)
    for node in g.nodes:
        for edge in node.edges.values():
            residual.insert_edge(edge.from_node, edge.to_no
de, edge.weight)

    done = False
    while not done:
      ❶ last: list = find_augmenting_path_bfs(residual)
        if last[sink] > -1:
            min_value: float = residual.min_residual_on_pat
h(last)
            residual.update_along_path(last, min_value)
        else:
            done = True
    return residual
```

The only significant change from Listing 14-1 is the use of the function `find_augmenting_path_bfs()` to conduct the search for an augmenting path ❶.

An Example

Figure 14-12 shows an example of running the Edmonds-Karp algorithm on a graph with 8 nodes and 11 edges, where node 0 is the source node and node 7 is the sink node. Figure 14-12(a) represents the state of the `ResidualGraph` before the first iteration. None of the edges' capacities are used. Each subsequent step of the algorithm is depicted *after* each augmenting path is updated; the bolded edges indicate the augmenting path used. For example, in Figure 14-12(b), the edges (0, 1), (1, 2), and (2, 7) form the augmenting path. The minimum residual was 3, and the subfigure shows the amount of used capacity after 3 more units of flow were added to this path.

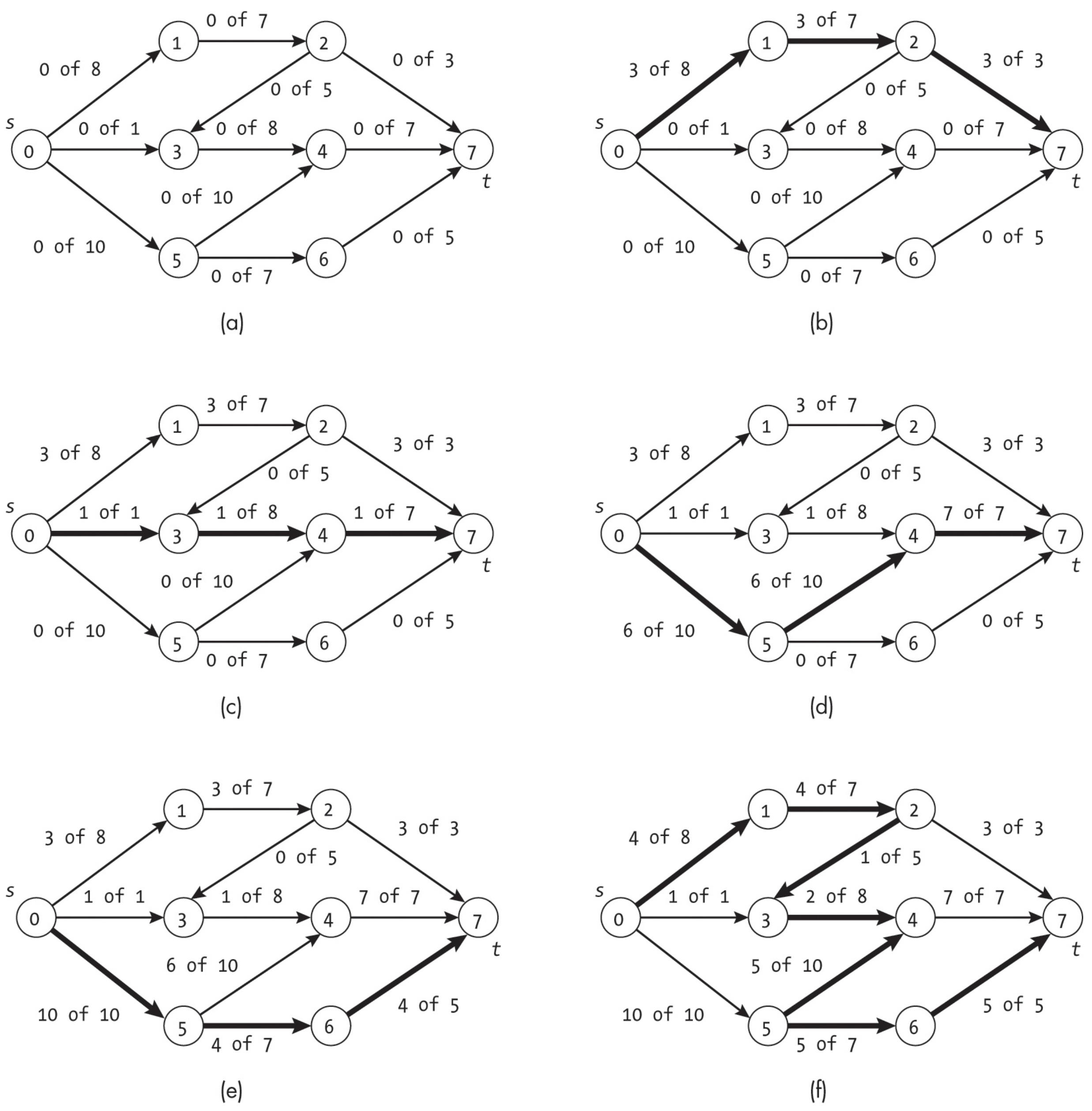


Figure 14-12: The steps of the Edmonds-Karp algorithm on a graph with eight nodes

Figure 14-12(f) shows a step where the algorithm uses the backward residual. After four rounds of following only forward edges, the search has hit a bottleneck and *reduces* the flow on the edge from node 5 to node 4 to free up more capacity. To understand how this helps, consider the flow from node 0 to node 5. Before the last step, it is already maxed out. The edge cannot handle more than 10 units of flow. However, that flow is not being used optimally. By reducing the flow from node 5 to node 4, we can send more of that flow through node 6 to the sink. This leaves node 4 with less incoming flow than outgoing flow. To fix this disparity, we need to push more flow through an alternate path. In this case, the extra unit of flow comes to node 4 through the path [0, 1, 2, 3, 4].

Modeling Increasingly Complex Real-World Situations

Our maximum-flow algorithm placed restrictions on the structure of the graph to simplify reasoning about the algorithm. These constraints included limiting the graph to a single source and a single sink and prohibiting anti-parallel edges. This section examines how we can relax several of these limitations to model increasingly complex real-world situations.

Multiple Sources

Many real-world flow networks contain more than a single source node. For example, consider the more realistic view of the wastewater problem we have been using throughout this chapter. Rather than a single incoming pipe, it's far more likely that the network will include pipes entering from each building connected to the system. Even if we model at the city level, we can expect new sources to join the network from the surrounding suburbs. Figure 14-13(a) shows a network with three source nodes.

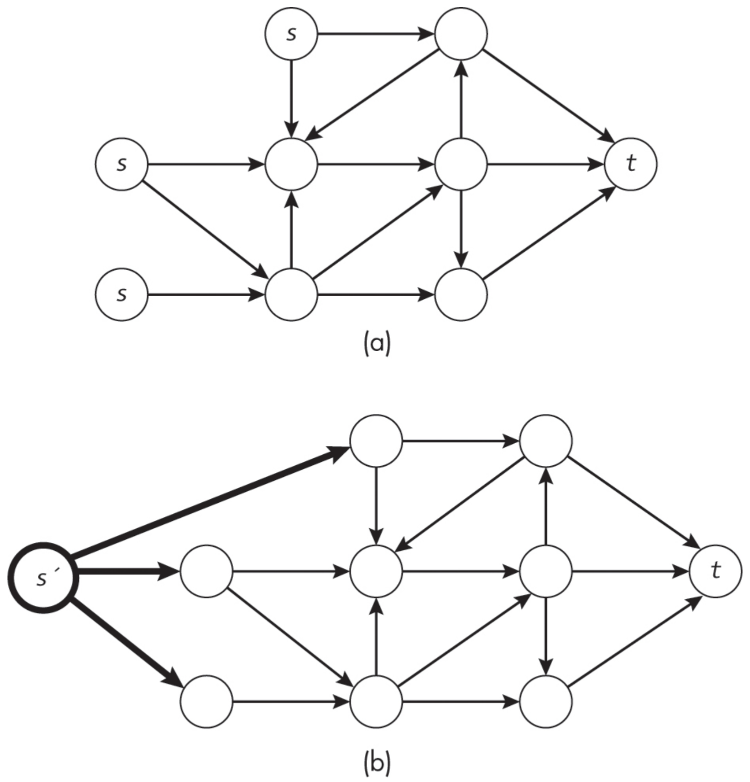


Figure 14-13: A flow graph with multiple sources (a) and the corresponding model with a single aggregate source (b)

Luckily, we can easily extend the flow network model by adding a new artificial source node *s*′ that effectively supplies each of the previous sources. The new source node is connected by directed edges to each of the previous source nodes. In turn, those previous sources now become internal nodes in our extended model, as shown in Figure 14-13(b). The bolded arrows indicate new edges added from the new node to the previous source nodes.

Of course, our artificial source node does not exist in reality. A city's storm drains are not fed by a secret super-drain. Instead, the aggregated source serves as a convenient mathematical abstraction that allows us to assign the source of all flow back to a single (virtual) node.

Adding the new node and edges raises the question of how we choose the capacity of those new edges. If we set their capacity too low, these edges will serve as a bottleneck, preventing us from accurately modeling the problem. However, it doesn't matter if we set the edge weights too high, because the bottleneck will then be the already-existing bottleneck in the original network. We can therefore use infinite capacities along those new edges to supply the previous sources with all the flow they can handle. In the context of the wastewater system, these would be gigantic pipes that vastly exceed the flow of anything the engineers could actually build.

We create a helper function to augment an arbitrary graph with multiple sources to add the aggregate source, as shown in Listing 14-2.

```
def augment_multisource_graph(g: Graph, sources: list) -> i
nt:
 ❶ new_source: Node = g.insert_node()

 ❷ for old_source in sources:
        g.insert_edge(new_source.index, old_source, math.in
f)
    return new_source.index
```

Listing 14-2: Transforming a multi-source graph into one with a single source

The code inserts a new node into the graph ❶. For each of the previous sources, it then creates a new edge from the new source to the previous source with infinite capacity ❷. Finally, the code returns the index of the new source for us to use when calling the Ford-Fulkerson algorithm.

Multiple Sinks

Just as many real-world problems have multiple sources, we often encounter networks with multiple sinks. Consider the interstate highway system, for example, where cars and trucks flow along the roads to numerous divergent destinations. Figure 14-14(a) shows a network with two sink nodes.

We can adapt the approach we used for the multi-source problem to handle multiple sinks. We create a new aggregated sink *t*′ and create directed edges from each previous sink to the new aggregate, as shown in Figure 14-14(b). The new node and edges are bolded. We assign each of these edges enough capacity so they cannot generate a new bottleneck.

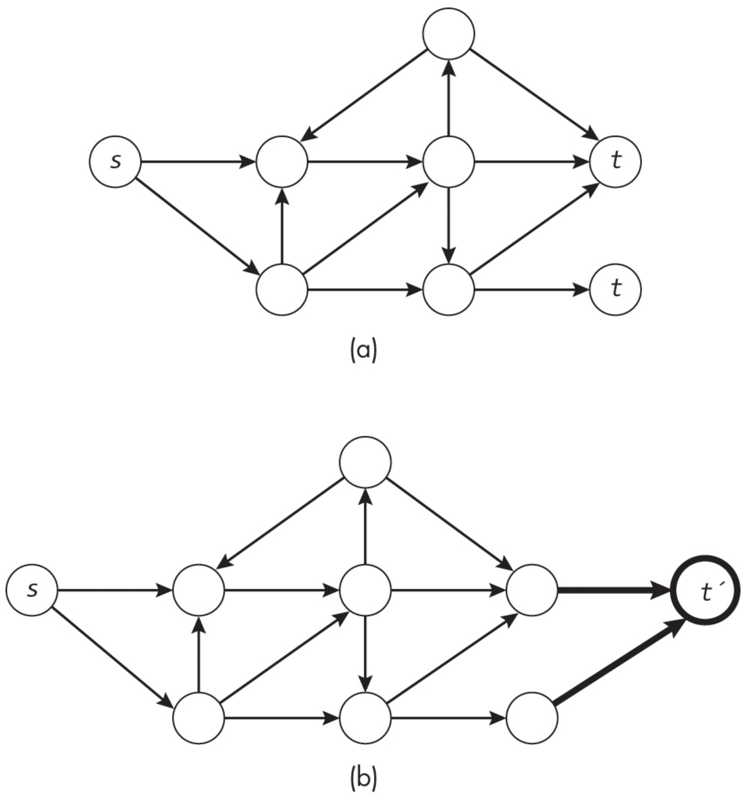


Figure 14-14: A flow graph with multiple sinks (a) and the corresponding model with a single aggregate sink (b)

Again, we provide a helper function to augment a given graph with multiple sinks:

```
def augment_multisink_graph(g: Graph, sinks: list) -> int:
    new_sink: Node = g.insert_node()

    for old_sink in sinks:
        g.insert_edge(old_sink, new_sink.index, math.inf)
    return new_sink.index
```

The code follows the form of the `augment_multisource_graph()` function in Listing 14-2. It inserts a new sink node into the graph, creates edges with sufficient capacity to each of the old sinks, and returns the index of this new node.

Anti-parallel Edges

It is also unrealistic to prohibit anti-parallel edges in real-world use cases. Continuing with the example of the interstate highway system, nearly every highway is a two-way road: you can travel Route 90 from Cleveland to Buffalo or from Buffalo to Cleveland.

We can use another mathematical trick to support such real-world cases while maintaining the restriction that the graph must not have anti-parallel edges. When dealing with a loop with edges (u, v) with capacity w_1 and (v, u) with capacity w_2, as shown in Figure 14-15(a), we can add a new node x and replace the edge (u, v) with the pair of edges (u, x) and (x, v), as shown in Figure 14-15(b). If we use the same capacity w_1 of the previous edge (u, v) for both edges (u, x) and (x, v), the total flow allowed through the expanded path is the same (w_1).

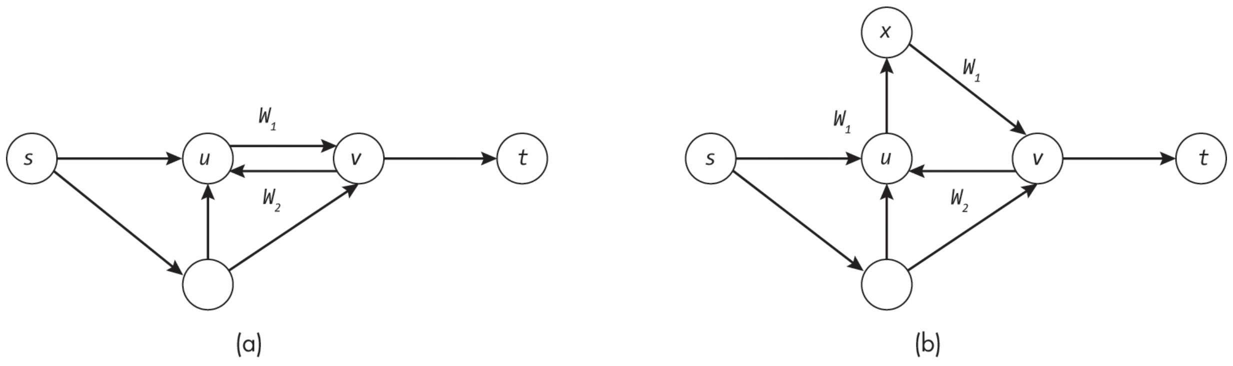


Figure 14-15: A graph with a loop containing two edges (a) and its augmented version (b)

We could define a single preprocessing step that iterates over all edges in the graph and inserts extra nodes and edges where needed. If the original graph contains an edge (origin, destination) and the edge's inverse (destination, origin), then we have anti-parallel edges and need to insert a single new node.

Why This Matters

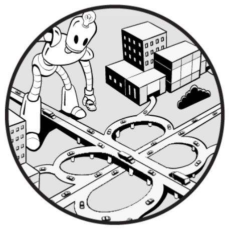
We can use the maximum-flow problem to answer a wide range of real-world analysis and optimization questions. Beyond simply finding the maximum flow from source to sink, the techniques for solving the maximum-flow problem provide crucial insights into the network itself: we can use the residual graph to find bottlenecks or discover which links have excess capacity. For example, suppose our analysis of a proposed wastewater processing system reveals that a pipe with capacity of 50 gallons per minute will be used only for 10 gallons per minute due to restrictions elsewhere in the network. We now know this pipe presents a clear cost-saving opportunity.

The algorithmic approaches in this chapter also provide new ways of thinking about and working with graphs. The `CapacityEdge` data structure is an expansion on the standard edge that allows for tracking dynamic amounts, and the paths through a `ResidualGraph` change as flow is applied. This is the first time we have seen an algorithm that needs to account for dynamic quantities related to edges.

As we will see in the next chapter, the maximum-flow algorithm has extensions to more general matching problems, including optimizing the connections between pairs of nodes. We will also see how these techniques can be applied to the more abstract problem of maximum-cardinality bipartite graph matching.

15

BIPARTITE GRAPH MATCHING



Many problems in business and logistics consist of matching items from two different sets. We might want to match people with jobs, conferences with locations, vehicles with routes, or adopted pets with homes. In each case, we must ask which item from one set is compatible with which item from the second set. This chapter explores this problem, called *bipartite graph matching*, in detail.

A *bipartite graph* is a graph consisting of two disjoint sets of nodes where each edge always has one end in each set. This graph naturally lends itself to the matching problem: each set of nodes represents one of the sets of items that we want to match, while edges indicate compatibility between the items.

This chapter begins by discussing the broader concept of matching on a graph before formally introducing undirected bipartite graphs and bipartite matching algorithms, showing along the way how graph matching encompasses a rich set of problems, from assigning partners for group work to scheduling jobs in a data center, and technical challenges.

Matching

A *matching* on an undirected graph is a set of edges that do not share any nodes. In other words, each edge in the matching connects two different nodes, and each node is adjacent to at most one edge. We can visualize matchings via pairing up students for project work using their friendship connections. Each edge in the matching represents two friends (the edge's two nodes) who will collaborate on the project. As a natural consequence of this pairwise assignment, we pair up only students with existing social connections and do not guarantee that all students will find a partner.

The general concept of matching opens up a range of problems we can solve. As concrete examples for this chapter, let's examine two particularly useful matching problems. First, finding a *maximum-cardinality matching* (sometimes shortened to *maximum matching*) consists of finding a matching with the most edges. This corresponds to finding an assignment of students to pairs that creates the most groups. Second, a *maximal matching* is any matching where no additional edges can be added without breaking the matching property. While a maximum-cardinality matching is always a maximal matching, the converse is not true.

Figure 15-1 shows examples of these two matching types. We cannot add any more edges to the maximal matching in Figure 15-1(a), represented by the bold edges, without reusing a node. Meanwhile, Figure 15-1(b) is both a maximal matching and a maximum-cardinality matching: we cannot create a matching for this graph with more than three edges.

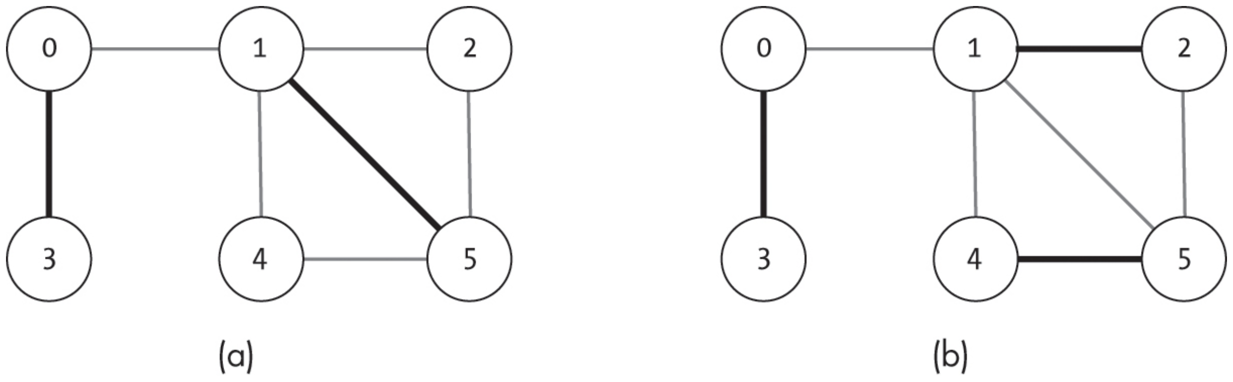


Figure 15-1: Maximal (a) and maximum-cardinality (b) matching

The problem of finding a *maximum-weight matching* consists of finding the matching in a weighted graph that maximizes the sum of the edge weights. This corresponds to prioritizing the strength of the students' friendships when allocating the groups. While this approach is useful in the context of maximizing a reward function such as student happiness, it does not necessarily lead to a maximum-cardinality matching. For example, Figure 15-2 shows a maximum-weight matching that is not a maximum-cardinality matching. Two pairs of nodes {0, 1} and {2, 5} are matched, while nodes 3 and 4 are left out.

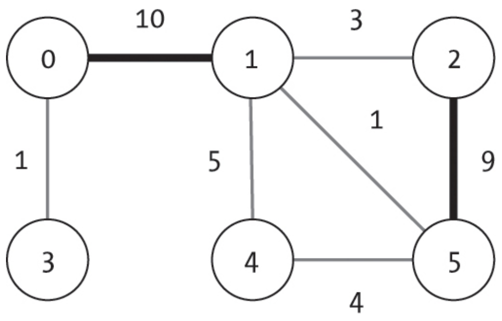


Figure 15-2: The maximum- weight matching on a graph

The list of possible matching problems continues beyond these initial examples. We could ask whether a graph has a *perfect matching*, where every node is included exactly once; find a maximum-cardinality matching that minimizes the sum of the edge weights; or find a matching that maximizes the weight while keeping below a given number of edges. Throughout the rest of the chapter, we will focus primarily on maximum-cardinality matching, one of the simplest and most broadly applicable formulations, in the context of a specific graph type.

Bipartite Graphs

As noted earlier, a *bipartite graph* can be partitioned into two disjoint sets of nodes such that no edge connects two nodes in the same set. Bipartite graphs are often visualized as two parallel lines of nodes, as shown in Figure 15-3. The left and right columns define the two sets of nodes. Every edge in the graph spans the columns.

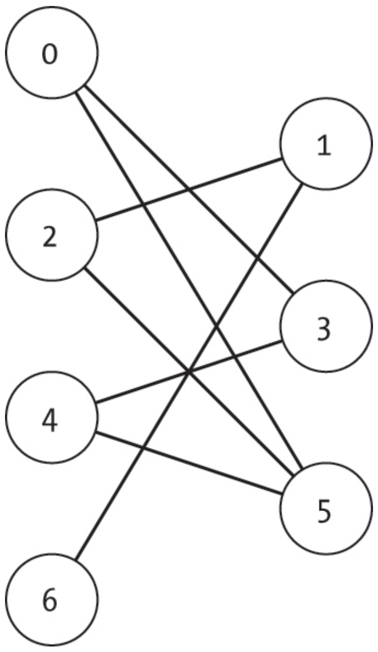


Figure 15-3: A bipartite graph with seven nodes

Bipartite graphs provide a natural model for pairwise matching problems. In a canonical example, the items on the left represent people and the ones on the right represent the jobs for which they are qualified, summarizing a complex set of constraints in a simple graph.

Although matching is the focus of this chapter, the usefulness of bipartite graphs does not end there. Bipartite graphs can model a range of phenomena, from physical bridges crossing a river to spies watching each other at a party.

Bipartite Labeling

Given an undirected graph, we can ask whether it is a bipartite graph and, if so, which node is part of which set. We can use the properties of a bipartite graph to perform both this check and the labeling of which nodes are in which set. We know that any path through an undirected bipartite graph must zigzag between the two sets, and a node can never have a neighbor within its own set. We use a simple search, either breadth-first or depth-first, to traverse the undirected graph and assign labels to nodes. The key is that we arbitrarily assign the first node one label and then alternate labels each time we cross an edge. If we ever find two neighbors with the same label, we know the graph is not bipartite.

We can picture the algorithm in the context of competing spy agencies at a cocktail party. The agencies consist of disjoint sets of spies, represented as nodes. Each edge signifies the connection between two people who are watching each other. The spies are well trained and can each keep an eye on multiple people at once—Spy A might be watching Spy B, Spy C, and Spy D!

A bored waiter, unaware of the true identities of anyone in the ballroom, uses the opportunity to determine which spies work together. They start by picking one spy at random and assigning them to the green team. They determine all the people that spy is watching and assign each of them to the yellow team. Then, for each of the newly discovered yellow members, the waiter determines who they are watching and assigns those people being watched to the green team. The process bounces back and forth, uncovering the affiliations of each person as the waiter serves various appetizers.

Of course, if the waiter ever finds a spy watching a member of their own team, they know they do not have a bipartite graph. Maybe the agencies sent internal affairs or there is a double agent. Regardless, the situation is no longer as clear as yellow versus green and probably is not something with which the waiter wants to be involved.

The Code

The code for bipartite graph labeling in Listing 15-1 uses a breadth-first search to iteratively explore the graph and label nodes as belonging to either the right or left side.

```
def bipartite_labeling(g: Graph) -> Union[list, None]:
    label: list = [None] * g.num_nodes
    pending: queue.Queue = queue.Queue()

❶   for start in range(g.num_nodes):
    ❷   if label[start] is not None:
            continue

    ❸   pending.put(start)
        label[start] = True
        while not pending.empty():
            current: int = pending.get()
        ❹   next_label = not label[current]
```

```
            for edge in g.nodes[current].get_edge_list():
                neighbor: int = edge.to_node
              ❺ if label[neighbor] is None:
                    pending.put(neighbor)
                    label[neighbor] = next_label
              ❻ elif label[neighbor] != next_label:
                    return None
    return label
```

Listing 15-1: Labeling nodes according to which side of the bipartite graph they occupy

The `bipartite_labeling()` function maintains a list mapping each node index to one of three states (unlabeled = `None`, right = `False`, or left = `True`) using a combination of Boolean and `None` values. The code starts by setting up the list of labels (`label`) and a queue (`pending`), which requires us to import Python's `queue` library. Each label is initialized to `None`, indicating that the algorithm has not yet seen a node and given it a label. This list will serve the roles of both tracking the seen nodes in breadth-first search and storing the labels.

The main body of the code is a repeated breadth-first search where an outer loop starts a new search on any unvisited node ❶. A `for` loop checks whether each potential starting node has already been seen by a previous search. If so (the node's label is not `None`), the code skips it ❷. If the node has not been seen, the code adds it to the `pending` queue for the breadth-first search, marks it as belonging to the left-hand side (label of `True`), and starts a new breadth-first search from that node ❸.

During each step of the breadth-first search, the code gets the current node being explored and uses its label to determine that of its neighbors ❹. That is, the current node's neighbors must all have the opposite label; otherwise, the graph is not bipartite. The code iterates over the node's edges and checks each neighbor. If the neighbor has not been seen (label is `None`) ❺, the code sets the label (marking it seen) and adds it to the `pending` queue. If the node has been seen, the code takes the opportunity to check the validity of its label ❻. If the label does not match what is expected, the graph has two connected nodes on the same side and therefore is not bipartite. It immediately returns `None` to indicate the problem.

If the code successfully finishes the series of breadth-first searches needed to explore every node, it returns a list of node labels as `True` or `False` values. Otherwise, it returns `None` to indicate the graph is not bipartite. As with other examples throughout the book, we need to import `Union` from Python's `typing` library to support type hints for these multiple return values.

An Example

Figure 15-4 shows the steps of the bipartite labeling algorithm working on an example seven-node graph. In Figure 15-4(a), an arbitrary node (0) is chosen, given the first label (`True`), and added to the queue to explore. This corresponds to the waiter selecting the first spy to watch and assigning them to the green team.

After exploring node 0, the algorithm finds two neighbors, as shown in Figure 15-4(b). It gives nodes 3 and 5 `False` labels to indicate they are in the opposite set from node 0. Both nodes are also added to the queue.

Figure 15-4(c) shows the search's state after exploring node 3 and discovering just one new neighbor, node 4. Since node 3 had a label of `False`, the search gives node 4 a label of `True`. The algorithm also checks all previously seen nodes, in this case node 0, to confirm their labels match the expected value. For the bored waiter, this step corresponds to watching the first person identified as a member of the yellow team. The waiter shifts behind a potted plant and notes that person 3 is watching person 0 back, as must be expected of any good spy. After a moment, the waiter notes that person 3 is also watching person 4. The waiter has found another member of the green team and notes this on a cocktail napkin.

The search continues through the graph one node at a time. At each step, the algorithm examines the full set of neighbors, labeling new ones and adding them to the queue as well as checking known neighbors' labels for consistency. The search ends in Figure 15-4(h) after every node has been checked.

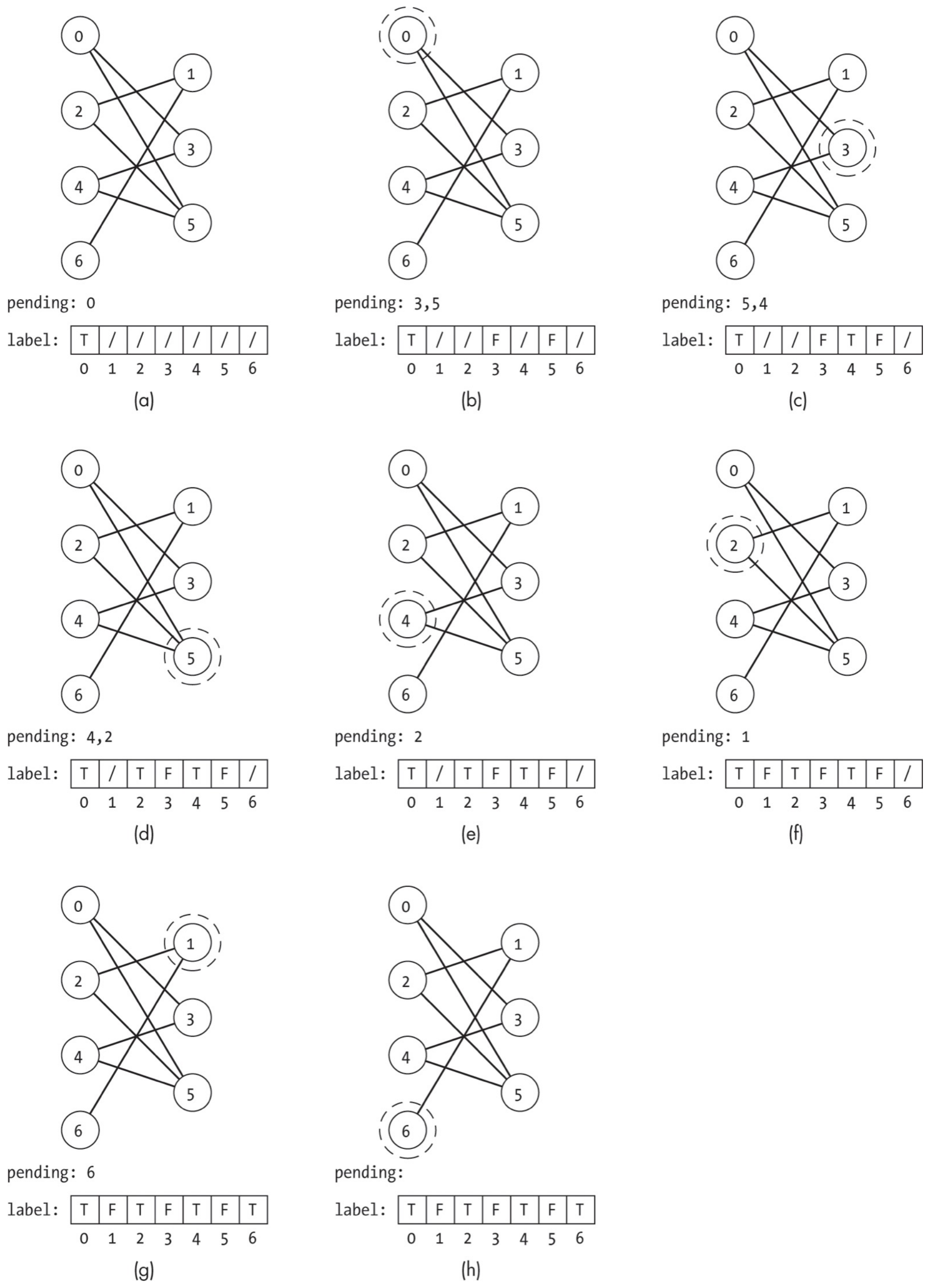


Figure 15-4: The steps of a successful bipartite graph check

We can also use this same algorithm to identify non-bipartite graphs. Figure 15-5 shows the same algorithm on a non-bipartite graph, formed by adding a single additional edge to the graph in Figure 15-4. For the first few steps, the search progresses similarly to that in Figure 15-4. An arbitrary initial node is chosen in Figure 15-5(a) and explored in Figure 15-5(b). The first hint of trouble occurs in Figure 15-5(c), where node 1 is labeled as occurring on the left-hand side because it is a neighbor of node 3. We can easily tell this is a mistake from the visual representation, but the algorithm does not have that information yet. From its perspective, node 1 could very well be on the left-hand side. The algorithm will not see the problem until it proceeds further.

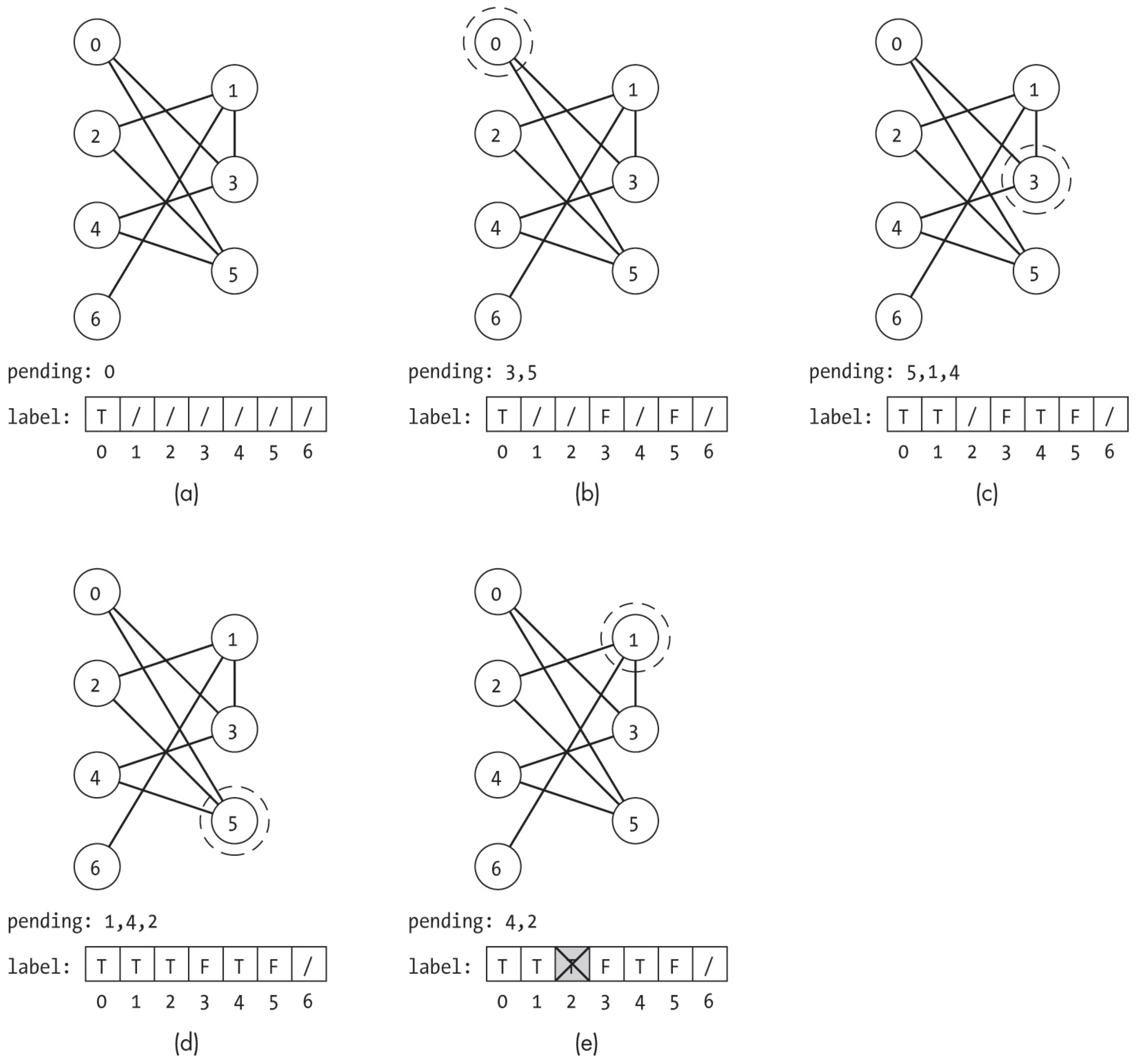


Figure 15-5: The steps of an unsuccessful bipartite graph check

The algorithm finally notices the problem in Figure 15-5(e), when it explores node 1. Since node 1's own label is `True`, it expects its neighbors to be `False`. This fails as soon as it checks node 2. Node 2 was previously labeled `True` while exploring node 5, but node 2 cannot be both `True` and `False`. The graph is not bipartite.

Use Cases

The *bipartite matching problem* consists of solving the matching problem on a bipartite graph and can be used to solve a number of real-world optimization and assignment problems. Due to the structure of the bipartite graph, each selected edge will join a single node from the left side with a single node from the right side. Depending on the task, the problem might be trying to maximize different criteria, such as the number of matches or the sum of the used edge weights. This formulation encompasses a wide range of real-world problems that we might not normally consider in the graph context, from job scheduling to planning office building organization to matching heroes with monsters in a magical dungeon.

Scheduling Jobs

Suppose a physics laboratory wants to maximize the number of concurrent simulations run on its machines. The machines vary in capability, but each can run only a single program at a time. The scientists each submit a program to the human scheduler and urge the scheduler to prioritize their own work first. Yet each program comes with its own requirements, such as high memory or a GPU. The scheduler must respect these constraints, limiting the number of valid assignments.

The scheduler, sensing the perfect opportunity to employ a bipartite matching algorithm, models the allocations as an undirected graph. They list the scientists' jobs in the left column and the computers in the right, then draw an edge from a program to a machine if the job can run on that machine. The jobs requiring high memory have edges to the high-memory machines, the jobs requiring GPUs have edges to the machines with GPUs, and so forth. Satisfied at their representation, they set about finding the maximum number of jobs that they can schedule at one time.

Assigning Office Space

The Happy Data Structures corporation is looking to move into a new office. While most teams are counting down to move-in day with anticipation of the new space, the planners are worrying about how to assign each team to a work area in the new building. Each workspace has constraints based on the properties of the area, including natural light, access to resources, and size.

After days of collecting long lists of requirements, the planners decide to model the problem as a bipartite-graph-matching problem. One set of nodes represents the teams and the other represents the workspaces. Spaces that are compatible with a team's needs are joined by an edge. Each team can be assigned only to a single space, and each space can have only a single team. The planners use bipartite graph matching between the teams and spaces to find an assignment of all teams to spaces that satisfy their constraints.

Planning Quest Battles

An adventuring party is exploring a magical dungeon when they stumble upon a room full of monsters. Each adventurer agrees to take on a single monster, but the group needs to (quickly) make the assignments. Some assignments are invalid: the wizard cannot fight the magic-resistant lizard, and the sword master cannot challenge the cloud of vapor.

In a twist on the canonical job allocation problem, the party models the problem as a bipartite graph assignment. They use one set of nodes to represent the adventurers and the other to represent the monsters. Compatible enemies are linked with an edge. Now they just need to efficiently match each adventurer with an enemy.

Exhaustive Algorithms

One straightforward approach to finding a variety of matchings, including maximum cardinality and maximum weight, is to simply try every combination of edges. We could enumerate all $2^{|E|}$ possible sets of used edges, throw away ones that use any node more than once, and score the rest based on our metric of interest. In this section, we briefly consider an algorithm based on depth-first search that performs this *exhaustive search*. This algorithm provides a baseline to which to compare more computationally efficient approaches.

Matching Data

To simplify and generalize the code in this section, we employ a wrapper data structure around the matching assignments that track both the assignments and the current score. The `Matching` object holds three pieces of information about the current matching:

`num_nodes (int)` Stores the total number of nodes in the graph

`assignments (list)` Stores a mapping of each node to its partner or `-1` if the node is unmatched

`score (float)` Stores the matching's score

The `assignments` list is bidirectional and stores the partners for both sides of the bipartite graph. For example, the inclusion of edge (0, 4) would result in `assignments[0]=4` and `assignments[4]=0`.

To implement this wrapper data structure, we define a constructor to create an empty matching and provide functions adding and removing edges from the matching:

```
class Matching:
    def __init__(self, num_nodes: int):
        self.num_nodes: int = num_nodes
        self.assignments: list = [-1] * num_nodes
        self.score: float = 0.0

    def add_edge(self, ind1: int, ind2: int, score: float):
        self.assignments[ind1] = ind2
        self.assignments[ind2] = ind1
        self.score += score

    def remove_edge(self, ind1: int, ind2: int, score: floa
t):
        self.assignments[ind1] = -1
        self.assignments[ind2] = -1
        self.score -= score
```

To simplify the example, neither the `add_edge()` nor `remove_edge()` functions check node validity or whether a node is currently assigned. In real-world production software, you'll often want to add checks that no node is used twice and that the added edge exists within the graph using similar checks to those in Chapter 1.

Exhaustive Scoring

We use an approach based on recursive depth-first search to enumerate which edges are used. Instead of exploring depth-first over the nodes, we explore over node assignments.

For example, consider the bipartite graph in Figure 15-6. Node 0 has three options for matching: nothing, node 1, or node 3. The same applies to node 2.

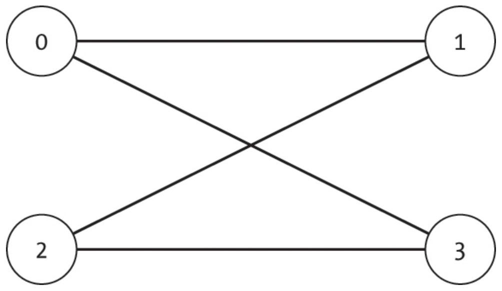


Figure 15-6: A bipartite graph with four nodes

We can visualize the space of potential matchings for the graph in Figure 15-6 as a tree where each level indicates the assignment of one of the left-hand nodes. This tree is shown in Figure 15-7. The first level of the tree splits into three scenarios based on the three different options for node 0. The left branch represents the options if we leave node 0 unpaired. The central branch represents the options with node 0 paired to node 1. The right branch represents the options with node 0 paired to node 3. We limit the search space by considering only potential matches that correspond to edges in the graph.

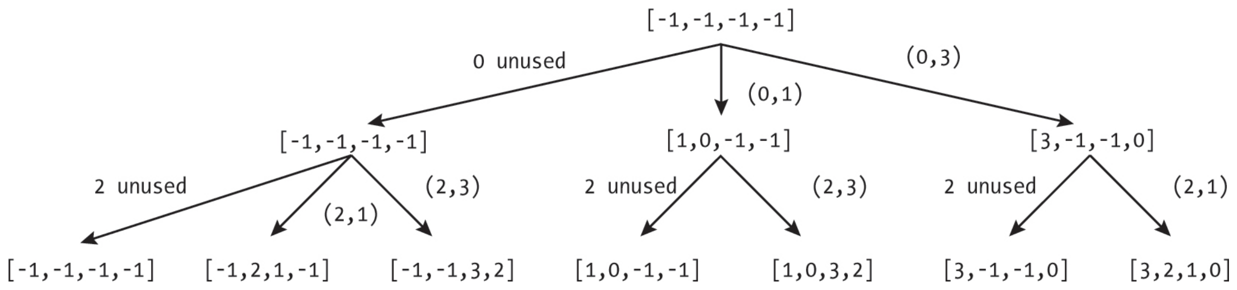


Figure 15-7: A search tree over the potential node-matching assignments

The second level of the tree in Figure 15-7 shows similar splits for the assignment of node 2. Since node 0 is already assigned in two of the three branches, those branches can split only into two sub-options.

Since this approach is performing an exhaustive search, it will work for a variety of bipartite matching problems, including maximum cardinality and maximum weight. As we will see, the only factor that changes is how we compute the matching's score.

The Code

We implement a search for the maximum-weight matching over this tree using a recursive algorithm:

```
def bipartite_matching_exh(g: Graph) -> Union[list, None]:
  ❶ labels: Union[list, None] = bipartite_labeling(g)
    if labels is None:
        return None

    current: Matching = Matching(g.num_nodes)
  ❷ best_matching: Matching = matching_recursive(g, labels,
current, 0)
    return best_matching.assignments

def matching_recursive(g: Graph, labels: list, current: Mat
ching,
                       index: int) -> Matching:
  ❸ if index >= g.num_nodes:
        return copy.deepcopy(current)
  ❹ if not labels[index]:
        return matching_recursive(g, labels, current, index
+ 1)

  ❺ best: Matching = matching_recursive(g, labels, current,
index + 1)
    for edge in g.nodes[index].get_edge_list():
      ❻ if current.assignments[edge.to_node] == -1:
            current.add_edge(index, edge.to_node, edge.weig
ht)
            new_m: Matching = matching_recursive(g, labels,
current, index + 1)
            if new_m.score > best.score:
                best = new_m
            current.remove_edge(index, edge.to_node, edge.w
eight)
    return best
```

The outer wrapper function `bipartite_matching_exh()` labels the sides of the graph ❶ and sets up the `Matching` data structure. It returns `None` if the graph is not bipartite (again necessitating the inclusion of `import Union` to support the type hints for multiple return values). It then calls the recursive function to do the matching ❷.

The recursive function `matching_recursive()` starts by checking whether it has hit the bottom of the search ❸, where everything on the left-hand side has been assigned (even if to `-1`). If there are no more nodes to assign, it returns a copy of the `Matching` as the best assignments found down this branch using the `deepcopy()` function from Python's `copy` library. The code performs the copy to effectively snapshot this match and separate it from the `current` object that it will continue to modify during the rest of the search. The use of `deepcopy()` requires the inclusion of `import copy` at the top of the file.

The code then checks whether this node is on the left-hand side of the graph ❹. Since the code makes assignments only from the left nodes, we skip the right ones by calling the recursive function with the current matching at the next index. While the code could be modified to test both sides, assigning the right-side nodes is unnecessary. Each edge can be used only once and is guaranteed to be adjacent to a left-side node.

The code then checks each available option for a match to the current node and saves the best one, starting with the choice of leaving the current node (`index`) unassigned by calling the recursive function with the current matching at the next index ❺. The code saves the best matching down this branch for later comparison as `best`. It then uses a `for` loop to iterate through each of the current node's neighbors, checking the current matching assignments so it can skip nodes that are already assigned to others ❻. The code adds each viable neighbor to the matching, uses the recursive function to get the best matching down that path, updates the `best` matching if necessary, and removes the edge from the matching. Finally, it returns the best matching down this entire branch.

We can change the implementation from the maximum-weight matching in the code to a maximum-cardinality matching by changing the score passed to the matching. That is, we swap `1.0` for `edge.weight` when adding or removing edges. With this change, the search selects matchings with the higher number of assigned edges instead of the greater total weight.

An Example

We can visualize this function by looking at the state of the current matching each time the algorithm hits the bottom of the recursion. Figure 15-8 shows the first nine times the algorithm hits the end of the recursion on an example graph.

Since the algorithm tests the no-assignment (`-1`) branch first, the recursion encounters the full assignment `[-1,-1,-1,-1]` in Figure 15-8(a). After evaluating this empty matching, the search backtracks and tests alternate assignments for node 4 while keeping the assignments for nodes 0 and 2 from previous decisions fixed. This produces the matchings in Figures 15-8(b), 15-8(c), and 15-8(d). It is not until Figure 15-8(f) that the algorithm evaluates a matching with two edges used.

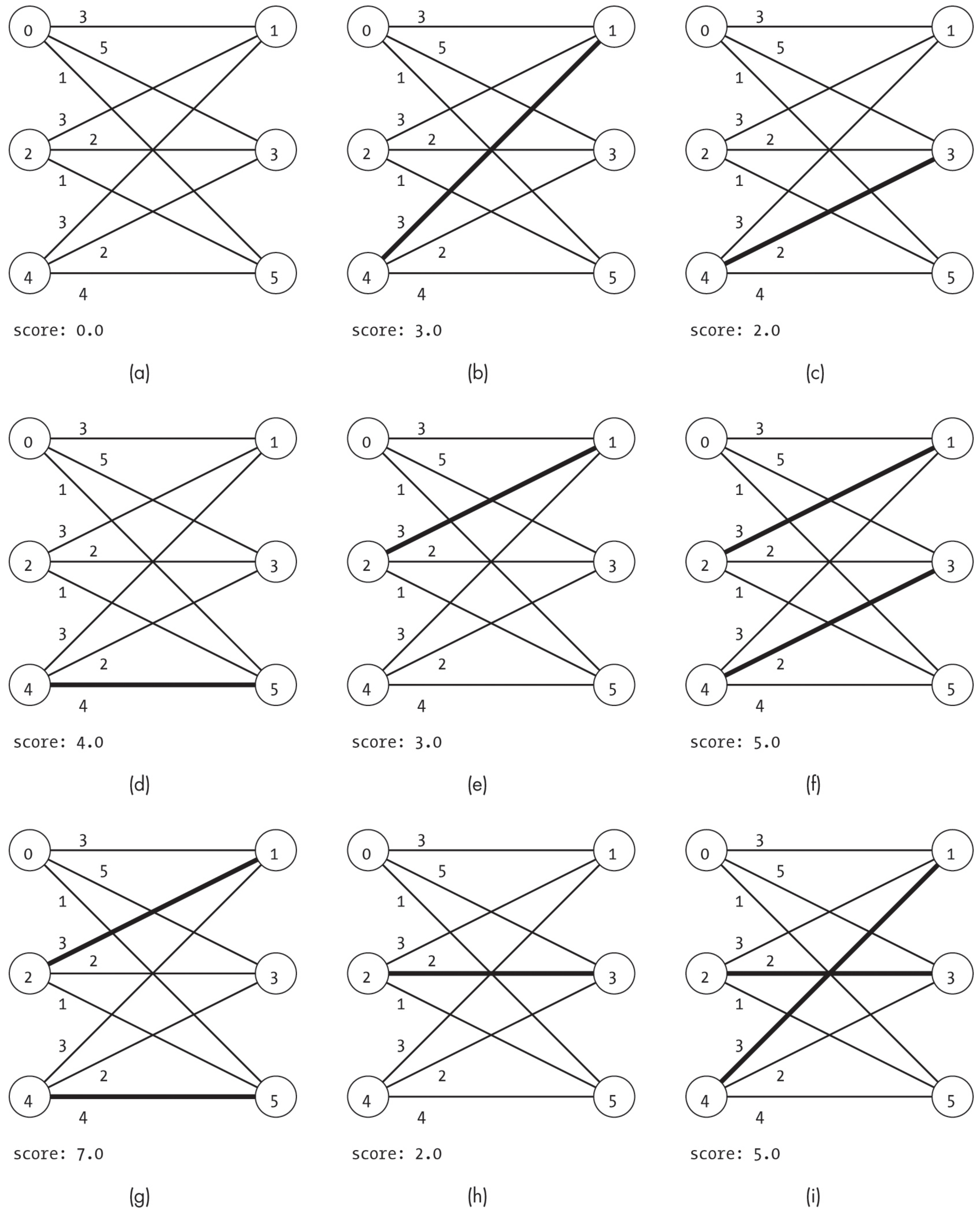


Figure 15-8: The first nine steps of the exhaustive search algorithm

While the exhaustive algorithm is both complete and generalizable, it is far from efficient, especially on large graphs. Even the search of the six-node graph in Figure 15-8 explores 34 different assignments. The next section introduces one of the specialized algorithms that exist to efficiently solve individual matching problems.

Solving the Maximum-Cardinality Bipartite Problem

This section shows how to use the max-flow algorithms from the previous chapter to solve the maximum-cardinality bipartite problem efficiently. We can transform the maximum-cardinality bipartite problem directly into a maximum-flow problem by transforming the bipartite graph into a flow network with directed edges and unit weights. Figure 15-9(a) shows a bipartite graph and Figure 15-9(b) shows the result of this transformation. We add a single source node *s* that feeds all the nodes in the left column. We add a single sink node *t* to accept flow from the nodes in the right column. Each edge in the graph is directed from left to right and has a capacity of 1. (To reduce clutter, the illustration does not include the capacity of the edges.)

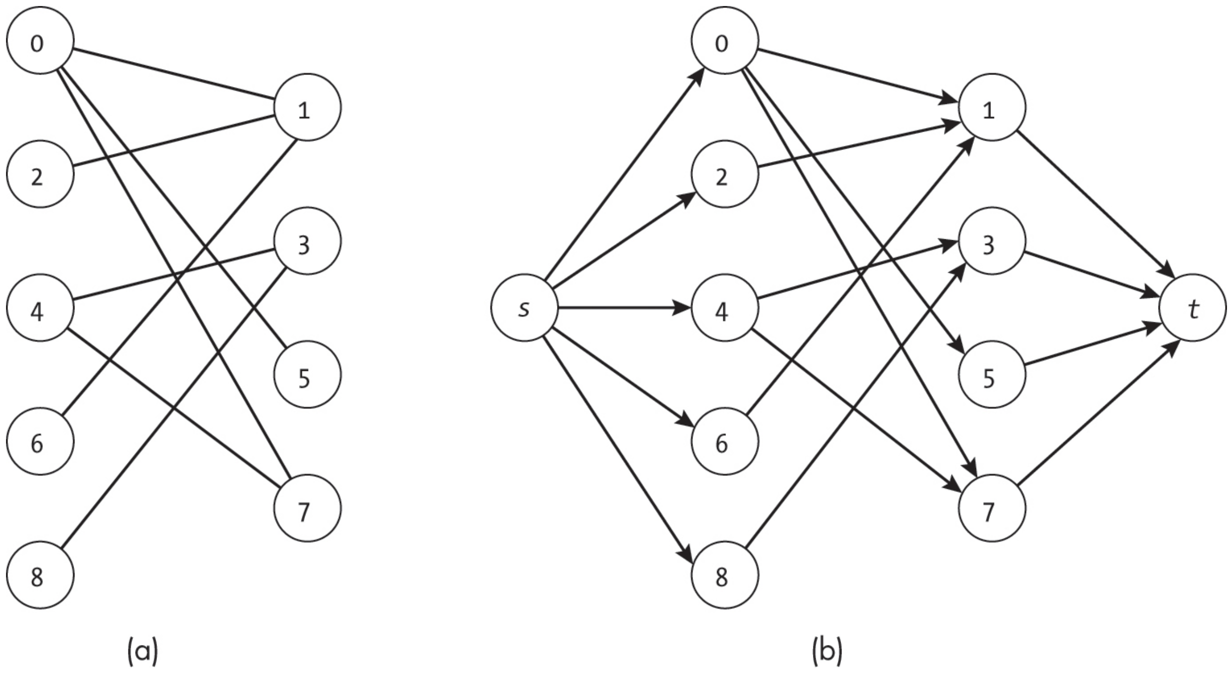


Figure 15-9: A bipartite graph (a) and its flow network version (b)

Using this setup, the source node can supply each of the left-hand nodes with up to a single unit of flow. Similarly, each node in the right-hand column can provide at most 1 unit of flow to the sink. Given the constraint that the flow into a node must equal the flow out of it, each node on the left can send flow to at most 1 unit of flow to a node on the right. Each node on the right can accept at most 1 unit of flow from a node on the left. The maximum flow is equal to the maximum number of pairs we can assign.

The Code

The code for performing maximum-cardinality bipartite matching with the max-flow algorithm reuses the Edmonds-Karp implementation from Chapter 14 to do all the heavy lifting. Much of the wrapper function itself transforms the graph by adding a source and sink node, then later pruning the unneeded edges:

```
def bipartite_matching_max_flow(g: Graph) -> Union[list, No
ne]:
    num_nodes: int = g.num_nodes

    labeling: Union[list, None] = bipartite_labeling(g)
    if labeling is None:
        return None

  ❶ extended: Graph = Graph(g.num_nodes + 2, undirected=Fal
se)
    for node in g.nodes:
        for edge in node.edges.values():
            if labeling[edge.from_node]:
                extended.insert_edge(edge.from_node, edge.t
o_node, 1.0)

  ❷ source_ind: int = num_nodes
    sink_ind: int = num_nodes + 1
    for i in range(num_nodes):
        if labeling[i]:
            extended.insert_edge(source_ind, i, 1.0)
        else:
            extended.insert_edge(i, sink_ind, 1.0)

  ❸ residual: ResidualGraph = edmonds_karp(extended, source
_ind, sink_ind)
```

```
❹ result: list = [-1] * g.num_nodes
    for from_node in range(residual.num_nodes):
        if from_node != source_ind:
            edge_list: dict = residual.edges[from_node]
            for to_node in edge_list.keys():
                if to_node != sink_ind and edge_list[to_nod
e].used > 0.0:
                    result[from_node] = to_node
                    result[to_node] = from_node
    return result
```

The `bipartite_matching_max_flow()` function returns either a list of assignments (in the same bidirectional format as `Matching` class's `assignments` list) or `None` if the graph is not bipartite.

To build the augmented graph, the code first must know which nodes are on which side of the bipartite graph. It reuses the `bipartite_labeling()` function from Listing 15-1, taking the opportunity to also check for non-bipartite graphs.

The code then builds an augmented graph. First, it creates a new (directed) graph with two extra nodes for the source and sink ❶. Second, it adds directed edges, each with a capacity of `1`, going from the left column to the right column, using the `labeling` list to determine in which column a node belongs. Finally, the code connects the source and sink ❷. It adds edges from the source to each of the original graph's left-hand nodes and edges from each of the original graph's right-hand nodes to the sink.

The code runs the Edmonds-Karp algorithm on the graph to find the maximum flow through it ❸. Each `CapacityEdge` in the resulting graph (`residual`) that has used flow are connected. However, the overall graph also still contains the source node, the sink node, and all their corresponding edges. The code iterates through the edges and fills in the assignments for each `CapacityEdge` where `used` is greater than zero, the origin is not the source node, and the destination is not the sink node ❹.

An Example

Figure 15-10 shows the steps the maximum-flow algorithm takes as it identifies a matching on the example bipartite graph in Figure 15-9. Edges at capacity (`used` equals 1) after each step of the algorithm are bolded.

We can picture this algorithm in the context of the job-scheduling algorithm introduced in “Use Cases” on page 263. Job 0 (node 0) is the most flexible job, with the ability to run on three of the four machines. In contrast, jobs 2, 6, and 8 are limited to running on specific machines.

Figure 15-10(a) shows the initial state where the algorithm has added the source node, the sink node, and all the corresponding edges. No edges carry any flow yet. This corresponds to starting in an empty state with no jobs scheduled to run on any machine.

Figures 15-10(b) and 15-10(c) show the state of assignments after the code finds the first two augmenting paths. During the first round, it discovers that the unassigned job 0 can run on the unused machine 1 and makes that assignment. In the second round, it does the same with job 4 and machine 3.

Things get interesting at this point, as shown in Figure 15-10(d). By assigning job 0 to machine 1, the algorithm has blocked both jobs 2 and 6, as both can only run on machine 1. Luckily, the algorithm can handle this by finding a new augmenting path: (*s*, 2), (2, 1), (1, 0), (0, 5), (5, *t*). In doing so, it pushes flow backward along the edge (0, 1), unassigning job 0 from machine 1. The result is an assignment with one more job scheduled.

Figure 15-10(e) shows a similar multi-step augmenting path. This new path consists of edges (*s*, 8), (8, 3), (3, 4), (4, 7), and (7, *t*). The code unassigns job 4 from machine 3 and then assigns job 4 to machine 7 and job 8 to machine 3.

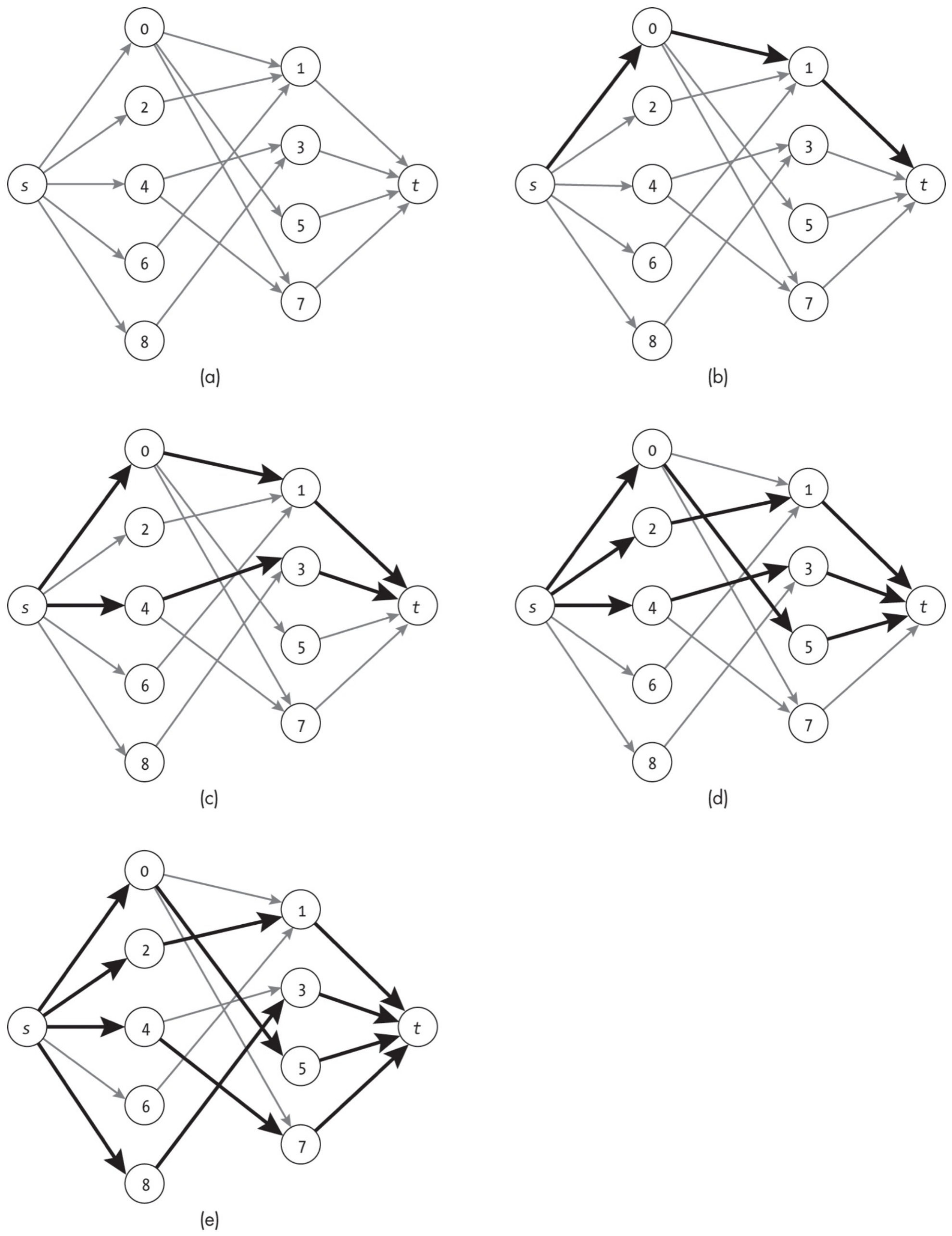


Figure 15-10: The max-flow algorithm operating on an augmented bipartite graph

After the maximum-flow algorithm completes, it has produced the graph in Figure 15-10(e). To produce the matching list, it then walks the edges of the residual graph. It skips all edges connected to either the source node *s* or the sink node *t*, since those were not part of the original bipartite graph. It saves all connections with nonzero flow. The result with matched edges (0, 5), (2, 1), (4, 7), and (8, 3) is shown in Figure 15-11. The unused connections in the original bipartite graph are shown in thin gray lines for reference.

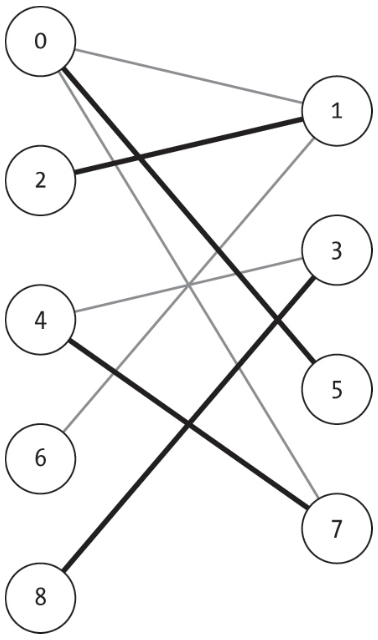


Figure 15-11: A bipartite graph with edges in the maximum-cardinality matching bolded

The maximum-flow algorithm will only find a single maximum-cardinality matching that is not necessarily unique. In Figure 15-11, for example, alternative matchings are possible. Instead of including the edge (2, 1), we could have included the edge (6, 1).

Why This Matters

Bipartite graphs enable us to translate a range of assignment problems to equivalent graph algorithms, allowing access to a wealth of powerful graph algorithms. In doing so, we can solve problems that we might not initially think of as graph-based. A clear example of this flexibility is the maximum-cardinality matching problem, which transforms the problem of matching items from two disjoint sets into a graph problem that we can solve with maximum-flow algorithms.

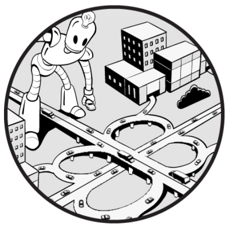
In the next section of the book, we switch topics and consider a variety of computationally challenging problems on graphs. Chapter 16 introduces the problem of assigning colors to graph nodes such that no two neighbors share a color. Chapter 17 considers a variety of other useful node assignment problems. Finally, Chapter 18 extends the examination of challenging problems to that of finding specific types of paths through graphs.

PART V

HARD GRAPH PROBLEMS

16

GRAPH COLORING



Graph coloring is a conceptually simple but computationally complex problem with a range of real-world applications. At its core, it consists of assigning a color to each node in an undirected graph such that no pair of nodes sharing an edge have the same color. Variations of this problem include minimizing the number of colors used or finding an assignment using only a fixed number of colors.

We can easily visualize the importance of the graph-coloring problem in terms of a map of Europe. We need to assign each country a color such that no two adjacent countries have the same color. If we use green for both France and Belgium, viewers may not be able to see the border. Beyond maps, the constraints enforced by the graph-coloring problem lend themselves to a range of real-world optimization tasks.

This chapter begins by formally defining the graph-coloring problem and discussing real-world use cases in more depth. We then examine several approaches to solving the task and discuss why the problem of using a fixed number of colors (or minimizing the number used) is surprisingly difficult.

The Graph-Coloring Problem

The *graph-coloring problem* consists of assigning colors to each of the $|V|$ nodes in an undirected graph such that no two nodes sharing an edge have the same color. Formally, we can define the problem as follows:

> Given a graph, defined by a set of nodes V and edges E, and a set of colors C, find an assignment of nodes to colors such that for any two nodes $u \in V$ and $v \in V$ connected by an edge, $(u, v) \in E$, then $color(u) \neq color(v)$.

We can define the *minimum graph-coloring problem* as finding the minimum number of colors such that a graph has a valid graph coloring.

In this chapter, we use the graph node's `label` field to store the color for that node. Colors will be represented by integers starting at 1, with the value of `None` indicating an unassigned node (`node.label == None`).

Listing 16-1 defines a simple `is_graph_coloring_valid()` function that checks whether a graph has a valid coloring. This checker both provides a good overview of the mechanics of the graph-coloring problem and is also a helpful utility function for testing.

