4 Informed search algorithms

This chapter covers

- Defining informed search
- Learning how to solve the minimum spanning tree problem
- Learning how to find the shortest path using informed search algorithms
- Solving a real-world routing problem using these algorithms

In the previous chapter, we covered blind search algorithms, which are algorithms in which no information about the search space is needed. In this chapter, we'll look at how search can be further optimized if we utilize some information about the search space during the search.

As problems and search spaces become larger and more complex, the complexity of the algorithms themselves increases. I'll start by introducing informed search algorithms, and then we'll discuss minimum spanning tree algorithms and shortest path search algorithms. A routing problem will be presented as a real-life application to show how you can use these algorithms.

4.1 Introducing informed search

As we discussed in the previous chapter, *blind search algorithms* work with no information about the search space, other than the information needed to distinguish the goal state from the others. Like the colloquial expression, "I'll know it when I see it," blind search follows a set framework of rules (e.g., breadth-first, depth-first, or Dijkstra's algorithm) to systematically navigate the search space. *Informed search algorithms* differ from blind search algorithms in the sense that the algorithm uses knowledge acquired during the search to guide the search itself. This knowledge can take the form of distance to target or incurred costs.

For example, in the 8-puzzle problem, we might use the number of misplaced tiles as a heuristic to determine how far any given state is from the goal state. In this way, we can determine at any given iteration of the algorithm how well it is performing and modify the search method based on current conditions. The definition of "good performance" depends on the heuristic algorithm being used.

Informed search algorithms can be broadly classified into those that solve for minimum spanning tree (MST) problems and those that compute the shortest path between two specific nodes or states, as outlined in figure 4.1.

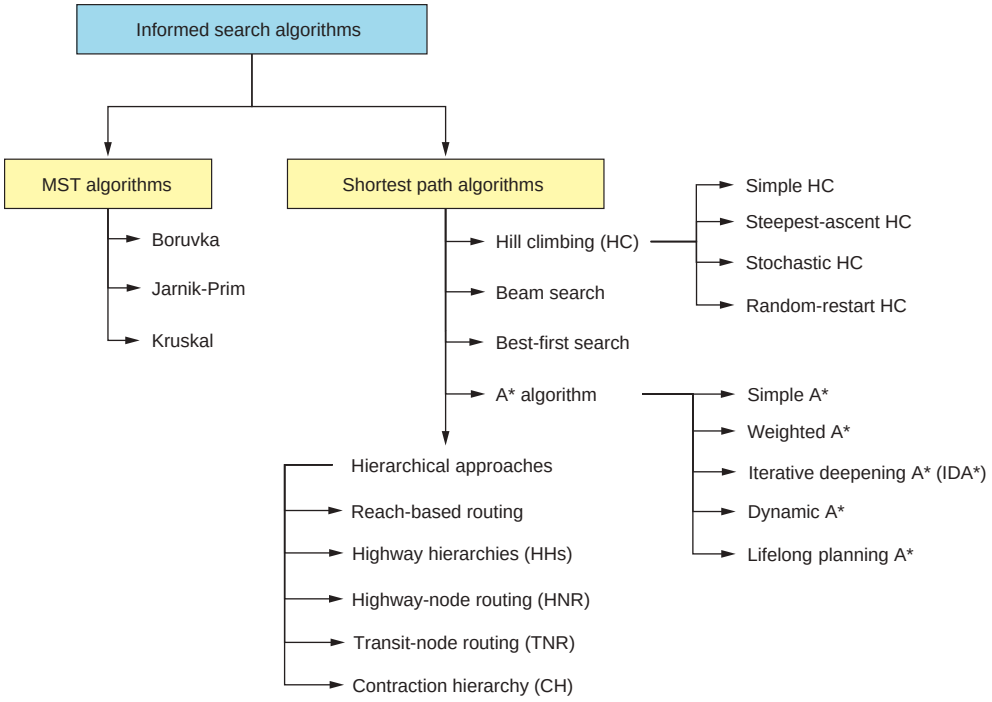


Figure 4.1 Examples of informed search algorithms. Each algorithm has multiple variants based on improvements, specific use cases, and specialized domains.

