3 Blind search algorithms

This chapter covers

- Applying different graph types
- Graph search algorithms
- Using graph traversal algorithms to find a path between two nodes
- Using blind search algorithms to find the shortest path between two nodes in a graph
- Solving a real-world routing problem using graph search algorithms

You were introduced to deterministic and stochastic algorithms in chapter 2. In this chapter, we will focus on deterministic algorithms, specifically blind search algorithms, and their applications in exploring tree or graph structures and finding the shortest path between nodes. Using these algorithms, you can explore a maze from an initial state to a goal state, solve *n*-puzzle problems, figure out the distance between you and any other person on a social media graph, search a family tree to determine the exact relationship between any two related people, or find the shortest path between any origin (e.g., your home) and any destination. Blind search algorithms are important, as they are often more efficient and practical to use when dealing with simple, well-defined problems.

3.1 Introduction to graphs

A *graph* is a nonlinear data structure composed of entities known as *vertices* (or nodes) and the relationships between them, known as *edges* (or *arcs* or *links*). This data structure does not follow a sequential pattern, making it *nonlinear*, unlike arrays, stacks, or queues, which are linear structures.

A graph can be represented mathematically by G, where $G = (V, E)$. V represents the set of nodes or vertices, and E represents the set of edges or links. Various attributes can also be added as components to the edge tuple, such as edge length, capacity, or any other unique properties (e.g., road material). Graphs can be classified as undirected, directed, multigraph, acyclic, and hypergraphs.

An *undirected graph* is one where a set of nodes are connected using bidirectional edges. This means that the order of two connected nodes is not essential.

NetworkX is a commonly used Python library for creating, manipulating, and studying the structure, dynamics, and functions of graphs and complex networks (see appendix A for more information about graph libraries). The following listing shows how you can use NetworkX to create an undirected graph.

Listing 3.1 Creating an undirected graph using NetworkX

```
import networkx as nx
import matplotlib.pyplot as plt

graph = nx.Graph()

nodes = list(range(5))
graph.add_nodes_from(nodes)

edges = [(0,1),(1,2), (1,3), (2,3),(3,4)]
graph.add_edges_from(edges)

nx.draw_networkx(graph, font_color="white")
```

Generate a list of nodes from 0 to 4.

Define a list of edges.

The output of this code is shown in figure 3.1. The actual layout you get might be different, but the connections among the vertices will be the same as shown here.

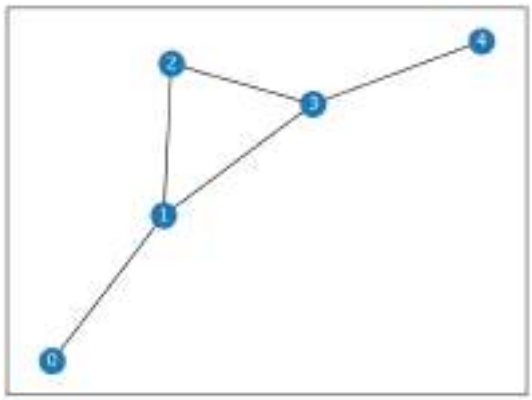


Figure 3.1 An undirected graph

A *directed graph* is a graph in which a set of nodes are connected using directional edges. Directed graphs have many applications, such as representing flow constraints (e.g., one-way streets), relations (e.g., causal relationships), and dependencies (e.g., tasks that depend on the completion of other tasks). The following listing shows how to use NetworkX to create a directed graph.

Listing 3.2 Creating a directed graph using NetworkX

```
import networkx as nx
import matplotlib.pyplot as plt

graph = nx.DiGraph()
nodes = list(range(5))
edges = [(0,1),(1,2), (1,3), (2,3),(3,4)]
graph.add_edges_from(edges)
graph.add_nodes_from(nodes)
nx.draw_networkx(graph, font_color="white")
```

DiGraph allows for directed edges.

The code output is shown in figure 3.2. Note the arrows indicating the edge directions.

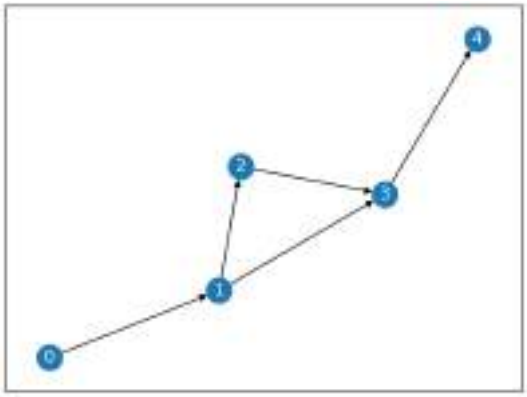


Figure 3.2 A directed graph

A *multigraph* is a graph in which multiple edges may connect the same pair of vertices. These edges are called *parallel edges*. Multigraphs can be used to represent complex relationships between nodes, such as multiple parallel roads between two locations in traffic routing, multiple capacities and demands in resource allocation problems, and multiple relationships between individuals in social networks, to name just a few. Unfortunately, NetworkX is not particularly good at visualizing multigraphs with parallel edges. This listing shows how you can use NetworkX in conjunction with the Matplotlib library to create a multigraph.

Listing 3.3 Creating a multigraph using NetworkX

```
import networkx as nx
import matplotlib.pyplot as plt

graph = nx.MultiGraph()
nodes = list(range(5))
edges = [(0,1),(0,1),(4,3),(1,2), (1,3), (2,3),(3,4),(0,1)]
graph.add_nodes_from(nodes)
graph.add_edges_from(edges)

pos = nx.kamada_kawai_layout(graph)
ax = plt.gca()

for e in graph.edges:
    ax.annotate("",xy=pos[e[0]], xycoords='data', xytext=pos[e[1]],
    ➥textcoords='data', arrowprops=dict(arrowstyle="-",
    ➥connectionstyle=f"arc3, rad={0.3*e[2]}"),zorder=1)

nx.draw_networkx_nodes(graph, pos)      #C
nx.draw_networkx_labels(graph,pos, font_color='w')      #C

plt.show()
```

Node positions are generated using the Kamada-Kawai path-length cost function.

Draw each edge one at a time, modifying the curvature of the edge based on its index (i.e., the second edge between nodes 0 and 1).

Draw nodes and node labels.

It is worth noting that `kamada_kawai_layout` attempts to position nodes on the space so that the geometric (Euclidean) distance between them is as close as possible to the graph-theoretic (path) distance between them. Figure 3.3 shows an example of a multigraph generated by this code.

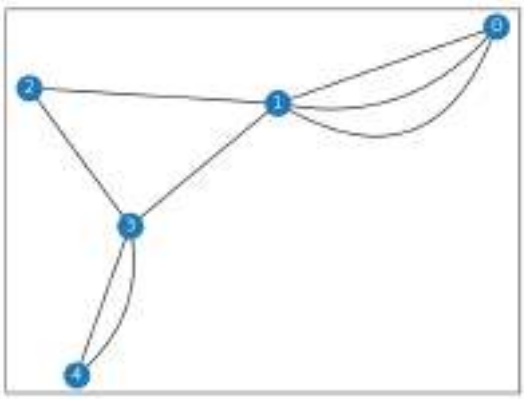


Figure 3.3 Example of a multigraph. Notice the three parallel edges connecting nodes 0 and 1, as well as the two edges connecting nodes 3 and 4.

As the name implies, an *acyclic graph* is a graph without cycles. A *tree*, as a specialized case of a graph, is a connected graph with no cycles or self-loops. In graph theory, a connected graph is a type of graph in which there is a path between every pair of

vertices. A *cycle*, also called a *self-loop* or a *circuit*, is an edge in a graph that connects a vertex (or node) to itself. In task scheduling, acyclic graphs can be used to represent the relationships between tasks where each node represents a task and each directed edge represents a precedence constraint. This constraint means that the task represented by the end node cannot start until the task represented by the start node is completed. We'll discuss the assembly line balancing problem in chapter 6 as an example of scheduling problems.

The following listing shows how you can use NetworkX to create and verify an acyclic graph. An example of an acyclic graph is shown in figure 3.4

Listing 3.4 Creating an acyclic graph using NetworkX

```
import networkx as nx
import matplotlib.pyplot as plt

graph = nx.DiGraph()
nodes = list(range(5))
edges = [(0,1), (0,2),(4,1),(1,2),(2,3)]
graph.add_nodes_from(nodes)
graph.add_edges_from(edges)

nx.draw_networkx(graph, nx.kamada_kawai_layout(graph), with_labels=True,
➥ font_color='w')
plt.show()

nx.is_directed_acyclic_graph(graph)
```

Check if the graph is acyclic.

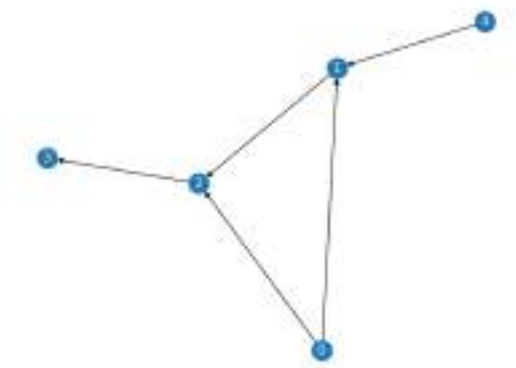


Figure 3.4 An acyclic graph—no path cycles back to any starting node.

A *hypergraph* is a generalization of a graph in which the generalized edges (called *hyperedges*) can join any number of nodes. Hypergraphs are used to represent complex networks because they can capture higher-order many-to-many relationships. They're used in domains such as social media, information systems, computational geometry,

computational pathology, and neuroscience. For example, a group of people working on a project can be represented by a hypergraph. Each person is represented by a node, and the project is represented by a hyperedge. The hyperedge connects all the people working on the project, regardless of how many people are involved. The hyperedge can also contain other attributes, such as the project's name, the start and end dates, the budget, etc.

The following listing shows how you can use HyperNetX (HNX) to create a hypergraph. HNX is a Python library that enables us to model the entities and relationships found in complex networks as hypergraphs. Figure 3.5 shows an example of a hypergraph.

Listing 3.5 Creating a hypergraph using HyperNetX

```
import hypernetx as hnx

data = {
     0: ("A","B","G"),
     1: ("A","C","D","E","F"),
     2: ("B","F"),
     3: ("A","B","D","E","F","G")
}
H = hnx.Hypergraph(data)
hnx.draw(H)
```

The data for the hypergraph comes as key-value pairs of hyperedge name/hyperedge node groups.

Create a hypergraph for the provided data.

Visualize the hypergraph.

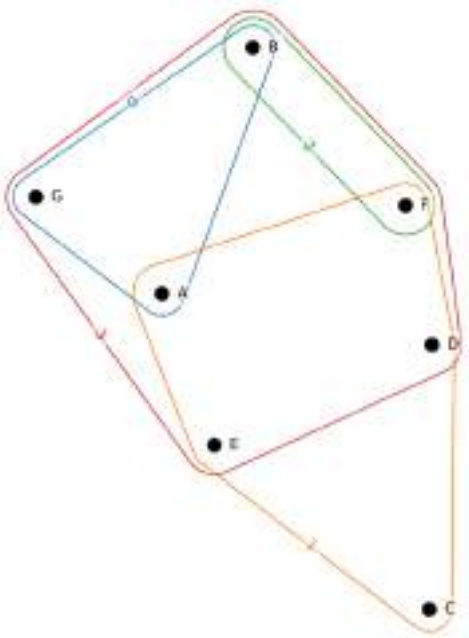


Figure 3.5 An example of a hypergraph. Hyperedges can connect more than two nodes, such as hyperedge 0, which links nodes A, B, and G.