```
def is_graph_coloring_valid(g: Graph) -> bool:
    for node in g.nodes:
      ❶ if node.label is None:
            return False
        for edge in node.get_edge_list():
            neighbor: Node = g.nodes[edge.to_node]
          ❷ if neighbor.label == node.label:
                return False
    return True
```

Listing 16-1: Checking whether a graph's coloring is valid

The code uses a `for` loop to iterate through each node in the graph, first checking whether the node has any color assigned ❶. If not, the graph's coloring is incomplete and thus invalid. If the node does have a color, the code uses a second `for` loop to iterate through each of the node's neighbors and check whether the two nodes share a color ❷. If two neighbors share a color, the coloring is invalid, and the code returns `False`. If the code makes it through every neighbor for every node without finding a match, it returns `True`.

We represent different node colors in the illustrations in this chapter using hashing arranged in different orientations, as shown in Figure 16-1.

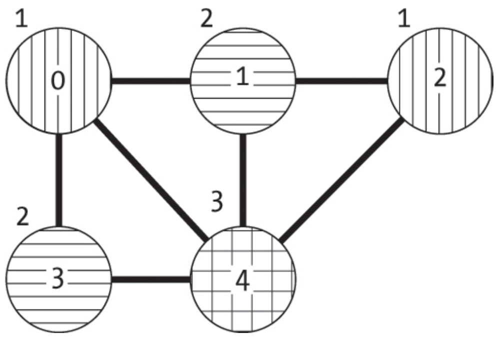


Figure 16-1: A graph with a valid assignment of colors

For the figures in this chapter, we'll note the color number outside each node and number the colors starting from 1.

Use Cases

We can use the graph-coloring formulation to model a range of real-world problems, including coloring maps, making seating arrangements at conferences, assigning parking spaces, and protecting valuable treasure within a labyrinth.

Coloring Maps

The classic use case for graph coloring stems from the daily needs of cartographers and map publishers throughout the world. To distinguish areas on a map, different regions must be shaded with different colors. For a map of New England like that shown in Figure 16-2, we might choose to color Connecticut in green and Rhode Island in orange.

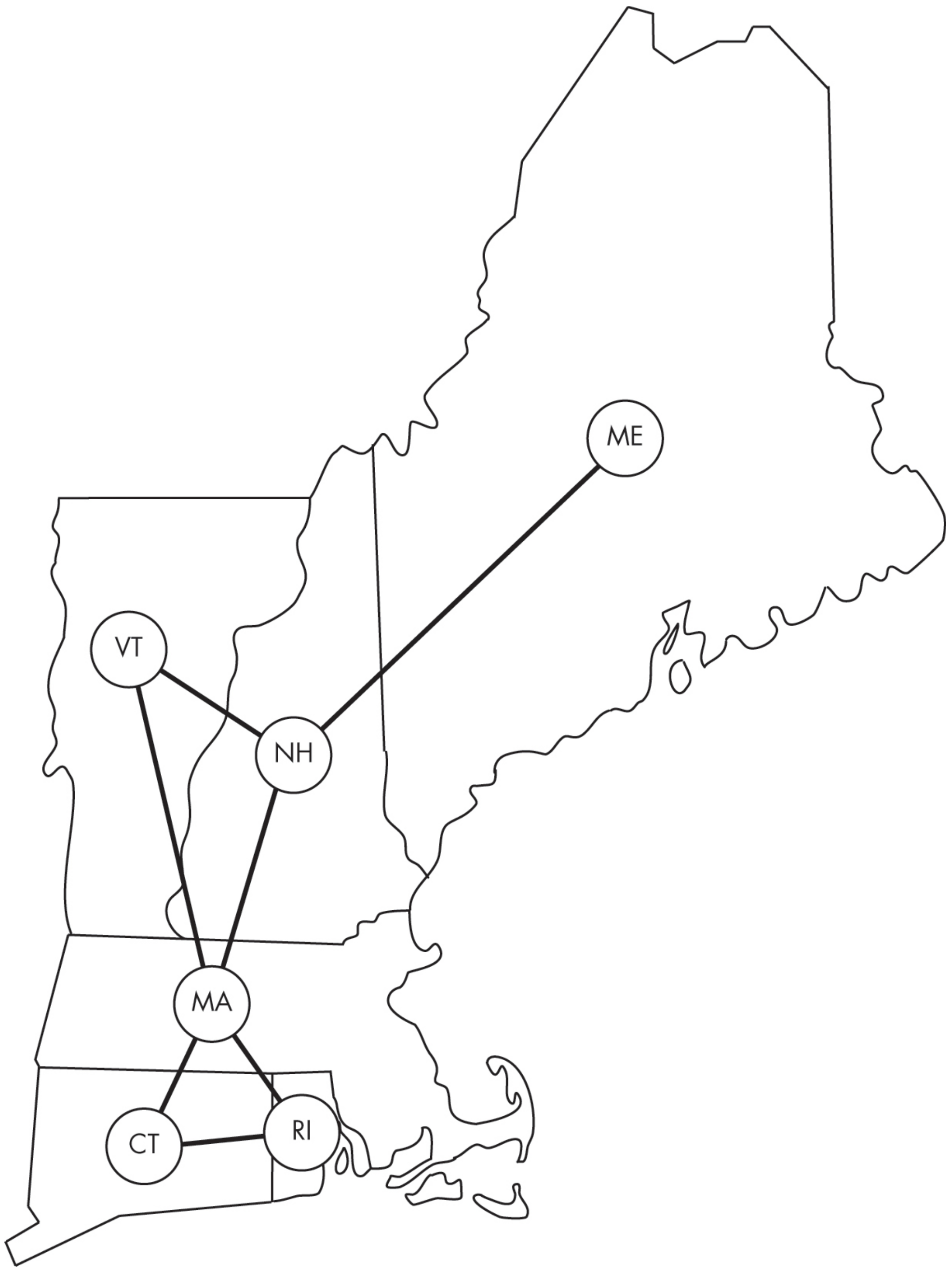


Figure 16-2: A map of New England and the overlaid graph representation

We can translate the map-coloring problem into a graph-coloring problem by creating a node for each region and adding edges between any two regions that share a border. In Figure 16-2, the Rhode Island node has edges to the nodes for both Connecticut and Massachusetts. The cartographer's goal then becomes finding a valid graph coloring.

Organizing Seating Arrangements

The annual convention of Coffee-Drinking Data Structure Enthusiasts is a joyous but surprisingly political affair. Deep divergences exist within the community in terms of preferences for both coffee and data structures, including rivalries between the light and dark roast camps, over 30 different groups of loudly declared favorite data structures, and the inevitable fighting over programming languages that finds a way into any discussion about computer science. Not all differences in preference lead to an argument, but the ones that do result in hours of pedantic shouting. Every year the organizers face the daunting task of assigning tables at the opening banquet such that there will be no loud feuds.

The organizers have long used this gathering as an opportunity to test new graph-coloring algorithms. Modeling each attendee as a node and a strong difference in opinion as an edge, the Conference Chair for Banquet Seating tries to find an assignment of attendees to tables such that no two people at the same table have a strong disagreement. The nodes' colors are the table assignments.

In the simplest case, the organizer could break up the tables into the smallest coherent factions. They might assign a table for the sole espresso-drinking, Fortran-writing hash table fan in attendance. This is equivalent to assigning the node a completely unique color. However, this approach is needlessly wasteful, resulting in almost as many tables as attendees. The aforementioned hash table fan gets along quite well with espresso and dark roast coffee drinkers, users of Fortran or Cobol, and aficionados of most data structures. The Conference Chair's goal is to minimize the number of tables (graph colors) while ensuring that no shouting matches ensue.

Assigning Parking Spaces

The Data Structures and Coffee bookstore sees a consistent stream of customers throughout the day. To meet demand, the owners decide to bring on more staff. After extensive interviews, the owners hire six employees. They draw up a schedule with those new hires working set shifts throughout the day as shown in Figure 16-3(a). One question remains, however: How many parking spaces should they set aside for staff?

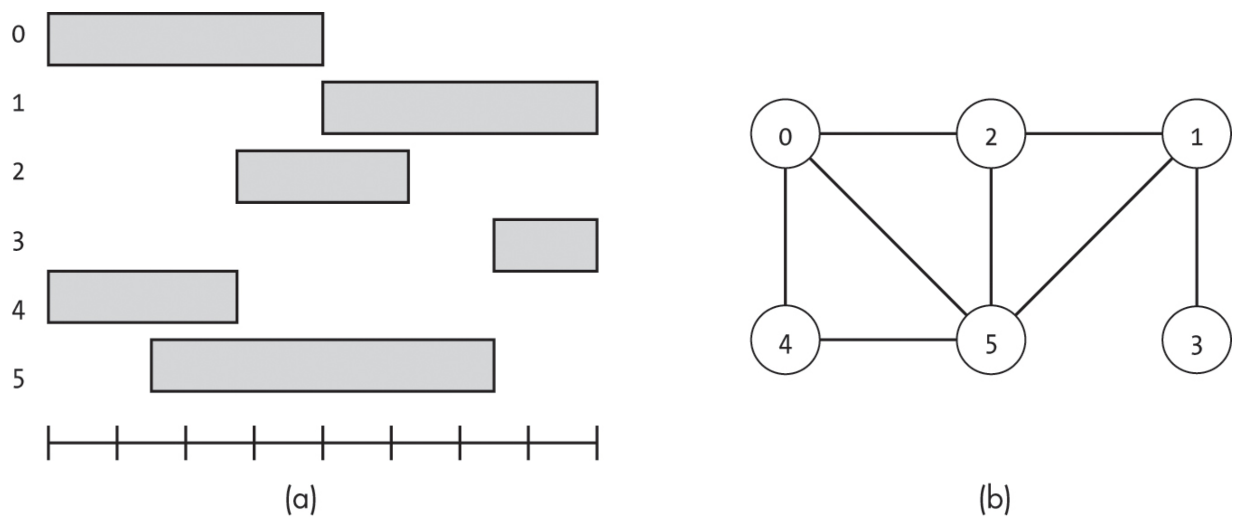


Figure 16-3: Employee work times (a) and the corresponding graph of parking conflicts (b)

Using their data structures expertise, the owners quickly reduce the question to a graph-coloring problem. As shown in Figure 16-3(b), each employee becomes a node in the graph with edges between any two employees whose schedules overlap and thus will need parking spaces at the same time. For example, the schedule of employee 1 overlaps with that of employees 2, 3, and 5. However, it does not overlap with employees 0 or 4, which means that employee 1 could share a parking space with either of the latter two employees. If the owners can find a graph coloring containing at most *C* colors, then they can safely reserve at most *C* parking spaces.

Planning Magical Labyrinths

An evil wizard decides to construct a magical labyrinth to protect their most prized possession, a horn that plays their favorite song. Seeking to protect the Horn of Beautiful Music from relentless adventurers, the wizard fills their labyrinth with an assortment of traps and monsters.

The wizard quickly runs into a problem, however. Due to the large number of passageways (edges), many of the rooms (nodes) connect. They do not want to be seen as lazy by having the same monsters in two adjacent rooms. Skill in labyrinth creation is a reputational necessity; even the most junior adventurers lose respect for architects who fill room after room with the same challenges. Yet the wizard is looking to keep costs down by bulk-ordering monsters. They need to determine the smallest number of monsters (colors) they can use such that no two adjacent rooms contain the same monster.

Graph-Coloring Algorithms

Computer scientists and mathematicians have developed a range of algorithms to solve graph-coloring problems. However, each solution comes with trade-offs. Some algorithms use heuristics that find solutions but might require too many colors; others are expensive on large graphs.

Graph coloring is an *NP-hard problem*, which informally means that there is no known algorithm whose worst-case running time scales as polynomial with respect to the size of the data. There is literally an exponential number of states to consider: $|V|^C$ states for a problem with $|V|$ nodes and C colors. Yet it is not all bad news. While this problem is constrained by its behavior in worst-case scenarios, many of the algorithms perform well in practice and can be applied to numerous everyday problems.

The algorithms in this section search for valid graph colorings. If they find one, they return `True` to indicate their success and set the color assignments in the nodes' `label` field, rather than returning the assignments in a separate data structure. The algorithms return `False` if they are unable to find a valid set of assignments.

Exhaustive Search

An *exhaustive search* through all possible node-to-color assignments is a comprehensive approach that is guaranteed to find a valid graph coloring if one exists:

```
def graph_color_brute_force(g: Graph, num_colors: int) ->
 bool:
    options: list = [i for i in range(1, num_colors + 1)]

 ❶ for counter in itertools.product(options, repeat=g.num
_nodes):
     ❷ for n in range(g.num_nodes):
            g.nodes[n].label = counter[n]
     ❸ if is_graph_coloring_valid(g):
            return True
```

```
❹ for n in range(g.num_nodes):
        g.nodes[n].label = None
   return False
```

The code uses the `product` function of Python's itertools package to enumerate every possible combination of color assignments ❶. Initially, every value of `counter` is assigned to the first color (`1`). During each iteration, the counter changes.

In each iteration of the `for` loop, the code copies the assignments into the node labels ❷ and checks whether they are valid ❸. If so, it immediately returns `True` to indicate it has found a solution. Otherwise, the loop continues forward to the next combination. If the code does not find a successful combination, it returns `False`. Before doing so, it resets all the node's label assignments to `None` because there is no valid coloring ❹.

Of course, the cost of an exhaustive search can be prohibitively expensive as the size of the graph grows. If the graph has $|V|$ nodes and uses C colors, we could end up testing $|V|^C$ assignments before either finding a valid one or determining that no C-color assignment will work. Figure 16-4 shows the first six iterations of an exhaustive search with three colors on the five-node graph from Figure 16-1. The search starts with all nodes the same color, as shown in Figure 16-4(a), and moves through the assignments.

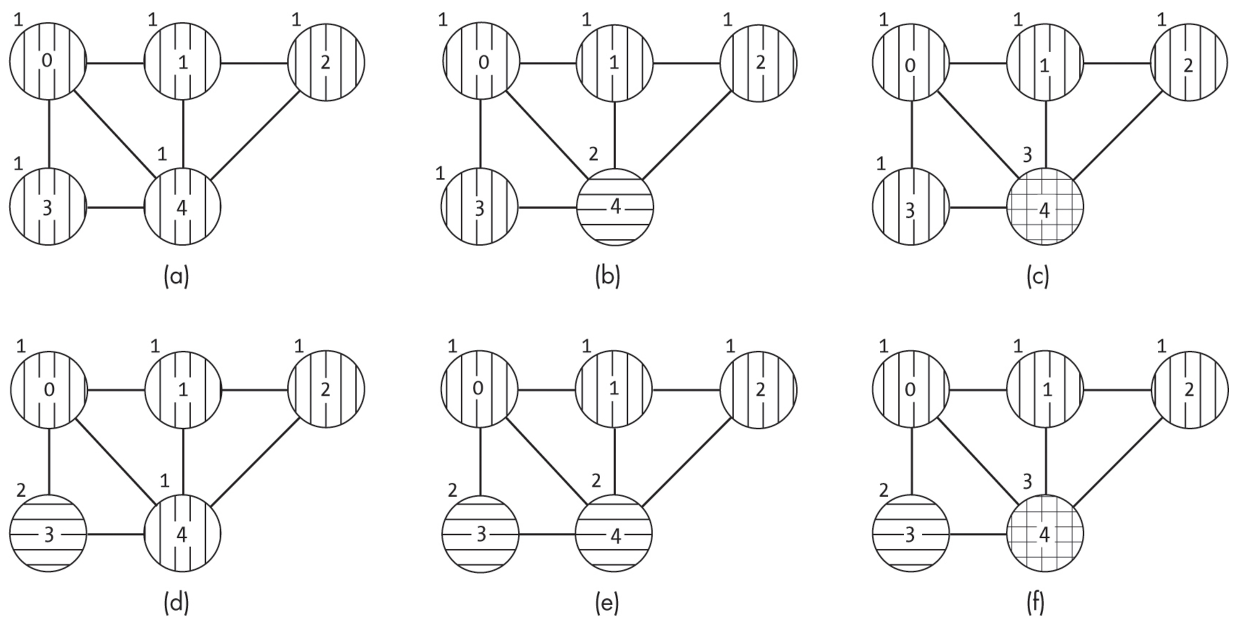


Figure 16-4: The first six iterations of an exhaustive search

As shown in Figure 16-4, exhaustive search can waste a significant amount of time checking one obvious bad state after another. It provides simplicity and completeness at the cost of efficiency. Unless our graph has no edges, it's clearly not possible for all the nodes to share the same color. Imagine the frustration of a human using this approach as they are forced to test yet another combination they know won't work because there is a clear conflict somewhere else in the graph.

Backtracking Search

We can also implement the iterator-based exhaustive search from the previous section using a *recursive backtracking search*. Unlike the depth-first searches used in earlier chapters, especially Chapter 4, this backtracking search does not explore individual nodes through their neighbors. Instead, the search state is the set of color assignments itself. We recursively explore all possible assignments of colors to nodes, backtracking when the solution is infeasible. Each state in our search corresponds to a partial assignment of colors to nodes, as shown in Figure 16-5.

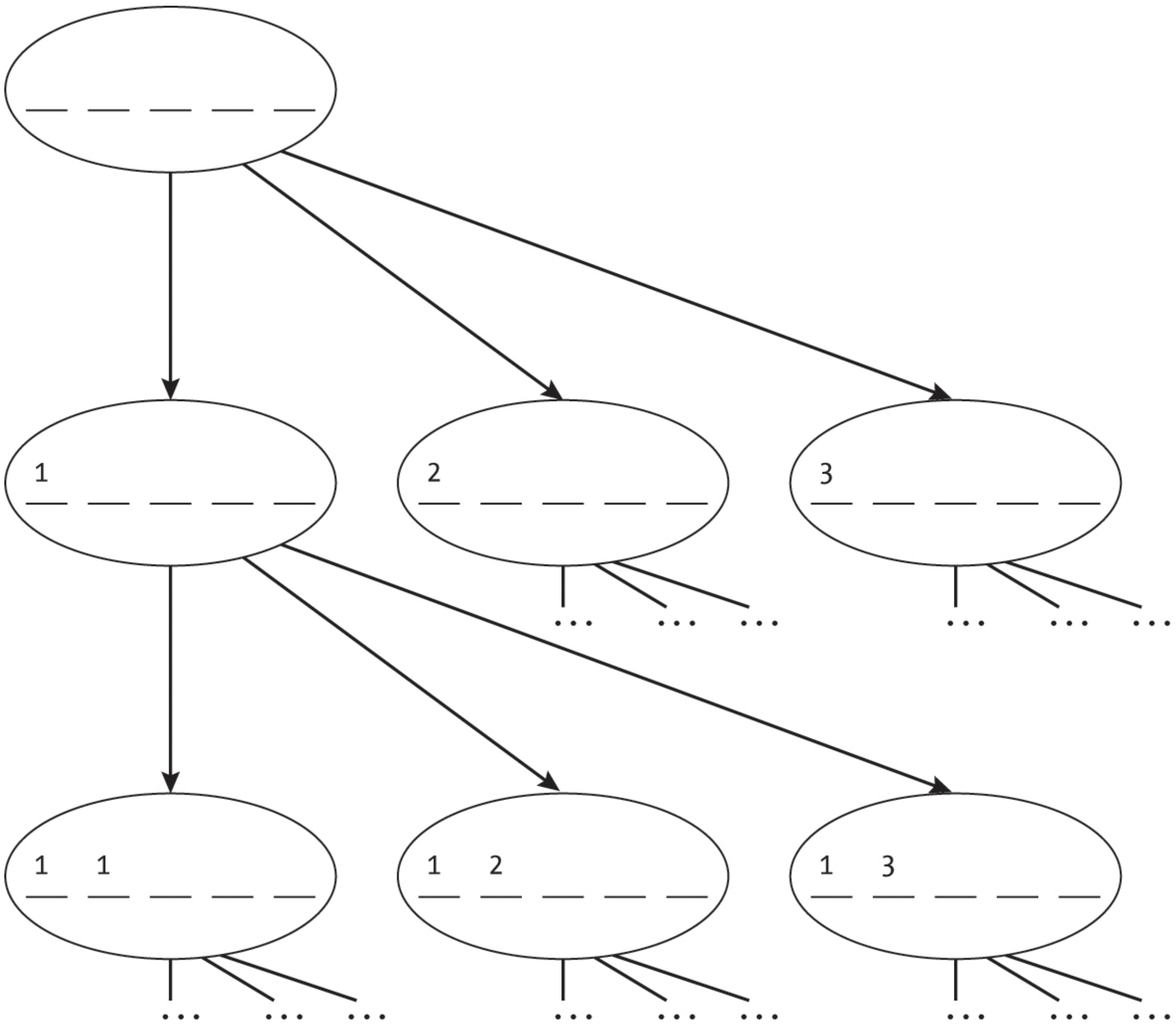


Figure 16-5: A backtracking search branching out over different assignments

We can first model the same search space as the iterator-based exhaustive search, using a depth-first search without any pruning (we'll improve on this approach shortly). This search progresses to an adjacent state by assigning a color to the next unassigned node, as shown in Listing 16-2.

```
def graph_color_dfs(g: Graph, num_colors: int, index: int=
0) -> bool:
    if index == g.num_nodes:
        return is_graph_coloring_valid(g)

    for color in range(1, num_colors + 1):
        g.nodes[index].label = color
      ❶ if graph_color_dfs(g, num_colors, index + 1):
            return True

  ❷ g.nodes[index].label = None
    return False
```

Listing 16-2: A recursive exhaustive search of color assignments

The `graph_color_dfs()` code uses a recursive search to assign each color to each node. It starts with the base case, checking whether all nodes have been assigned and, if so, whether the assignment is valid. If there are more nodes to assign, then the code iterates through all possible colors for the current node. For each color, the code then continues the recursive search on the next node (by index). It returns `True` if the assignment leads to a valid solution ❶. If the search has not found a valid assignment, it resets the current node's color to `None` ❷ and backtracks by returning `False`.

The backtracking search implementation in Listing 16-2 is only an alternative implementation of the iterator-based exhaustive search. It doesn't improve efficiency. Figure 16-6 shows how the search will iterate through the same first dead end as did the exhaustive search in Figure 16-4. After it progresses all the way to the dead end in Figure 16-6(e), the algorithm backtracks and tries a new assignment for node 4, as shown in Figure 16-6(f).

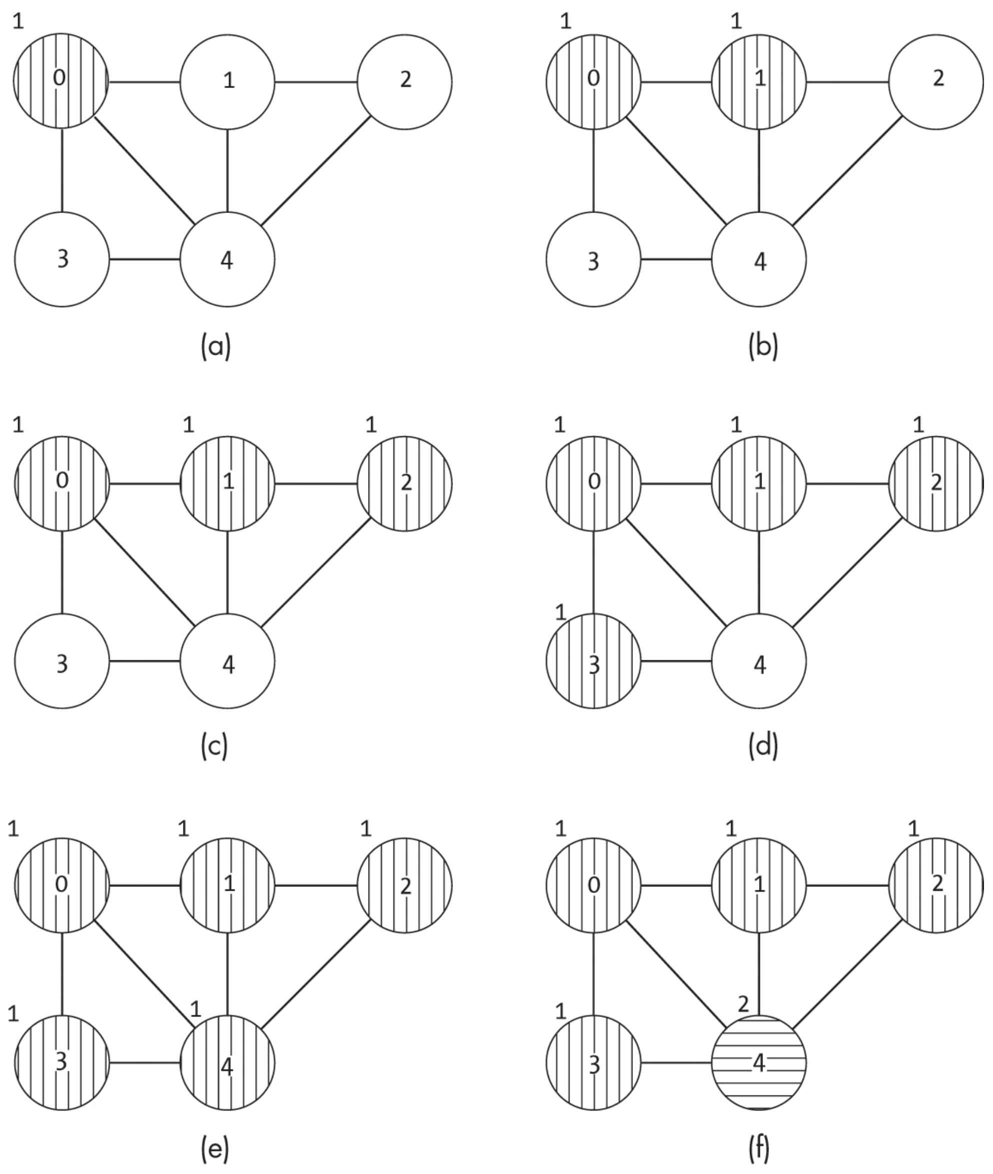


Figure 16-6: The first six steps of the backtracking search coloring on a graph

However, we can modify this search to greatly improve efficiency by *pruning* early and exploring only valid paths. Before assigning a color to a node, we can check whether that assignment produces a conflict. If so, we can skip not only that assignment but also all later recursions from it, as shown in Figure 16-7, where we skip the entire subtree of possibilities stemming from assigning color = 1 to both nodes 0 and 1. Instead, once we have assigned color = 1 to node 0, we consider only colors 2 and 3 for the adjacent node 1.

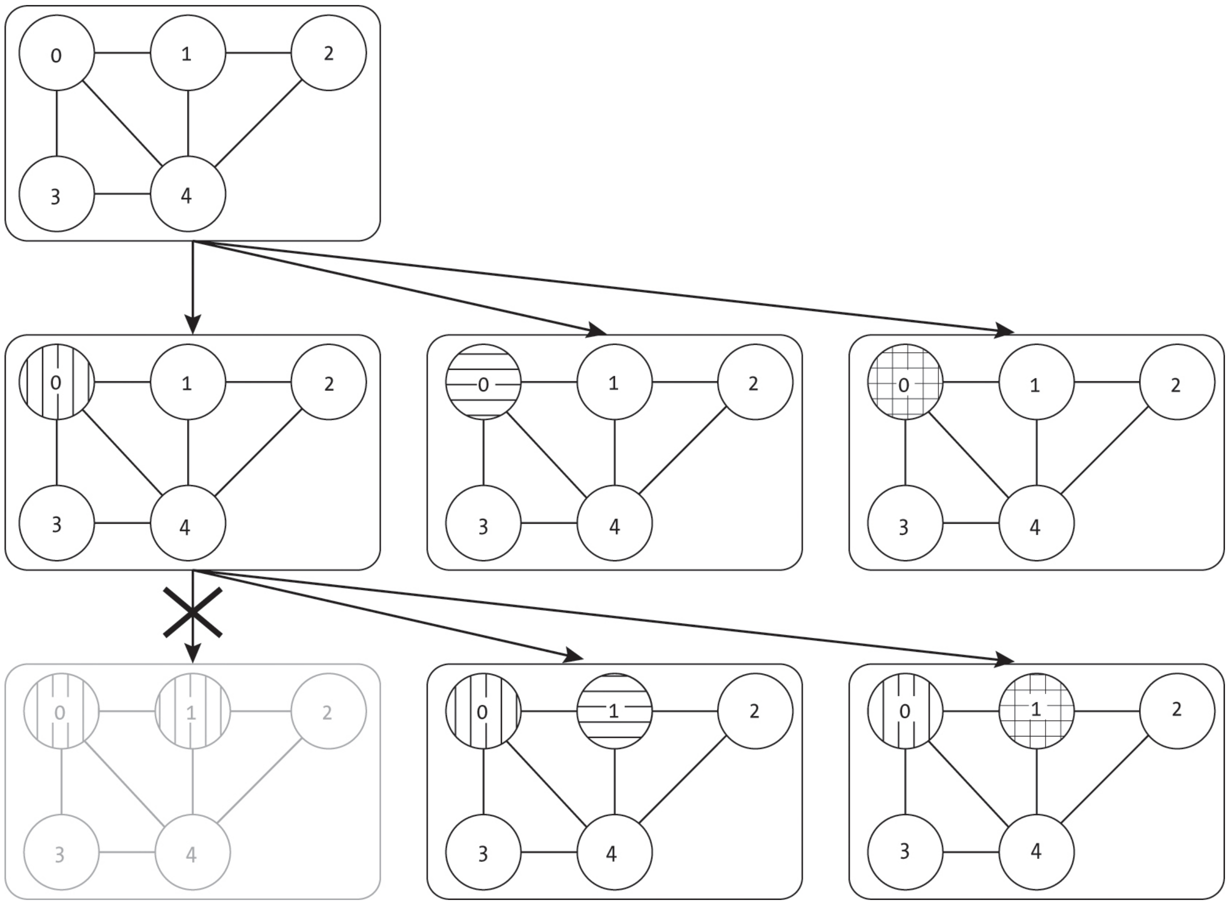


Figure 16-7: The first few steps of a backtracking search with pruning for graph coloring

The code for *backtracking search with pruning* requires a small change. Before assigning a color to a node, we check whether any neighbors already have that assignment. This simple check prevents us from progressing down deep dead ends:

```
def graph_color_dfs_pruning(g: Graph, num_colors: int, ind
ex: int=0) -> bool:
    if index == g.num_nodes:
        return True

    for color in range(1, num_colors + 1):
      ❶ is_usable: bool = True
        for edge in g.nodes[index].get_edge_list():
            if g.nodes[edge.to_node].label == color:
                is_usable = False
        if is_usable:
          ❷ g.nodes[index].label = color
          ❸ if graph_color_dfs_pruning(g, num_colors, inde
x + 1):
                return True
            g.nodes[index].label = None

    return False
```

The code again starts with the base case, checking whether all nodes have been assigned and, if so, returning `True`. The code does not need to check for the validity of the current assignment, because it will do that before assigning each color.

If there are more nodes to explore (assign), then the code iterates through all the possible colors for the current node. It first checks whether any neighboring node uses this color and, if so, marks it as unusable for the current node ❶. If the color is usable, the code continues with the recursive exploration, assigning the color to the node ❷ and recursively proceeding to the next node ❸. As with the approach in Listing 16-2, the code returns `True` if it finds a valid assignment and `False` if it is forced to backtrack.

Figure 16-8 shows a backtracking search with pruning on an example graph with five nodes and $C = 3$.

During its first few steps, the search progresses through assigning valid colors to node 0 in Figure 16-8(a), node 1 in Figure 16-8(b), node 2 in Figure 16-8(c), and node 3 in Figure 16-8(d). When it reaches node 4, it realizes it is at a dead end: none of the three potential colors can be assigned to this node. The search backtracks to where it assigned node 3, but this doesn't help because at that point, node 3 had only one possible valid assignment. The search backtracks again and tries a different assignment for node 2 in Figure 16-8(e). When the search hits its next dead end in Figure 16-8(f), it backtracks all the way to where it assigned node 1 and tries a new color there, as shown in Figure 16-8(g). With the new node 1 assignment, the search can find assignments for the remaining nodes without trouble.

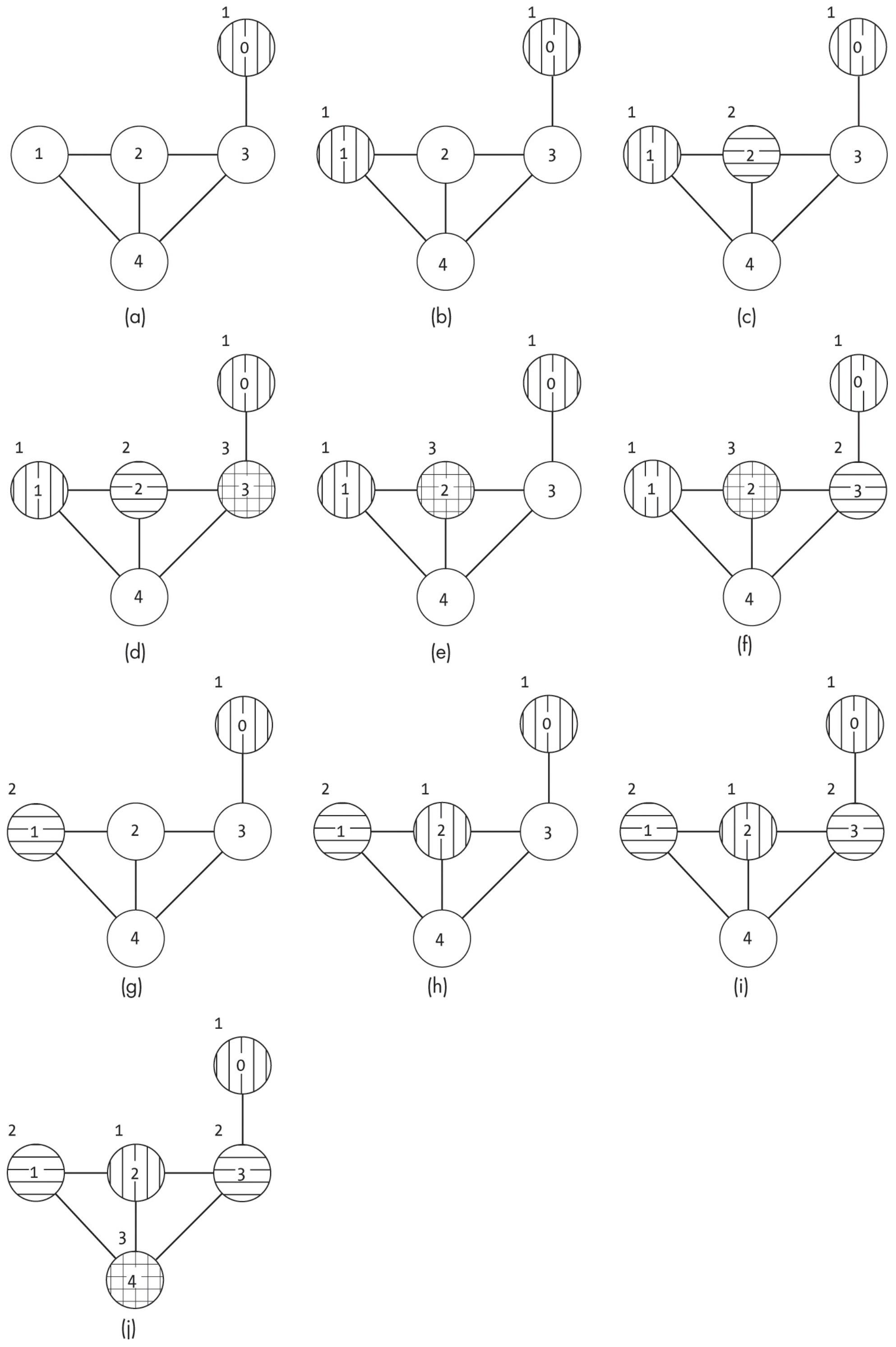


Figure 16-8: Ten steps of a backtracking search graph-coloring algorithm with pruning

Backtracking search with pruning is like a methodical conference planner with a good eraser. They start by making initial table assignments one by one. For each assignment, they check whether there is a known conflict at the table. If so, they skip that assignment and avoid wasted effort —there is no point in searching for the rest of a solution if they already know it will result in a screaming match over the merits of B-trees versus red-black trees. This validity check only helps so much, however. The planner still reaches dead ends where the current person has no valid table. An ardent LISP programmer will have nowhere to go if every table already has at least one Python aficionado. The conference planner takes out their trusty eraser, sighs deeply, and starts backtracking to an earlier point where they could have made a different assignment.

Greedy Search

Beyond these exact but computationally expensive solutions, we can consider heuristic approaches. The *greedy approach* to graph coloring considers one node at a time, picking the first color that does not violate any constraints with the already assigned neighbors. Unlike the exhaustive algorithms described in this section, we define this greedy search without respect to a maximum number of colors. While it will always find some solution, the search's greedy nature means it will not always use the smallest possible number of colors.

The code for the greedy search approach starts with a helper function that finds the first valid color for a node by determining which colors are used by its neighbors, then choosing the first color not in that set, as shown in Listing 16-3.