Several algorithms have been proposed to solve MST problems:

- *Borůvka's algorithm* finds an MST in a graph for which all edge weights are distinct. It also finds a minimum spanning forest, in the case of a graph that is not connected. It starts with each node as its own tree, identifies the cheapest edge leaving each tree, and then merges the trees joined by these edges. No edge presorting is needed or maintained in a priority queue.
- *Jarník-Prim's algorithm* starts from the root vertex and finds the lowest-weight edge from an MST vertex to a non-MST vertex and adds it to MST at each step.
- *Kruskal's algorithm* sorts edges by increasing weight and starts from the least-weighted edge to form a small MST component and then grows them into one large MST. I will describe this algorithm in more detail in the next section.

Hill climbing (HC), beam search, the A* algorithm, and contraction hierarchies (CH) are examples of informed search algorithms that can be used to find the shortest path between two nodes:

- *Hill climbing* is a local search algorithm that continuously moves in the direction of optimizing the objective function, increasing in the case of maximization problems, or decreasing in the case of minimization problems.
- *Beam search* explores a graph or tree by expanding the most promising node within a limited predefined set.
- *The A* algorithm* combines both the cost accrued up to a node and heuristic information, such as the straight-line distance between this node and the destination node, to select new nodes for expansion.
- *Hierarchical approaches*, such as reach-based routing, highway hierarchies (HHs), highway-node routing (HNR), transit-node routing (TNR), and contraction hierarchy (CH), are hierarchical approaches that take into consideration node importance and try to prune the search space by admissible heuristics.

The next section introduces the concept of an MST and presents an algorithm that can generate an MST for any given graph.

4.2 Minimum spanning tree algorithms

Imagine that you are the infrastructure manager for a small, remote rural town. Unlike most towns, there isn't really a main street or downtown area, so most points of interest are scattered. Additionally, budget cuts in previous years have left the roads either damaged or non-existent. The damaged roads are all buried under mud and are essentially impassable. You've been given a small budget to fix or build roads to improve the situation, but the money isn't enough to repair all the existing roads or to build new ones. Figure 4.2 shows a map of the town, as well as the locations of the existing damaged roads.

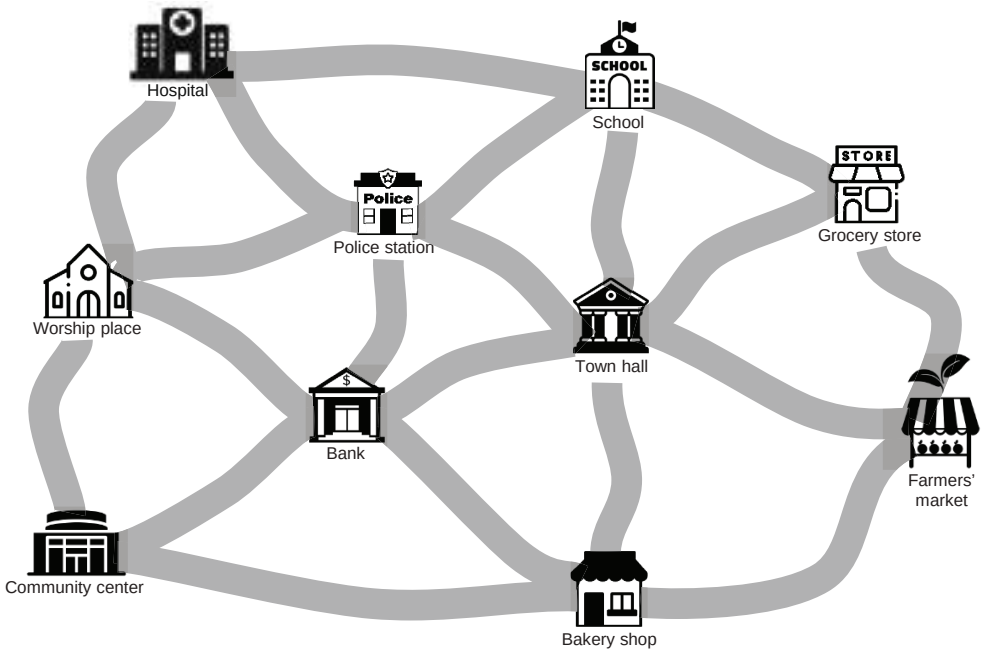


Figure 4.2 The muddy city problem. The roads in this town are badly damaged, but there isn't enough money to repair them all.