Graphs can also be weighted or unweighted. In a *weighted graph*, a weight, or value, is assigned to each edge. For example, in the case of road networks, the edges could have weights that represent the cost of traversing the road. This weight could represent distance, time, or any other metric. In telecommunications networks, the weight might represent the cost of utilizing that edge or the strength of the connections between the communication devices.

Listing 3.6 shows how you could create and visualize a weighted graph between telecommunication devices. The weights in this example represent the speed of connections between the devices in Mbps. Running this code generated the weighted graph in figure 3.6.

Listing 3.6 Creating a weighted graph using NetworkX

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.Graph()

G.add_node("Device1", pos=(0,0))
G.add_node("Device2", pos=(0,2))
G.add_node("Device3", pos=(2,0))
G.add_node("Device4", pos=(2,2))

G.add_weighted_edges_from([("Device1", "Device2", 45.69),
                           ("Device1", "Device3", 56.34),
                           ("Device2", "Device4", 18.5)])

pos = nx.get_node_attributes(G, 'pos')
nx.draw_networkx(G, pos, with_labels=True)
nx.draw_networkx_edge_labels(G, pos,
➥edge_labels={(u, v): d['weight'] for
➥u, v, d in G.edges(data=True)})
plt.show()
```

Create an empty weighted graph.

Add nodes to the graph (representing devices).

Add weighted edges to the graph (representing connections).

Get node position attributes from the graph.

Draw the graph.

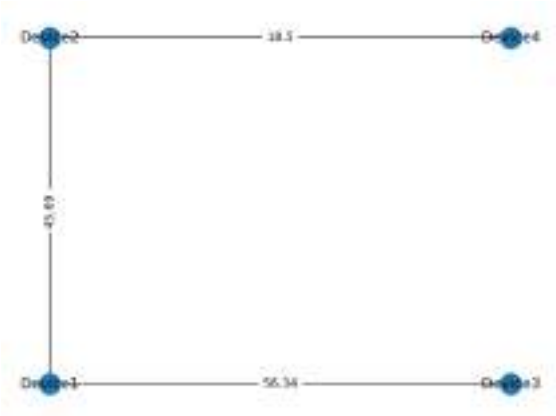


Figure 3.6 Example of a weighted graph

Graphs are everywhere. Search engines like Google see the internet as a giant graph where each web page is a node, and two pages are joined by an edge if there is a link from one page to the other. A social media platform like Facebook treats each user profile as a node on a social graph, and two nodes are said to be connected if they are each other's friends or have social ties. The concept of "following" a user, such as on a platform like X (previously Twitter), can be represented by a directional edge, where user *A* can follow user *B*, but the reverse is not necessarily true. Table 3.1 shows the meanings of nodes and edges on different social media platforms.

Table 3.1 Examples of graphs in the context of social media

Social media platform	Nodes	Edges	Type of edge
Facebook	Users, groups, posts, and events	Friendship, group membership, messages, creation of posts, and reactions on posts	Undirected: a like, or react, or comment Directed: a friend request
X (previously Twitter)	Users, groups, unregistered persons, and posts	Following, group membership, messages, creation of posts, and reactions on posts	Undirected: a mention or a retweet Directed: the following relationship (when you follow a person, they do not automatically follow you back)
LinkedIn	Users, groups, unregistered persons, posts, skills, and jobs	Connections, group membership, posting, reactions on posts, messages, endorsements, invitations, recommending jobs	Undirected: an endorsement or recommendation Directed: a connection
Instagram	Users, comments, containers for publishing posts, hashtags, media (e.g., photo, video, story, or album), and pages (Facebook page)	Relationships between users such as following, liking, and commenting	Undirected: a like or a comment Directed: a follow relationship
TikTok	Users, videos, hashtags, locations, and keywords	Relationships between users such as following, liking, and commenting	Undirected: a like or comment Directed: a follow relationship

In a road network graph, the nodes represent landmarks such as intersections and points of interest (POI), and the edges represent the roads. In such a graph, most of the edges are directed, meaning that they have specific directions, and they may have additional information such as length, speed limit, capacity, etc. Each edge is a two-endpoint connection between two nodes, where the direction of the edge represents the direction of traffic flow. A *route* is a sequence of edges connecting the origin node to the destination node.

OSMnx is a Python library developed to simplify the retrieving and manipulating of data from OpenStreetMap (OSM; openstreetmap.org). OSM is a crowdsourced

geographic database of the world (see appendix B for more information about how to fetch data from open geospatial data sources). OSMnx lets you download filtered data from OSM and returns the network as a NetworkX graph data structure. It can also convert a text descriptor of a place into a NetworkX graph (see appendix A for more information about graph and mapping libraries). The following listing uses the University of Toronto as an example.

Listing 3.7 University of Toronto example

```
import osmnx as ox
import matplotlib.pyplot as plt

place_name = "University of Toronto"

graph = ox.graph_from_address(place_name)
ox.plot_graph(graph,figsize=(10,10))
```

A graph_from_address can also take city names and mailing addresses as input.

Figure 3.7 shows an OSM map of the area around the St. George campus of the University of Toronto. The graph shows the edges and nodes of the road network surrounding the campus in downtown Toronto.

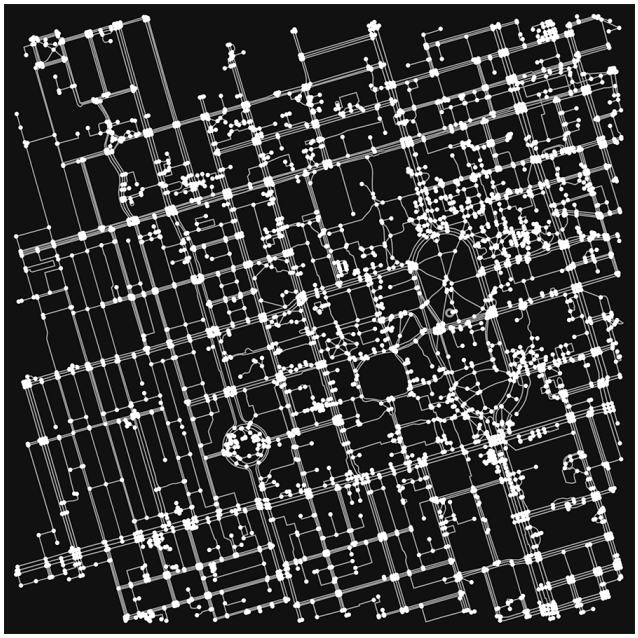


Figure 3.7 St. George campus, University of Toronto

While the map may look visually interesting, it lacks the context of surrounding geographic features. Let's use the folium library (see appendix A) to create a base layer map with street names, neighborhood names, and even building footprints.

```
graph = ox.graph_from_address(place_name)
ox.folium.plot_graph_folium(graph)
```

Figure 3.8 shows the road network surrounding the St. George campus.

Figure 3.8 Road network around St. George campus, University of Toronto

Suppose you want to get from one location to another on this campus. For example, imagine you're starting at the King Edward VII equestrian statue near Queen's Park in Toronto, and you need to cross the campus to attend a lecture at the Bahen Centre for Information Technology. Later in this chapter, you will see how you can calculate the shortest path between these two points.

For now, let's just plot these two locations on the map using the folium library. Figure 3.9 shows the folium map and markers.

Listing 3.8 Plotting with folium

```
import folium

center=(43.662643, -79.395689)
source_point = (43.664527, -79.392442)
destination_point = (43.659659, -79.397669)

m = folium.Map(location=center, zoom_start=15)
folium.Marker(location=source_point,icon=folium.
➥Icon(color='red',icon='camera', prefix='fa')).add_to(m) #E
folium.Marker(location=center,icon=folium.Icon(color='blue',
➥icon='graduation-cap', prefix='fa')).add_to(m) #E
folium.Marker(location=destination_point,icon=folium.Icon(color='green',
➥icon='university', prefix='fa')).add_to(m)
```

The GPS coordinates (latitude and longitude) of the University of Toronto

The GPS coordinates of the equestrian statue as a source point

The GPS coordinates of the Bahen Centre for Information Technology as the destination

Create a map centered around a specified point.

Add markers with icons.

Figure 3.9 Visualizing points of interest using folium markers

The output of the code is interactive and allows for features such as zooming, panning, and even layer filtering (when enabled). Appendix A provides more details about map visualization libraries in Python.

3.2 Graph search

As I mentioned in chapter 2, search algorithms can be broadly classified into deterministic and stochastic algorithms. In *deterministic search*, the search algorithm follows a rigorous procedure, and its path and the values of both the design variables and the functions are repeatable. The algorithm will follow the same path for the same starting point whenever you run the program, whether it's today or ten years in the future. In *stochastic search*, on the other hand, the algorithm always has some randomness, and the solution is not exactly repeatable. Each time you run the algorithm, you may get slightly different results.

Based on the availability of information about the search space or domain knowledge (e.g., the distance from the current state to the goal), deterministic search algorithms can be broadly classified into *blind* (or *uninformed*) search and *informed* search, as illustrated in figure 3.10. Some of these algorithms, such as Kruskal's minimum spanning tree (MST) algorithm, will be covered in the next chapter. This chapter focuses on blind search algorithms. Blind search is a search approach where no information about the search space is needed.

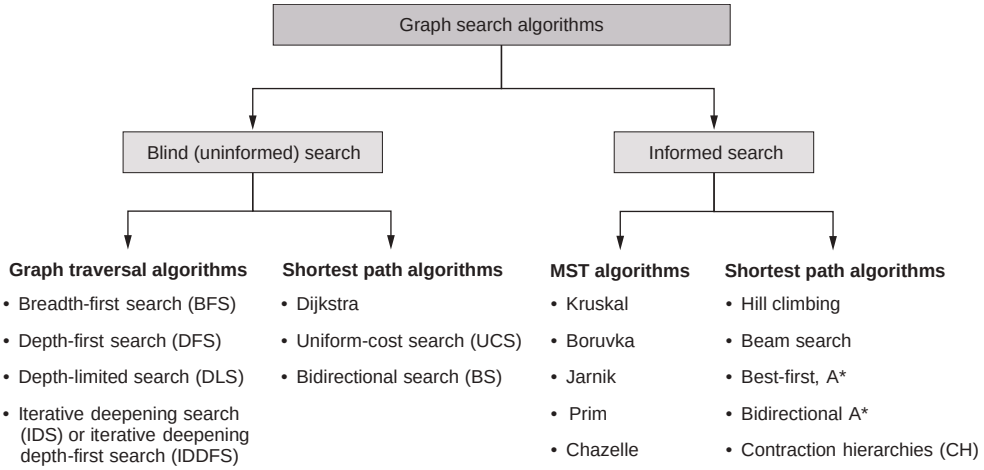


Figure 3.10 Graph search methods

A blind search may conclude upon discovering the first solution, depending on the algorithm's termination criteria. However, the search space may contain numerous valid but non-optimal solutions, so a blind search may return a solution that meets all the requirements but does so in a non-optimal way. An optimal solution can be found by running a blind search following an exhaustive search or brute-force strategy to find all the feasible solutions, which can then be compared to select the best one. This is similar to applying the British Museum algorithm, which finds a solution by checking all possibilities one by one. Given the fact that blind search treats every node in the graph or tree equally, this search approach is often referred to as a *uniform search*.

Examples of blind search algorithms include, but are not limited to, the following:

- *Breadth-first search* (BFS) is a graph traversal algorithm that builds the search tree by levels.
- *Depth-first search* (DFS) is a graph traversal algorithm that first explores nodes going through one adjacent to the root, then the next adjacent, until it finds a solution or it reaches a dead end.
- *Depth-limited search* (DLS) is a DFS with a predetermined depth limit.
- *Iterative deepening search* (IDS), or *iterative deepening depth-first search* (IDDFS), combines DFS's space efficiency and BFS's fast search by incrementing the depth limit until the goal is reached.
- *Dijkstra's algorithm* solves the single-source shortest-path problem for a weighted graph with non-negative edge costs.

- *Uniform-cost* search (UCS) is a variant of Dijkstra's algorithm that uses the lowest cumulative cost to find a path from the source to the destination. It is equivalent to the BFS algorithm if the path cost of all edges is the same.
- *Bidirectional* search (BS) is a combination of forward and backward search. It searches forward from the start and backward from the goal simultaneously.

The following sections discuss graph traversal algorithms and shortest path algorithms, focusing on BFS, DFS, Dijkstra's algorithm, UCS, and BS as examples of blind search approaches.

3.3 Graph traversal algorithms

Graph traversal is the process of exploring the structure of a tree or a graph by visiting the nodes following a specific, well-defined rule. This category of graph search algorithms only seeks to find a path between two nodes without optimizing for the length of the final route.

3.3.1 Breadth-first search

Breadth-first search (BFS) is an algorithm where the traversal starts at a specified node (the source or starting node) and follows the graph layerwise, thus exploring all of the current node's neighboring nodes (those directly connected to the current node). Then, if a result has not been found, the algorithm searches the next-level neighbor nodes. This algorithm finds a solution if one exists, assuming that a finite number of successors, or branches, always follow any node. Algorithm 3.1 shows the BFS steps.

Algorithm 3.1 Breadth-first search (BFS)