```
def first_unused_color(g: Graph, node_index: int) -> int:
    used_colors: set = set()
    for edge in g.nodes[node_index].get_edge_list():
        neighbor: Node = g.nodes[edge.to_node]
      ❶ if neighbor.label is not None:
            used_colors.add(neighbor.label)

  ❷ color: int = 1
    while color in used_colors:
        color = color + 1
    return color
```

Listing 16-3: Finding a valid color assignment for a node

The `first_unused_color()` function collects the colors seen in the neighboring nodes in a `set` data structure `used_colors`, which allows it to easily insert new colors and check whether a color has already been used. The code then iterates through all the node's neighbors. For each neighbor, it adds that node's color to the `used_colors` set. It skips unassigned nodes (`neighbor.label == None`) because they do not provide a conflict ❶. The code finishes by using a `while` loop to find the first color that does not occur in the `used_colors` set ❷. While this is not terribly efficient, the loop will always find *some* color that could be used.

Given the helper function, this greedy search can be implemented in a single loop:

```
def graph_color_greedy(g: Graph) -> bool:
    for idx in range(g.num_nodes):
        g.nodes[idx].label = first_unused_color(g, idx)
    return True
```

The `graph_color_greedy()` function iterates through all nodes using the index variable `idx`. For each node, it calls the helper function from Listing 16-3 to find the first color that does not conflict with an assigned neighbor. For consistency with other algorithms in this chapter, the function returns `True` to indicate that it has found a valid coloring.

We can visualize the greedy algorithm through the eyes of the Conference Chair for Banquet Seating at the data structure convention. The organizer iterates through the list of attendees one by one, assigning each to a table before moving on to the next. For each attendee, they review the list of available tables and check whether a conflict would arise with any of the attendees already assigned there. This is equivalent to checking whether the current attendee (node) has a conflict (shares an edge) with any of the table's other occupants (already assigned nodes). If so, the organizer moves on to the next table on the list. If they run out of tables, they sigh, mumble unkind words about the absurdity of programming language fights, and add a new table to the floor.

Figure 16-9 illustrates this greedy search. During the first iteration in Figure 16-9(a), the code assigns a color to node 0. In Figure 16-9(b), it then considers node 1. Since that node shares an edge with node 0, the search cannot reuse color = 1. Instead, it assigns node 1 to color = 2. When it considers node 2 in Figure 16-9(c), the only assigned neighbor has color = 2, so the search can assign node 2 color = 1. This search continues until it has assigned all nodes a color, as shown in Figure 16-9(e).

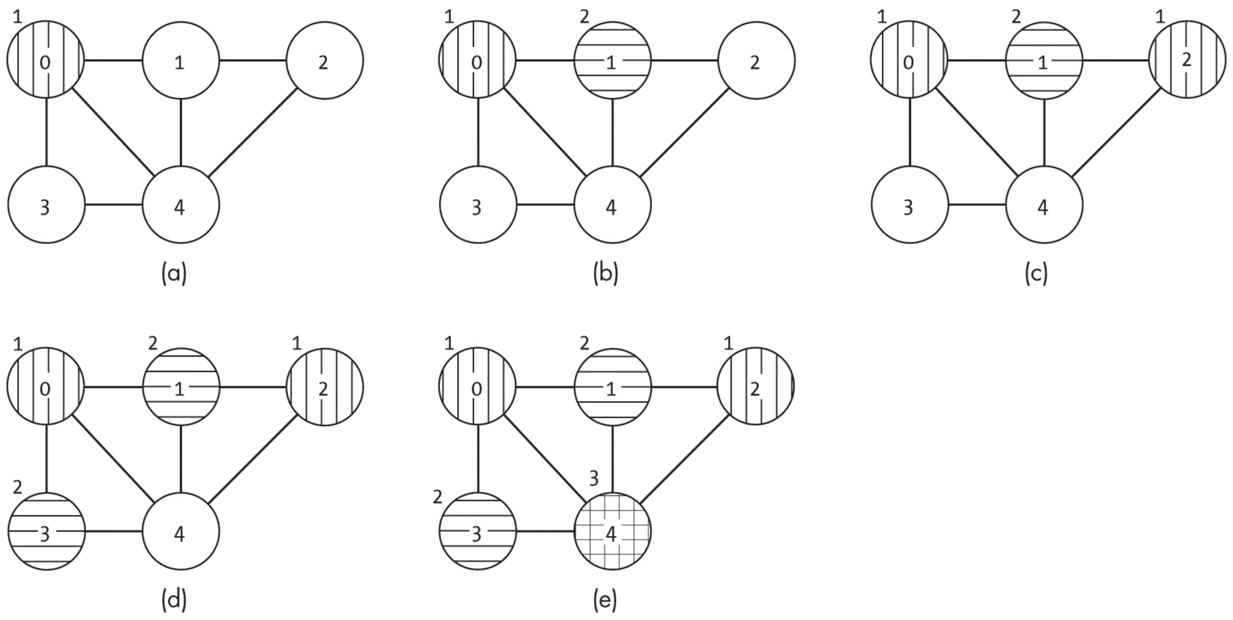


Figure 16-9: The five steps of a greedy coloring algorithm

Given enough colors, the greedy algorithm will find a valid coloring for the graph. However, this assignment is not guaranteed to use the fewest colors. Instead, the order in which the nodes are assigned has a significant impact on how many colors the greedy algorithm will require. Consider Figure 16-10, which shows two valid ways of coloring the same graph.

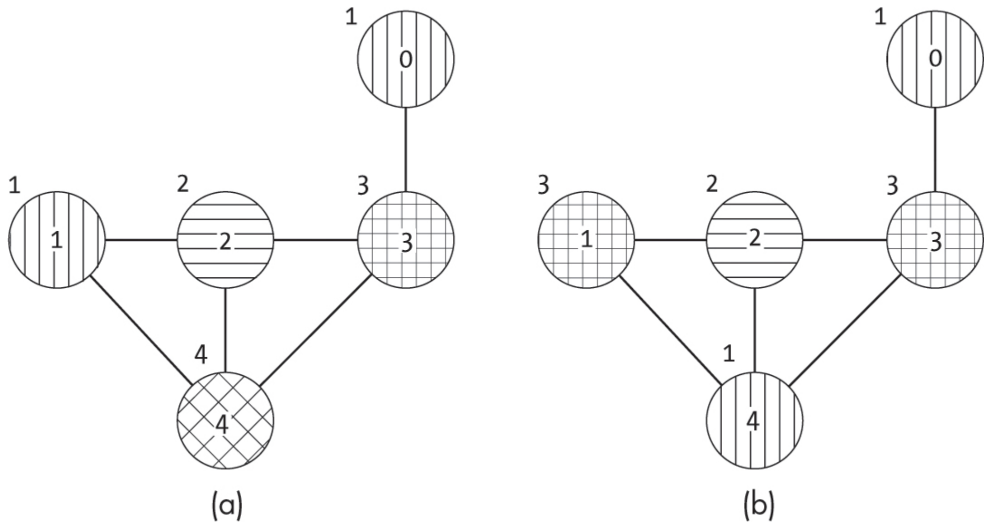


Figure 16-10: A graph where the greedy coloring approach finds a solution with four colors (a) while there exists a solution with only three (b)

Figure 16-10(a) shows the graph coloring produced by the greedy algorithm. Because the search assigns the same color to nodes 0 and 1, it must use a fourth color on node 4. In contrast, depth-first search with pruning would find an optimal coloring requiring only three colors, as shown in Figure 16-10(b). The trade-off is speed versus optimality. While the greedy search will occasionally use more than the minimum number of colors, its lack of backtracking makes it faster.

Node Removal

Another heuristic algorithm worth discussing is the *removal algorithm* proposed by a team of scientists at IBM to find nonconflicting assignments of variables to CPU registers in a compiler. This algorithm works by iteratively simplifying the problem if possible. As with the example of assigning parking spaces, the paper's authors defined register assignment as a graph-coloring problem that uses nodes to represent variables, edges to capture which variables were in use at the same time, and colors to represent each of the CPU's registers. The number of colors is fixed at *C* by the chip's architecture. The algorithm's goal is to determine whether it can find a graph coloring that uses *C* or fewer colors.

As discussed in their paper "Register Allocation via Coloring" by George Chaitin et al., the IBM team proposed a multistep approach to register allocation that includes a node removal algorithm for generating the color assignments. This algorithm relies on the insight that if a node has fewer than *C* edges, we can trivially assign it a color after we have assigned colors to its neighbors. We simply review its neighbors' colors and use any color that doesn't occur among those nodes, reusing our `first_unused_color()` function from Listing 16-3. Therefore, we can initially ignore a node with fewer than *C* edges and focus on the more difficult cases: nodes with *C* or more neighbors. In fact, we can go one step further and temporarily remove any node with fewer than *C* edges from the graph, along with its edges, while we deal with the remaining nodes. We then re-add that node when it is time to assign its color.

Based on this insight, the algorithm iteratively checks the nodes in the current graph and removes any that have *C* or fewer edges, along with those edges. It adds these nodes to a stack to revisit once it has dealt with the more difficult cases. As it removes nodes and edges, new nodes will drop below the *C* neighbor threshold and can be removed as well. It knows that it will be able to easily find a color for those nodes using `first_unused_color()` when it returns to them after assigning colors to the neighbors.

If the algorithm can remove every node from the graph, we know it has a valid coloring with *C* colors. If the search tracks what it removes in a stack, it can pop items from the stack to reverse the operations and reassemble the graph, using it to effectively re-add the nodes to the graph and assign colors along the way.

The code for the removal algorithm uses this two-phase approach:

```
def graph_color_removal(g: Graph, num_colors: int) -> boo
l:
    removed: list = [False] * g.num_nodes
    node_stack: list = []
  ❶ g2 = g.make_copy()

    removed_one: bool = True
    while removed_one:
        removed_one = False
        for node in g2.nodes:
          ❷ if not removed[node.index] and node.num_edges
() < num_colors:
                node_stack.append(node.index)
              ❸ all_edges: list = node.get_sorted_edge_lis
t()
                for edge in all_edges:
                    g2.remove_edge(edge.from_node, edge.to
_node)

                removed[node.index] = True
                removed_one = True

  ❹ if len(node_stack) < g.num_nodes:
        return False

  ❺ while len(node_stack) > 0:
        current: int = node_stack.pop()
        g.nodes[current].label = first_unused_color(g, cur
rent)

    return True
```

The code starts by creating multiple helper data structures. The `removed` array stores a Boolean for each node, allowing the code to quickly check whether a node is still in the graph. The list `node_stack` stores information about the nodes removed from the graph and the order in which they were removed. The code also makes a copy of the graph (`g2`), allowing it to remove edges without modifying the original graph ❶.

The code then enters a `while` loop that continues as long as the function has removed at least one node in the previous iteration (as tracked by the Boolean `removed_one`). Within the `while` loop, the code iterates through each node in the graph, checking whether that node has already been removed and how many neighbors it has ❷. If the node has not been removed and has fewer than *C* (`num_colors`) neighbors, the code adds the node to `node_stack`, removes all its edges ❸, and marks the node as removed. Technically, the code removes only the edges from the graph; the nodes' removals are captured through the `removed` array. This allows us to stably iterate over the graph's nodes in the `for` loop and doesn't impact the accuracy of the algorithm.

If the code does not manage to remove all the nodes from the graph and add them to the stack, it has failed in finding a valid color assignment ❹. In this case, the code returns `False`. If there is a valid assignment, the code assigns the colors one at a time ❺. As it pops each node from the stack, it uses `first_unused_color()` from Listing 16-3 to choose a valid color. Since the node had fewer than `num_colors` neighbors when it was added to the stack, it now must have fewer than `num_colors` neighbors that have been assigned a color. Thus `first_unused_color()` will choose a valid color in the range [1, `num_colors`].

We can picture the removal algorithm in the context of our conference planner by employing the key phrase "I'll just deal with this attendee later." Any time the conference planner sees an attendee with fewer than *C* conflicts, they dismissively say, "This person isn't going to be a problem. I can find a table for them. I'll just deal with them after I've handled the difficult attendees." Outsiders might see this as procrastination, but graph-coloring enthusiasts recognize it as a key algorithmic insight.

Figures 16-11 shows the first phase of this code's operation for *C* = 3. In this stage, the nodes are removed one at a time. During the first iteration of the `while` loop, three nodes are removed. Node 2 has fewer than *C* neighbors and is added to the stack in Figure 16-11(b). Node 3 is removed next in Figure 16-11(c). At this point, node 4 has fewer than *C* neighbors and can also be removed, as shown in Figure 16-11(d).

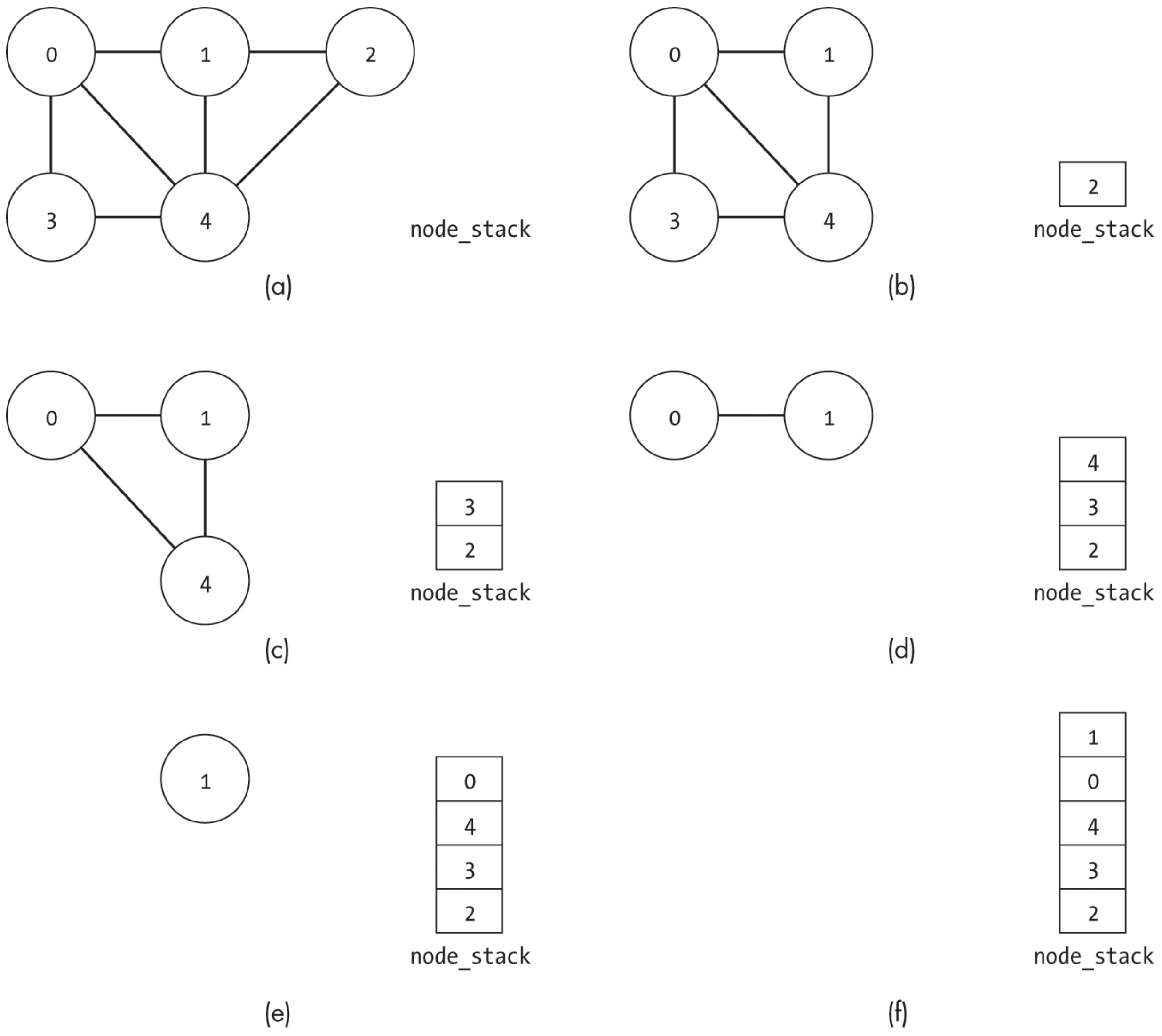


Figure 16-11: The first phase of the removal graph-coloring algorithm

The algorithm has now passed through each node in the graph one time. Since it has removed at least one node during this iteration, it restarts from node 0 and checks again. In Figure 16-11(e) it removes node 0, which has only one remaining neighbor. Finally, it removes node 1 in Figure 16-11(f).

The second phase of the algorithm, shown in Figure 16-12, labels and "re-adds" the nodes. The algorithm starts by popping node 1 from the stack and assigning it color 1, as shown in Figure 16-12(a). In Figure 16-12(b), the algorithm pops node 0 from the stack and assigns node 0 the first color that does not conflict with a neighbor. This process continues for nodes 4, 3, and 2 in Figures 16-12(c), 16-12(d), and 16-12(e), respectively.

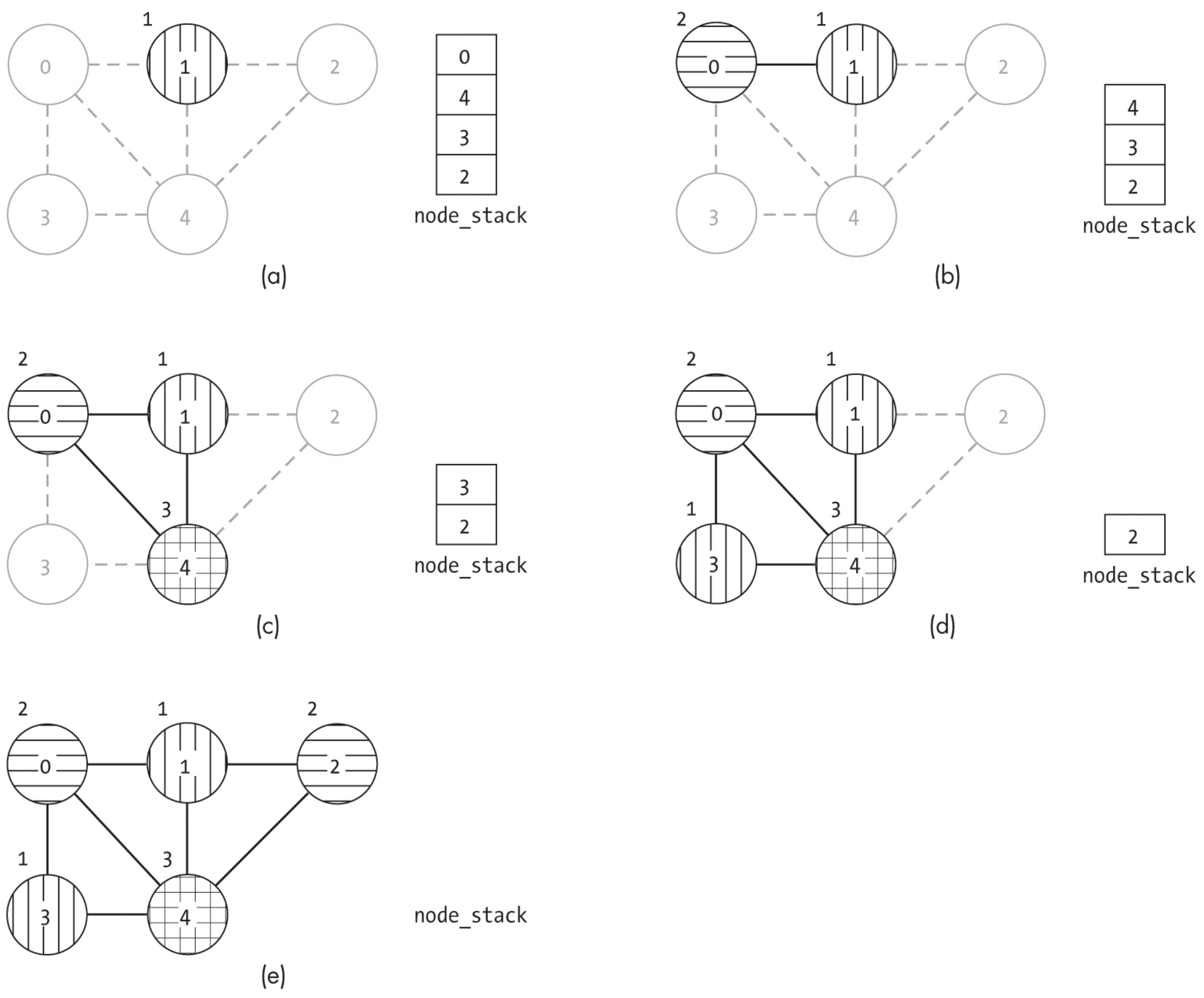


Figure 16-12: The second phase of the removal graph-coloring algorithm

Unfortunately, this heuristic approach is not sufficient to solve every graph. A graph coloring can sometimes use fewer than *C* colors despite a cluster of interconnected nodes with at least *C* edges. In Figure 16-13, for example, the removal algorithm would fail for *C* = 3 despite a valid coloring with only two colors. Since every node has three neighbors, the removal algorithm cannot remove any. It is stuck.

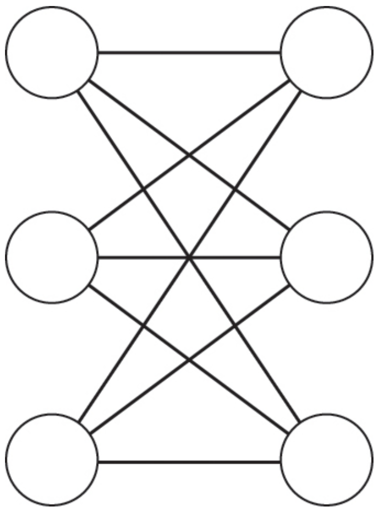


Figure 16-13: An example graph for which the removal algorithm fails

Yet it is obvious that we could create a valid assignment for the graph in Figure 16-13 with only two colors. We could assign all nodes on the left a color of 1 and all nodes on the right a color of 2. Because the edges connect only left nodes to right nodes, there would be no conflicts. In fact, we could solve this particular case with the bipartite labeling algorithm from Chapter 15.

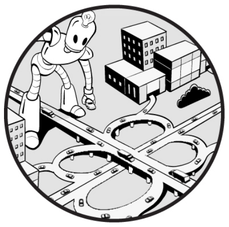
Why This Matters

The problem of finding an assignment of graph nodes to colors has a range of real-world use cases, from planning out magical labyrinths to assigning parking spaces. What makes this problem interesting is that there is no known algorithm that efficiently solves every case. Instead, we need to rely on either exhaustive searches or heuristics. This has led to the development of a variety of approaches aimed at providing good performance under different real-world conditions.

In the next chapter we examine similar assignment problems that do not have a known efficient solution. We look at a range of different branching searches based on the backtracking depth-first search from this chapter, as well as considering a variety of heuristics and the use of randomized algorithms to find solutions.

17

CLIQUES, INDEPENDENT SETS, AND VERTEX COVERS



In the previous chapter, we saw how the seemingly simple problem of assigning colors rapidly explodes into costly searches. Here, we consider the similarly challenging problems of assembling sets of nodes that satisfy various criteria: maximum cliques, maximum independent sets, and minimum vertex covers.

For each of these problems, we want to find the largest or smallest set of nodes that fulfills some criteria based on immediate neighbors or adjacent edges. While it is easy to check whether a single proposed solution satisfies various constraints, it can be computationally expensive to find the best solution. Like the graph-coloring problem, these problems are classified as NP-hard. Again, we can attack these problems with either heuristic or exhaustive approaches.

This chapter begins by returning to the exhaustive backtracking search with pruning that was introduced in the previous chapter, adapting it to exhaustively search for solutions to each of the three problems covered here. In addition, we consider a variety of greedy or heuristic approaches. We'll also discuss real-world applications for each problem, from choosing office locations with cliques to avoiding grudges with independent sets to building guard towers with vertex covers.

Backtracking Search for Sets of Nodes

For each of the problems in this chapter, we want to find a set of nodes that satisfies given constraints. We use a modified version of the backtracking search with pruning introduced in Chapter 16 to find potential solutions by exploring the different assignments for whether each node is included in the set. As with their use in graph coloring, these backtracking searches enumerate all valid solutions. While they'll check every possible valid assignment, they are rarely efficient.

The basic concept behind this search is to explore every possible set of nodes by considering the nodes one at a time and branching the search into two paths at each decision point. In the first path, the search explores the possible sets constructed without including the current node in the set. In the second path, it explores those possible sets constructed with the current node included in the set.

Figure 17-1 shows this approach with each node's inclusion in the set marked as `True` (included) or `False` (excluded). The empty entries in the list indicate nodes we have yet to decide whether to include. At each level, the search considers the next unassigned node and branches out over both potential assignments.

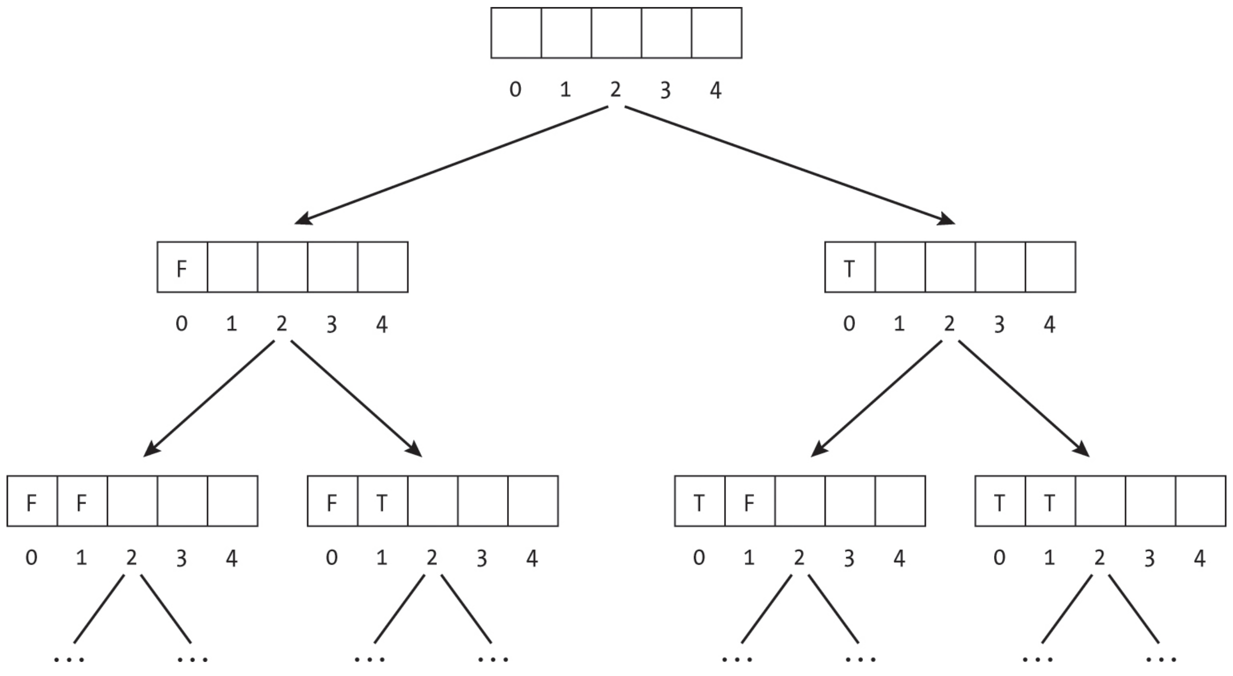


Figure 17-1: A backtracking search to exhaustively try all set assignments

Because we split each branch into two subbranches in Figure 17-1, the number of possible options doubles at each level. For a tree with N decisions, we explore 2^N full assignments. In the case of subsets of graph nodes, we consider each node as a separate decision, so $N = |V|$ and we have $2^{|V|}$ options to explore. While pruning invalid paths will help remove some obviously infeasible results, it will not save us from the full explosion of complexity this search can entail.

We can think of this search as a method of solving a puzzle in a magical dungeon. Upon entering the cold stone room, we find five massive switches along the wall. We know from our previous studies of magical dungeons that only one correct configuration of switches will unlock the door to where the treasure is hidden. Unfortunately, the dungeon's designer was not simply trying to create a fun puzzle; they want to protect their treasure and therefore provide absolutely no hints. While it doesn't take long to check any single guess, we might need to try every combination to find the right one.

Determined to get the treasure, we start with a guess for the leftmost switch (Off), then the second from the left (Off), and so forth until all the switches are in the Off position. When the vault door inevitably doesn't open, we backtrack to the last decision point (where we had set the rightmost switch to Off) and try the On option. When that doesn't work, we backtrack further (to the second rightmost switch), change that to On, and once again explore each possible setting for the final switch. We should consider ourselves lucky that the dungeon designer only had the budget for five switches, meaning we need to test only $2^5 = 32$ settings. But we find it hard to muster such positive thoughts as we backtrack again and again.

For all the algorithms in this chapter, we describe the same two algorithmic approaches for finding solutions. We start by describing an approximate greedy search to establish the fundamentals of the problem and the factors that impact the solutions. We then show how to adapt backtracking search for that problem and how to add pruning.

Cliques

Cliques are subsets of nodes within an undirected graph that are fully connected. Formally, we say that a clique is a set of nodes $V' \subseteq V$ such that:

$$(u, v) \in E \text{ for all } u \in V' \text{ and } v \in V'$$

In a social network, a clique would be a set of people who are all friends with each other.

Figure 17-2 shows a graph with two shaded subsets of nodes. The shaded nodes {1, 2, 5} in Figure 17-2(a) form a clique because the graph contains an edge between each pair of nodes in the subset. In contrast, the shaded nodes {0, 1, 4} in Figure 17-2(b) do not form a clique, as there is no edge between nodes 0 and 4 nor between nodes 1 and 4.

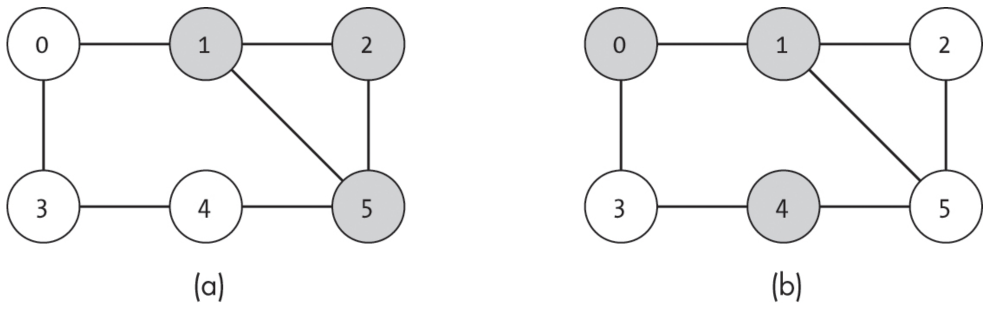


Figure 17-2: A graph with a subset of nodes forming a clique (a) and a non-clique subset of nodes (b)

We can determine whether a set of nodes forms a clique by checking each pair of nodes and confirming that an edge between them exists, as shown in Listing 17-1.

```
def is_clique(g: Graph, nodes: list) -> bool:
    num_nodes: int = len(nodes)
    for i in range(num_nodes):
        for j in range(i + 1, num_nodes):
            if not g.is_edge(nodes[i], nodes[j]):
                return False
    return True
```

Listing 17-1: Checking if a set of nodes forms a valid clique

The code uses a pair of `for` loops to iterate over each pair of nodes in the list and check whether the corresponding edge exists. If the edge is missing, the code immediately returns `False`. If the code successfully examines all pairs of nodes in the list without finding any missing edges, it returns `True`.

We can visualize this check as a nosy outsider in a social network. Hearing tales of the Great Friend Group within a high school, the skeptical outsider proclaims, "There's no way they all actually like each other," and sets out to expose the group's hidden divisions. Firmly in junior detective mode, they corner each person and ask them about their relationship with all the other members of the group: "Are you really friends with Jonny? How about Suzy?" It is not until they have confirmed every pairing is genuine that they finally abandon their skepticism.

While determining whether a given set of nodes forms a clique is straightforward, it's significantly more difficult to build the largest possible clique in a graph. The problem of finding the *maximum clique* consists of finding the largest subset of nodes $V' \subseteq V$ in the graph that form a valid clique. This problem is significantly more difficult than finding an arbitrary clique in the graph because the validity of a node's membership depends on the other nodes in the clique. If adding nodes one by one, early choices can take us in suboptimal directions and exclude later nodes.

Use Cases

We can visualize the importance of cliques by considering locations (nodes) that need to be directly joined by transportation routes (edges). The kingdom's Guild of Adventurers, Explorers, and Cartographers is looking to set up regional headquarters in locations with magical dungeons. After spending hours debating the importance of various criteria, including such considerations as dungeon difficulty and access to fresh produce, they conclude that the number one priority is easy transportation between the offices. After all, the guild offices share a single list of open quests for their members. If an adventurer in the city of Old Melbourne learns of a promising quest at the Cliffs of Indecision, they will want to travel there directly. The guild leaders enlist their senior cartographers to find the largest set of cities such that each city is directly connected by a road. The cartographers, familiar with the problem of finding maximum cliques, set to work enumerating the possibilities.

In a less fantastical world, we might want to use maximum clique detection to choose locations for businesses with direct transportation connections or computational nodes with direct links. Each of these problems consists of finding fully connected subsets within a graph.

Greedy Search

We can define a greedy algorithm for building cliques by starting with an arbitrary node as our clique and continually adding compatible nodes. We always choose to add new nodes that would keep our clique *valid*, which is any node that shares edges with each of the clique members.

Listing 17-2 shows how to list the options for clique expansion by checking each node to see whether we could add it to the set and still have a valid clique.

```
def clique_expansion_options(g: Graph, clique: list) -> li
st:
    options: list = []
    for i in range(g.num_nodes):
      ❶ if i not in clique:
            valid: bool = True
            for j in clique:
              ❷ valid = valid and g.is_edge(i, j)
            if valid:
                options.append(i)
    return options
```

Listing 17-2: Checking which nodes can be added to a clique

The code iterates through each node in the graph and tests whether that node could be added to the clique, first checking whether the node is already part of the clique ❶. If not, the code checks a node's validity by checking that it shares an edge with every node in the current clique ❷. If those tests pass for each node in the clique, the code adds the current node to the list of expansion options.

This function could help the Great Friend Group identify prospective members. Every student in the school is a prospective candidate. For each student who is not already in the group, the friend group's chosen representative asks every member of the group, "Are you two friends?" If the prospective member is already friends with all members of the existing group (there is an edge from the new node to every node in the group), the current members quickly welcome in their mutual friend.

In Listing 17-3, we construct a greedy algorithm that incrementally builds a clique one node at a time.

```
def clique_greedy(g: Graph) -> list:
    clique: list = []
    to_add: list = clique_expansion_options(g, clique)
    while len(to_add) > 0:
      ❶ clique.append(to_add[0])
        to_add = clique_expansion_options(g, clique)
    return clique
```

Listing 17-3: A greedy algorithm to find cliques

The code starts with an empty list to represent the clique being constructed. It uses a `while` loop to continually find a list of potential options with the `clique_expansion_options()` function from Listing 17-2 and adds the first option from the returned list to the clique ❶. It stops and returns the list when there are no more nodes that can be added to the current clique (`len(to_add) == 0`).

When adding nodes one at a time, we immediately run into the question, "Which node do we add next?" In the code in Listing 17-3, we added just the first option, but this could be a *terrible* choice. Consider what happens if we apply this greedy algorithm to the graph in Figure 17-3. As written, the greedy algorithm would choose node 0 first and ultimately return {0, 1} instead of the larger clique {1, 2, 4, 5}.

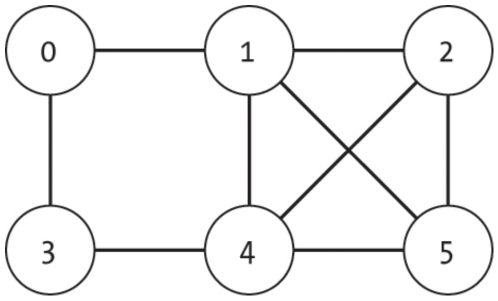


Figure 17-3: A graph for which the greedy search for a maximum clique fails

The greedy search is not guaranteed to find the maximum clique because decisions at each iteration of greedy search are not independent. Each time the algorithm adds a node *u* to the clique, this prevents it from adding any future nodes that do not have an edge to *u*. We can easily get stuck in a local maximum by adding the wrong node early on. We could improve our selection heuristics, such as by choosing nodes with the most edges, but this only helps so much. To build a maximum clique, we need a more comprehensive (and expensive) search.

Backtracking Search

Backtracking search for a maximum clique recursively tries to set one node of the graph as either a member or non-member of the clique, as shown in Listing 17-4. At each level of recursion, the search function takes the clique built so far (`clique`) and the next node to test (`index`) and recursively tests all combinations of unassigned nodes, returning the biggest clique found down that branch of the search. This branching effectively tests all $2^{|V|}$ possible subsets of nodes, while using pruning to cut off invalid options.