There are several ways to approach this problem, ranging from the not feasible (repair all the damaged roads and live with the consequences of a bankrupt town) to the overly conservative (only fix a few roads, or none at all, and ignore all the complaining townspeople). This problem is typically known as the muddy city problem, where various nodes in a graph must be connected while minimizing the edge weights. These weights can present the cost of fixing or paving the road, which may vary depending on the road's condition, length, and topology.

The mathematical way of solving the muddy city problem involves the idea of a minimum spanning tree (MST). A spanning tree, in general, is a cycle-free or loop-free subgraph of an undirected graph that connects all the vertices of the graph with the minimum number of edges. In figure 4.3, the left tree shows a graph *G* with nodes from A to F, while the middle and right trees show spanning trees of *G*. Notice that generic spanning trees do not require the edges to be weighted (i.e., to have a length, speed, time, or cost associated with them).

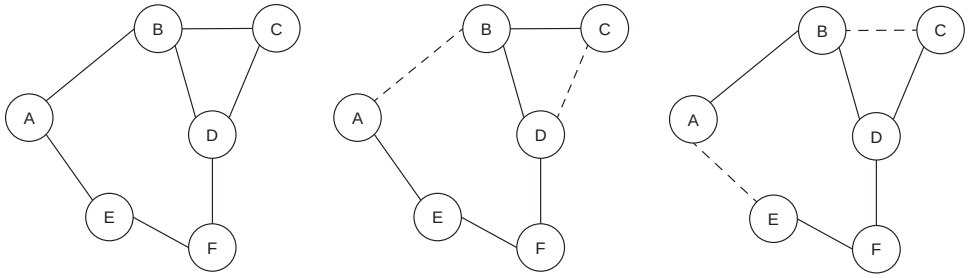


Figure 4.3 Examples of spanning trees. The middle and right trees reach every node of graph *G* with no loops or cycles.

An MST or minimum-weight spanning tree of an edge-weighted graph is a spanning tree whose weight (the sum of the weights of its edges) is no larger than the weight of any other spanning tree.

If $G=(V, E)$ is a graph, then any subgraph of G is a spanning tree if both of the following conditions are met:

- The subgraph contains all vertices V of G.
- The subgraph is connected with no circuits and no self-loops.

For a given spanning tree T for a graph G, the weight w of the spanning tree is the sum of the weights of all the edges in T. If the weight of T is the lowest of the weights of all the possible spanning trees of G, then we can call this an MST.

The previously described muddy city problem will be solved as an MST. Kruskal, Borůvka, Jarník-Prim, and Chazelle are all examples of algorithms that can be used to find an MST. Algorithm 4.1 shows the pseudocode for Kruskal's algorithm.

Algorithm 4.1 Kruskal's algorithm

```
Input: Graph G = (V, E) with each edge e ∈ E having a weight w(e)
Output: A minimum spanning tree T

Create a new graph T:= ∅ with the same vertices as G, but with no edges.
Define a list S containing all the edges in the graph G
Sort the edges list S in ascending order of their weights.
For each edge e in the sorted list:
   If adding edge e to T does not form a cycle:
      Add this edge to T.
   Else:
      Skip this edge and move to the next edge in the list.
Continue this process until all the edges are processed.
Return T as the minimum spanning tree of graph G.
```

To better understand these steps, let's apply Kruskal's algorithm to solve the muddy city problem. Figure 4.4 shows the original graph. The numbers near the edges represent edge weights, and no edges have been added to the MST yet. The following steps will generate the MST by hand iteration.