```
Inputs: Source node, Destination node
Output: Route from source to destination

Initialize queue ← a FIFO initialized with source node
Initialize explored ← empty
Initialize found ← False

While queue is not empty and found is False do
    Set node ← queue.dequeue()
    Add node to explored
    For child in node.expand() do
        If child is not in explored and child is not in queue then
            If child is destination then
                Update route ← child route()
                Update found ← True
            Add child to queue
Return route
```

BFS uses the queue as a data structure to maintain the states to be explored. A queue is a first in, first out (FIFO) data structure, where the node that has been sitting on the queue for the longest time is the next node to be expanded. BFS dequeues a state off the queue and then enqueues its successors back on the queue.

Let's consider the 8-puzzle problem (sometimes called the *sliding-block problem* or *tile-puzzle problem*). The puzzle consists of an area divided into a 3 × 3 grid. The tiles are numbered 1 through 8, except for an empty (or blank) tile. The blank tile can be moved by swapping its position with any tile directly adjacent (up, down, left, right). The puzzle's goal is to place the tiles so that they are arranged in order. Variations of the puzzle allow the empty tile to end up either at the first or last position. This problem is an example of a well-structured problem (WSP) with the following well-defined components:

- States—Location of the blank and location of the eight tiles
- Operator (successor)—Blank moves left, right, up, and down
- Goal—Match the state given by the goal state
- Solution/path—Sequence through state space
- Stopping criteria—An ordered puzzle (reaching the Goal state)
- Evaluation criteria—Number of steps or path cost (the path length)

Figure 3.11 illustrates the BFS steps for solving the 8-puzzle problem and the search tree traversal order. In this figure, the state represents the physical configuration of the 8-puzzle problem, and each node in the search tree is a data structure that includes information about its parent, children, depth, and the cost of the path from the initial state to this node. Level 1 nodes are generated from left to right by moving the blank title left, up, and right respectively. Moving forward, level 2 nodes are generated by expanding the previously generated nodes in level 1, avoiding the previously explored nodes. We keep repeating this procedure to traverse all the possible nodes or until we hit the goal (the shaded grid). The number of steps to reach the goal will depend mainly on the initial state of the 8-puzzle board. The highlighted numbers show the order of traverse. As you can see, BFS progresses horizontally before it proceeds vertically.

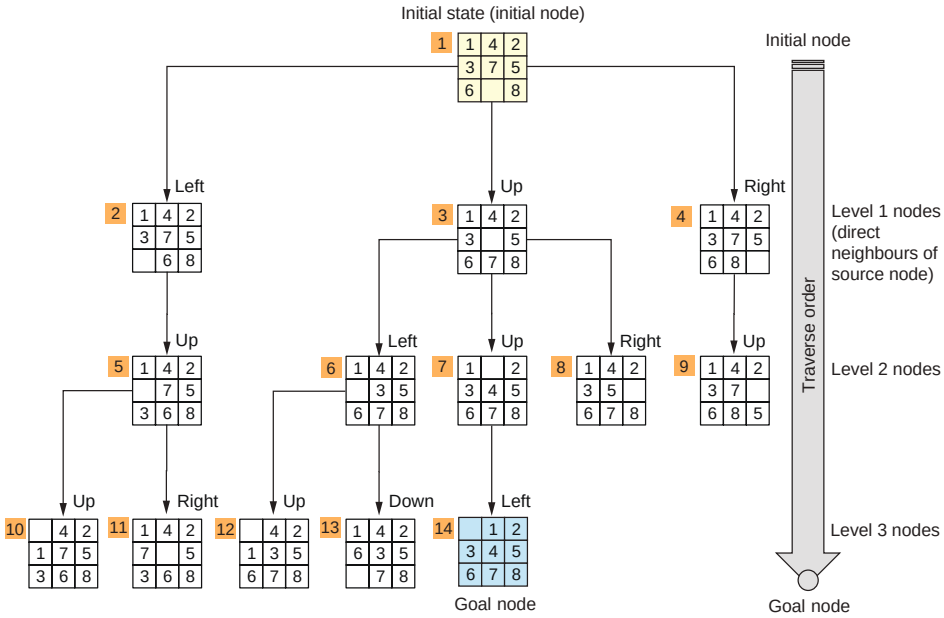


Figure 3.11 Using BFS to solve the 8-puzzle problem

Listing 3.9 utilizes a generic BFS algorithm developed for this book, which can be found in the Optimization Algorithm Tools (optalgotools) Python package (see appendix A for installation instructions). The algorithm takes starting and goal states as inputs and returns a `solution` object. This `solution` object contains the actual result and some performance metrics, such as processing time, maximum space used, and the number of solution states explored. The `State` class and `visualize` function are defined in the complete listing available in the book's GitHub repo. The `State` class helps manage some data structures and utility functions, and it will allow us to reuse this problem's structure later with different algorithms.

Listing 3.9 Solving the 8-puzzle problem using BFS

```
#!pip install optalgotools
from optalgotools.algorithms.graph_search import BFS

init_state = [[1,4,2], [3,7,5], [6,0,8]]

goal_state = [[0,1,2], [3,4,5], [6,7,8]]


init_state = State(init_state)
goal_state = State(goal_state)


if not init_state.is_solvable():
    print("This puzzle is not solvable.")
```

The BFS algorithm is imported from a library called optalgotools.

See the State class in the complete listing.

Some boards are not solvable.

```
else:

    solution = BFS(init_state, goal_state)
    print(f"Process time: {solution.time} s")
    print(f"Space required: {solution.space} bytes")
    print(f"Explored states: {solution.explored}")
    visualize(solution.result)
```

See the visualize function in the complete listing.

This is an example solution, given the preceding inputs:

```
Process time: 0.015625 s
Space required: 624 bytes
Explored states: 7
```

Figure 3.12 shows the state changes following the BFS algorithm.

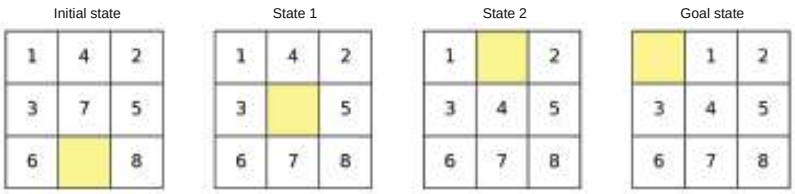


Figure 3.12 The step-by-step BFS solution using Python. BFS searches for a solution but does not consider optimality.

To really understand how BFS works, let's look at the steps involved in a simple path-planning problem. This problem finds a collision-free path for a mobile robot or autonomous vehicle from a start position to a given destination amidst a collection of obstacles.

1. Add the source node to the queue (figure 3.13).

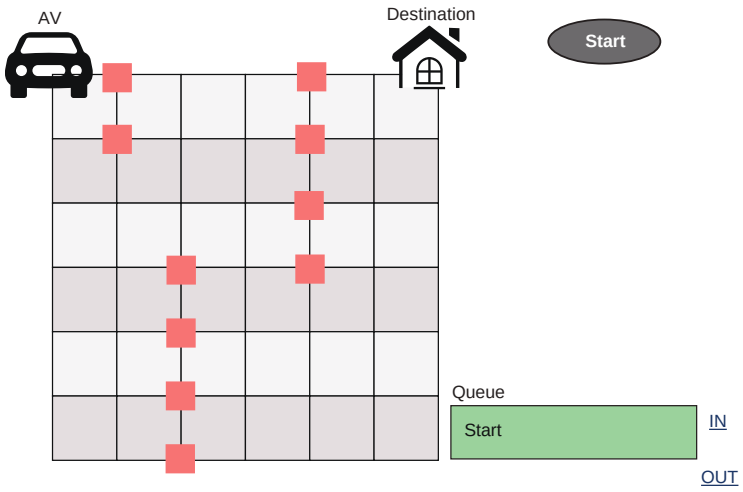


Figure 3.13 Solving the path-planning problem using BFS—step 1

2 The robot can only move to the south (S) node, as the east (E) and southeast (SE) nodes are obstructed (figure 3.14).

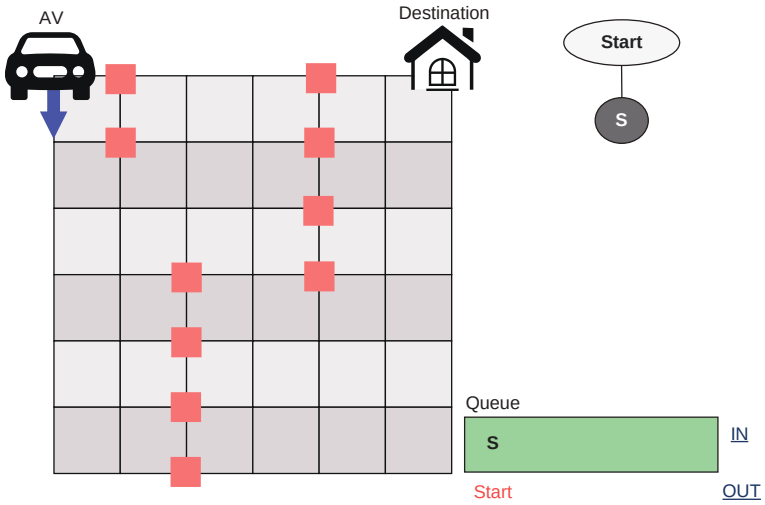


Figure 3.14 Solving the path-planning problem using BFS—step 2

3 Take S out (FIFO), and explore its neighboring nodes, S and SE, with E being an obstructed node (figure 3.15).

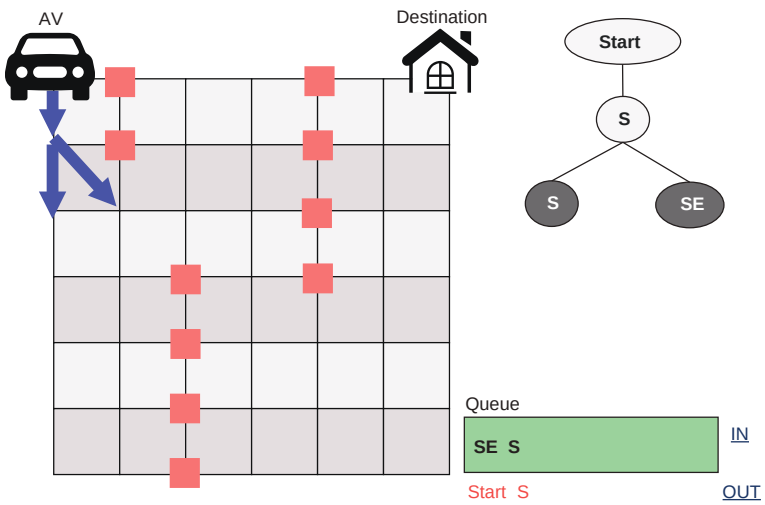


Figure 3.15 Solving the path-planning problem using BFS—step 3

4 Take S out (FIFO), and explore its neighboring nodes, S and SE (figure 3.16).

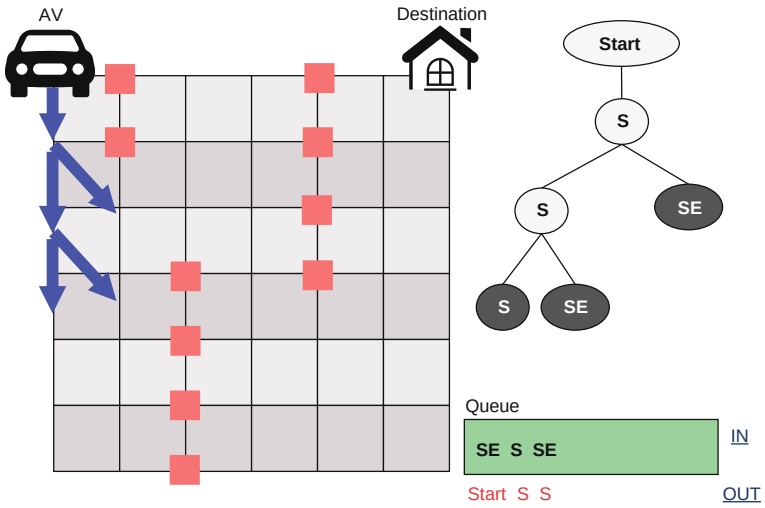


Figure 3.16 Solving the path-planning problem using BFS—step 4

5 Take SE out (FIFO), and explore its neighboring nodes, E and NE (figure 3.17).

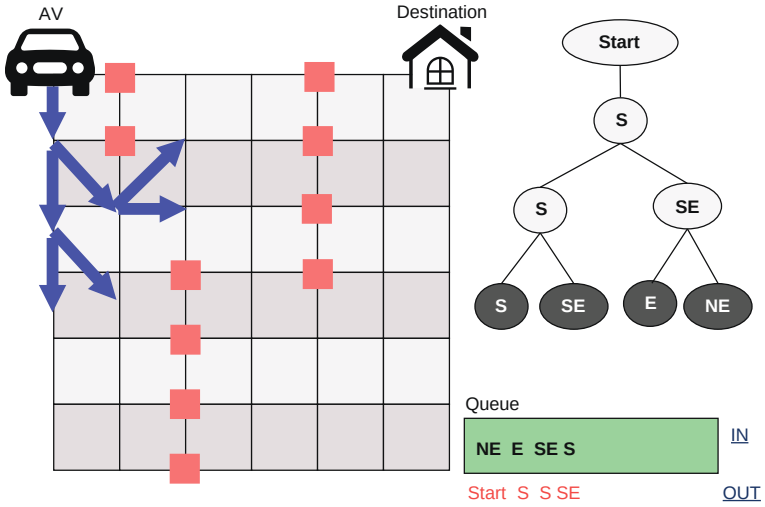


Figure 3.17 Solving the path-planning problem using BFS—step 5

6 The FIFO queue continues until the destination node is found (figure 3.18). For simplicity, assuming that the robot wants to reach node E shown in figure 3.18, we can trace back up the tree to find the path from the source node to the goal, which will be Start-S-SE-E.

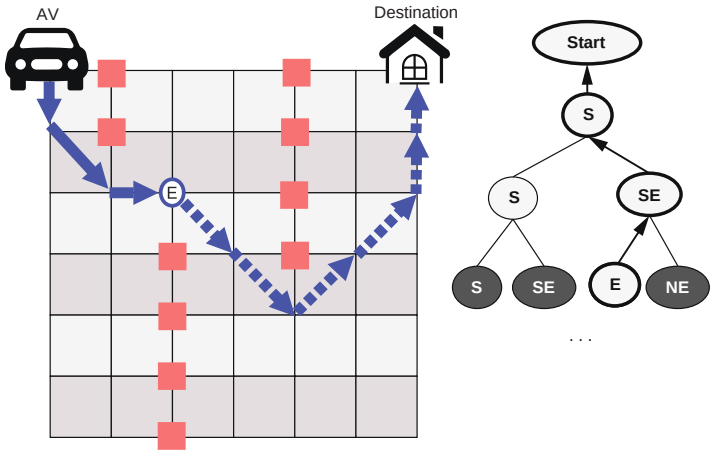


Figure 3.18 Solving the path-planning problem using BFS—final routes for an intermediate goal node E and the final destination

In BFS, every node generated must remain in memory. The number of nodes generated is at most $O(b^d)$, where b represents the maximum branching factor for each node (i.e., the number of children the node has) and d is the depth one must expand to reach the goal. In the previous example, with E as a goal node (b=2, d=3), the total number of traversed nodes is 2^3=8, including the start node.

Aside from the algorithm's ability to solve the problem at hand, algorithm efficiency is evaluated based on run time (time complexity), memory requirements, and the number of primitive operations required to solve the problem in the worst case. Examples of these primitive operations include, but are not limited to, expression evaluation, variable value assignment, array indexing, and method or function calls.

Big *O* notation

Big *O* notation describes the performance or complexity of an algorithm, usually under the worst-case scenario. Big *O* notation helps us answer the question, "Will the algorithm scale?"

To obtain the big *O* notation for a function f(*x*), if f(*x*) is a sum of several terms, the one with the largest growth rate is kept, and all others are omitted. Moreover, if f(*x*) is a product of several factors, any constants (terms in the product that do not depend on *x*) are omitted.

As an example, let's look at the ticket pricing problem presented in chapter 1: $f(x) = -20x^2 + 6200x - 350000$. Assume that x is a vector with size n that represents n different ticket prices. This function is the sum of three terms, of which the one with the highest growth rate is the one with the largest exponent as a function of x, namely $-20x^2$. We can now apply the second rule: $-20x^2$ is a product of -20 and x^2, in which the first factor does not depend on x. Dropping this factor results in the simplified form x^2. Thus, we say that $f(x)$ is a big O of n^2, where n is the size of the decision variable x. Mathematically we can write $f(x) \in O(n^2)$ (pronounced "order n squared" or "O of n squared"), which represents a quadratic complexity (i.e., the growth rate is proportional to the square of the size of the ticket price vector).

Table 3.2 shows examples of algorithm complexities, and figure 3.19 shows examples of big *O* notations.

Table 3.2 Algorithm complexity

Notation	Name	Effectiveness	Description	Examples
$O(1)$	Constant	Excellent	Running time does not depend on the input size. As the input size grows, the number of operations is not affected.	Variable declaration Accessing an array element Retrieving information from a hash-table lookup Inserting and removing from a queue Pushing and popping on a stack
$O(\log n)$	Logarithmic	High	As the input size grows, the number of operations grows very slowly. Whenever n doubles or triples, etc., the running time increases by a constant.	Binary search
$O(n^c)$, $0 < c < 1$	Fractional power or sublinear	High	As the input size grows, the number of operations is replicated in multiplication.	Testing graph connectedness Approximating the number of connected components in a graph Approximating the weight of the minimum spanning tree (MST)
$O(n)$	Linear	Medium	As the input size grows, the number of operations increases linearly. Whenever n doubles, the running time doubles.	Printing out an array's elements Simple search Kadane's algorithm
$O(n \log n) = O(\log n!)$	Linearithmic, loglinear, or quasilinear	Medium	As the input size grows, the number of operations increases slightly faster than linear.	Merge sort Heapsort Timsort
$O(n^c)$, $c > 1$	Polynomial or algebraic	Low	As the input size grows, the number of operations increases as the exponent increases.	Minimum spanning tree (MST) Matrix determinant
$O(n^2)$	Quadratic	Low	Whenever n doubles, the running time increases four-fold. The quadratic function is practical for use only on small problems.	Selection sort Bubble sort Insertion sort

Table 3.2 Algorithm complexity (*continued*)

Notation	Name	Effectiveness	Description	Examples
$O(n^3)$	Cubic	Low	Whenever n doubles, the running time increases eightfold. The cubic function is practical for use only on small problems.	Matrix multiplication
$O(c^n)$, $c > 1$	Exponential	Very low	As the input size grows, the number of operations increases exponentially. It is slow and usually not appropriate for practical use.	Power set Tower of Hanoi Password cracking Brute force search
$O(n!)$	Factorial	Extremely low	Extremely slow, as all possible permutations of the input data need to be checked. The factorial algorithm is even worse than the exponential function.	Traveling salesman problem Permutations of a string

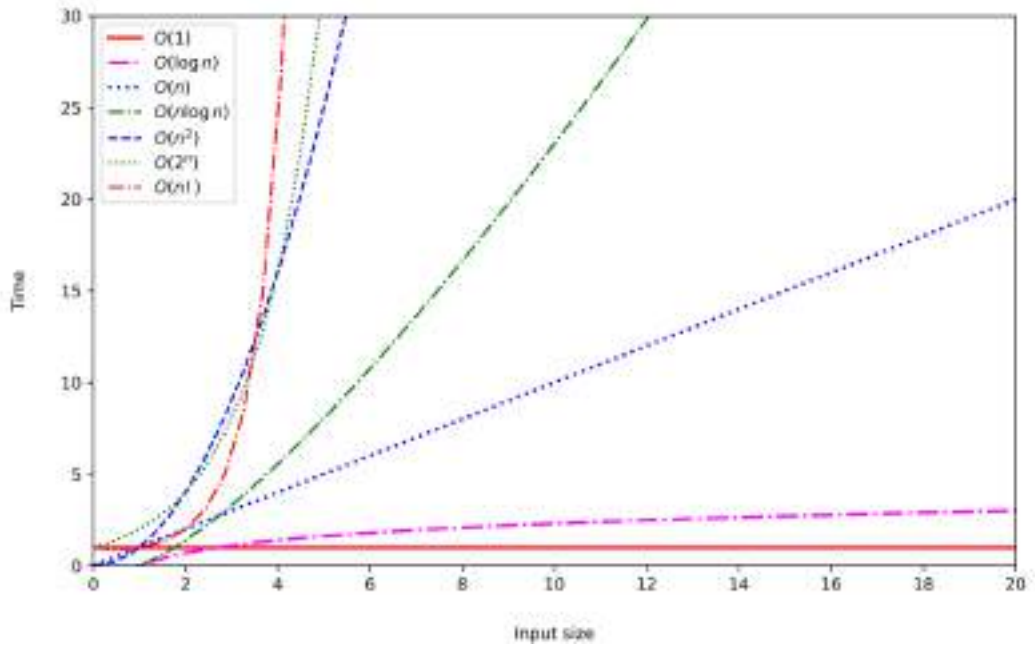


Figure 3.19 Examples of big *O* notations

Assume a computer with a processor speed of one million operations per second is used to handle a problem of size $n = 20{,}000$. Table 3.3 shows the running time according to the big *O* notation of the algorithm used to solve this problem.

Table 3.3 Algorithm complexity and the running time

Big *O*	Running time
$O(1)$	10^{-6} seconds
$O(\log n)$	14×10^{-6} seconds
$O(n)$	0.02 seconds
$O(n \log n) = O(\log n!)$	0.028 seconds
$O(n^2)$	6.66 minutes
$O(n^3)$	92.6 days
$O(c^n), c = 2$	$1262.137 \times 10^{6015}$ years
$O(n!)$	$5768.665 \times 10^{77331}$ years (this is many orders of magnitude larger than the age of the universe, which is around 13.7 billion years)

For a huge workspace where the goal is deep, the number of nodes could expand exponentially and demand a large memory requirement. In terms of time complexity, for a graph G = (V, E), BFS has a running time of $O(|V| + |E|)$, since each vertex is enqueued at most once and each edge is checked either once (for a directed graph) or at most twice (for an undirected graph). The time and space complexity of BFS is also defined in terms of a branching factor b and the depth of the shallowest goal d. Time complexity is $O(b^d)$, and space complexity is also $O(b^d)$.

Let's consider a graph with a constant branching factor $b = 5$, nodes of size 1 KB, and a limit of 1,000 nodes scanned per second. The total number of nodes N is given by the following equation:

$$N = \frac{b(d+1) - 1}{b - 1} \tag{3.1}$$

Table 3.4 shows the time and memory requirements to traverse this graph using BFS.

Table 3.4 BFS time and space complexity

Depth *d*	Nodes *N*	Time	Memory
2	31	31 ms	31 KB
4	781	0.781 second	0.78 MB
6	19,531	5.43 hours	19.5 MB
8	488,281	56.5 days	488 MB
10	12,207,031	3.87 years	12.2 GB
12	305,175,781	96.77 years	305 GB
14	7,629,394,531	2,419.26 years	7.63 TB

Next, we'll take a look at the counterpart to the BFS algorithm, which searches deep into a graph first, rather than breadth-wise.

3.3.2 Depth-first search

Depth-first search (DFS) is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by first going as deep as possible into the graph. Then, when it reaches the last layer with no result (i.e., when a dead end is reached), it backtracks up a layer and continues the search. In DFS, the deepest nodes are expanded first, and nodes of equal depth are ordered arbitrarily. Algorithm 3.2 shows the DFS steps.

Algorithm 3.2 Depth-first search (DFS)