```
def maximum_clique_recursive(g: Graph, clique: list, inde
x: int) -> list:
 ❶ if index >= g.num_nodes:
        return copy.copy(clique)
 ❷ best: list = maximum_clique_recursive(g, clique, index
+ 1)

 ❸ can_add: bool = True
    for n in clique:
        can_add = can_add and g.is_edge(n, index)

    if can_add:
        clique.append(index)
        candidate: list = maximum_clique_recursive(g, cliq
ue, index + 1)
     ❹ clique.pop()

     ❺ if len(candidate) > len(best):
            best = candidate

    return best
```

Listing 17-4: Recursively exploring possible cliques

The code for backtracking search starts by checking whether it has reached the termination condition (iterated past the last node in the graph) ❶. If so, there is nothing left to check, and `clique` is the largest subset down this branch of the search. The code returns a copy of the current clique to effectively snapshot the state and separate it from the `clique` object that it will continue to modify during the rest of the search.

If the search has not reached the end of the recursion, the code tries building cliques both with and without `index`. It tests the subset without `index` by calling `maximum_clique_recursive()` with the current `clique` and the index of the next node ❷, then saves the best result down that branch for comparison.

Before testing the subset with the current node, the code avoids exploring invalid subtrees by checking whether this node is compatible with the current `clique` ❸. As with `clique_expansion_options()` in Listing 17-2, the `maximum_clique_recursive()` function checks that the prospective node `index` has edges to all the nodes in the current clique. If even a single edge is missing, adding `index` would result in an invalid clique. The code skips recursive exploration on such invalid sets.

If the current node is compatible with the current clique, the code tries adding it to the clique and recursively testing the remaining options. It then cleans up the `clique` data by removing `index` so that the `clique` list can continue to be used in other branches ❹. The code compares the results of the two branches and keeps the larger valid subset of nodes ❺.

We call the function in Listing 17-4 with an initial `clique=[]` and `index=0` or use a wrapper function:

```
def maximum_clique_backtracking(g: Graph) -> list:
    return maximum_clique_recursive(g, [], 0)
```

Figure 17-4 shows a visualization of the search. Each level shows the algorithm branching on the inclusion (or not) of a single node. The first level considers whether to include node 0. The second level considers including node 1. Nodes assigned to the clique are shaded, excluded nodes are white, and unassigned nodes are dashed circles.

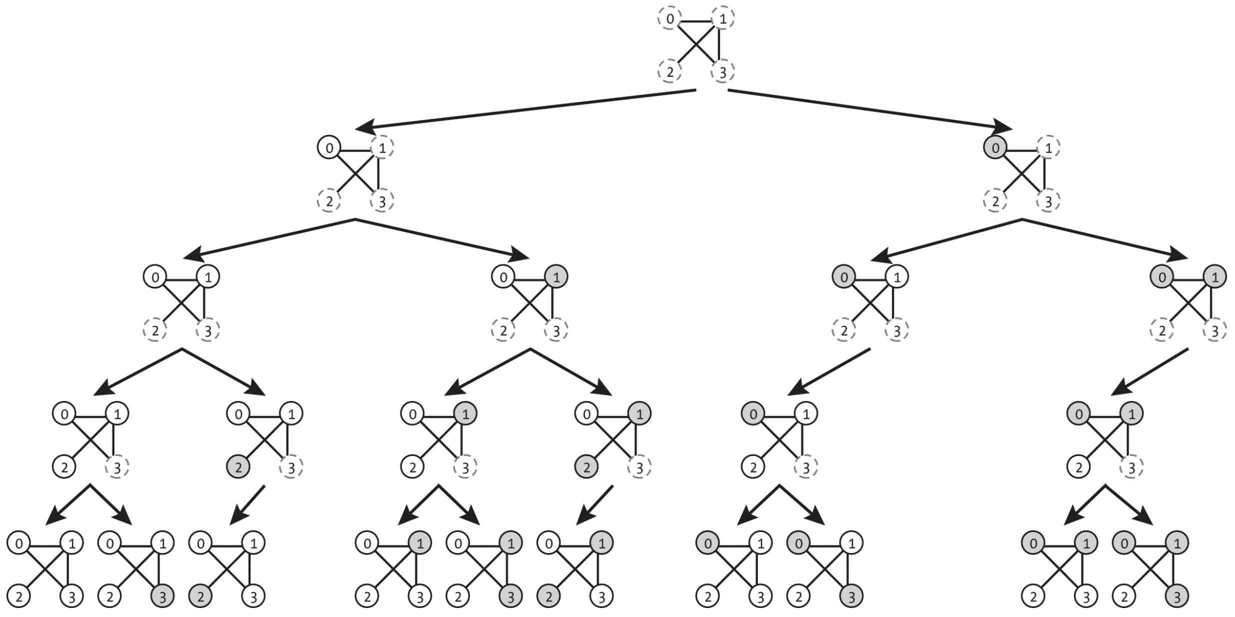


Figure 17-4: The steps of a backtracking search for the maximum clique

The subgraphs in Figure 17-4 show the state of the `clique` list at the start of each function call. As you can see, the function follows only branches that contain valid cliques. For example, when evaluating `clique=[0]` and `index=2`, the search cannot follow the right-hand branch because {0, 2} is not a valid clique. As a result, the search tests only 10 of the 16 possible full combinations, as shown in the final row.

Independent Sets

An *independent set* is effectively the opposite of a clique. We define an independent set within an undirected graph as a subset of nodes such that no two nodes in the set are neighbors. Formally, an independent set is a set of nodes $V' \subseteq V$ such that:

$$(u, v) \notin E \text{ for all } u \in V' \text{ and } v \in V'$$

We can envision choosing an independent set as planning the world's most awkward party: we invite a group of people from our school or office such that no one at the party likes anyone else.

Figure 17-5 shows a graph with two shaded subsets of nodes. The shaded nodes {0, 2, 4} in Figure 17-5(a) form an independent set because the graph does not contain any edges connecting these nodes. In contrast, the shaded nodes {0, 1, 4} in Figure 17-5(b) do not form an independent set, as there is an edge between nodes 0 and 1.

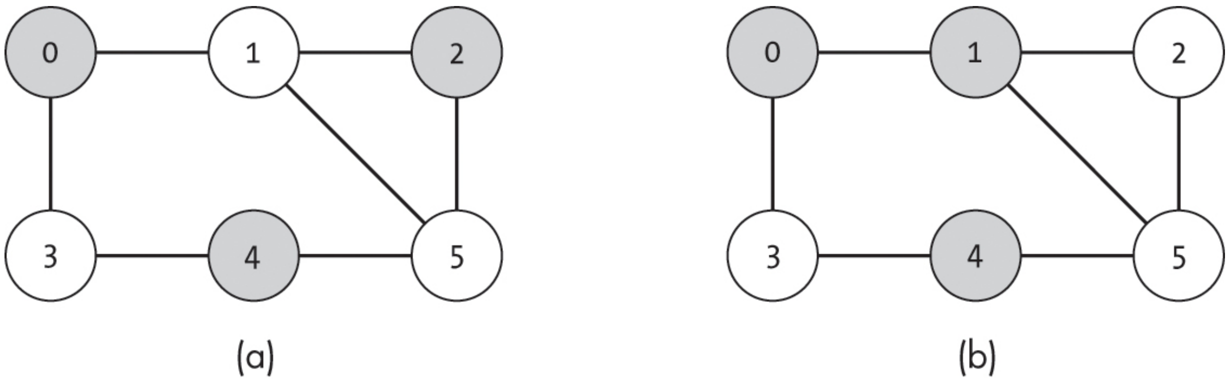


Figure 17-5: A graph with independent (a) and non-independent (b) subsets of nodes

Determining whether a set of nodes forms an independent set requires us to check each pair of nodes and confirm there is no edge between them:

```
def is_independent_set(g: Graph, nodes: list) -> bool:
    num_nodes: int = len(nodes)
    for i in range(num_nodes):
        for j in range(i + 1, num_nodes):
            if g.is_edge(nodes[i], nodes[j]):
                return False
    return True
```

This code is almost identical to the clique-checking algorithm in Listing 17-1. It iterates over each pair of nodes in the list and checks whether they violate the independent set criteria.

In the context of our awkward party, the `is_independent_set()` function plays the role of another skeptical outsider. Unable to bear the silence, they insist, "Some people here must be friends." They query every member of the party about their relationships with each other attendee, asking, "Are you sure you're not friends with them?" and "How about them?" Only when every single question comes back that no pair are friends do they admit that the awkward atmosphere is understandable and that the host must be a bit of a jerk.

As with cliques, generating large independent sets can be difficult because adding a single node to our independent set may impact the validity of other nodes. The problem of finding the *maximum independent set* consists of finding the largest subset of nodes $V' \subseteq V$ in the graph that form a valid independent set.

Use Cases

We can visualize the importance of independent sets by considering a problem where we want to select people (nodes) without negative connections (edges). Imagine being tasked with building a functional project team in a highly dysfunctional organization. Every employee holds a collection of grudges against their coworkers for "misplaced" lunches or forgotten birthdays. In fact, the HR department has gone so far as to build a graph indicating pairwise grudges. Each node represents an employee and undirected edges indicate mutual ill will. The problem of finding a team of employees who do not dislike each other consists of finding a set of nodes within this graph such that no two share an edge.

Alternatively, we can visualize the independent set problem in the context of designing a magical dungeon. Aiming to provide adventurers with an appropriate but not impossible challenge, an evil wizard resolves not to include boss-level monsters in any two adjacent rooms. They model the dungeon level as a graph with nodes as rooms and the tunnels between them as edges. They then set about finding the largest possible independent set of rooms that will contain their boss-level monsters. The rest of the rooms can consist of low-level slimes to give the adventurers a break.

Greedy Search

As with the clique algorithm, we can define a greedy algorithm that builds independent sets one node at a time by adding compatible nodes. We list the options for independent expansion by checking whether each node is valid, as shown in Listing 17-5.

```
def independent_set_expansion_options(g: Graph, current: l
ist) -> list:
    options: list = []
    for i in range(g.num_nodes):
        if i not in current:
            valid: bool = True
            for j in current:
             ❶ valid = valid and not g.is_edge(i, j)
            if valid:
                options.append(i)
    return options
```

Listing 17-5: Finding nodes that can be added to an independent set

The code iterates through each node in the graph and tests whether that node could be added to the independent set. For this function, the code checks that the node under consideration does not share an edge with any node in the current set ❶. Only if those checks pass for every node in the set (and `valid` is still `True`) does the code add the current node to the list of options.

Rather than choosing just any viable node, we can often extend greedy searches by choosing the next node with a heuristic. This heuristic will not guarantee correct results 100 percent of the time, but it can help guide the set construction in better directions. One reasonable heuristic for the independent set problem is to choose nodes with the fewest number of edges, which are likely to have fewer conflicts with other nodes and thus be more compatible with our needs. In the context of our dysfunctional organization, this corresponds to choosing team members who hold the fewest grudges.

Listing 17-6 shows how we can codify this heuristic by modifying Listing 17-5 to return the feasible node with the fewest number of edges.

```
def independent_set_lowest_expansion(g: Graph, current: li
st) -> int:
 ❶ best_option: int = -1
    best_num_edges: int = g.num_nodes + 1

    for i in range(g.num_nodes):
     ❷ if i not in current and g.nodes[i].num_edges() < b
est_num_edges:
            valid: bool = True
         ❸ for j in current:
                valid = valid and not g.is_edge(i, j)
            if valid:
                best_num_edges = g.nodes[i].num_edges()
                best_option = i
    return best_option
```

Listing 17-6: Finding the node with the fewest edges that is compatible with the independent set

The code largely mirrors that of Listing 17-5 but tracks the best node seen (`best_option`) and its number of edges (`best_num_edges`). It starts by setting the best node seen to the invalid option `-1` and the best number of edges to more than could be adjacent to a single node ❶. The code then uses a `for` loop to check each node for feasibility. However, before the feasibility test itself, the code checks whether the node under consideration has fewer edges than the best found so far ❷. If not, it does not matter whether the node is feasible, as the code will not be returning it anyway. It can therefore skip the feasibility test and move on to the next node.

The actual feasibility test is identical to that in Listing 17-5 ❸. The code uses a `for` loop through the existing independent set to test each node against the current candidate `i`. Only if those checks pass for every node in the set does the code save the current node as the new `best_option`. After exhausting all possible nodes, the code returns the best found.

We can build the greedy search by continually adding the best candidate to the independent set:

```
def independent_set_greedy(g: Graph) -> list:
    i_set: list = []
    to_add: int = independent_set_lowest_expansion(g, i_se
t)
    while to_add != -1:
        i_set.append(to_add)
        to_add = independent_set_lowest_expansion(g, i_se
t)
    return i_set
```

The code starts with an empty list `i_set` and uses the `independent_set _lowest_expansion()` function from Listing 17-6 to find the best-looking node to add. It uses a `while` loop to continually find and add nodes until no other nodes can be added.

In the context of our dysfunctional organization example, the greedy search algorithm for finding independent sets consists of building a team one person at a time by always selecting the employee with the fewest grudges who is compatible with everyone previously selected. We start by selecting employees (nodes) who have no conflicts (no edges). We can always add these employees to an independent set. Next, we consider employees with only a single conflict, and so forth, always skipping employees who are incompatible with anyone on the current team.

Greedy search will not always find the maximum independent set. Despite the use of an informative heuristic, this greedy search can still make suboptimal choices that relegate the solution to a local minimum. Consider what happens if we apply this greedy algorithm to the graph in Figure 17-6.

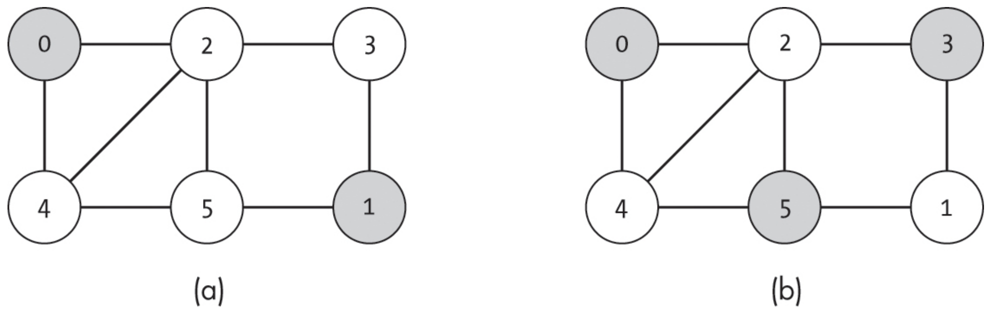


Figure 17-6: The results of a greedy search for maximum independent sets (a) and the true maximum independent set (b)

As shown in Figure 17-6(a), the greedy search will choose node 0 and then node 1, locking itself into a local minimum. If the search had instead selected node 3 as its second choice, as shown in Figure 17-6(b), it would have found {0, 3, 5}.

Backtracking Search

The backtracking search for constructing a maximum independent set again tries to label each node a member or non-member of the set. This branching allows the function to test all combinations of nodes and return the largest independent set found through each branch of the search. At each level of recursion, the function in Listing 17-7 takes the independent set (`current`) constructed so far and the next node to test (`index`).

```
def maximum_independent_set_rec(g: Graph, current: list, i
ndex: int) -> list:
 ❶ if index >= g.num_nodes:
        return copy.copy(current)

 ❷ best: list = maximum_independent_set_rec(g, current, i
ndex + 1)

 ❸ can_add: bool = True
    for n in current:
        can_add = can_add and not g.is_edge(n, index)

    if can_add:
        current.append(index)
      ❹ candidate: list = maximum_independent_set_rec(g, c
urrent, index + 1)
        ❺ current.pop()

            if len(candidate) > len(best):
                best = candidate

    return best
```

Listing 17-7: Recursively exploring possible independent sets

Following the same pattern as the `maximum_clique_recursive()` function from Listing 17-4, the `maximum_independent_set_rec()` function starts by testing whether it has reached the end of the recursion and there are no nodes left to check ❶. If so, it returns a copy of the current independent set as the best found down this branch.

If the search has not reached the end of the recursion, it tries building out independent sets both with and without `index`. It tests the subset without `index` by calling the function with the current independent set and the index of the next node ❷. This effectively skips adding the current index and moves on to considering later nodes. The code saves the best result found down that branch as the baseline for other branches.

The code then checks whether the current node (`index`) is compatible with the independent set under construction ❸. If the current node shares an edge with any node in `current`, adding it would result in an invalid independent set. The code explores only paths that result in valid independent sets (`can_add == True`).

If the current node is compatible with the current set, the code tries adding the node to `current` and recursively testing the remaining options ❹. Afterward, it cleans up the `current` list by removing `index` so that it can continue to use the list in other branches ❺. The code compares the results of the two branches and keeps the larger valid subset of nodes.

We call the function in Listing 17-7 with an initial `current=[]` and `index=0` or use a wrapper function:

```
def maximum_independent_set_backtracking(g: Graph) -> lis
t:
    return maximum_independent_set_rec(g, [], 0)
```

Figure 17-7 shows a visualization of the search where each level depicts the algorithm branching on the inclusion (or not) of a single node. Nodes assigned to the independent set are shaded, nodes excluded are white, and unassigned nodes are dashed circles.

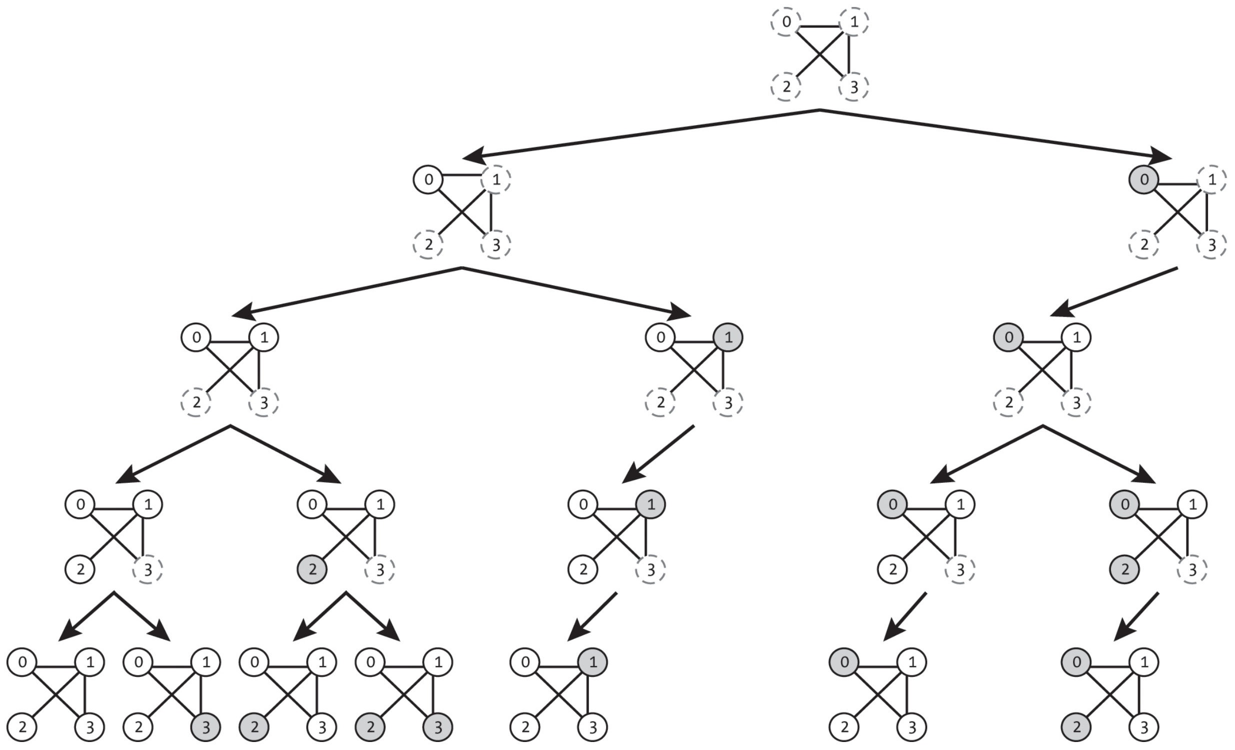


Figure 17-7: The exploration of a backtracking search for maximum independent sets

The subgraphs in Figure 17-7 show the state of `current` at the start of each function call. The first level considers whether to include node 0; the second considers including node 1. Since the function explores only branches containing valid independent sets, it reaches only 7 of the 16 possible full assignments.

Vertex Cover

Whereas the problems of finding cliques and independent sets both focus on whether a pair of nodes are neighbors, the problem of *vertex cover* considers the edges that each node touches. We define a vertex cover within an undirected graph as a subset of nodes such that every edge has at least one endpoint in the set. In other words, each edge is covered by at least one vertex (node). Formally, the vertex cover is a set of nodes $V' \subseteq V$ such that:

For every edge $(u, v) \in E$, we have $u \in V'$, $v \in V'$, or both.

We can envision vertex covers in the context of a kingdom that consists of an archipelago of islands (nodes) connected by bridges (edges). To maintain security, the kingdom constructs tall watchtowers to survey each bridge. The watchtower on each island views every bridge touching the island, allowing the kingdom to be strategic about the towers' locations. However, each bridge (edge) must end on at least one island containing a watchtower (selected node).

Figure 17-8 shows a graph with two shaded subsets of nodes. The shaded nodes {1, 3, 5} in Figure 17-8(a) form a vertex cover because each edge touches at least one shaded node. In contrast, the shaded nodes {0, 1, 4} in Figure 17-8(b) do not form a vertex cover, as the edge (2, 5) is not covered by any node in the set.

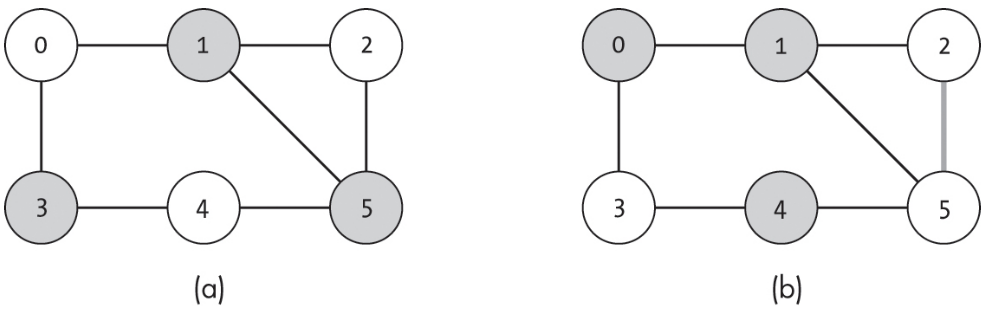


Figure 17-8: A graph with a subset of nodes forming a vertex cover (a) and a subset of nodes that is not a vertex cover (b)

Determining whether a set of nodes forms a vertex cover requires us to check whether each edge in the graph is covered by at least one node in the set:

```
def is_vertex_cover(g: Graph, nodes: list) -> bool:
  ❶ node_set: set = set(nodes)
    for edge in g.make_edge_list():
      ❷ if edge.from_node not in node_set and edge.to_node
not in node_set:
            return False
    return True
```

This code starts by creating a set of the included nodes (`node_set`) to enable fast lookups by using the `set` data structure instead of searching through a `list` ❶. It then loops through each edge in the graph and checks whether both the origin and destination nodes are missing from the set ❷. If neither node is included in the set, the edge is not covered, and the function immediately returns `False`. If the code makes it through all edges, it returns `True`.

The problem of finding the *minimum vertex cover* consists of finding the smallest subset of nodes $V' \subseteq V$ in the graph that form a vertex cover. This problem has direct analogies in cost savings. In the watchtower example, the kingdom wants to build the minimum number of watchtowers that will secure every bridge.

Use Cases

The problem of vertex cover arises naturally in the context of maintenance. Imagine that an evil but tidy wizard has constructed a magical dungeon. Knowing that they cannot leave passages unswept or risk torches burning out, they need to station a crew of emergency-repair minions near each passage. After adventurers blunder down tunnels, knocking stones loose in their fights with monsters, the minions rush forth to repair the damage. For the sake of expediency, the wizard needs to station a crew in at least one of the two rooms at the end of each passage. To minimize costs, the wizard meticulously finds the smallest number of crews they can employ.

In a non-magical context, we might be interested in employing maintenance crews or toll collectors for transportation networks. To keep costs low, we plan a single set of tollbooths through which all incoming and outgoing traffic to the island flows.

Greedy Search

We can build on the greedy algorithm presented for independent sets to create a greedy approach for finding a vertex cover using a subset of the nodes, as shown in Listing 17-8. This time, we use the heuristic of selecting the node that covers the largest number of uncovered edges.

```
def vertex_cover_greedy_choice(g: Graph, nodes: list) -> i
nt:
  ❶ edges_covered: set = set([])
    for index in nodes:
        for edge in g.nodes[index].get_edge_list():
            edges_covered.add((edge.from_node, edge.to_nod
e))
            edges_covered.add((edge.to_node, edge.from_nod
e))

    best_option: int = -1
    best_num_edges: int = 0
    for i in range(g.num_nodes):
        new_covered: int = 0
        for edge in g.nodes[i].get_edge_list():
          ❷ if (edge.from_node, edge.to_node) not in edges
_covered:
                new_covered = new_covered + 1

        if new_covered > best_num_edges:
            best_num_edges = new_covered
            best_option = i

    return best_option
```

Listing 17-8: Heuristically selecting a node to add to a vertex cover

In comparison to the code in Listing 17-6, this code does additional bookkeeping to track edges already covered in the set `edges_covered`. It starts by creating an empty set of covered edges ❶. Because of our `Graph` class's implementation of undirected edges, the code adds each undirected edge in both directions to `edges_covered`.

The main `for` loop is similar to the heuristic for independent sets, where the code iterates through each node in the graph and checks its heuristic value. In this case, the code counts how many of the current node's edges would be newly covered ❷, keeps the best option seen so far, and returns it. If there are no nodes that would increase the number of covered edges (that is, `nodes` already forms a valid vertex covering), the code returns `-1`.

We create a full greedy algorithm by putting a loop around the selection logic within Listing 17-8:

```
def vertex_cover_greedy(g: Graph) -> list:
    nodes: list = []
    to_add: int = vertex_cover_greedy_choice(g, nodes)
    while to_add != -1:
        nodes.append(to_add)
        to_add = vertex_cover_greedy_choice(g, nodes)
    return nodes
```

The code starts with an empty list of nodes (`nodes`) to represent the current selection and uses `vertex_cover_greedy_choice()` from Listing 17-8 to add nodes one by one until it has constructed a valid vertex covering and no additions would increase the coverage.

Note that we could improve the efficiency of this greedy algorithm by maintaining the `edges_covered` set in the outer loop and passing it into the `vertex_cover_greedy_choice()` function. This avoids the cost of recomputing it with each iteration. For the context of this description, we intentionally recompute `edges_covered` so as to keep the selection function stand-alone.

As was the case with all the other greedy algorithms in this chapter, the greedy algorithm for minimum vertex cover is not guaranteed to be optimal. A seemingly good-looking initial choice might prove to be suboptimal in the context of the entire graph.

Imagine the planner in our watchtower example working on the islands shown in Figure 17-9. Determined to keep costs low, the planner selects the island with the highest number of bridges (node 0) for first watchtower. This is generally a good strategy, as that node covers the most edges. However, in this case, it leads to the suboptimal solution shown in Figure 17-9(a). By choosing node 0 first, the planner needs to choose three more islands to cover the rightmost edges. Worse, they will repeat this mistake again and again. As long as the greedy algorithm is using deterministic choices, it will always produce the same results.

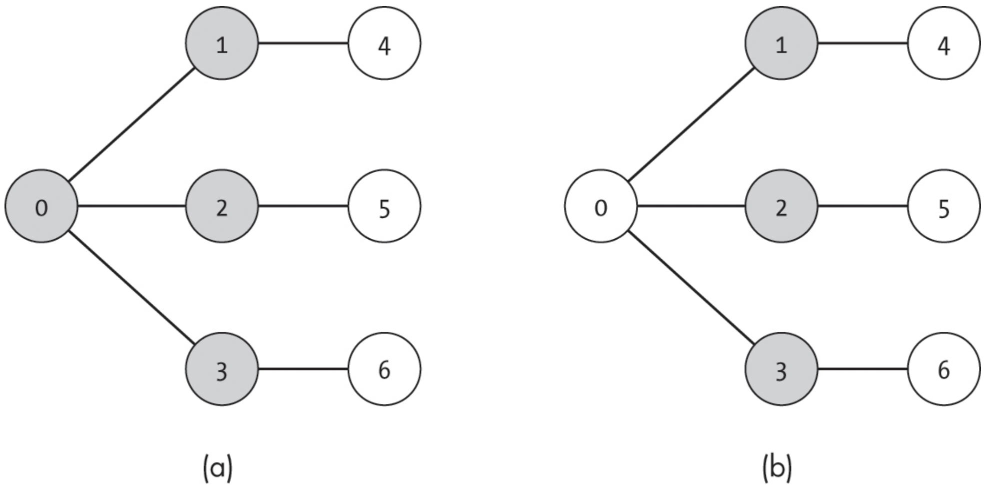


Figure 17-9: The results of a non-optimal greedy search (a) compared to the optimal solution (b) on a minimum vertex cover problem

In contrast, Figure 17-9(b) shows a vertex cover that uses fewer nodes. Once we have included nodes 1, 2, and 3, we no longer need to include node 0.

Backtracking Search

While the backtracking search for minimum vertex cover follows a similar node-by-node approach to both maximum clique and independent set construction, constructing a vertex cover by adding nodes does not offer the same pruning opportunities. Our general pruning approach requires us to start with a valid solution and skip choices that make our candidate set invalid. However, a subset of a valid vertex cover may not cover each edge and thus may not be a valid vertex cover itself. We therefore cannot start with an empty set and build up from there.

We regain the opportunity to prune if we start with a full set of nodes and remove them one by one, instead of adding nodes to a set. Since the set of all nodes is itself a valid vertex cover, we regain the constraint that we are following only branches that remain valid vertex covers.

At each level of recursion, the backtracking search function takes the current vertex cover (`current`) and the next node to test for removal (`index`) and explores the possibilities if it both does and does not remove that node, as shown in Listing 17-9.

```
def minimum_vertex_cover_rec(g: Graph, current: set, inde
x: int) -> set:
 ❶ if index >= g.num_nodes:
        return copy.copy(current)

    best: set = minimum_vertex_cover_rec(g, current, index
+ 1)

    can_remove: bool = True
    for edge in g.nodes[index].get_edge_list():
     ❷ can_remove = can_remove and edge.to_node in curren
t

    if can_remove:
        current.remove(index)
        candidate: set = minimum_vertex_cover_rec(g, curre
nt, index + 1)
     ❸ current.add(index)

        if len(candidate) < len(best):
            best = candidate

    return best
```

Listing 17-9: Recursively exploring possible vertex covers

The code starts by testing whether it has reached the end of the recursion and there are no nodes left to check ❶. If so, it returns a copy of the current vertex cover as the best found down this branch.

If the function has not reached the end of the recursion, the code tries vertex covers with and without `index`. In contrast to the searches in Listings 17-4 and 17-7, however, it is considering whether to *remove* `index`. The default option is to leave `index` in the set by recursively calling the function with the current set and the index of the next node. The code saves the result of this branch as the baseline best result.

Before removing the node, the code checks whether this removal would break the vertex cover. For the set to remain valid without `index`, all the edges currently covered by that node must be covered by another node in current ❷. The code checks this by iterating over each of the current node's edges and checking whether the corresponding neighbor (edge.to_node) is in current.

If it is viable to remove the current node, the code tries removing index from current and recursively testing the remaining options. It then cleans up the current data by re-adding index so the set can be used in other branches ❸. The code compares the results of the two branches and keeps the smaller valid subset of nodes.

We call the function in Listing 17-9 with an initial current equal to a set of all node indices and index=0 by using a wrapper function:

```
def minimum_vertex_cover_backtracking(g: Graph) -> list:
    current: set = set([i for i in range(g.num_nodes)])
    best: set = minimum_vertex_cover_rec(g, current, 0)
    return list(best)
```

Figure 17-10 provides a visualization of this search where each level shows the algorithm branching on the removal (or not) of a single node. Nodes assigned to the vertex cover are shaded, excluded nodes are white, and unassigned nodes are dashed circles initially included in the vertex cover.

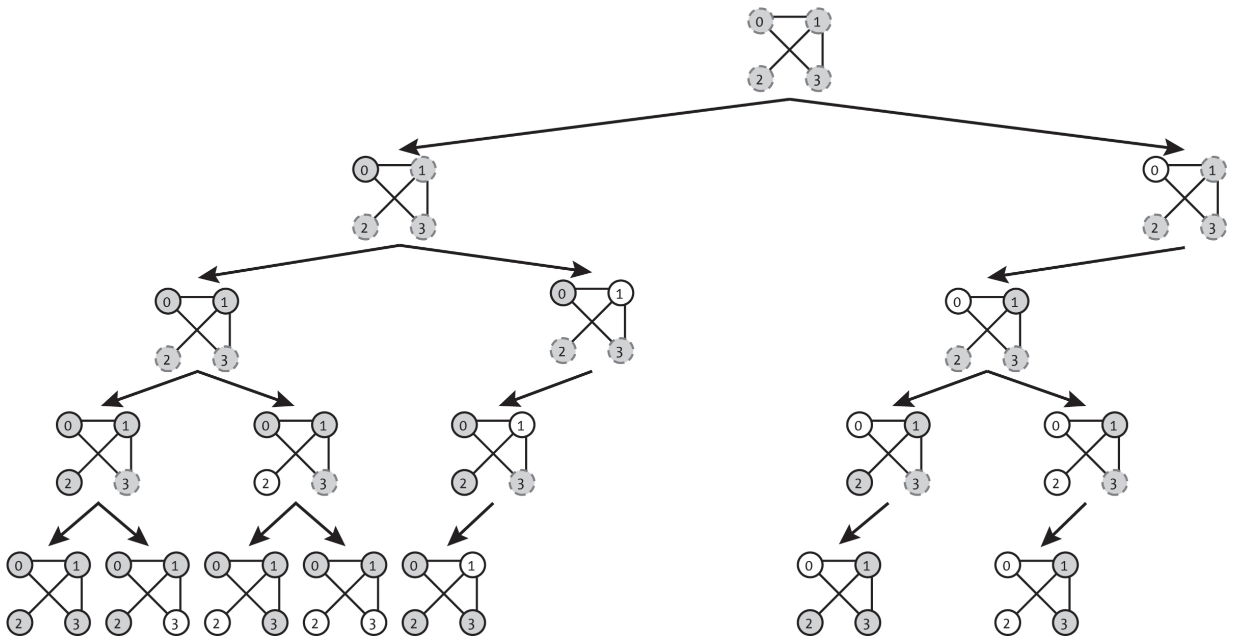


Figure 17-10: A backtracking search for finding vertex covers

The subgraphs in Figure 17-10 show the state of `current` at the start of each function call. Since the function explores only branches containing valid vertex covers, it reaches only 7 of the possible 16 full assignments.

Randomized Algorithms

Another approach to solving the types of assignment problems discussed in this chapter is to evaluate solutions using *randomized algorithms*. Such algorithms use a random number generator to select which node to add to or remove from the set next. At first this might seem unlikely to work. Deterministically minded users might exclaim, "Why add a random node when we could add the best node with greedy search? Won't we waste a lot of time on bad choices?" While randomized algorithms can and will explore suboptimal choices, they offer a few important advantages to consider.

First, randomized algorithms avoid the local minima that can trap greedy algorithms. As we saw in Figure 17-9(a), greedy searches can lead to suboptimal solutions by making each choice in isolation. In contrast, the randomized algorithm will occasionally guess a good solution, like the one in Figure 17-9(b).

Second, a randomized algorithm lends itself well to parallelization: we can run many randomized searches in parallel (without significant coordination), then compare the best results found in each. This is equivalent to having multiple watchtower planners perform their own randomized searches and compare the results, perhaps as part of a kingdom-wide competition where contestants vie to produce the best watchtower plan. Each group can work in isolation without the need for kingdom-wide coordination.

In its simplest form, there is nothing to prevent a randomized search from trying the same solution multiple times. While we could use additional tracking to avoid or at least discourage evaluating duplicate options, this adds complexity and, in the case of parallel searches, the need for coordination. In this section, we focus on the basics of how randomization works and thus keep the implementations simple.

Basic Randomized Search

The simplest randomized search selects viable options completely at random. Let's consider how this works in the context of finding maximum independent sets. We can use the `independent_set_expansion_options()` function from Listing 17-5 as follows to provide a list of feasible options:

```
def independent_set_random(g: Graph) -> list:
    i_set: list = []
    options: list = independent_set_expansion_options(g, i
_set)
    while len(options) > 0:
      ❶ index: int = random.randint(0, len(options)-1)
        i_set.append(options[index])
      ❷ options = independent_set_expansion_options(g, i_s
et)
    return i_set
```

The code starts with an empty independent set (`i_set`) and a list of all nodes as potential options (`options`). The code operates by continuously choosing one of the feasible nodes at random ❶, adding it to the independent set, and rebuilding the set of feasible expansion options ❷. The code uses the `randint()` function from Python's `random` library to select a node, requiring the inclusion of `import random` at the top of the file. The loop continues until there are no more options to add, at which point the code returns the current independent set.

Despite being randomized, this function is guaranteed to produce valid independent sets. During each iteration, the algorithm considers only expansions from a list of feasible options, meaning the independent set remains valid after each addition. We can use a loop to keep searching for a better solution until we hit some maximum number of iterations:

```
def build_independent_set_random(g: Graph, iterations: in
t) -> list:
    best_iset: list = []
    for i in range(iterations):
        current_iset: list = independent_set_random(g)
        if len(current_iset) > len(best_iset):
            best_iset = current_iset
    return best_iset
```

The code starts with an empty independent set (`best_iset`) as the best result seen so far. It then uses a `for` loop to generate and test more options. During each iteration, the code generates a random independent set using `independent_set_random()` and compares its size to the best it has seen so far. It tracks the largest independent set seen as `best_iset` and returns it after testing `iterations` options.

We can picture this search in the context of the earlier example of building teams in a dysfunctional organization. The planner is determined to build the biggest team but lacks the time to do an exhaustive search. Panicked by their tight deadline, they resolve to build 100 random but valid teams and present the best one to their boss. For each of their 100 attempts to create a team, they use random selection to make sure they at least have the possibility of trying options they have not previously considered. After 100 tries, they write up the best team and run to their boss's office to meet the deadline.

Like the greedy search, the randomized search is not guaranteed to find the optimal solution. However, unlike the greedy search, the randomized search can avoid making the same mistake over and over.

Weighted Randomized Search

One potential downside of completely randomized searches is that we have an equal probability of picking a promising node or a terrible node. While there must be some probability of selecting each node to fully explore the solution space, we are not constrained to selecting nodes with equal probability. There's no reason we need to give the office diplomat (who has no interpersonal conflicts) and office troublemaker (who has ongoing feuds with half the company) equal shots at being on the team.

A *weighted randomized algorithm* uses information about the problem structure to define a custom probability distribution for selecting nodes. As a simple example, consider selecting the next node in the maximum independent set problem. Given a subset of nodes $V' \subseteq V$ representing the current independent set, we can define a set of viable candidates C as the set of nodes that are not already in V' and do not share an edge with a node in V'. Formally, we say:

$$\text{For each } u \in C \text{ and } v \in V'\text{: } u \neq v \text{ and } (u, v) \notin E.$$

Given this candidate set C, we can define a probability distribution $p(v)$ of selecting node $v \in V$ where:

$$p(v) = 0 \text{ if } v \notin C$$

and

$$\textstyle\sum_v p(v) = 1$$

For example, we could assign each node a weight that is inversely proportional to the number of adjacent edges, making it more likely that we will select nodes with fewer neighbors.

Why This Matters

For all the problems covered in this chapter and the previous one, it's easy to evaluate a proposed solution but difficult to find the best solution. We've examined a variety of approaches for solving NP-hard graph assignment problems, including greedy searches, randomized searches, exhaustive searches, and customized (heuristic) algorithms. Yet no known approach is efficient for all cases.

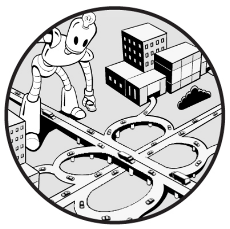
These problems represent only a subset of the NP-hard graph problems. While they do not have known general-purpose efficient algorithms, they often correspond to vital real-world questions. Therefore, it is important to understand not only the structure of the problems but also practical techniques for solving them.

We presented two approaches for each of the problems in this chapter—an approximate greedy solution and an exhaustive solution using backtracking search—to illustrate the problems and the factors making them computationally difficult. These approaches barely scratch the surface of the range of techniques that have been studied. For example, the interested reader can find a bounded approximation algorithm for vertex cover in Cormen, Leiserson, Rivest, and Stein's *Introduction to Algorithms*, 4th edition (MIT Press, 2022). Russell and Norvig's *Artificial Intelligence: A Modern Approach*, 4th edition (Pearson, 2020), provides a good introduction to the powerful world of constraint satisfaction algorithms and applying those to problems such as graph coloring.

In the next chapter, rather than selecting nodes for a set, we tackle the problem of choosing which edges to traverse as part of a tour through the graph.

18

TOURS THROUGH GRAPHS



The problem of designing an optimal sightseeing tour lends itself perfectly to a graph formulation. Depending on our travel preferences, we might want to visit each of a set of major landmarks exactly once, minimize the total distance we travel, or visit every shop-lined street in our destination city. Each of these formulations corresponds to a classic graph problem: planning paths through a graph.

In this chapter, we consider several variations of this core problem. *Hamiltonian paths* visit each node exactly once and can help us plan trips where we want to visit a discrete set of sites. Solving the *traveling salesperson problem* allows us to visit every site while minimizing the total distance we need to walk. Finally, *Eulerian paths* traverse each edge only once and can help us plan trips where we want to wander down each street without repetition.

These types of path-planning problems have a host of real-world applications beyond vacation planning, such as solving logistical problems within the shipping industry. Unfortunately, many also come with significant computational challenges. While the problem of computing Eulerian paths has an efficient computational solution, both Hamiltonian paths and the traveling salesperson problem are NP-hard. We build upon the techniques from the previous chapters to create exhaustive algorithms for solving these problems.

Hamiltonian Paths and Cycles

A *Hamiltonian path*, named after the mathematician William Hamilton, is a path through a graph that visits each node exactly once. We can view the problem in terms of itinerary planning for a thorough yet easily bored tourist. The tourist balances two competing goals. First, they need to ensure they visit every attraction in the city. They do not want to miss a single one. Second, they want to avoid the annoyance of seeing the same spot twice. After all, if you've seen a giant clock tower once, do you really need to see it again?

Figure 18-1 shows a Hamiltonian path [0, 1, 2, 5, 4, 3] on a graph with six nodes. The path starts at node 0, progresses through nodes 1, 2, 5, and 4, and terminates at node 3. Each node is visited only once.

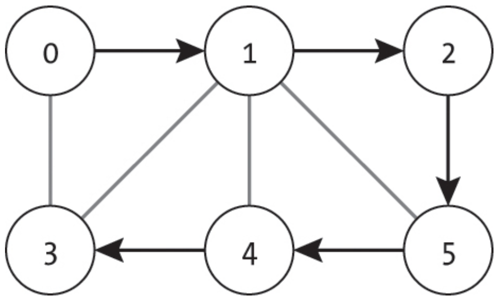


Figure 18-1: A Hamiltonian path

More useful for the tourist's case is the *Hamiltonian cycle* (or circuit), which starts and ends at the same node while traveling to each node exactly once. While the tourist wants to avoid repeat destinations as much as possible, they want to both start and end the journey at their hotel—an acceptable trade-off for not having to carry their luggage through the city.

While the tourist example uses a predefined node (the hotel) as the starting and ending location, Hamiltonian cycles can start or end at any node in the graph. The example Hamiltonian cycle in Figure 18-2 might start at node 0 or any of the other five nodes.

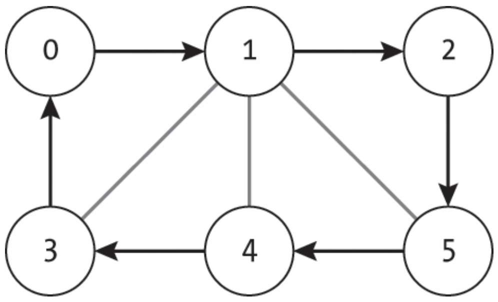


Figure 18-2: A Hamiltonian cycle

Because the path in Figure 18-2 forms a loop, all nodes along it must be reachable from themselves. As we will see, this same flexibility does not apply to Hamiltonian paths in general.

Validating Hamiltonian Paths

Determining whether a path is Hamiltonian requires us to check that each node is visited once. We define a checker function `is_hamiltonian_path()` that takes a path as a list of visited nodes:

```
def is_hamiltonian_path(g: Graph, path: list) -> bool:
    num_nodes: int = len(path)
  ❶ if num_nodes != g.num_nodes:
        return False

    visited: list = [False] * g.num_nodes
  ❷ prev_node: int = path[0]
    visited[prev_node] = True

    for step in range(1, num_nodes):
        next_node: int = path[step]

      ❸ if not g.is_edge(prev_node, next_node):
            return False
      ❹ if visited[next_node]:
            return False

        visited[next_node] = True
        prev_node = next_node
    return True
```

The code starts by checking for potential validity by confirming the path length is equal to the number of nodes in the graph ❶. If there are fewer steps in the path, then not every node could have been visited. If there are more, then some node must have been visited more than once. This also handles the edge cases of an empty path and an empty graph.

If the path is non-empty, the code sets up the data structures to use a Boolean array tracking whether each node has been visited (`visited`) and the previous node seen along the path (`prev_node`). The algorithm starts the check by setting `prev_node` to the first node in the path and marking it visited ❷.

Most of the algorithm consists of a `for` loop through the path. At each step, it checks for an edge between the previous node and current node ❸, then checks that the current node has not yet been visited ❹. If either check fails, the path is not a valid Hamiltonian path, and the code returns `False`. If both checks succeed, the algorithm marks the current node as seen and sets `prev_node` to the current node. If the code makes it through the entire path, it knows that it has visited the same number of nodes as in the graph and has not visited any twice. It returns `True` to indicate success.

Finding Hamiltonian Paths with Depth-First Search

While the problem of finding Hamiltonian paths is NP-hard, we can define an exhaustive search that, though costly, will find such paths. We use a variation of depth-first search that, instead of visiting each graph node once, explores all paths through a node.

Consider the graph in Figure 18-3(a). This graph has the valid Hamiltonian path [0, 1, 3, 2, 4] shown in Figure 18-3(b). However, the depth-first search from Chapter 4 will not visit the nodes in this order but will explore node 2 before node 3.

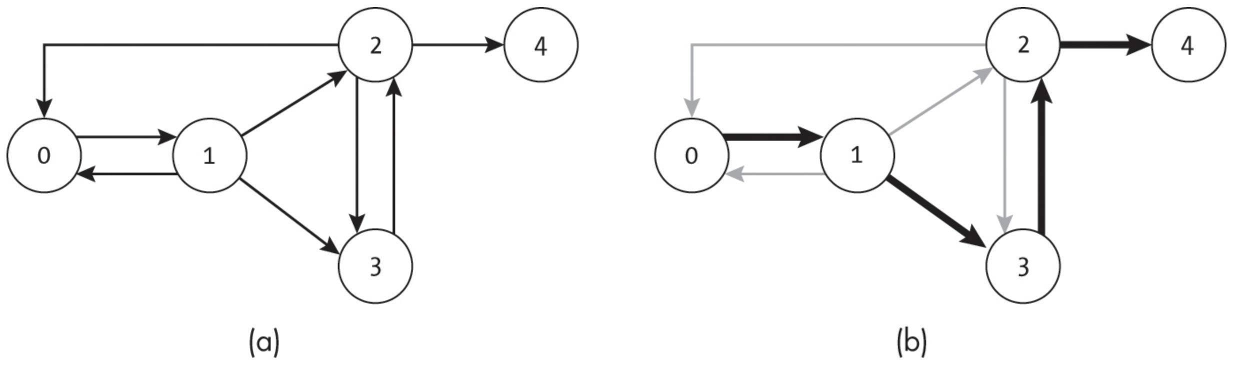


Figure 18-3: A Hamiltonian path that does not match the visit order of the depth-first search

To find valid Hamiltonian paths, we must extend this depth-first search to backtrack and try different paths. The search must reset the nodes visited *after* the current node to unseen so that it can try different paths to get to those nodes.

The code for Hamiltonian depth-first search in Listing 18-1 uses the standard depth-first search with a few modifications.

```
def hamiltonian_dfs_rec(g: Graph, current: int, path: lis
t,
                        seen: list) -> Union[list, None]:
    path.append(current)
    seen[current] = True
  ❶ if len(path) == g.num_nodes:
        return path

    for edge in g.nodes[current].get_edge_list():
        n: int = edge.to_node
        if not seen[n]:
          ❷ result: Union[list, None] = hamiltonian_dfs_re
c(g, n, path, seen)
            if result is not None:
                return result

  ❸ _ = path.pop()
    seen[current] = False
    return None
```

Listing 18-1: A recursive function for searching for Hamiltonian paths

The hamiltonian_dfs_rec() function takes the graph (g), the current node index (current), the path so far as a list of nodes (path), and the Boolean list of nodes visited (seen). It returns a list of nodes representing the path if one is found and None otherwise. We must import Union from Python's typing library to support the type hint of the return value.

The code starts by adding the current node to the path and marking it seen. It then checks whether it has visited g.num_nodes unique nodes ❶. If so, path is a valid Hamiltonian path, and the function returns it.

The core search logic occurs when the path has yet to be completed. The code iterates over the outgoing edges with a for loop and recursively tests unvisited neighbors ❷. If it finds a valid Hamiltonian path along any of these explorations (result is not None), it returns the path immediately. In this case, the function exits without resetting either the nodes' seen value or the path list because it will not continue testing alternate paths.

If the code makes it through each outgoing edge without finding a valid path, it backtracks to the previous node. The code removes the current node from the path and marks it unseen ❸. This allows the search to visit the node via a different path. The code returns None to indicate that it was unable to find a path along this branch.

We define a wrapper that initiates a search from each possible starting node:

```
def hamiltonian_dfs(g: Graph) -> Union[list, None]:
    seen: list = [False] * g.num_nodes
    for start in range(g.num_nodes):
        path: Union[list, None] = hamiltonian_dfs_rec(g, s
tart, [], seen)
        if path is not None:
            return path
    return None
```

The hamiltonian_dfs() function initializes the seen list to all False and uses a for loop to start a recursive search from each starting node. As soon as it finds a path (a non-None result), it returns that path. If it cannot find a valid Hamiltonian path using any starting node, it returns None.

Figure 18-4 shows an example of this updated search. When visiting node 1 in Figure 18-4(a), the search has two options of where to go next, nodes 2 or 3. It starts by exploring node 2, as shown in Figure 18-4(b), which leads to a dead end at node 3, which leaves node 4 unvisited. Unable to visit node 4, it has not found a valid Hamiltonian path. It must backtrack and try alternate paths.

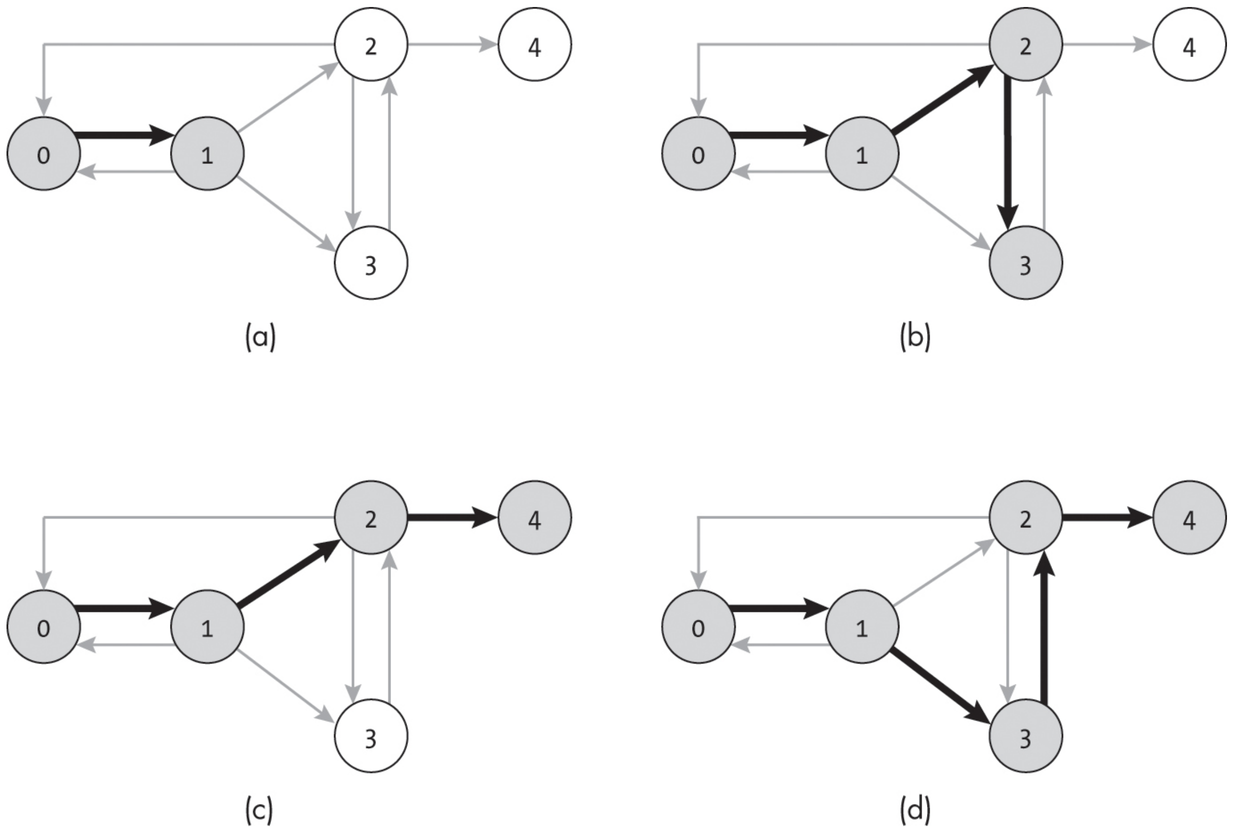


Figure 18-4: The steps along the depth-first-based Hamiltonian path search

The search backtracks to node 2. It marks node 3 unvisited because it is no longer including that node in its current path. The search considers the other paths out of node 2. It has already rejected the edge (2, 0) since it visited node 0 earlier. This leaves the edge (2, 4), as shown in Figure 18-4(c). Unfortunately, taking this edge leads to a dead end, which does not visit node 3. It is blocked again.

The search must backtrack all the way to node 1 and try the path to node 3, as shown in Figure 18-4(d). It resets both nodes 2 and 4 to unvisited and returns to the state shown in Figure 18-4(a). This allows it to travel from node 3 to node 2 and on to node 4.

Unfortunately, it may be insufficient to perform a single depth-first search. Unlike Hamiltonian cycles, the starting node of the search can affect whether the search finds a Hamiltonian path. Figure 18-5 shows a graph for which we can find Hamiltonian paths of [1, 0, 2] when starting from node 1 or [2, 0, 1] when starting from node 2, but no Hamiltonian path starting from node 0.

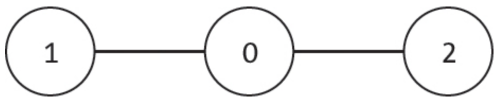


Figure 18-5: A graph with no Hamiltonian path starting at node 0

To combat this problem, we can run a separate depth-first search using each possible starting node. We keep searching until we have exhausted all the starting nodes or found a valid path.

The Traveling Salesperson Problem

The *traveling salesperson problem* is an extension to finding a Hamiltonian cycle that accounts for edge weights. Modeled after the problem of a traveling salesperson planning their itinerary through multiple cities, the goal of this problem is to find a path that (1) starts and ends at the same node, (2) travels to each node exactly once, and (3) minimizes the sum of the edge weights.

Figure 18-6 shows an example graph (a) and the lowest-cost traveling salesperson route through it (b). Spending a few minutes manually trying different paths quickly reveals the difficulty of the problem: the number of possible paths explodes even for simple graphs like this one.

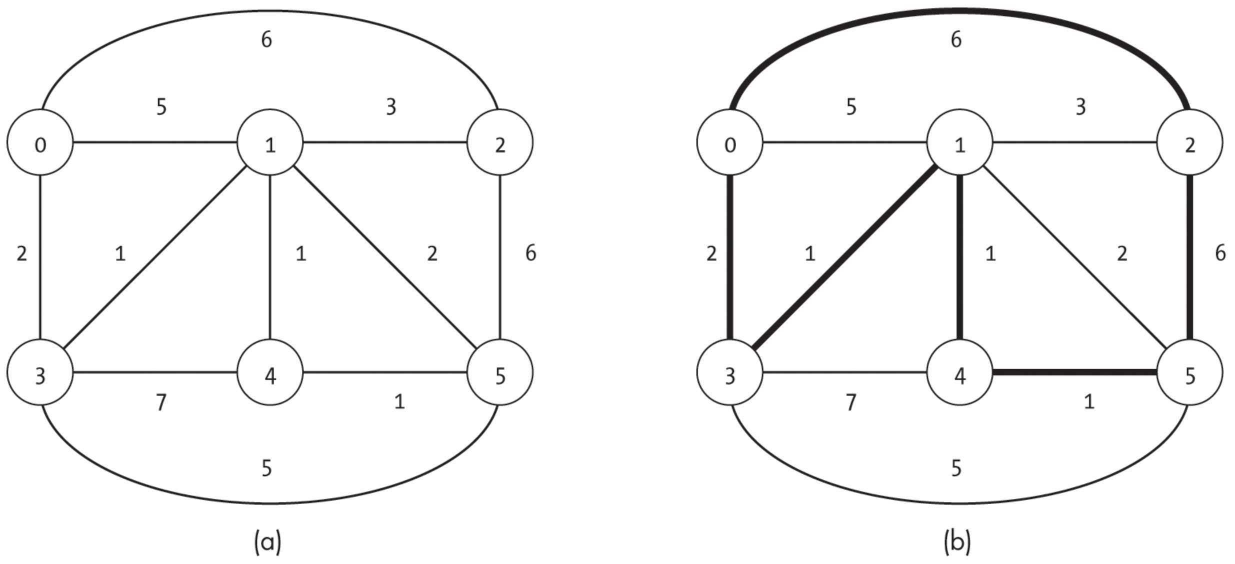


Figure 18-6: An example graph with 6 nodes and 11 edges (a) and an optimal traveling salesperson path (b)

The many concrete applications of this task to real-world routing problems make it critical in a range of domains such as shipping and logistics. Because of this importance, computer scientists and mathematicians have devoted significant time and effort to studying the traveling salesperson problem and have developed numerous approaches, including heuristic searches and approximation algorithms. In this section, we build off the approaches from previous sections and examine a single exhaustive approach based on depth-first search.

Depth-First Search

We can directly adapt the depth-first search algorithm from Hamiltonian paths to account for path cost. In doing so, we need to make three changes. First, since we are looking for cycles rather than paths, we update the algorithm to return to the starting node. Second, we no longer stop as soon as we have found the first valid result. Instead, we continue searching until we have evaluated all valid Hamiltonian cycles so that we can find the lowest-cost one. Third, we track the best path seen so far and its cost.

We define the algorithm to perform a depth-first search over paths, resetting each node's `seen` label as the search backtracks from that node. As with the search for Hamiltonian paths, this allows us to try alternate paths out of each node. Each time we reach the end of the recursion by completing a

Hamiltonian cycle, we return a copy of the path and its cost. The calling function compares the paths and costs from each of the recursive calls and returns the best one.

Unlike the algorithm for Hamiltonian paths, we can begin this search from a single arbitrary node rather than from each starting node, since the result of the traveling salesperson problem is a cycle. The full cycle will have the same cost regardless of which node it uses as a starting and ending point.

The Code

We base the code for the traveling salesperson problem on the code for Hamiltonian path search in Listing 18-1. Again, we start with the recursive function, as shown in Listing 18-2.

```
def tsp_dfs_rec(g: Graph, path: list, seen: list, cost: fl
oat) -> tuple:
    current: int = path[-1]

  ❶ if len(path) == g.num_nodes:
        last_edge: Union[Edge, None] = g.get_edge(current,
path[0])
        if last_edge is not None:
            return (cost + last_edge.weight, path + [path
[0]])
        else:
            return (math.inf, [])

    best_path: list = []
    best_score: float = math.inf
    for edge in g.nodes[current].get_edge_list():
        n: int = edge.to_node
        if not seen[n]:
          ❷ seen[n] = True
            path.append(n)

          ❸ result: tuple = tsp_dfs_rec(g, path, seen, cos
t + edge.weight)
          ❹ seen[n] = False
```

```
            _ = path.pop()

            if result[0] < best_score:
                best_score = result[0]
                best_path = result[1]

    return (best_score, best_path)
```

Listing 18-2: A recursive function for the traveling salesperson problem

The `tsp_dfs_rec()` function takes the graph (`g`), the path so far (`path`), the Boolean list of nodes visited (`seen`), and the cost so far (`cost`). It extracts the index of the current node as the current final node in `path`.

The function starts by considering the base case for the recursion where all the nodes have been visited ❶. It checks whether the path can be made into a cycle by returning to the starting node (`path[0]`). If so, the code returns a tuple with the cycle's cost and a new copy of the full cycle (`path + [path[0]]`). If there is no edge back to the starting node, the code returns an infinite cost to indicate that it is not a valid cycle.

If the algorithm has more nodes to explore, the code iterates over the current node's outgoing edges with a `for` loop and recursively tests unvisited neighbors ❸. Unlike in the code for Hamiltonian paths in Listing 18-1, the code augments ❷ and resets ❹ both the `seen` list and `path` in the calling function. This simplifies the earlier logic for the base case. After recursively exploring the neighbor, the code checks whether it has found a better result and, if so, saves it. It finishes by returning the best cost and cycle found through this branch.

We define a wrapper that sets up the data structures and initiates the search:

```
def tsp_dfs(g: Graph) -> tuple:
    if (g.num_nodes == 1):
        return (0.0, [0])

    seen: list = [False] * g.num_nodes
    path: list = [0]
    seen[0] = True
    return tsp_dfs_rec(g, path, seen, 0.0)
```

The code starts by checking the edge case where the graph has a single node and returns the appropriate answer. It then sets up the initial `seen` and `path` lists for the search, starting the path at node 0 and marking it seen. Finally, it runs the search and returns the result.

The recursive function shown in Listing 18-2 is a basic implementation of the traveling salesperson algorithm. We could improve its efficiency with additional pruning. For example, we could prune the current path if its cost exceeds the best cost so far. Similarly, we could incorporate heuristics like exploring the neighbors in order of increasing edge weight to focus on potentially lower-cost paths. As noted earlier, the vast variety of optimizations and heuristics for the traveling salesperson problem far exceeds the scope of this chapter.

An Example

Figure 18-7 shows the results of running this search on the graph from Figure 18-6(a). Each subfigure shows the base case where the algorithm has found a Hamiltonian cycle, with the path highlighted in bold and the cost listed below it.

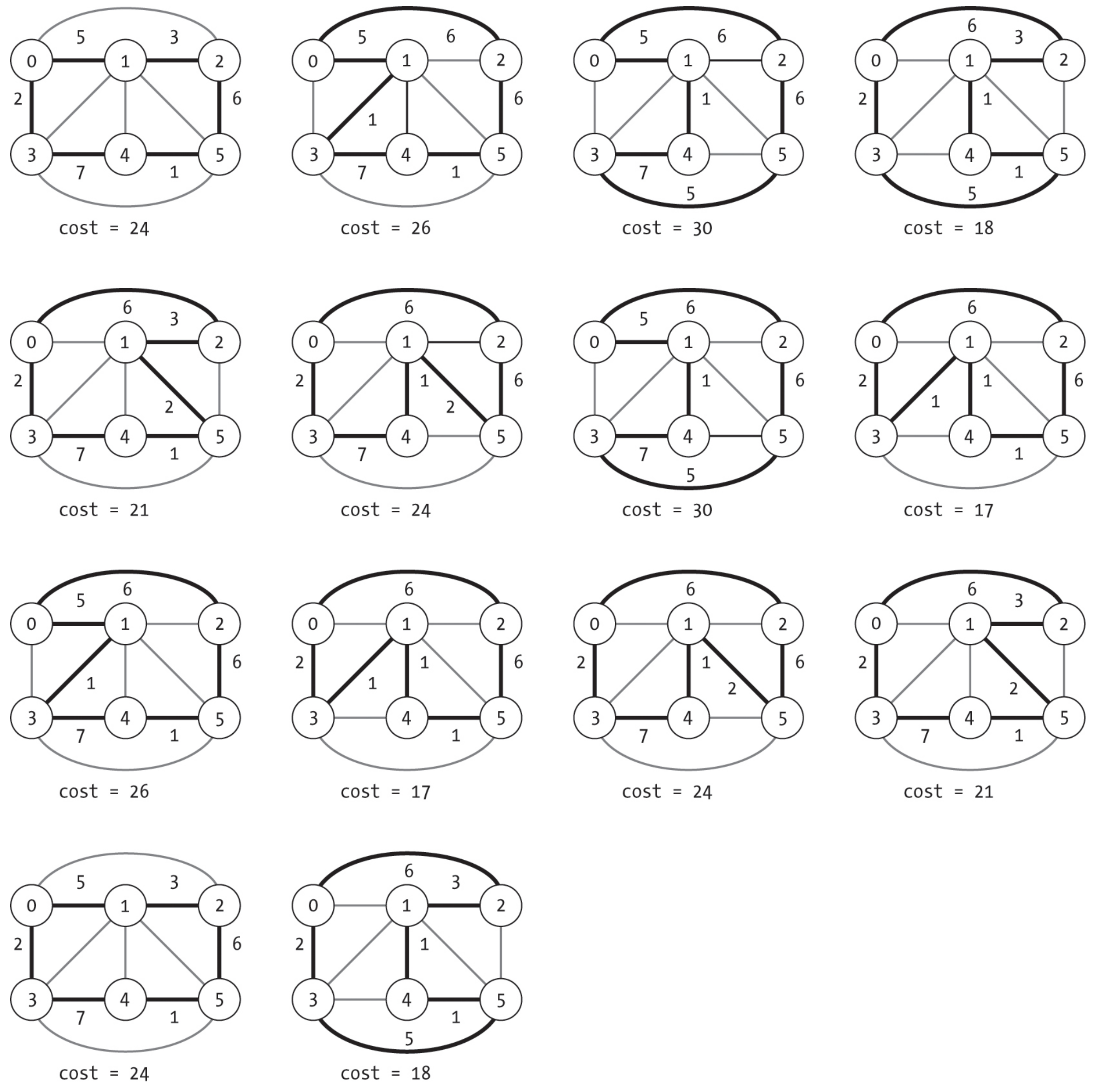


Figure 18-7: The 14 Hamiltonian cycles explored during the depth-first search

Each path appears in Figure 18-7 multiple times because the algorithm will find the cycle in both directions. For example, the first subfigure corresponds to the path [0, 1, 2, 5, 4, 3, 0], while the second to last corresponds to the path [0, 3, 4, 5, 2, 1, 0].

Eulerian Paths and Cycles

An *Eulerian path*, which is named after the mathematician Leonhard Euler, is a path through a graph that traverses each edge exactly once. We can view the problem in terms of an efficient window-shopping tourist. Determined to survey all the stores in a city, our tourist searches for a path that will take them down every road exactly one time. They refuse to miss out on potential finds by skipping a road or to waste their time passing stores they have already seen. An *Eulerian cycle* is an Eulerian path that starts and ends at the same node, providing the ideal planning tool if the tourist wants to start and end at their hotel but is concerned about traveling each road exactly once.

NOTE

Remember that, as described in Chapter 3, we are using the definition of path that is common in computer science texts and allows repeated nodes. This differs from the formal graph theory definition of a path, which does not allow for repeated nodes. In graph theory, this problem may be referred to as finding an Eulerian trail.

Figure 18-8 shows an Eulerian cycle on a graph with six nodes. The path starts at node 0 and consists of [0, 1, 2, 5, 1, 3, 4, 5, 3, 0]. While it revisits nodes, it traverses each of the nine edges only once. The tourist may pass through the same intersection multiple times but will venture past each street's shop windows only a single time.

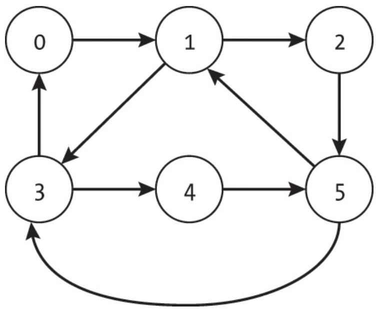


Figure 18-8: An Eulerian cycle

Not all graphs contain Eulerian paths. Figure 18-9 shows an undirected graph where no Eulerian path is possible. After moving from node 1 to any other node, the search would need to use the same edge to return to node 1.

Since nodes 0, 2, and 3 are connected only to node 1, any path through all edges would need to return to node 1.

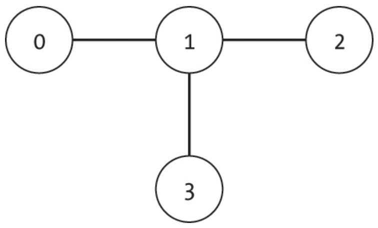


Figure 18-9: A graph without an Eulerian path

Leonhard Euler devised a simple and effective way to test whether a connected, undirected graph has an Eulerian cycle:

A connected, undirected graph has an Eulerian cycle if and only if all nodes have an even degree.

Using this test, we can define a helper function for testing if a graph both is fully connected and has an Eulerian cycle, as shown in Listing 18-3.