```
Inputs: Source node, Destination node
Output: Route from source to destination

Initialize Stack ← a LIFO initialized with sourcenode
Initialize Explored ← empty
Initialize Found ← False

While stack is not empty and found is False do
    Set node ← stack.pop()
    Add node to explored
    For child in node.expand() do
        If child is not in explored and child is not in stack then
            If child is destination then
               Update route ← child.route()
               Update found ← True
            Add child to stack
Return route
```

As you may have noticed, the only difference between DFS and BFS is in how the data structure works. Rather than working down layer by layer (FIFO), DFS drills down to the bottommost layer and moves its way back to the starting node, using a last in, first out (LIFO) data structure known as a *stack*. The stack contains the list of discovered nodes. The most recently discovered node is pushed onto the top of the LIFO stack. Subsequently, the next node to be expanded is popped from the top of the stack, and all of its successors are then added to the stack.

Figure 3.20 shows the DFS solution for the 8-puzzle problem we looked at before, based on moving the blank tile. As you can see, when the algorithm reaches a dead end or terminal node (such as node 7), it goes back to the last decision point (node 3) and proceeds with another alternative (node 8 and so on). In this example, a depth bound of 5 is placed to constrain the node expansion. This depth bound makes nodes 6, 7, 10, 11, 13, 14, 16, 17, 22, 23, 26, and 27 terminal nodes in the search tree (i.e., they have no successors).

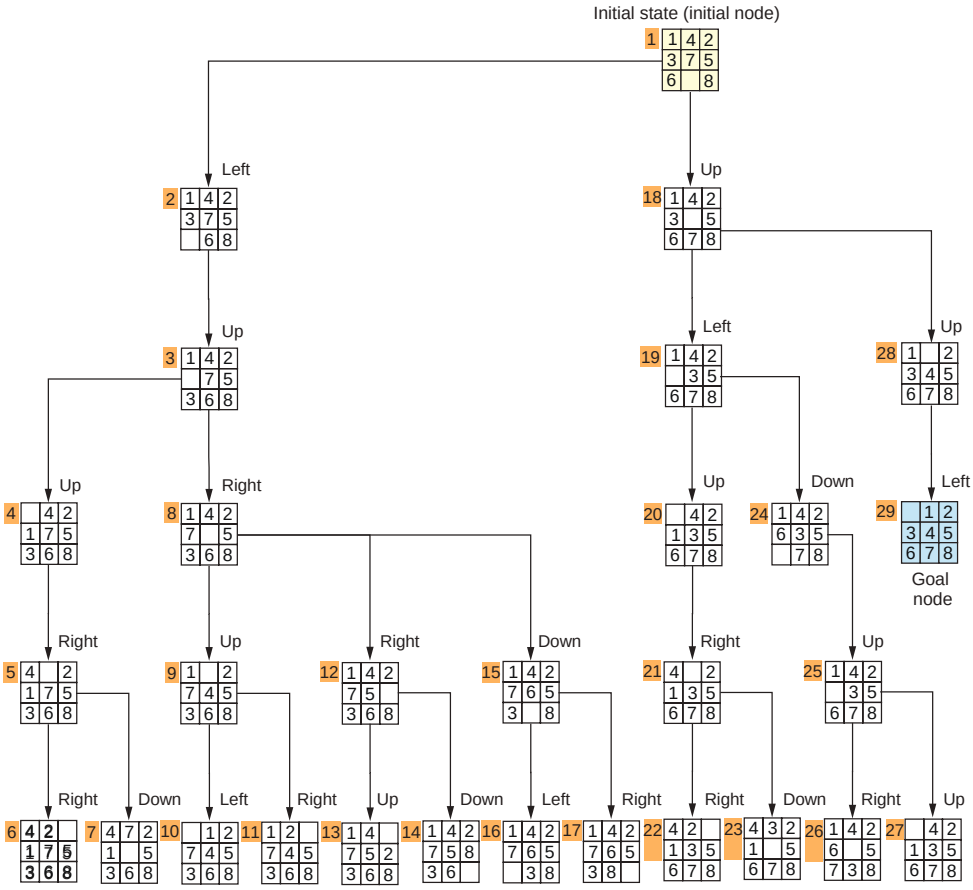


Figure 3.20 Using DFS to solve the 8-puzzle problem

As you can see in listing 3.10, we only need to change the algorithm in the code to use DFS. I've also omitted the solution visualization, the reason for which you'll see shortly. The `State` class is defined in the complete listing available in the book's GitHub repo.

Listing 3.10 Solving the 8-puzzle problem using DFS

```
from optalgotools.algorithms.graph_search import DFS

init_state = [[1,4,2],[3,7,5],[6,0,8]]
goal_state = [[0,1,2],[3,4,5],[6,7,8]]

init_state = State(init_state)
goal_state = State(goal_state)
```

```
if not init_state.is_solvable():
     print("This puzzle is not solvable.")
else:
     solution = DFS(init_state, goal_state)
     print(f"Process time: {solution.time} s")
     print(f"Space required: {solution.space} bytes")
     print(f"Explored states: {solution.explored}")
     print(f"Number of steps: {len(solution.result)}")
```

Some puzzles are not solvable.

The inputs for DFS are the same as for BFS.

Here's the output of this code run with the preceding inputs:

```
Process time: 0.5247 s
Space required: 624 bytes
Explored states: 29
Number of steps: 30
```

As you can see, DFS is not great when dealing with very deep graphs, where the solution may be located closer to the top. You can also see why I opted not to visualize the final solution: there are a lot more steps in the solution than we had in BFS! Because the solution to this problem is closer to the root node, the solution generated by DFS is a lot more convoluted (30 steps) than with BFS.

Revisiting the path-planning problem, DFS can be used to generate an obstacle-free path from the start location to the destination as follows:

1. Add the source node to the stack (figure 3.21).

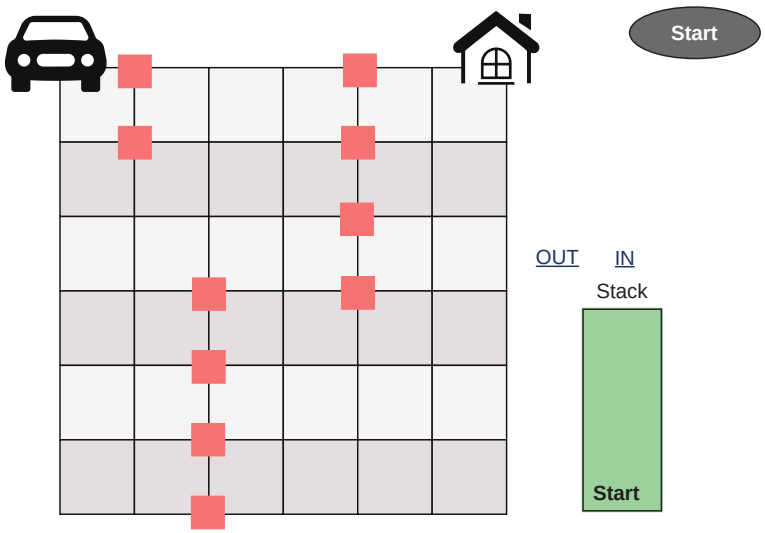


Figure 3.21 Solving the path-planning problem using DFS—step 1

2. Explore the S node, as the E and SE nodes are obstructed (figure 3.22).

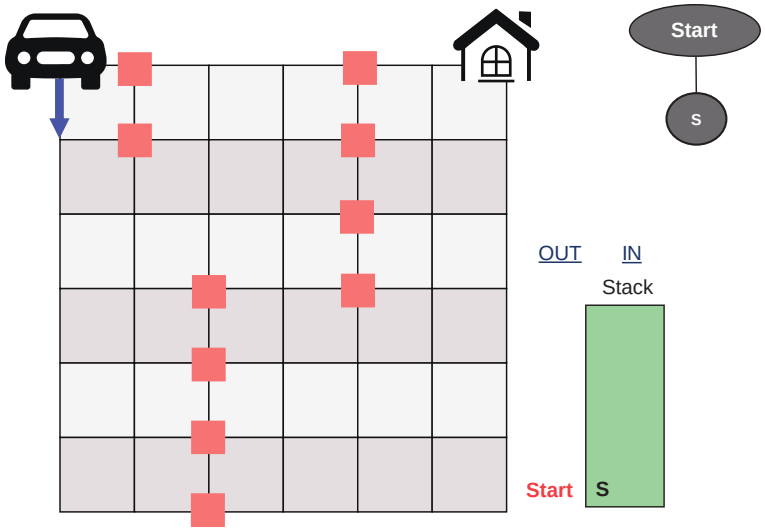


Figure 3.22 Solving the path-planning problem using DFS—step 2

3 Take S out (LIFO), and explore its neighboring nodes, S and SE, as E is an obstructed node (figure 3.23).

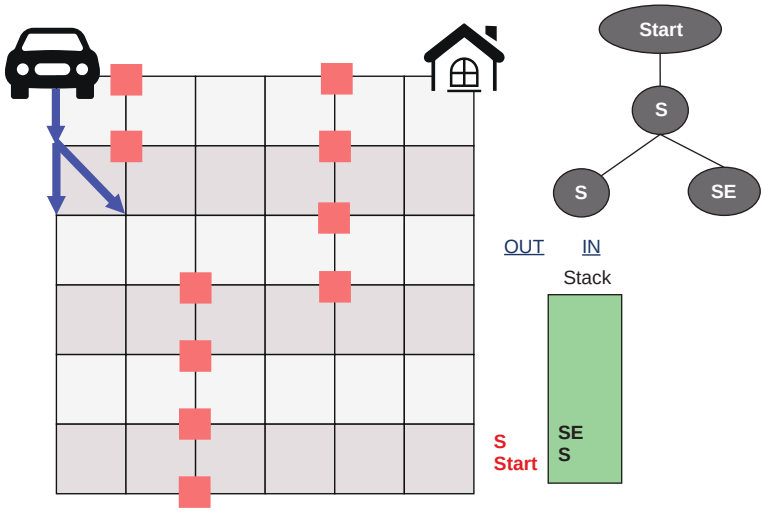


Figure 3.23 Solving the path-planning problem using DFS—step 3

4 Take SE out (LIFO), and explore its neighboring nodes, SW, S, E, and NE (figure 3.24).

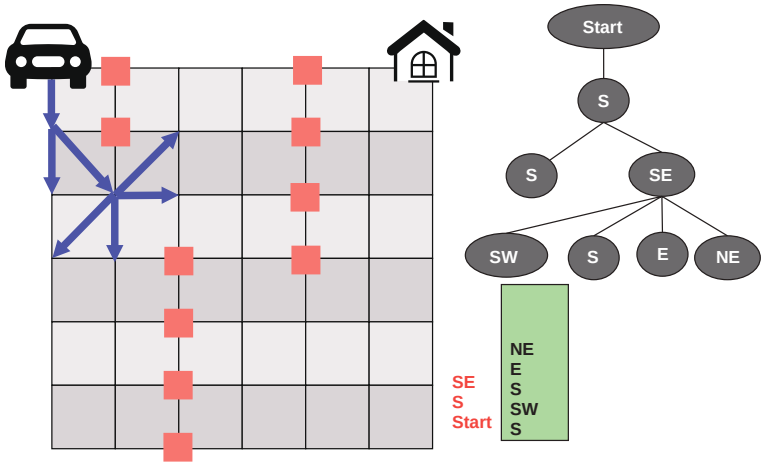


Figure 3.24 Solving the path-planning problem using DFS—step 4

5 The next node to be expanded would be NE, and its successors would be added to the stack. The LIFO stack continues until the goal node is found. Once the goal is found, you can then trace back through the tree to obtain the path for the vehicle to follow (figure 3.25).

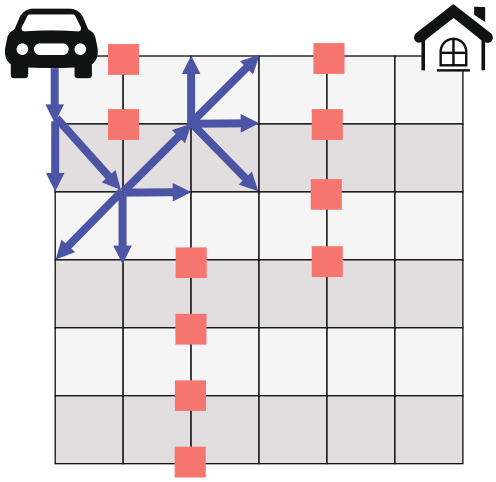


Figure 3.25 Solving the path-planning problem using DFS—step 5

DFS usually requires considerably less memory than BFS. This is mainly because DFS does not always expand every single node at each depth. However, DFS could continue down an unbounded branch forever in the case of a search tree with infinite depth, even if the goal is not located on that branch.

One way to handle this problem is to use *constrained depth-first search*, where the search stops after reaching a certain depth. Time complexity of DFS is $O(b^d)$ where b is the branching factor and d is the maximum depth of the search tree. This is terrible if d is much larger than b, but if solutions are found deep in the tree, it may be much faster than BFS. The space complexity of DFS is $O(bd)$, which is linear space! This space complexity represents the maximum number of nodes to be stored in memory.

Table 3.5 summarizes the differences between BFS and DFS.

Table 3.5 BFS versus DFS

	Breadth-first search (BFS)	Depth-first search (DFS)
Space complexity	More expensive	Less expensive. Requires only $O(d)$ space, irrespective of the number of children per node.
Time complexity	More time efficient. A vertex at a lower level (closer to the root) is visited first before visiting a vertex that is at a higher level (far away from the root).	Less time efficient
When it is preferred	• If the tree is very deep • If the branching factor is not excessive • If the solution appears at a relatively shallow level (i.e., the solution is near the starting point in the tree) • Example: Search the British royal family tree for someone who died a long time ago, as they would be closer to the top of the tree (e.g., King George VI).	• If the graph or tree is very wide with too many adjacent nodes • If no path is excessively deep • If solutions occur deeply in the tree (i.e., the target is far from the source) • Example: Search the British royal family tree for someone who is still alive, as they would be near the bottom of the tree (e.g., Prince William).

In applications where the weights of the edges in a graph are all equal (e.g., all length 1), the BFS and DFS algorithms outperform shortest path algorithms like Dijkstra's in terms of time. Shortest path algorithms will be explained in the next section.

3.4 Shortest path algorithms

Suppose that you were looking for the quickest way to go from your home to work. Graph traversal algorithms like BFS and DFS may eventually get you to your destination, but they certainly do not optimize for the distance traveled. We'll discuss Dijkstra's algorithm, uniform-cost search (UCS), and bidirectional Dijkstra's search as examples of blind search algorithms that try to find the shortest path between a source node and a destination node.

3.4.1 Dijkstra's search

Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a fully connected graph with non-negative edge path costs, producing a shortest-path tree. Dijkstra's algorithm was published in 1959, and it's named after Dutch computer scientist Edsger Dijkstra. This algorithm is the base of several other graph search algorithms that are commonly used to solve routing problems in popular navigation apps, as illustrated in figure 3.26. The algorithm follows dynamic programming approaches where the problem is recursively divided into simple subproblems. Dijkstra's algorithm is uninformed, meaning it does not need to know the target node beforehand and doesn't use heuristic information.

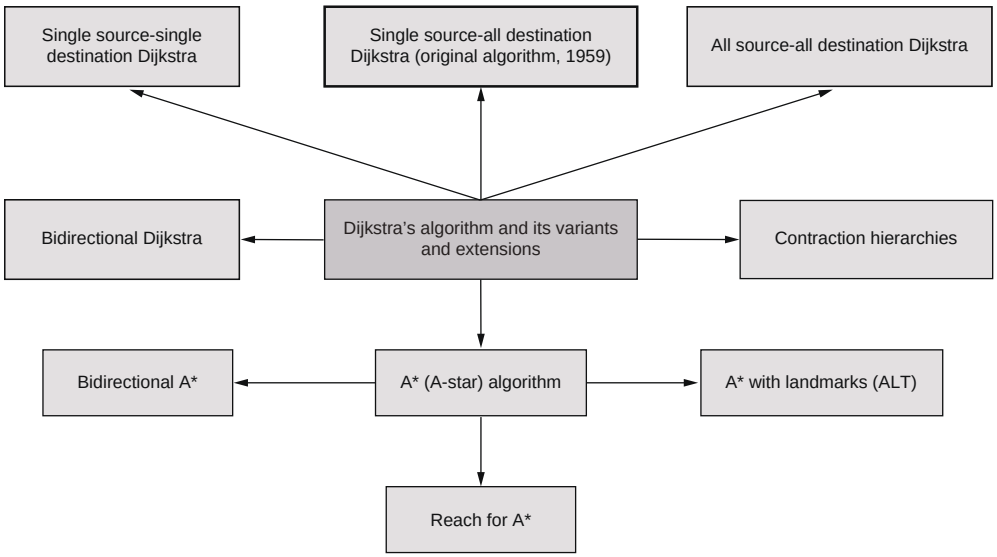


Figure 3.26 Dijkstra's algorithm and examples of its variants and extensions

Algorithm 3.3 shows the steps of the original version of Dijkstra's algorithm for finding the shortest path between a known single source node to all other nodes in the graph or tree.

Algorithm 3.3 Dijkstra's algorithm

```
Inputs: A graph with weighted edges and a source node
Output: Shortest path from the source to all other nodes in the graph

Initialize shortest_dist ← empty
Initialize unrelaxed_nodes ← empty
Initialize seen ← empty
```

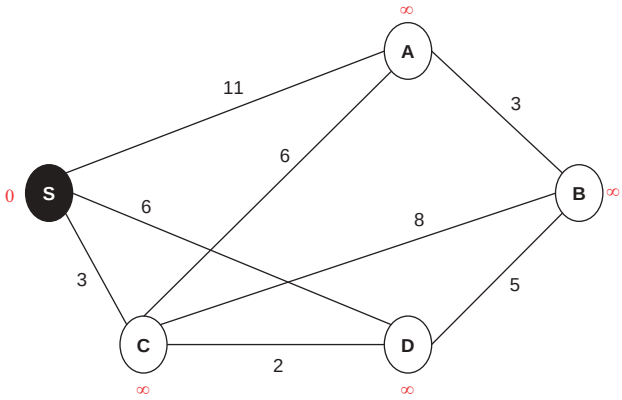
```
For node in graph
    Set shortest_dist[node] = Infinity
    Add node to unrelaxed_nodes
    Set shortest_dist[source] ← 0