```
def has_eulerian_cycle(g: Graph) -> bool:
 ❶ components: list = dfs_connected_components(g)
    for i in range(g.num_nodes):
     ❷ if components[i] != 0:
            return False

     ❸ degree: int = g.nodes[i].num_edges()
        if i in g.nodes[i].edges:
            degree += 1
        if degree % 2 == 1:
            return False
    return True
```

Listing 18-3: Checking if a graph is fully connected and has an Eulerian cycle

The code starts by using the `dfs_connected_components()` function from Chapter 4 to label each node's component ❶. It then uses a `for` loop to examine each node and check that it is part of the same component ❷ and has an even degree.

For completeness, the degree computation in `has_eulerian_cycle()` handles the case of undirected self-loops ❸. As noted in Chapter 2, edges forming self-loops in undirected graphs are counted twice for the degree, since they contact the node on each end.

If the code finds a disconnected component or a node with an odd degree, it immediately returns `False`. If it checks all nodes without a problem, it returns `True`.

Validating Eulerian Paths

To determine whether a path is a valid Eulerian cycle, we need to check that each edge is used a single time. We define a checker function that takes a path as a list of nodes:

```
def is_eulerian_cycle(g: Graph, path: list) -> bool:
    num_nodes: int = len(path)
  ❶ if num_nodes == 0:
        return g.num_nodes == 0

  ❷ used: dict = {}
    for node in g.nodes:
        for edge in node.get_edge_list():
            used[(edge.from_node, edge.to_node)] = False

    prev_node: int = path[0]
    for step in range(1, num_nodes):
        next_node: int = path[step]
      ❸ if not g.is_edge(prev_node, next_node):
            return False
      ❹ if used[(prev_node, next_node)]:
            return False

      ❺ used[(prev_node, next_node)] = True
        if g.undirected:
            used[(next_node, prev_node)] = True

        prev_node = next_node

  ❻ for value in used.values():
        if not value:
            return False
❼ return path[0] == path[-1]
```

The `is_eulerian_cycle()` code starts by checking the edge case of an empty path, which is assumed to be valid only if the graph has no nodes ❶. If the path does have edges, the code builds a dictionary `used` that maps each edge in the graph to a Boolean indicating whether that edge has been visited ❷.

The main body of the code consists of a `for` loop that walks the path, using a combination of the previous node (`prev_node`) and the current node (`next_node`) to identify the current edge. The function immediately returns `False` if the path uses an edge that does not exist ❸ or has already been traversed ❹. Otherwise, the code marks the edges as visited ❺, taking care to mark entries for both directions if the edges are undirected.

The code completes by checking that it has visited each of the edges and returning `False` if it finds an unvisited edge ❻. (Alternatively, we could structure the code to count the total number of edges and the number of edges seen, handling the undirected case correctly, and just compare the counts.) The function uses a final check that the start and end nodes are the same, meaning the path is a cycle ❼.

Finding Eulerian Cycles with Hierholzer's Algorithm

Unlike the previous two problems in this chapter, the problem of finding Eulerian cycles is not NP-hard, and there exists an efficient method for finding an Eulerian cycle in a graph. The mathematician Carl Hierholzer developed an algorithm for extracting the Eulerian cycle in graphs that have one. *Hierholzer's algorithm* operates by repeatedly finding cycles over unused edges and removing those cycles from the graph. Since the algorithm requires that the graph does have an Eulerian cycle, we use the Euler's degree-based test (and the code in Listing 18-3) to precheck the graph.

The primary insight behind this approach is that if a graph has an Eulerian cycle, we can build this full cycle from a series of potentially smaller cycles. We call these smaller cycles *subloops* to distinguish them from the full Eulerian cycle. The algorithm starts by finding any cycle in the graph and removing its edges, which may leave some edges in the graph.

Since the graph has a full Eulerian cycle that uses all the edges, the algorithm can insert these remaining edges into the full path by splicing in additional subloops that each start and end at the same node in the current path.

Figure 18-10 shows an example of this algorithm. In Figure 18-10(b), the search finds an initial cycle [0, 1, 2, 5, 3, 0] that uses five edges and visits the five shaded nodes. It then removes these edges, as shown in Figure 18-10(c).

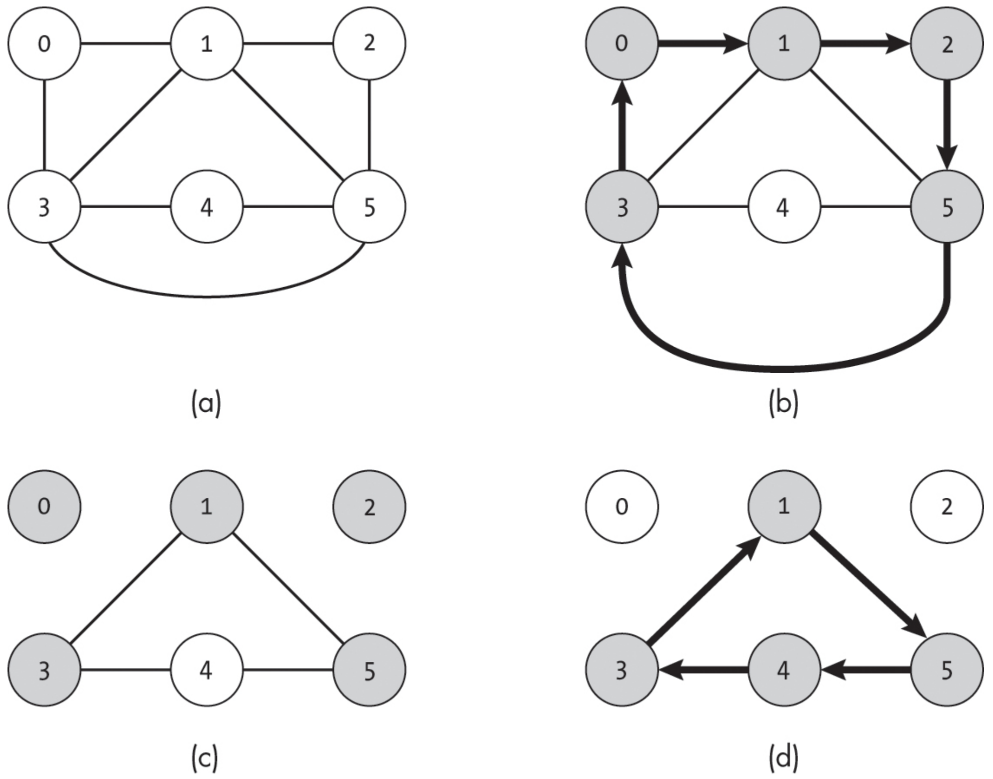


Figure 18-10: The graph before (a), during the first step (b), after the first step (c), and during the second step (d) of Hierholzer's algorithm

Next, the algorithm looks for a cycle that starts and ends at one of the previously visited nodes but travels unused edges. Figure 18-10(d) shows the cycle [1, 5, 4, 3, 1]. We can splice this new loop into the full path by inserting it in place of the occurrence of node 1 for an overall path of [0, 1, 5, 4, 3, 1, 2, 5, 3, 0].

Depending on how the algorithm selects which node to visit next, different implementations can ultimately explore different subloops and produce different Eulerian cycles for the same graph. For example, the code in this section will explore the nodes in Figure 18-10(a) so as to produce the final Eulerian cycle from Figure 18-8: [0, 1, 2, 5, 1, 3, 4, 5, 3, 0].

To extract an Eulerian cycle from a graph, we must follow the subloops through the graph:

```
def hierholzers(g: Graph) -> Union[list, None]:
  ❶ if not has_eulerian_cycle(g):
        return None

    g_r: Graph = g.make_copy()
    options: set = set([0])
    full_cycle: list = [0]

    while len(options) > 0:
      ❷ start: int = options.pop()
        current: int = start
        subcycle: list = [start]

      ❸ while current != start or len(subcycle) == 1:
          ❹ neighbor: int = list(g_r.nodes[current].edges.
keys())[0]
            subcycle.append(neighbor)
            g_r.remove_edge(current, neighbor)

          ❺ new_num_edges: int = g_r.nodes[current].num_ed
ges()
            if new_num_edges > 0:
                options.add(current)
            elif new_num_edges == 0 and current in option
s:
                options.remove(current)

            current = neighbor

      ❻ if g_r.nodes[start].num_edges() == 0 and start in
 options:
            options.remove(start)

        loc: int = full_cycle.index(start)
        ❼ full_cycle = full_cycle[0:loc] + subcycle + full_c
ycle[loc+1:]

    return full_cycle
```

The code starts by confirming the graph has an Eulerian cycle using the `has_eulerian_cycle()` function from Listing 18-3 ❶. If this check fails, the code returns `None` to indicate the lack of an Eulerian cycle. The code relies on importing `Union` from the `typing` library to support the type hints for multiple return types. If the check passes, the code sets up the initial data structures, including a full copy of the graph that can be modified (`g_r`), a set of seen nodes that it can use as starting points for subloops (`options`), and a list that tracks the Eulerian cycle constructed so far (`full_cycle`). The code will iteratively build up `full_cycle` by following subloops and inserting them into `full_cycle`.

The main body of the algorithm is a `while` loop that continues finding new cycles while there exists a visited node with unused edges (`options` is not empty). The `options` set provides a list of nodes from which the code can start a new subloop. The code pops an arbitrary node from `options` ❷ and starts traversing a cycle.

The code traverses the new cycle by using an inner `while` loop that explores until it has completed a circuit and returned to the cycle's starting node ❸. The loop condition also tests that the new cycle has taken at least one step before terminating. If `len(subcycle) == 1`, the loop continues because the path hasn't gone anywhere yet. During each step of the cycle traversal, the code selects the first key in the current node's `edges` dictionary as its next destination to visit (`neighbor`) ❹. It adds `neighbor` to the current loop being tracked and removes the edge from the copy of the graph.

The code updates `options` by considering the number of remaining edges from the current node ❺. If there is at least one remaining edge, it adds the node to `options` to indicate there are other paths to take. In contrast, if the code has just deleted the last edge adjacent to the current node, the code removes that node from the `options`. After the inner `while` loop completes, the code similarly discards the starting node if it has no remaining edges ❻.

After completing the inner loop, the code also inserts `subcycle` into `full_cycle` ❼. For simplicity's sake, we use a linear time find (`index()`

function) and build a new copy of `full_cycle`. With additional bookkeeping, we could use more efficient approaches to minimize the cost of this step.

Figure 18-11 shows the operation of Hierholzer's algorithm on a graph with eight nodes. Figure 18-11(a) shows the state of the graph, the `options` set, and the `full_cycle` list before the start of the algorithm. The remaining subfigures show the state of the algorithm after each iteration of the outer `while` loop. Edges traversed and removed during that iteration are highlighted in bold.

We can visualize this algorithm in terms of an officer of a city's tourism board planning a comprehensive tour. Their goal is to devise a path through the city that travels each street exactly once, giving visitors a full experience of the city without unnecessary repetition. They choose the city's premier hotel as the starting location (node 0) and set out walking. Throughout their trip, they record each street they travel and visit intersections with untraveled roads.

Figure 18-11(b) illustrates the results of the tour planner's first day. Taking untraveled roads, they complete a small cycle [0, 1, 2, 0] to arrive back at the hotel. At this point, they have no untraveled roads out of the current node. Undaunted by the number of streets left unexplored, they cross roads (0, 1), (1, 2), and (2, 0) off of their map. They also note that they could have taken different roads at intersections (that is, nodes) 1 and 2.

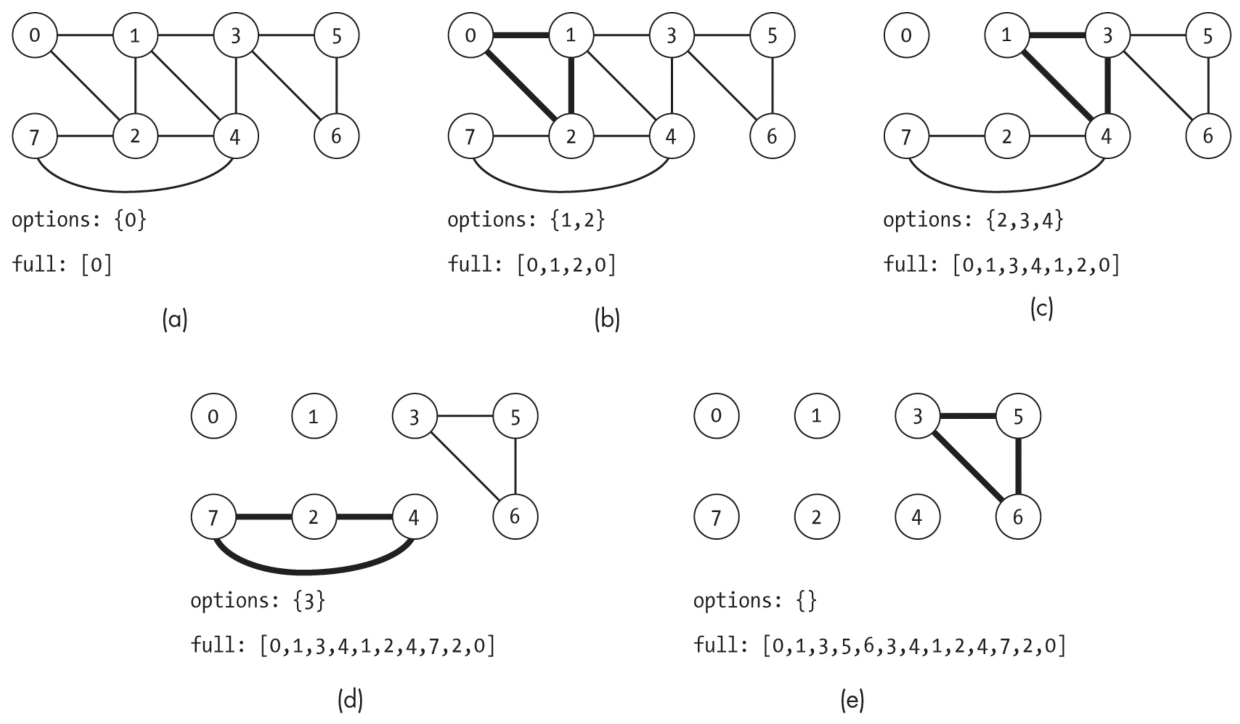


Figure 18-11: The steps of Hierholzer's algorithm on an example graph

The next day, the planner ventures to one of the nodes with untraveled edges to explore from there. As shown in Figure 18-11(c), they choose to start at node 1 because it was reachable on a previous cycle and had unexplored options. They complete another small cycle [1, 3, 4, 1] before returning to node 1 and discovering they have traversed all the adjacent streets. They update their map to remove roads (1, 3), (3, 4), and (4, 1) and note that there are unexplored streets branching out from intersections 2, 3, and 4. They splice today's path into yesterday's path at node 1, providing a combined path of [0, 1, 3, 4, 1, 2, 0].

The third day is similar, with the planner starting from node 2, as shown in Figure 18-11(d). They complete the cycle [2, 4, 7, 2], delete the traveled streets, and extend the combined path to [0, 1, 3, 4, 1, 2, 4, 7, 2, 0]. As they follow that day's tour, they notice that they have traveled all roads adjacent to nodes 2 and 4. They remove both nodes from their starting options, leaving only node 3.

The final day starts at node 3, as shown in Figure 18-12(e). The planner travels [3, 5, 6, 3] and splices it into the combined path for an Eulerian cycle of [0, 1, 3, 5, 6, 3, 4, 1, 2, 4, 7, 2, 0].

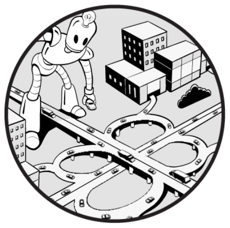
Why This Matters

The three problems discussed in this chapter—finding Hamiltonian and Eulerian paths and solving the traveling salesperson problem—have clear applications to a variety of real-world planning and optimization use cases. Rather than searching for a path from a given origin to a given destination like the problems in the previous two chapters, the problems covered here aim to find paths that visit each node or edge in the graph.

These problems provide a foundation on which to build more complex tasks. We could extend the Eulerian path problem by adding pairwise ordering constraints. For example, the tourist might need to visit the city's welcome center and buy tickets before they can ride the gondolas. A company might partition their cities among five salespeople, requiring them to assign cities as well as paths to each employee. The three formulations from this chapter barely scratch the surface of the interesting and complex questions we can ask.

The problems in this chapter also demonstrate that the difficulty of solving seemingly similar problems can actually vary greatly. While the tasks of finding Eulerian and Hamiltonian paths have similar real-world analogies, their worst-case computational costs vary significantly. Recognizing and understanding these types of differences is important when considering which approaches to use when tackling a novel problem.

CONCLUSION



Throughout this book, we've examined how to model a range of real-world problems and systems with graph data structures. We've also investigated some of the corresponding algorithms to solve these problems. We built upon basic searches, like depth-first and breadth-first search, to construct more complex algorithms for tasks like topological sort and labeling bipartite graphs. We also introduced a variety of more specialized algorithms, such as Kahn's algorithm for topological sort or the removal algorithm for graph coloring.

Yet this book only scratches the surface of the fascinating world of graphs. A huge volume of research exists on both the theoretical properties of graphs (the mathematical field of *graph theory*) and practical graph algorithms. For example, recent computer science research has continued to develop novel approaches for processing large graphs. A fully comprehensive coverage of graph algorithms is beyond the scope of a single book.

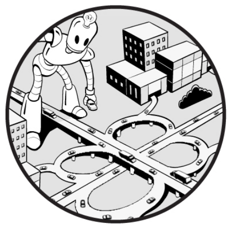
This book has aimed to introduce the fundamental concepts of graphs and explain various ways to operate on them. It has provided a foundation and basic toolbox. You should be able to adapt the ideas behind these algorithms to techniques beyond this book and easily dive into other graph-related topics.

As discussed throughout the book, we can often optimize algorithms and implementations to meet the needs of a given use case. As a future step in learning about graphs, I encourage you to try modifying the approaches you've learned so far to take advantage of the properties of a specific problem or to avoid unnecessary overhead. For example, rather than storing explicit `Node` data structures in a graph, it may be more effective in some cases to use a list of neighbors. In others, adding auxiliary data to the node's representation may help you avoid repeated computation.

Perhaps the most exciting aspect of computer science is the ability to continually explore and build new solutions.

A

CONSTRUCTING GRAPHS



This book has covered a range of algorithms that operate over graphs. To apply these algorithms to real-world problems, we need a mechanism for programmatically creating new graphs or loading them from files.

While the graph creation functions in Chapter 1 such as `insert_node()` and `insert_edge()` provide a full mechanism for defining graphs, it would be unrealistic to hard-code the creation of each graph. Beyond the tedium of listing out hundreds or thousands of `insert_edge()` statements, such an approach is error-prone.

In this appendix we introduce a variety of simple mechanisms for creating graphs and loading them from files. As with previous chapters, we optimize for understandability instead of storage size or computational efficiency. This means that some of the input formats we use, such as comma-separated values, will be simplistic. However, these formats are perfect for demonstrating what information we need to consider and how we can programmatically manipulate it to form a graph. Readers can build off of these approaches to construct formats optimized for their own applications.

This appendix makes use of Python's `csv` library to facilitate loading and parsing files. We encourage readers to explore the vast ecosystem of more specialized libraries that might further simplify their code.

Constructing Graphs from Edges

The `Graph` data structure we've used since Chapter 1 relies upon a constructor that creates an initial graph with no edges. The user can add edges to the graph using the `insert_edge()` function. In this section, we'll automate this flow to build graphs from lists of edges.

After creating a new graph from a given list of edges, we extend this functionality to read from and write to edge files. Along the way, we tackle the problem of nodes being specified by a generic text label rather than a node index.

Inserting Edges from a List

We can create a graph from a list of `Edge` data structures:

```
def make_graph_from_edges(num_nodes: int,
                          undirected: bool,
                          edge_list: list) -> Graph:
    g: Graph = Graph(num_nodes, undirected)
    for edge in edge_list:
        g.insert_edge(edge.from_node, edge.to_node, edge.w
eight)
    return g
```

In addition to the edge list, the code takes in the number of nodes (`num_nodes`) and whether the graph is undirected (`undirected`). It creates an initial graph structure with the correct number of nodes and undirected setting, then loops over each edge and inserts it into the graph. If the edges are undirected, the `insert_edge()` function in the `Graph` class inserts a directed edge in each direction.

In many cases, we could compute the number of nodes directly from the edge list by tracking the largest node index seen. However, this approach does not allow for graphs with fully disconnected nodes, because their index would never appear in the edge list and the resulting graph might miss later nodes. Instead, the code takes `num_nodes` as an input parameter.

Figure A-1 shows an example graph construction where the following directed edges are inserted:

```
[Edge(0,1,1.0), Edge(1,3,10.0), Edge(2,4,5.0), Edge(3,1,2.0),
                        Edge(1,2,3.0)]
```

Figure A-1(a) shows the graph after the first node is inserted. Each subfigure shows the state of the graph after one iteration of the code's `for` loop.

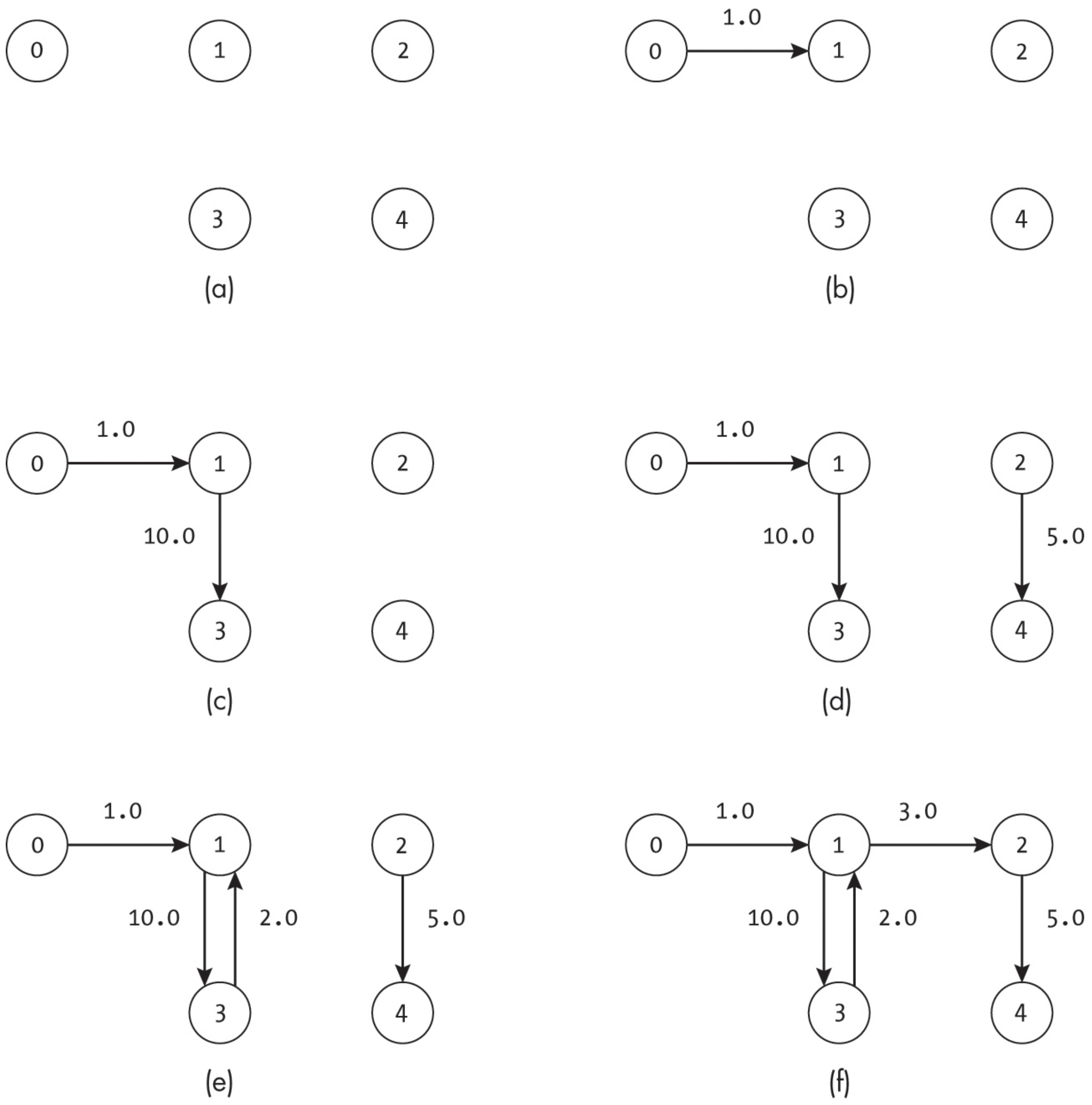


Figure A-1: The steps of adding edges to a graph with five nodes

We can use the `make_graph_from_edges()` function in conjunction with algorithms that return lists of edges, such as the `randomized_kruskals()` algorithm in Chapter 10.

Loading Edge Lists from Files

A natural approach to constructing graphs is to load the graph directly from a file. While there are myriad different formats we could use to store a graph's representation, we'll start with one of the simplest approaches: storing a graph as a file of comma-separated values (CSV). Each row indicates a single edge listing the originating node as a string label, the destination node as a string label, and an optional floating-point weight. Unconnected nodes are listed alone.

Using this CSV format, we can encode the undirected graph shown in Figure A-2 with the following data:

```
a,b
b,c,10.0
d,e,5.0
```

The data define a graph with five nodes a, b, c, d, and e, and three edges. Two of the edges are given explicit weights and the third uses a default value of 1.0.

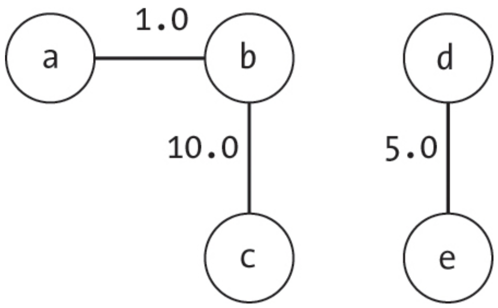


Figure A-2: A graph with five nodes and three weighted edges

We can extend this format to account for fully disconnected nodes, which will not appear in any edge, by allowing rows with a single entry. This single entry simply indicates the node exists so that the program includes it in the graph.

We can use this general format to encode a vast number of real-world phenomena. For example, we could use a single file to represent note-passing behavior in a classroom. Each node represents a student. Each row provides new information about the class (graph). A row can contain a single name to indicate the existence of a student (node). Alternatively, a row can indicate the directed note-passing behavior between a pair of students and includes `person1`, `person2`, `weight` to indicate how many times (represented by weight) person 1 passed a note to person 2.

The code to read such a CSV file makes use of Python's `open()` function to read the file and the csv package (which requires importing `csv`) to parse each row, as shown in Listing A-1.

```
def make_graph_from_weighted_csv(filename: str, undirecte
d: bool) -> Graph:
    g: Graph = Graph(0, undirected)
    node_indices: dict = {}

  ❶ with open(filename) as f:
        graph_reader = csv.reader(f, delimiter=',')
      ❷ for row in graph_reader:
            name1: str = row[0]
            if name1 not in node_indices:
                new_node: Node = g.insert_node(label=name
1)
                node_indices[name1] = new_node.index
            index1: int = node_indices[name1]

          ❸ if len(row) > 1:
                name2: str = row[1]
                if name2 not in node_indices:
                    new_node = g.insert_node(label=name2)
                    node_indices[name2] = new_node.index
                index2: int = node_indices[name2]

              ❹ if len(row) > 2:
                    weight: float = float(row[2])
                else:
                    weight = 1.0

                g.insert_edge(index1, index2, weight)
    return g
```

Listing A-1: Loading a graph from a CSV file

The code starts by creating the necessary data structures—an empty graph (`g`) and an empty dictionary (`node_indices`) that will map the string names to the corresponding node index.

The code then opens the file and parses it using `csv.reader` ❶. It iterates through every row in the file, reading up to three entries ❷. For each row, it extracts the name of the node from the first entry. Since the code reads in names, it needs to map those names to a corresponding index. If the node is in the graph already, the code can look it up from the `node_indices` dictionary. Otherwise, it has found a node that is not yet in the graph and must insert this node into the graph and the name into the dictionary. If the row has only a single entry, the function skips the rest of the logic and continues to the next row.

If the row has a second entry, however, that row specifies an edge ❸. The code extracts the second node's name. Again, it checks whether the node is in the graph and, if not, inserts it. The code checks for a third entry in the row, indicating a weight ❹. If no weight is provided, the code uses a default value of `1.0`. Finally, the code uses the combination of the two node indices and the weight to insert a new edge into the graph.

The code continues row by row in the CSV until it has read the entire file. At that point it returns the final graph. The file is closed automatically as the code exits the `with` statement.

One of the major complexities in this code is mapping the nodes' text strings to their indices. In theory, we could restrict our loader function to require that files supply each node's integer index. While this would remove the need for a mapping, it greatly reduces the usability. Anytime we want to load in a new graph, we must first construct a mapping from name to index and convert the file. In a later section, we will look at how to incorporate this common operation into the `Graph` data structure itself.

Saving Edge Lists to Files

We can use the same comma-separated values format to save our favorite graphs. Since most of the graphs we have considered have not used named nodes, we will use each node's integer index as its name in our CSV file. However, we still face the problem that the code in Listing A-1 assigns node indices as they are encountered. Ideally, we would like node 5 to stay node 5 when reloaded, regardless of the order in which the code encounters it in the edge list.

There are a couple potential solutions to this problem. We could forgo names altogether and just store the integer node indices in the CSV, modifying Listing A-1 to read the node names directly as integer indices. Alternatively, we can output one node name per row in order at the start of the CSV file to indicate their existence. This will ensure that the first node is mapped to index 0, the second to index 1, and so on.

For the following code, we use this second approach to stay consistent with Listing A-1. This code will work for an integer-based reader as well:

```
def save_graph_to_csv(g: Graph, filename: str):
 ❶ with open(filename, 'w', newline="\n") as f:
        graph_writer = csv.writer(f, delimiter=',')
     ❷ for node in g.nodes:
            graph_writer.writerow([node.index])

     ❸ for node in g.nodes:
            for edge in node.get_edge_list():
                graph_writer.writerow([edge.from_node, edg
e.to_node,
                                      edge.weight])
```

The code opens the file and writes to it using `csv.writer` ❶. It uses a `for` loop to iterate through each node, writing one node index per row ❷. This ensures that all nodes are captured in the CSV even if they are not part of any edges. Finally, a pair of `for` loops iterate through each edge in the graph and write out three values (origin node, destination node, and weight) representing the edge ❸. The file closes automatically as the code exits the `with` statement.

Inserting Nodes by Name

A significant portion of the code in Listing A-1 consists of handling the mapping of a node's name to its index. We look up each name string in the `node_indices` dictionary to get the index. If the name is not in the dictionary, we need to insert a new node and create the corresponding dictionary entry mapping the name to the new index. Depending on how you reference nodes in your program, this can be a common problem.

To simplify working with named nodes, we can incorporate this logic into the `Graph` class itself. We need to make two changes. First, we add the creation of an empty dictionary to the initialization function:

```
self.node_indices: dict = {}
```

We will use this dictionary whenever we need to map a name to an index.

Second, we add a function to the `Graph` class that performs both the lookup and, if necessary, the insertion, as shown in Listing A-2.

```
def get_index_by_name(self, name: str) -> int:
    if name not in self.node_indices:
        new_node: Node = self.insert_node()
        self.node_indices[name] = new_node.index
    return self.node_indices[name]
```

Listing A-2: Adding named nodes to a graph

The code starts by determining if this node already exists in the graph by checking whether its name is in the mapping. If not, the graph inserts the node into the graph using the original `insert_node()` function. It also inserts the name into the index map. The function finishes by returning the node's index.

We can picture this function in the context of a lazy teacher creating a seating chart. On the first day, the teacher allows the students to pick their own seats but does not bother to write down who is sitting where. Their seating chart is initially empty. As the class progresses, students raise their hands with questions. Each time this happens, the teacher glances down at their seating chart. If the student's name is listed, the teacher uses it. Otherwise, they peer at the student and, with a complete lack of tact, ask, "Who are you?" When the student finishes rolling their eyes and responding, the teacher adds the student to the seating chart. The chart progressively gets filled in as the students are acknowledged.

We can use these helper functions to rewrite the CSV reader code from Listing A-1 more compactly:

```
def make_graph_from_weighted_csv2(filename: str, undirected: bool) -> Graph:
    g: Graph = Graph(0, undirected)

    with open(filename) as f:
        graph_reader = csv.reader(f, delimiter=',')
        for row in graph_reader:
          ❶ index1: int = g.get_index_by_name(row[0])

            if len(row) > 1:
              ❷ index2: int = g.get_index_by_name(row[1])

                if len(row) > 2:
                    weight: float = float(row[2])
                else:
                    weight = 1.0
                g.insert_edge(index1, index2, weight)
    return g
```

The code follows the same flow as Listing A-1 but uses `get_index_by_name()` to do both the node mapping and lookup for the first ❶ and second nodes ❷ in each row.

Co-occurrences

Co-occurrence graphs represent which pairs of entities have previously co-occurred. In biology, they can be used to study interactions, such as which genes are often present together or which microbes interact. In sociology, they can model gatherings of groups or coauthorship of academic publications. And in pop culture, they can help answer critical questions about which movie stars appeared on-screen together, as discussed in Chapter 2.

The difficulty with ingesting co-occurrence data is that it usually consists of a list of sets instead of pairwise interactions. Our favorite online movie database probably does not list each pairing of movie stars from last summer's smash hit. Instead, it provides a single list labeled *cast* that includes everyone who appeared in the movie.

For example, when reviewing all the local theater company's recent productions, we might find the following cast lists:

(`A`, `B`, `C`)

(`D`, `E`)

(`A`, `D`)

(`C`, `F`, `G`, `H`)

(`C`, `E`)

Using a single cast list, we can easily see that `A` has appeared onstage with `C`. However, if we are interested in more complex questions such as how many degrees of separation there are between `A` and `H`, we need to construct a more comprehensive picture. Figure A-3 shows the co-occurrence graph based on these plays.

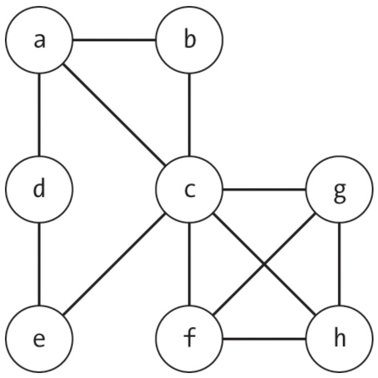


Figure A-3: An undirected graph with eight nodes representing the co-occurrence of actors in plays

To make the model more powerful, we can use edge weights to track how often two nodes have co-occurred. For example, Alice and Bob might have a stronger acting connection after appearing together in 10 different plays, while Diane's weaker connection to Alice comes from sharing the stage a single time. If you are looking for an introduction to the world-famous Alice, it is better to use the stronger connection through Bob.

We can extend the approach used to load graphs from CSVs to construct pairwise graphs from arbitrary lists. Instead of expecting at most three entries per row, we allow all co-occurring entries to be listed on a single row. The code then determines the pairwise edges implied by each list:

```
def make_graph_from_multi_csv(filename: str) -> Graph:
    g: Graph = Graph(0, undirected=True)
    with open(filename) as f:
        graph_reader = csv.reader(f, delimiter=',')
        for row in graph_reader:
            num_items: int = len(row)

            for i in range(num_items):
                index1: int = g.get_index_by_name(row[i])

                for j in range(i + 1, num_items):
                    index2: int = g.get_index_by_name(row
[j])
                    edge: Union[Edge, None] = g.get_edge(i
ndex1, index2)
                 ❶ if edge is not None:
                        weight = edge.weight + 1.0
                    else:
                        weight = 1.0
                 ❷ g.insert_edge(index1, index2, weight)
    return g
```

As with the earlier CSV reader, the code starts by creating an empty graph, opening the file, and parsing it with `csv.reader`. This time, the code restricts the graph to being undirected, as there is no implied directionality in these co-occurrence edges. The code also does not maintain a dictionary of names, but rather uses the helper function from Listing A-2 to insert new nodes and retrieve each node's index.

The code iterates through each row using a `for` loop. For each row, it uses another pair of nested `for` loops to iterate over all unique pairs of entities on each row. It looks up the nodes' indices using the `get_index_by_name()` function, inserting new nodes as needed. Before creating an edge from the pair, the code checks whether the edge exists ❶. We use `Union` from Python's `typing` library to allow `None` to be returned by the `get_edge()` function. If the edge already exists, the code retrieves its current weight and increments it by 1, then inserts a new edge with the updated weight ❷. Since `insert_edge()` overwrites the existing edges when a duplicate is inserted, the code effectively updates the weights of existing edges.

As with the previous CSV reader in Listing A-1, the code continues row by row until it has read the entire file. At that point it returns the final graph.

Spatial Points

When using graphs to model path planning or other physical problems, we often need to construct a graph from a series of spatial data points. We can do this by creating a graph containing one node for each spatial point and one edge between *every pair* of points.

Figure A-4 shows an example of such a representation. Figure A-4(a) shows five two-dimensional points: (0, 0), (1, 0), (1.2, 1), (1.8, 1), and (0.5, 1.5). Figure A-4(b) shows the graph representation with the edge weights capturing the distance between points.

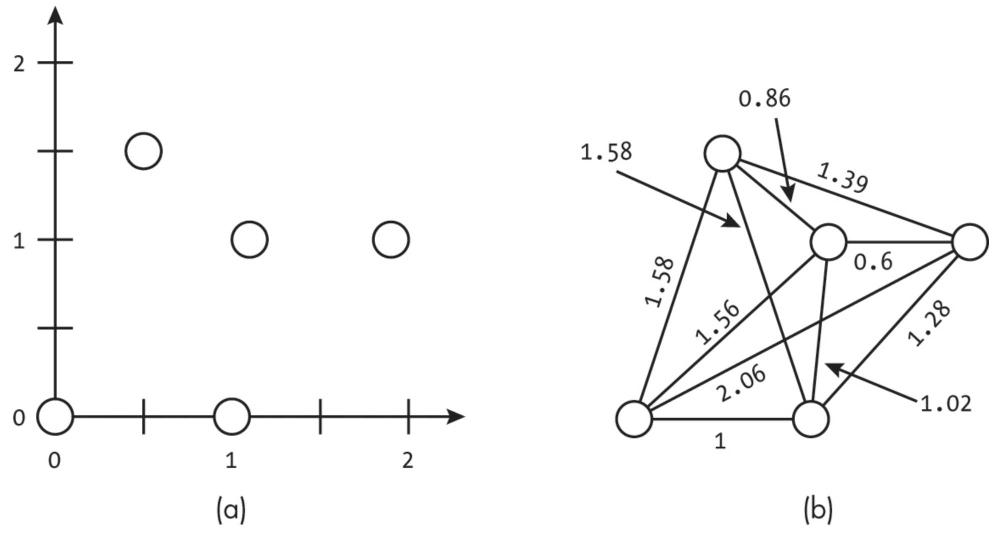


Figure A-4: A set of two-dimensional spatial points (a) and the corresponding graph representation (b)

We start by defining a helper class to store the spatial points and compute the distance between them:

```
class Point:
    def __init__(self, x: float, y: float):
        self.x: float = x
        self.y: float = y
    def distance(self, b) -> float:
        diff_x: float = (self.x - b.x)
        diff_y: float = (self.y - b.y)
      ❶ dist: float = math.sqrt(diff_x*diff_x + diff_y*dif
f_y)
        return dist
```

While using a `Point` class is not required, it allows us to easily swap in higher-dimensional points or alternative distance functions. The `distance()` function computes the Euclidean distance in two-dimensional space. Note that we will need to import `math` to use the square root function.

We can substitute in alternative distance functions by modifying the `distance()` function. For example, we could use Manhattan distance by changing the line at ❶ to the following:

```
        dist: float = abs(diff_x) + abs(diff_y)
```

We can write the code for constructing the graph from spatial points using a nested pair of loops:

```
def build_graph_from_points(points: list) -> Graph:
    num_pts: int = len(points)
    g: Graph = Graph(num_pts, undirected=True)

    for i in range(num_pts):
        for j in range(i + 1, num_pts):
            dist: float = points[i].distance(points[j])
            g.insert_edge(i, j, dist)
    return g
```

The code allocates a graph with one node for each point in the dataset, then uses a pair of `for` loops to iterate through each pair of points. For each pair, the code computes the distance between the points using the `distance()` function from the `Point` class and inserts an undirected edge with the corresponding weight into the graph. Once all edges are added, the function returns the completed graph.

Preconditions

In Chapter 9 we considered the problem of topological sort—ordering nodes according to the graph's directed edges. Figure A-5 shows an example of such an ordering, where each node is ordered along the horizontal direction.

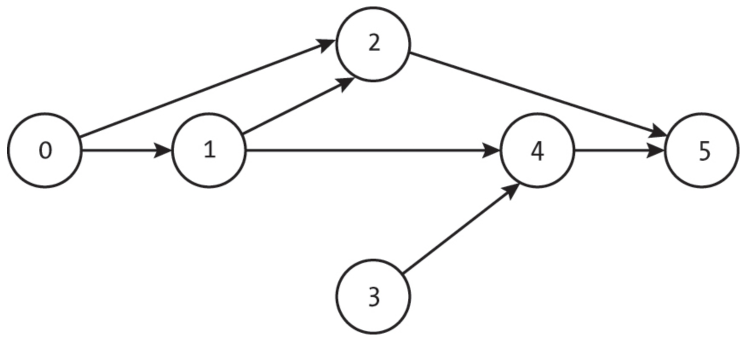


Figure A-5: Six nodes ordered by their relative dependencies

Topological sort allows us to list steps for an instruction manual, plan for courses with prerequisites, or compute the order in which to compile parts of a program. Each of these problems required us to specify the dependencies between nodes.

We define a simple function to construct a graph from a dictionary of preconditions. Each entry in the dictionary maps a node to a list of its dependencies. For example, the graph in Figure A-5 would be represented as:

```
{0: [], 1: [0], 2: [0, 1], 3: [], 4: [1, 3], 5: [2, 4]}
```

Nodes without a dependency, such as nodes 0 and 3, are represented using an empty list.

The code for constructing a graph from dependencies consists of loops that iterate through each node, as well as each node's dependencies:

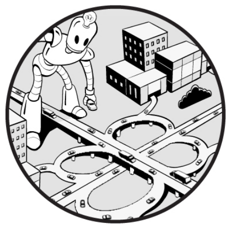
```
def make_graph_from_dependencies(dependencies: dict) -> Gr
aph:
    g: Graph = Graph(0, undirected=False)
    for node in dependencies:
        n_index: int = g.get_index_by_name(node)
        for prior in dependencies[node]:
            p_index: int = g.get_index_by_name(prior)
            g.insert_edge(p_index, n_index, 1.0)
    return g
```

The code starts by allocating an empty directed graph. It fills the graph by iterating over each key in the `dependencies` list with a `for` loop. For each entry, even those with an empty list of priors, the code uses `get_index_by _name()` to look up and potentially insert a new node. This code is sufficient to fill in the graph's nodes.

To fill in the edges, the code uses a second `for` loop to iterate over each node's dependency list. The dependency's index is retrieved (and a new node may be inserted) using `get_index_by_name()`. The code then inserts the edge *from* the dependency node *to* the current node. Since we are interested only in the ordering, the code uses a default `1.0` weight for all edges. The code concludes by returning the constructed graph.

B

MODIFIABLE PRIORITY QUEUES



Several algorithms in this book, such as Dijkstra's algorithm and A* search, use an augmented priority queue that allows the program to modify the priority of existing elements. For completeness, this appendix describes and provides the code for this data structure.

While many standard heap implementations support the addition and removal of items, they often do not support efficiently changing an item's priority. We'll provide a brief overview of heaps, then define a small extension to the standard priority queue that uses a dictionary to map each item's location in the heap. This mapping allows us to efficiently look up a given item and change its priority. Finally, we present code to implement a modifiable priority queue.

Heaps

The core of our priority queue data structure is the heap data structure. This section provides a short introduction to heaps using material adapted from my previous book, *Data Structures the Fun Way*. We'll cover just enough discussion of heaps to explain the code but won't go into significant depth; you can learn more about the details of heaps and their properties in the aforementioned book.

Heaps are variants of the binary tree that maintain a special ordered relationship between a node and its children. A *max heap* orders the elements according to the max heap property, which states that the value at any node in the tree is larger than or equal to the values of its child nodes. A *min heap* orders the elements according to the min heap property, which states that the value at any node in the tree is smaller than or equal to the values of its child nodes. For the priority queue, we use a max heap to order items by their priority.

Heap Items

We define each item in the priority queue with two variables. The item's *value* is the information we are storing about the object. This can be an integer (node's index), string (node's name), or even an object. The item's *priority* is the floating-point number that we use to determine which item is extracted next from the priority queue.

We use a wrapper data structure `HeapItem` to store the combination of an item's value and priority:

```
class HeapItem:
    def __init__(self, value, priority: float):
        self.value = value
        self.priority = priority

 ❶ def __lt__(self, other):
        return self.priority < other.priority

 ❷ def __gt__(self, other):
        return self.priority > other.priority
```

The data structure consists of a constructor that initializes the object and code to overload both the less-than ❶ and greater-than ❷ comparisons to compare the priorities. This data structure is not strictly necessary and adds some overhead; we could alternately use a tuple. However, we'll rely on `HeapItem` throughout this chapter to make the code more readable.

Array-Based Storage

While heaps are defined in terms of trees, we use a standard array-based implementation that is particularly efficient. Each element in the array corresponds to a node in the tree with the root node at index 1 (we skip index 0, as is conventional for heaps). Child node indices are defined relative to the indices of their parents; a node at index i, for instance, has children at indices $2i$ and $2i + 1$. We likewise compute the index of the parent of node i as `Floor`($i/2$). This indexing scheme, shown in Figure B-1, allows the algorithm to easily compute the index of a child based on that of the parent and the index of a parent based on a child.

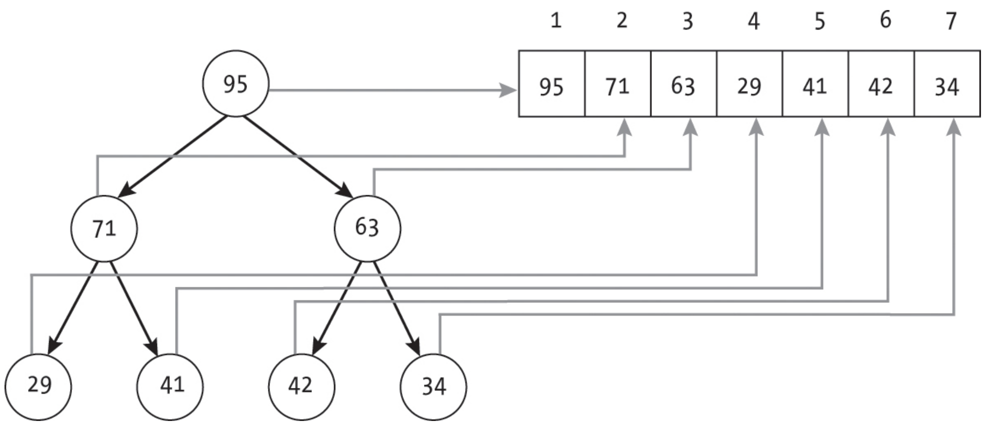


Figure B-1: A heap represented as a tree (left) and an array (right)

Although the items in the array are not in sorted order, the first item is always the front item in terms of priority. In a max heap, the first item in the array has the maximum priority. In a min heap, the first item has the minimum priority. This means we can access the "next" item from our heap with a simple lookup.

Element Swaps

Both insertion and removal of items involve breaking the heap property and swapping pairs elements to restore it. Anytime we have an item out of place, we can fix the ordering by either swapping it upward with its parent or swapping it downward with one of its children. We repeat this process, using the item's new position, until it has the correct location in the heap.

In the case of max heaps, we swap an element up if its priority is larger than that of its parent. In Figure B-2, for example, the element 56 is out of place. When we compare it to its parent in Figure B-2(a), we see that 56 is larger than 41, which violates the max heap property. We can fix this by swapping those two elements, placing 56 in the correct location with respect to its parent, as shown in Figure B-2(b).

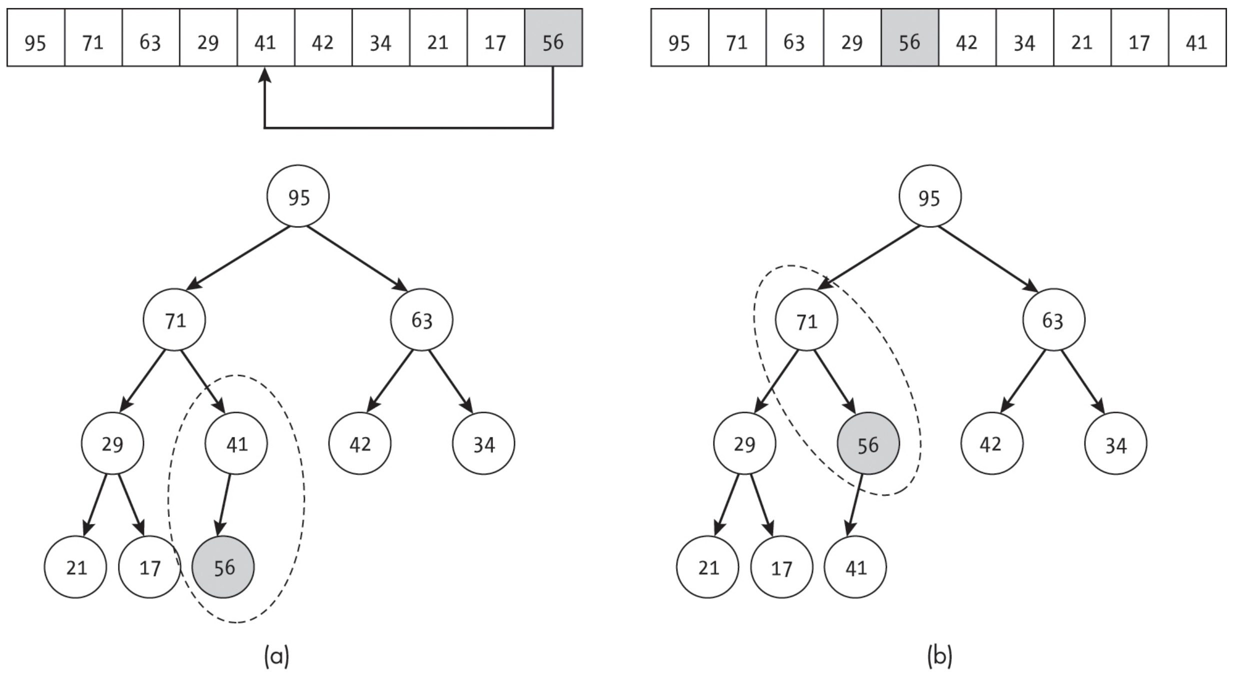


Figure B-2: A heap element that is out of place (a) and the resulting upward comparisons and swaps (b)

Similarly, we swap an element down in a max heap if its priority is smaller than either of its children. In this case, we must also choose which child to use for the swap, selecting the larger of the two children to maintain the max heap property. In Figure B-3, for instance, the element 29 is out of place. When we compare its priority to that of its children in Figure B-3(a), we see that 29 is smaller than 71 and 29 is smaller than 41, both of which violate the max heap property. We fix this by swapping element 29 with the larger of its two children, as shown in Figure B-3(b).

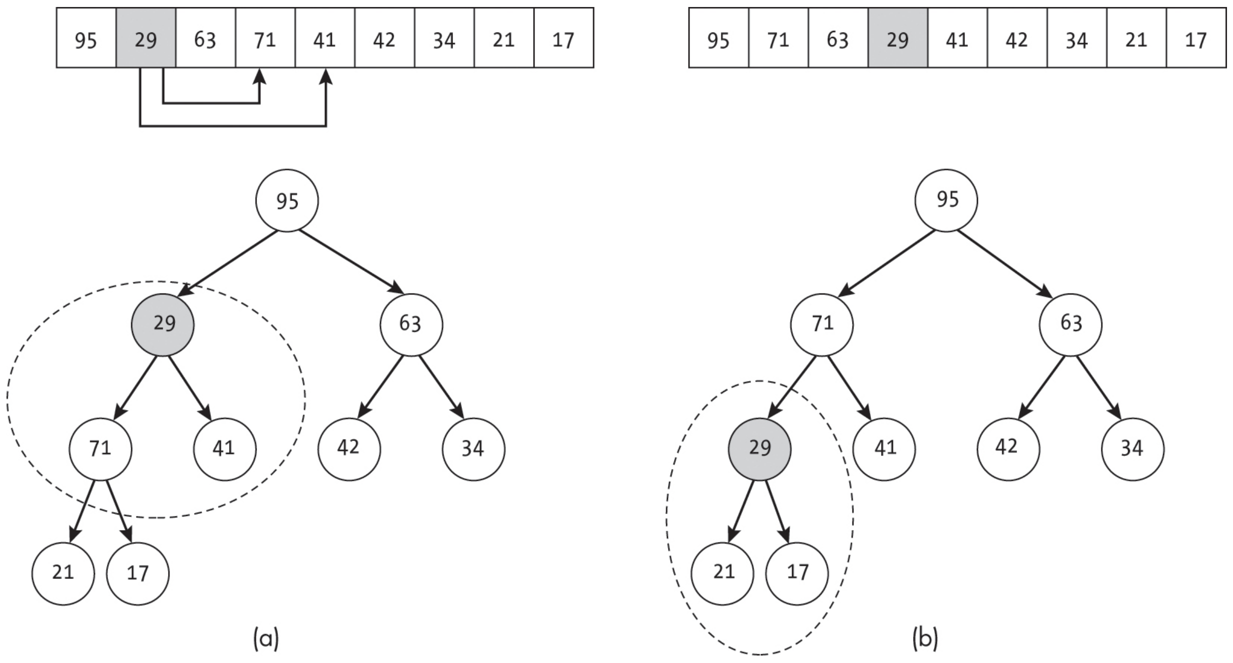


Figure B-3: A heap element that is out of place (a) and the resulting downward comparisons and swaps (b)

When fixing a min heap, we reverse the comparisons. We swap an element upward if its priority is smaller than that of its parent or downward if its priority is larger than either of its two children. When swapping downward, we choose the child with the smaller priority to maintain the min heap property.

Modifiable Priority Queue

The modifiable priority queue consists of a class wrapping a standard heap-based priority queue and a dictionary mapping items to their location in the heap's array. This structure is shown in Figure B-4 with the dictionary (indexed by the item's value) on the left and an array-based maximum heap on the right. As shown, each of the dictionary's entries points to the item's index in the heap array.

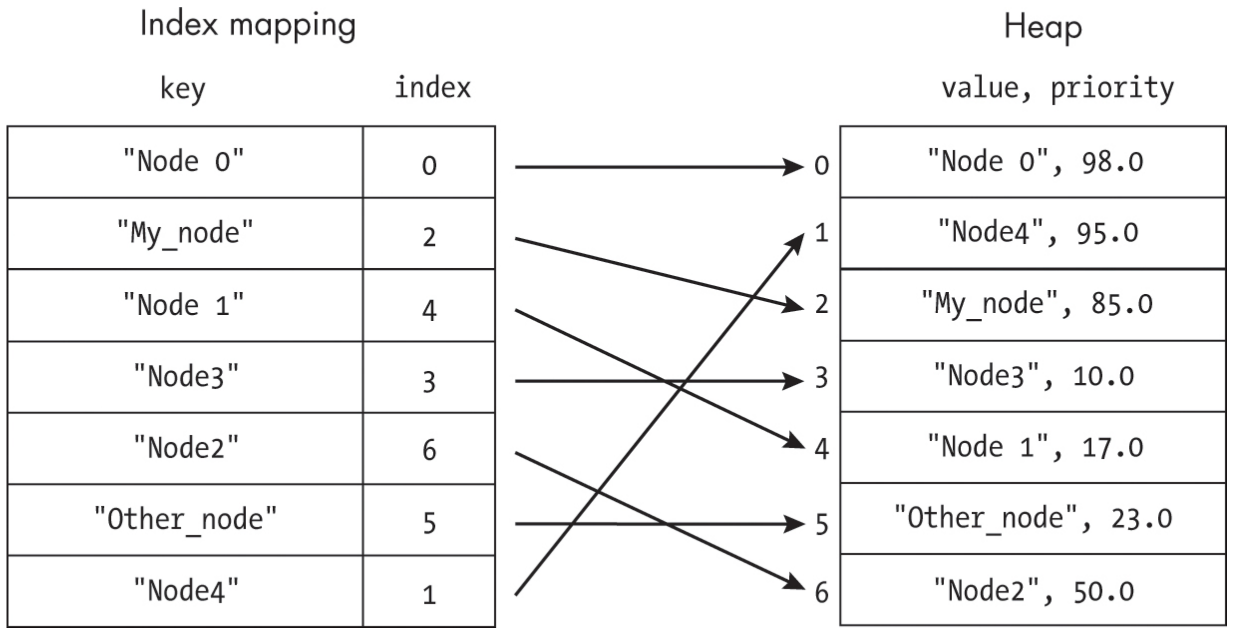


Figure B-4: The two data structures inside the priority queue class

Given these attributes, we define a simple interface that allows us to insert (value, priority) pairs into the priority queue, remove values from the front of the priority queue, and change the priority for an existing value. We also include several convenience functions for operations like getting the size or checking that a value is in the priority queue.

This appendix provides the code for the functions that make up the priority queue's interface:

`dequeue()` Removes the top item from the priority queue and returns its value

`enqueue(value, priority)` Inserts a new (value, priority) pair into the priority queue

`get_priority(value)` Returns the value's floating-point priority

`in_queue(value)` Returns a Boolean indicating whether a given value is in the priority queue

`is_empty()` Returns a Boolean indicating whether the priority queue is empty

`peek_top()` Returns the top item on the priority queue

`peek_top_priority()` Returns the priority of the top item in the queue

peek_top_value() Returns the value of the top item in the queue

size() Returns the number of items in the priority queue

update_priority(value, priority) Updates the priority of the item with a given value

The Data Structure

The `PriorityQueue` class provides a wrapper around the heap and dictionary of indices. It contains the following attributes:

array_size (int) Stores the total length of the heap array

heap_array (list) Stores `HeapItems`, which serves as the heap-ordered internal storage for the priority queue

last_index (int) Stores the index of the last element in the heap

is_min_heap (bool) Indicates whether a `PriorityQueue` object is a min heap (`True`) or max heap (`False`)

indices (dict) Stores a map of the `HeapItem`'s value to its index in `heap_array`, allowing the efficient lookup of items by their value

We define a constructor that initializes the attributes to those values of an empty priority queue and provides a few basic functions:

```
class PriorityQueue:
    def __init__(self, size: int = 100, min_heap: bool = F
alse):
        self.array_size: int = size
      ❶ self.heap_array: list = [None] * size
        self.last_index: int = 0
        self.is_min_heap: bool = min_heap
        self.indices: dict = {}

    def size(self) -> int:
        return self.last_index

    def is_empty(self) -> bool:
        return self.last_index == 0

    def in_queue(self, value) -> bool:
        return value in self.indices

    def get_priority(self, value) -> Union[float, None]:
        if not value in self.indices:
            return None
        ind: int = self.indices[value]
        return self.heap_array[ind].priority
```

The `PriorityQueue` constructor pre-allocates `heap_array` given an estimated size ❶. As we'll see later, the heap uses array doubling to increase the size if more than `array_size` elements are needed.

The `size()` and `is_empty()` functions both use the value of `last_index` to determine the number of items in the priority queue. Note that because we are using 1-indexing in this case, the number of elements in the `PriorityQueue` always equals `last_index` and the heap is empty when `last_index == 0`.

The next two functions use the priority queue's `indices` dictionary, which maps each item's value to its index in order to efficiently look up items. The `in_queue()` function checks whether a value is in the queue by checking whether it is in `indices`. The `get_priority()` function starts by checking whether the item is in the priority queue and, if not, returning `None`. Otherwise, it uses `indices` to find the correct `HeapItem` and return its priority. As with other functions that can return multiple types, we make use of `Union` from Python's `typing` library.

Defining Helper Functions

We also define several internal helper functions to support the heap operations. Since we allow the heap to be configurable as either a min heap or a max heap, these functions encapsulate the different logic needed for those two settings.

Checking Inversions

The first helper function checks whether a node and its parents are in the wrong ordering for the heap:

```
def _elements_inverted(self, parent: int, child: int) -> b
ool:
```

```
❶ if parent < 1 or parent > self.last_index:
        return False
    if child < 1 or child > self.last_index:
        return False

❷ if self.is_min_heap:
        return self.heap_array[parent] > self.heap_array[c
hild]
    else:
        return self.heap_array[parent] < self.heap_array[c
hild]
```

The code for `_elements_inverted()` starts with bounds checks on the indices of the node's parent and its child ❶. If either index is invalid, the code returns `False`. Since we will use this function later in this section to determine whether nodes need to be swapped within the heap, this prevents swaps past the bounds of the array. The check accounts for the heap's use of an array starting at index 1 by disallowing index 0.

The code then branches on whether it is dealing with a min heap or max heap ❷. In the former case, the code checks whether the elements are inverted by checking whether the parent's priority is larger than the child's priority. In the case of a max heap, the code checks for inversion by checking whether the parent's priority is smaller than the child's priority. Because we overloaded these two comparisons within the `HeapItem`, the code is always checking the item's relative priorities.

Swapping Elements

The second helper function swaps two elements in the heap's array. This operation requires a bit of extra logic because we have to not only swap the objects but also update their corresponding entries in the `indices` dictionary:

```
def _swap_elements(self, index1: int, index2: int):
❶ if index1 < 1 or index1 > self.last_index:
        return
    if index2 < 1 or index2 > self.last_index:
        return
    item1: HeapItem = self.heap_array[index1]
    item2: HeapItem = self.heap_array[index2]
    self.heap_array[index1] = item2
    self.heap_array[index2] = item1

  ❷ self.indices[item1.value] = index2
    self.indices[item2.value] = index1
```

Again, the code for `_swap_elements()` starts with bounds checking ❶, returning early if either index is out of bounds. The code then extracts both heap items, swaps their positions in the array, and updates their indices in the dictionary ❷.

Propagating Elements Upward

The third helper function implements the upward propagation described earlier in the appendix:

```
def _propagate_up(self, index: int):
    parent: int = int(index / 2)
  ❶ while self._elements_inverted(parent, index):
        self._swap_elements(parent, index)
        index = parent
        parent = int(index / 2)
```

The `_propagate_up()` code starts by computing the index of the parent. It then uses a `while` loop to keep swapping the element upward while it has the wrong ordering relative to its parent ❶. Because `_elements_inverted()` returns `False` when either index is out of bounds, the loop will also terminate when the element in question reaches the front of the array (index 1 and parent index 0).

Each time the loop finds that the element is still out of place, it swaps that element with its parent. The code then updates the element's index and computes the index of the new parent.

Propagating Elements Downward

The final helper function implements the downward propagation described earlier:

```
def _propagate_down(self, index: int):
    while index <= self.last_index:
        swap: int = index
        if self._elements_inverted(swap, 2*index):
            swap = 2*index
        if self._elements_inverted(swap, 2*index+1):
            swap = 2*index + 1

     ❶ if index != swap:
            self._swap_elements(index, swap)
            index = swap
        else:
         ❷ break
```

The `_propagate_down()` code uses a `while` loop to continually swap the element downward until it either is the last element in the array or is not inverted with respect to its children. The code checks both the left and right child using the `_elements_inverted()` function, which also handles bounds checking for the array. If it finds a child with an inverted priority ❶, the code performs the swap. Otherwise, it breaks out of the loop ❷.

Adding Items

We add (*enqueue*) a new element to the heap by first appending it to the back of the array, which corresponds to the first empty space in the bottom level of the tree. Since this location does not account for the item's priority, we have likely broken the heap property. We rectify this by swapping the item upward until it is in the correct location.

The code for enqueuing performs the heap insertion as well as the additional bookkeeping:

```
def enqueue(self, value, priority: float):
 ❶ if value in self.indices:
        self.update_priority(value, priority)
        return
```

```
  ❷ if self.last_index == self.array_size - 1:
         old_array: list = self.heap_array
         self.heap_array = [None] * self.array_size * 2
         for i in range(self.last_index + 1):
             self.heap_array[i] = old_array[i]
         self.array_size = self.array_size * 2

     self.last_index = self.last_index + 1
     self.heap_array[self.last_index] = HeapItem(value, pri
ority)
     self.indices[value] = self.last_index
     self._propagate_up(self.last_index)
```

The code for the `enqueue()` function starts by checking whether the object already exists in the priority queue by determining whether its value is in `indices` ❶. If so, it updates the item's priority and returns. The code does not insert items with duplicate values.

The code next checks that the list has sufficient space for new elements ❷. If not, it must allocate more space before inserting the item and uses array doubling to increase the size.

Finally, the code inserts the element at the end of `heap_array`. It marks this location in the `indices` dictionary for later lookups, then uses `_propagate_up()` to fix any ordering problems the insertion caused. Importantly, the `_propagate_up()` function uses the `_swap_elements()` function, which updates `indices`. Therefore, although the code initially sets `indices[value]` to `last_index`, it correctly updates this index mapping throughout the process.

Removing Items

We remove (*dequeue*) the top node by replacing it with the last value in the array. This jumps the last node in our heap to the root of the tree, very likely breaking the heap property in the process. We correct the relative ordering by propagating the item down the tree until it is no longer out of order with respect to its children.

The code for dequeuing performs the heap removal as well as the additional bookkeeping:

```
def dequeue(self):
    if self.last_index == 0:
        return None

  ❶ result: HeapItem = self.heap_array[1]
    new_top: HeapItem = self.heap_array[self.last_index]
    self.heap_array[1] = new_top
    self.indices[new_top.value] = 1

    self.heap_array[self.last_index] = None
    self.indices.pop(result.value)
    self.last_index = self.last_index - 1

    self._propagate_down(1)
    return result.value
```

The code for the `dequeue()` function starts by checking whether the queue is empty and, if so, returns `None`. (Depending on the context of the code, we might alternatively want to raise an error.)

If the queue is not empty, the code updates the heap and index mapping. First, it swaps the last element in `heap_array` into the first position, saving the old root as `result` ❶. Second, it removes the former top element (`result`) from both the array and the index map. Third, it fixes any breakages to the heap property using the `_propagate_down()` function. Finally, it returns the result's value.

Modifying Priorities

The final operation supported by the modifiable priority queue is to change the priority of an element. This involves looking up the element's location in `heap_array`, changing the priority, and fixing any resulting breakages of the heap property using either the `_propagate_up()` or `_propagate _down()` functions. The majority of this code consists of determining which of the two propagation functions it should use:

```
def update_priority(self, value, priority: float):
    if not value in self.indices:
        return
    index: int = self.indices[value]
    old_priority: float = self.heap_array[index].priority
    self.heap_array[index].priority = priority

    if self.is_min_heap:
        if old_priority > priority:
            self._propagate_up(index)
        else:
            self._propagate_down(index)
    else:
        if old_priority > priority:
            self._propagate_down(index)
        else:
            self._propagate_up(index)
```

The code starts by determining if the value is in the priority queue by checking that `value` is in the `indices` dictionary. If not, there is nothing to update, and it can return immediately.

If the value is in the priority queue, the code looks up and saves the item's current index (`index`) and priority (`old_priority`), then sets the new priority. At this point the code must determine which propagation function to use, depending on whether the object is a min heap and whether the new priority is larger or smaller than the old priority. The code uses `_propagate_up()` if the object is a min heap and the old priority is larger, or if the object is a max heap and the old priority is smaller. Alternatively, it uses `_propagate_down()` if the object is a min heap and the old priority is smaller, or if the object is a max heap and the old priority is larger.

Peek Functions

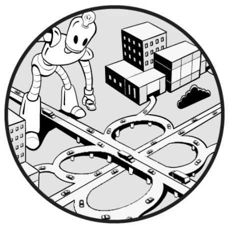
In addition to the standard priority enqueue/dequeue type functions presented in this appendix, we provide a few additional convenience functions to perform operations that allow us to peek at the top value—that is, to look at it without dequeuing it. We can return the entire `HeapItem` for the top item (or `None` if the queue is empty), the item's value, or the item's priority:

```
def peak_top(self) -> Union[HeapItem, None]:
    if self.is_empty():
        return None
    return self.heap_array[1]

def peek_top_priority(self) -> Union[float, None]:
    obj: Union[HeapItem, None] = self.peak_top()
    if not obj:
        return None
    return obj.priority

def peek_top_value(self):
    obj: Union[HeapItem, None] = self.peak_top()
    if not obj:
        return None
    return obj.value
```

Each of these three functions provides a useful mechanism for examining the “best” item in the priority queue without modifying the queue and can be useful for debugging.

C

UNION-FIND



Kruskal's algorithm, randomized maze generation, and single-linkage clustering from Chapter 10 all use a data structure called `UnionFind` to represent the disjoint sets of nodes corresponding to the different connected components in a graph. This data structure allows the algorithms to efficiently (1) determine whether two nodes are already in the same connected component and (2) merge two different components. For completeness, this appendix describes and provides the code for this data structure.

We begin by providing a very brief overview of union-find data structures, followed by just enough code to implement the algorithms in this book. We encourage interested readers to explore additional resources. The "Union-Find" chapter of Daniel Zingaro's *Algorithmic Thinking*, 2nd edition (No Starch Press, 2023), provides an accessible introduction to these fascinating data structures as well as additional optimizations.

The Union-Find Data Structure

The *union-find* data structure (also called a *disjoint sets* data structure) is commonly viewed as a list of trees (also called a *forest* of trees). Each item is represented as a tree node and each set is encoded as a tree. Items are considered to be in the same set if and only if they are in the same tree.

Figure C-1 shows an example union-find data structure with 11 items organized into three sets: {0, 1, 6, 7, 10}, {3, 5, 9}, {2, 4, 8}. As shown in the figure, the trees are not restricted to be binary (at most two children per node), nor do they enforce an ordering over the elements.

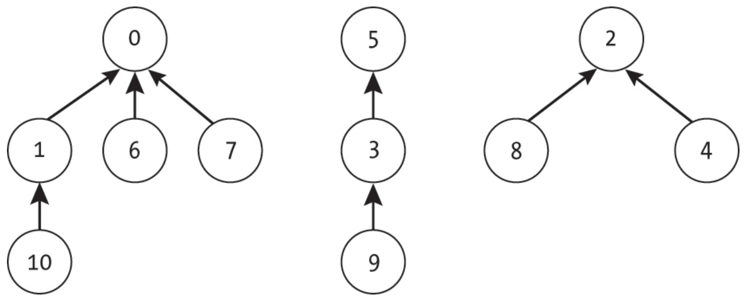


Figure C-1: Three disjoint sets represented as trees

Each set in this data structure is uniquely identified by the index number of the root node. The trees in Figure C-1 have labels 0, 5, and 2 from left to right. We can easily retrieve the set label for any item by traversing from that item's node to the root of the tree. For example, we could identify the set label for item 9 by progressing from node 9 to node 3 to node 5 and then returning 5.

We create the union of sets by combining trees. There are a variety of ways to append one tree to another. In this appendix, we'll use the common optimization of appending the root node of the tree with fewer nodes to the root node of a larger tree. Figure C-2 shows an example of combining the sets rooted at 0 and 2 into a single set. Since the tree rooted at node 2 has fewer nodes, we set the parent pointer of node 2 to node 0, effectively adding the subtree as a child.

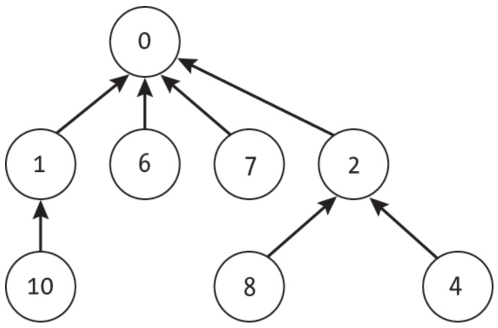


Figure C-2: Combining two trees

For illustration purposes, in this section we describe a minimalist `UnionFind` data structure that uses an explicit forest-of-trees implementation to make the tree-based operations clear. More efficient optimizations are possible, such as array-based implementations of the data structure and the use of path compression to reduce tree height.

UnionFind

The `UnionFind` data structure partitions elements into different (and disjoint) sets such that each element belongs to exactly one set. For Kruskal's algorithm, these sets represent the different connected components within the graph. Nodes within the same component are part of the same set.

As discussed in Chapter 10, this data structure is powerful because it facilitates performing two operations very quickly. The first operation is to determine whether two points are in the same set, which is necessary for determining if two nodes are already connected. The second operation is merging two sets, which is necessary for connecting components.

This appendix provides the code for the following functions that make up the union-find interface:

`are_disjoint(i, j)` Determines whether two elements `i` and `j` are in different sets

`union_sets(i, j)` Merges the set with element `i` and the set with element `j` into a single set

`find_set(i)` Returns a unique label for the set containing element `i`

UnionFindNode

Since we only ever need to travel up the tree (rather than down), each node only needs to store two pieces of information: its own index number and the pointer to its parent. Nodes do not need to store pointers to their children. We can define a minimal `UnionFindNode` as follows:

```
class UnionFindNode:
    def __init__(self, label: int):
        self.label = label
        self.parent = None
```

We initially set the parent of a node to `None` to indicate that it is a root node.

UnionFind Class

Our minimal `UnionFind` object tracks three pieces of information:

`nodes` (`list`)	A list of `UnionFindNode` objects indexed by their label
`set_sizes` (`list`)	A list that maps the set's label to its size
`num_disjoint_sets` (`int`)	The number of disjoint sets

We use a list to store the nodes because we only need to support contiguous integer labels for the algorithms in this book. However, we could support more general labels, such as strings, by using a dictionary to map each label to its corresponding `UnionFindNode`. The `num_disjoint_sets` attribute could be computed from the other attributes but is explicitly stored for simplicity.

Using these attributes, we define a constructor to set up the initial state of the union-find data structure:

```
class UnionFind:
    def __init__(self, num_sets: int):
        self.nodes: list = [UnionFindNode(i) for i in rang
e(num_sets)]
        self.set_sizes: list = [1 for i in range(num_set
s)]
        self.num_disjoint_sets: int = num_sets
```

The constructor takes the number of items (`num_sets`) and constructs both the full list of nodes (`nodes`) and the list of sizes for each set (`set_sizes`). Since the items all start in disjoint sets, the constructor initializes the `num_sets` sizes to 1 for each item. Finally, it sets the count of disjoint sets (`num_disjoint_sets`).

Finding the label of a set corresponds to walking up the tree and returning the label of the root node:

```
def find_set(self, label: int) -> int:
    if label < 0 or label >= len(self.nodes):
        raise IndexError

  ❶ current: UnionFindNode = self.nodes[label]
    while current.parent is not None:
        current = current.parent
    return current.label
```

The `find_set()` function starts by checking the bounds of the label and raising an `IndexError` if the label is out of bounds. It then starts at the current node ❶ and uses a `while` loop to walk up the tree to the parent. It returns the label of the parent node as the identifier for the set.

The `are_disjoint()` function uses two calls to `find_set()` to extract the set labels for each item and test whether they are equal:

```
def are_disjoint(self, label1: int, label2: int) -> bool:
    return self.find_set(label1) != self.find_set(label2)
```

If the set labels are the same, the items must share a root node and thus be in the same set.

Taking the union of two sets consists of appending the trees:

```
def union_sets(self, label1: int, label2: int):
  ❶ set1_label: int = self.find_set(label1)
    set2_label: int = self.find_set(label2)
    if set1_label == set2_label:
        return
```

```
 ❷ if self.set_sizes[set1_label] < self.set_sizes[set2_la
bel]:
        small = set1_label
        large = set2_label
    else:
        small = set2_label
        large = set1_label
 ❸ self.nodes[small].parent = self.nodes[large]
    self.set_sizes[large] += self.set_sizes[small]
    self.set_sizes[small] = 0
    self.num_disjoint_sets -= 1
```

The `union_sets()` function starts by finding the label for each set and checking whether they are already equal to each other ❶. If so, there is nothing to be done and the function returns. If not, the function uses the `set_sizes` list to determine which tree has fewer nodes ❷ and appends the root node of the smaller tree as a child of the larger tree's root node ❸. Finally, the function updates the remaining data by computing the new size of the larger tree, setting the size entry of the smaller tree to `0` (since it is no longer a disjoint set), and updating the number of disjoint sets.

INDEX