While unrelaxed_nodes is not empty do
    Set node ← unrelaxed_nodes.pop()
    Add node to seen
    For child in node.expand() do
        If child in seen then skip
        Update distance ← shortest_dist[node] + length of edge to child
        If distance < shortest_dist[child] then
           Update shortest_dist[child] ← distance
           Update child.parent ← node
Return shortest_dist
```

Dijkstra's algorithm and its variants presented in the code for this book are all modified to require a target node. This improves the processing time when working with large graphs (e.g., road networks).

Let's look at how Dijkstra's algorithm finds the shortest path between any two nodes in a graph. The priority queue is used to pop the element of the queue with the highest priority according to some ordering function (in this case, the shortest distance between the node and the source node).

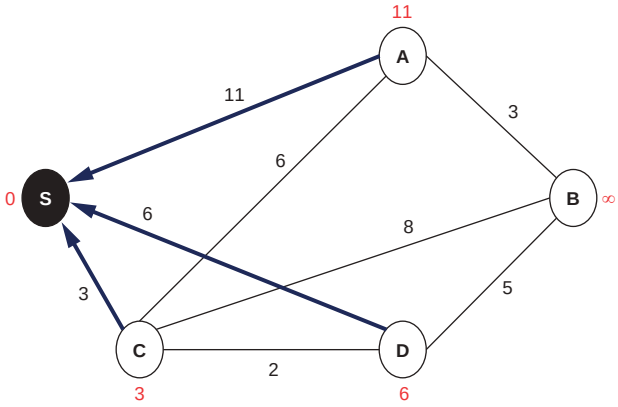
0 Initial list, no predecessors: priority queue = {} (figure 3.27).



From S to	Distance	Predecessor
S	0	-
A	∞	-
B	∞	-
C	∞	-
D	∞	-

Figure 3.27 Finding the shortest path using Dijkstra's algorithm—step 0

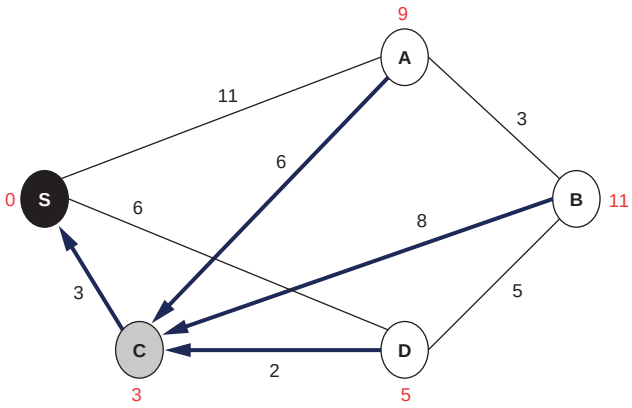
1 The closest node to the source node is S, so add it to the priority queue. Update the cumulative distances (i.e., distances from the source node *S* to get to the node) and predecessors for A, C, and D. Priority queue = {S} (figure 3.28).



From S to	Distance	Predecessor
S	0	-
A	11	S
B	∞	-
C	3	S
D	6	S

Figure 3.28 Finding the shortest path using Dijkstra's algorithm—step 1

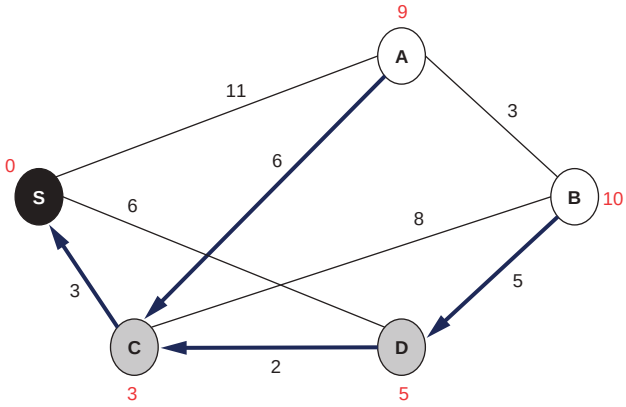
2 The next closest node is C, so add it to the priority queue. Update the distances and predecessors for A and D. Priority queue = {S, C} (figure 3.29).



From S to	Distance	Predecessor
S	0	-
A	~~11~~ 9	C
B	~~∞~~ 11	C
C	3	S
D	~~6~~ 5	C

Figure 3.29 Finding the shortest path using Dijkstra's algorithm—step 2

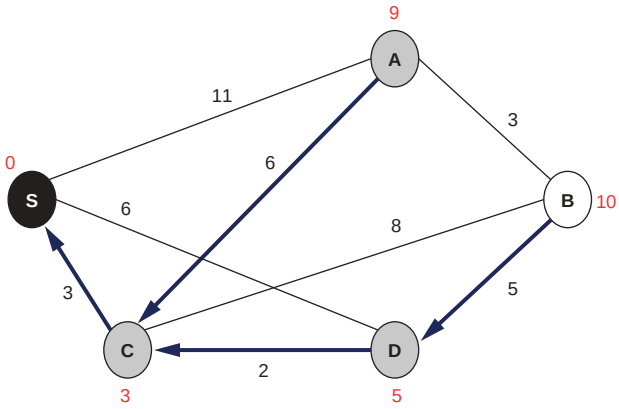
3 The next closest node is D, so add it to the priority queue. Update the distances and predecessor for B. Priority queue = {S, C, D} (figure 3.30).



From S to	Distance	Predecessor
S	0	-
A	9	C
B	~~11~~ 10	D
C	3	S
D	5	C

Figure 3.30 Finding the shortest path using Dijkstra's algorithm—step 3

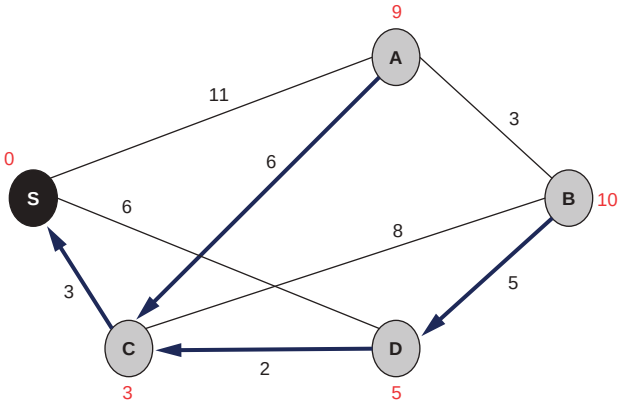
4 The next closest node to the source node is A, so add it to the priority queue. Priority queue = {S, C, D, A} (figure 3.31).



From S to	Distance	Predecessor
S	0	-
A	9	C
B	10	D
C	3	S
D	5	C

Figure 3.31 Finding the shortest path using Dijkstra's algorithm—step 4

5 The next step is to add the remaining node B to complete the search (figure 3.32). Priority queue = {S, C, D, A, B}. All nodes are now added.



From S to	Distance	Predecessor
S	0	-
A	9	C
B	10	D
C	3	S
D	5	C

Figure 3.32 Finding the shortest path using Dijkstra's algorithm—step 5

Once the search is complete, you can choose your goal node and find the shortest path from the table. For example, if the goal node is A, the shortest path between S and A is S-C-A with length 9. Likewise, if the goal node is B, the shortest path between S and B is S-C-D-B with a distance of 10.

Note that we can't use Dijkstra's search on our 8-puzzle problem as Dijkstra's search requires knowledge of the entire problem space beforehand. While the problem has a finite number of possible states (exactly 9!/2), the scale of that solution space makes the Dijkstra's search not very feasible.

3.4.2 Uniform-cost search (UCS)

The uniform-cost search (UCS) algorithm is a blind search algorithm that uses the lowest cumulative cost to find a path from the origin to the destination. Essentially, the algorithm organizes nodes to be explored either by their cost (with the lowest cost as the highest priority) for minimization problems, or by their utility (with the highest utility as the highest priority) in the case of maximization problems.

As nodes are popped from the queue, we add the node's children to the queue. If a child already exists in the priority queue, the priorities of both copies of the child are compared, and the lowest cost (the highest priority) in a minimization problem is accepted. This ensures that the path to each child is the shortest one available. We also maintain a visited list so we can avoid revisiting nodes that have already been popped from the queue. UCS behaves like BFS when all the edge costs in the graph are equal or identical. In this case, UCS will expand nodes in the same order as BFS—level by level or breadth-first. Algorithm 3.4 shows the steps of the UCS algorithm.

Algorithm 3.4 Uniform-cost search (UCS)

```
Inputs: A graph with edges, a source node, a destination node
Output: Shortest path from source to destination in the graph
```

```
Initialize priority_queue ← source
Initialize found ← False
Initialize seen ← source

While priority_queue is not empty and found is False do
    Set node ← priority_queue.pop()
    Update seen ← node
    Update node_cost ← cumulative distance from source
    If node is destination then
        Update route ← node.route()
        Update found ← True
    For child in node.expand() do
        If child in priority_queue then
            If child.priority < priority_queue[child].priority then
               Set priority_queue[child].priority = child.priority
        Else
            Update priority_queue ← child
        Update priority_queue[child].priority ← node_cost
Return route
```

UCS is a variant of Dijkstra's algorithm that is useful for large graphs because it is less time-consuming and has fewer space requirements. Whereas Dijkstra's adds all nodes to the queue at the start with an infinite cost, UCS fills the priority queue gradually. For example, consider the problem of finding the shortest path between every node pair in a graph. As a graph's size and complexity grows, it quickly becomes apparent that UCS is more efficient, as it does not require knowing the entire graph beforehand. Table 3.6 shows the difference in processing time between Dijkstra's and UCS on graphs of different sizes. These numbers were collected using the code in Comparison.ipynb, available in the book's GitHub repo, on an Intel Core i9-9900K at 3.60 GHz without multiprocessing or multithreading.

Table 3.6 UCS versus Dijkstra's

Graph size = |V| + |E|	Dijkstra time	Uniform-Cost Search (UCS) time
108	0.25 s	0.14 s
628	84.61 s	58.23 s
1,514	2,082.97 s	1,360.98 s

Note that running UCS on our 8-puzzle problem requires a distance property for each state (this defaults to 1), and it generates decent results overall (around 6.2 KB of space used and 789 states explored). It is important to note that because the edge lengths are all equal, UCS cannot prioritize new nodes to explore. Thus, the solution loses the advantage of shortest path algorithms, namely, the ability to optimize for a more compact solution. In the next chapter, you'll see ways of calculating artificial distances between these states, ultimately generating solutions quickly and minimizing the number of steps required.

3.4.3 Bidirectional Dijkstra's search

Bidirectional search simultaneously applies forward search and backward search. As illustrated in figure 3.33, it runs a search forward from the initial source state S→G and backward from the final goal state G→S until they meet.

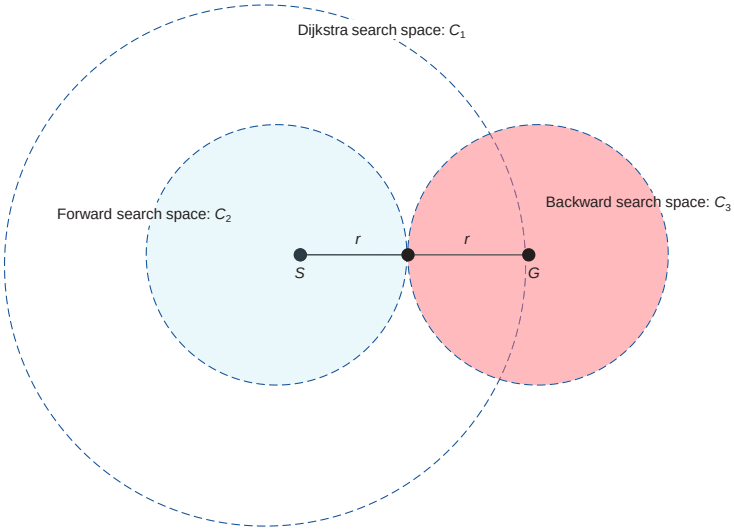


Figure 3.33 Bidirectional Dijkstra

As shown in figure 3.33, the Dijkstra's search space is $C_1 = 4\pi r^2$, and the bidirectional Dijkstra's search space is represented by $C_2 + C_3 = 2\pi r^2$. This means that we reduce the search space by about a factor of two. The following algorithm shows the steps of the bidirectional Dijkstra's algorithm.

Algorithm 3.5 Bidirectional Dijkstra's

```
Inputs: A graph, a source node, a destination node
Output: Shortest path from source to destination in the graph

Initialize frontier_f ← initialized with source
Initialize frontier_b ← initialized with destination
Initialize explored_f ← empty
Initialize explored_b ← empty
Initialize found ← False
Initialize collide ← False
Initialize altr_expand ← False

While frontier_f is not empty and frontier_b is not empty and not collide and
not found do
    If altr_expand then
        Set node ← frontier_f.pop()
        Add node to explored_f
```

```
        For child in node.expand() do
            If child in explored_f then continue
            If child is destination then
                Update route ← child.route()
                Update found ← True
            If child in explored_b then
                Update route ← child.route() + reverse(overlapped.route())
                Update collide ← True
            Add child to frontier_f
        Update altr_expand ← not altr_expand
    Else
        Update node ← frontier_b.pop()
        Add node to explored_b
        For child in node.expand() do
            If child in explored_b then continue
            If child is origin then
               Update route ← child.route()
               Update found ← True
            If child in explored_f then
               Update route ← reverse(child.route()) + overlapped.route()
               Update collide ← True
            Add child to frontier_b
        Update altr_expand ← not altr_expand
Return route
```

This approach is more efficient because of the time complexities involved. For example, a BFS search with a constant branching factor b and depth d would have an overall $O(b^d)$ space complexity. However, by running two BFS searches in opposite directions with only half the depth $(d/2)$, the space complexity becomes $O(b^{d/2} + b^{d/2})$ or simply $O(b^{d/2})$, which is significantly lower.

Figure 3.34 shows the difference between the Dijkstra's and bidirectional Dijkstra's algorithms in exploring 50,841 nodes in the City of Toronto.

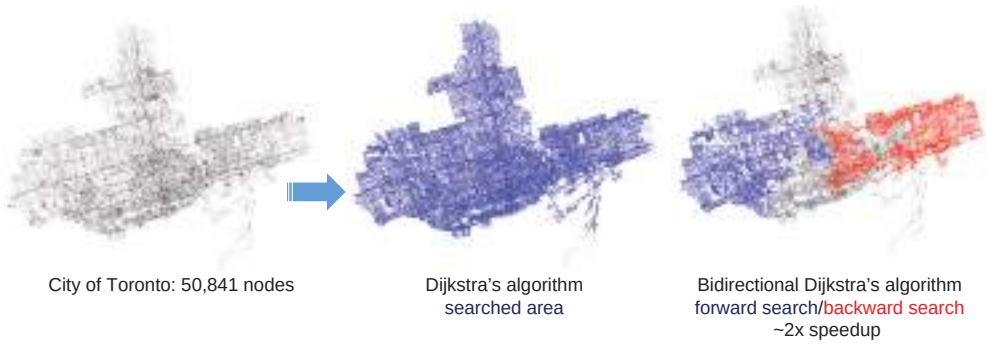


Figure 3.34 Dijkstra's vs. bidirectional Dijkstra's—forward exploration from the left and backward exploration from the right

3.5 Applying blind search to the routing problem

Puzzle games and simple grid routing problems are nice for understanding how an algorithm works. However, it's time we look at some real-world examples and outcomes of using these algorithms. For example, imagine that you are visiting the King Edward VII equestrian statue at Queen's Park in Toronto when you suddenly remember you have a meeting at the Bahen Centre for Information Technology at the University of Toronto. I initially presented this problem when we first discussed road network graphs at the beginning of this chapter. There are a couple of assumptions we'll make when considering this problem:

- You aren't able to open a navigation app or call a friend for help, as your phone is out of battery power.
- You know your destination is somewhere in Toronto, but you have no clue where it is with reference to your starting location. (In later chapters, you'll learn how knowing your destination's direction can help generate near-optimal solutions in a very short amount of time.)
- Once you start using a rule for routing to your destination, you'll stick to that rule.

Let's look at how we might be able to simulate our pathfinding skills using BFS, DFS, Dijkstra's, UCS, and bidirectional Dijkstra's. The code for this example is located in the book's GitHub repo (Comparison.ipynb). Figures 3.35 to 3.37 show the routes generated by these blind search algorithms.

Figure 3.35 Shortest path generated using BFS. BFS searches each layer first before moving to the next. This works best for graphs that are not very broad and that have a solution near the root node.

Figure 3.36 Shortest path generated using DFS. DFS searches as deep in the graph as possible before backtracking. This works best when the graph is not very deep and solutions are located further away from the root node.

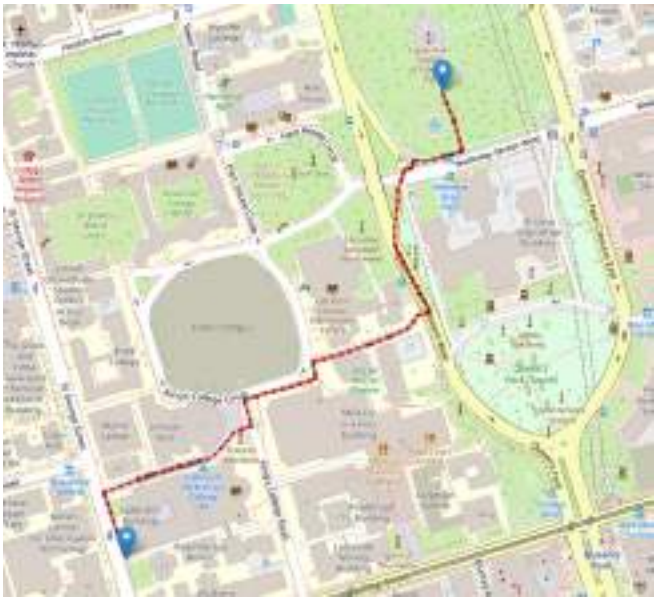


Figure 3.37 Shortest path generated using Dijkstra's, UCS, and bidirectional Dijkstra's. All three of these algorithms will produce the same solution (the optimal routing) but will handle memory use and node exploration differently.

It is worth noting that the `dijkstra_path` function in NetworkX uses Dijkstra's method to compute the shortest weighted path between two nodes in a graph. Our optalgotools package also provides an implementation for different graph search algorithms such as BFS, DFS, Dijkstra's, UCS, and bidirectional Dijkstra's. The implementation of Dijkstra's algorithm in optalgotools has been modified to work with our OSM data because graphs generated from maps will naturally have self-loops and parallel edges. Parallel edges may result in a route that is not the shortest available, as the route length depends heavily on which parallel edge was chosen when a particular path was generated. In figure 3.38, the shortest path from 0 to 2 may be returned as having a length of 7 if the top edge connecting 0 and 1 is chosen when calculating that path, versus a length of 3 when selecting the bottom edge.

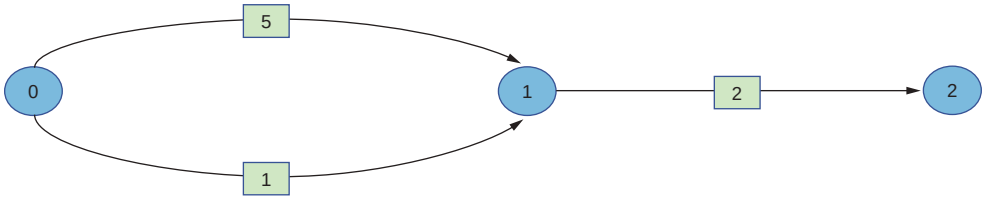


Figure 3.38 Parallel edges may be problematic because finding the shortest path depends on which parallel edge is selected during graph exploration.

Self-loops also cause trouble for the original Dijkstra's algorithm. If a graph contains a self-loop, the shortest path to a node might come from itself. At that point, we would be unable to generate a route (figure 3.39).

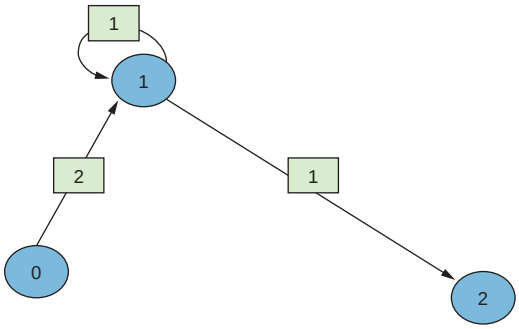


Figure 3.39 Self-loops may disrupt the chain of parent-child nodes, which prevents us from retracing the route after a solution has been found.

These two problems are generally easy but nontrivial to avoid. For parallel edges, we select the edge with the lowest weight (shortest length) and discard any other parallel edge. With self-loops, we can ignore the loop entirely, as negative-weight loops

do not exist in most routing problems (a road cannot have a negative length), and positive-weight loops cannot be part of the shortest path. Additionally, the version of Dijkstra's algorithm used in this book terminates upon finding the target node, as opposed to the traditional implementation, which ends only when the shortest path from the root node to all other nodes is found.

Table 3.7 compares BFS, DFS, Dijkstra's, and UCS with regards to path length, process time, space required, and the number of explored nodes. As you can see from these results, Dijkstra's, UCS, and the bidirectional Dijkstra's algorithms produce optimal results, with varying degrees of time and space cost. While both BFS and DFS find feasible solutions in the shortest time, the solutions delivered are not optimal and, in the case of DFS, are not even plausible. On the other hand, DFS requires knowing the entire graph beforehand, which is costly and sometimes not very practical. Much of selecting an appropriate search algorithm for a specific problem involves determining the ideal balance between processing time and space requirements. In later chapters, we'll look at algorithms that produce near-optimal solutions and that are often used when optimal solutions are either impossible or impractical to find. Note that all these solutions are feasible; they all produce a valid (if sometimes convoluted) path from point A to point B.

Table 3.7 Comparing BFS, DFS, Dijkstra's, and UCS, where *b* is the branching factor, *m* is the maximum depth of the search tree, *d* is the shallowest graph depth, *E* is the number of edges, and *V* is the number of vertices.

Algorithm	Cost (meters)	Process time (s)	Space (bytes)	Explored nodes	Worst-case time	Worst-case space	Optimality
BFS	955.962	0.015625	1,152	278	$O(b^d)$	$O(b^d)$	No
DFS	3347.482	0.015625	1,152	153	$O(b^m)$	$O(bm)$	No
Dijkstra's	806.892	0.0625	3,752	393	$O(\|E\| + \|V\| \log \|V\|)$	$O(\|V\|)$	Yes
UCS		0.03125	592	393	$O((b + \|E\|) * d)$	$O(b^d)$	Yes
Bidirectional Dijkstra's		0.046875	3,752	282	$O(b^{d/2})$	$O(b^{d/2})$	Yes

In the next chapter, we will look at how search can be optimized if we utilize domain-specific knowledge instead of searching blindly. We'll dive right into informed search methods and see how we can use these algorithms to solve minimum spanning tree and shortest path problems.

Summary

- Conventional graph search algorithms (blind and informed search algorithms) are deterministic search algorithms that explore a graph either for general discovery or for explicit search.
- A graph is a nonlinear data structure consisting of vertices and edges.
- Blind (uninformed) search is a search approach where no information about the search space is used.
- Breadth-first search (BFS) is a graph traversal algorithm that examines all the nodes in a search tree on one level before considering any of the nodes on the next level.
- Depth-first search (DFS) is a graph traversal algorithm that starts at the root or an initial node or vertex, follows one branch as far as possible, and then backtracks to explore other branches until a solution is found or all paths are exhausted.
- Depth-limited search (DLS) is a constrained version of DFS with a predetermined depth limit, preventing it from exploring paths beyond a certain depth.
- Iterative deepening search (IDS), or iterative deepening depth-first search (IDDFS), combines DFS's space efficiency and BFS's fast search by incrementing the depth limit until the goal is reached.
- Dijkstra's algorithm solves the single-source shortest path problem for a weighted graph with non-negative edge costs.
- Uniform-cost search (UCS) is a variant of Dijkstra's algorithm that uses the lowest cumulative cost to find a path from the source to the destination. It is equivalent to the BFS algorithm if the path costs of all edges are the same.
- Bidirectional search (BS) is a combination of forward and backward search. It searches forward from the start and backward from the goal simultaneously.
- Selecting a search algorithm involves determining the target balance between time complexity, space complexity, and prior knowledge of the search space, among other factors.