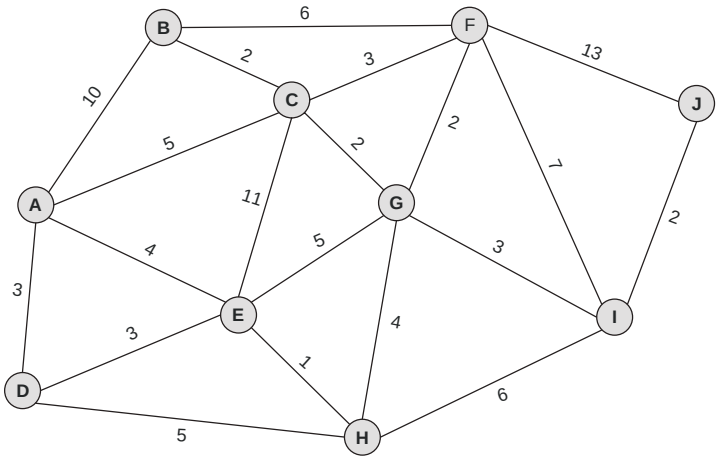


Figure 4.4 Solving the muddy city problem using Kruskal's algorithm—original graph

1. The shortest edge is E-H with a length of 1, so it is highlighted and added to the MST (figure 4.5).

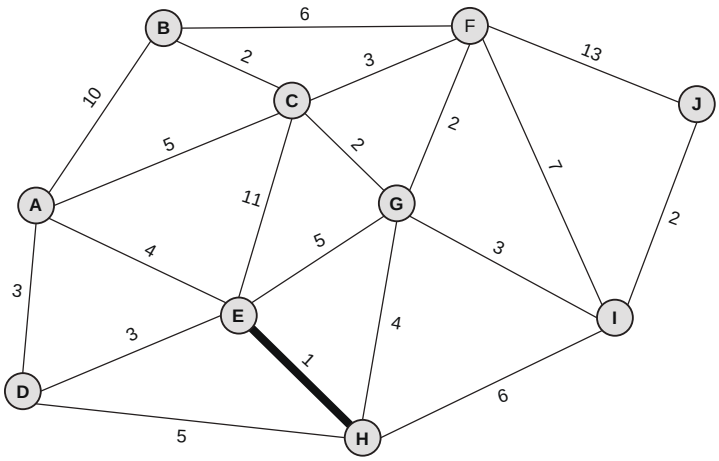


Figure 4.5 Solving the muddy city problem using Kruskal's algorithm—step 1

2. B-C, C-G, G-F, and I-J are now the shortest edges with lengths of 2. B-C is chosen arbitrarily and is highlighted, followed by C-G, G-F, and I-J, as they don't form a cycle (figure 4.6).

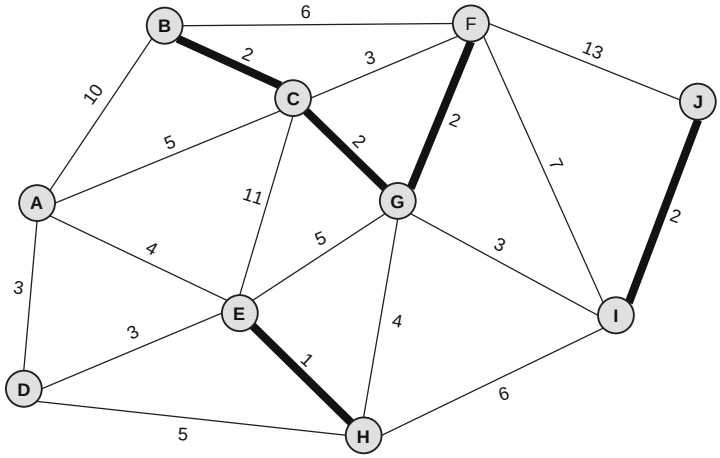


Figure 4.6 Solving the muddy city problem using Kruskal's algorithm—step 2

3 C-F, F-J, G-I, A-D, and D-E are now the shortest edges with lengths of 3. C-F cannot be chosen, as it forms a cycle. A-D is chosen arbitrarily and is highlighted, followed by D-E and G-I. F-J cannot be chosen, as it forms a cycle (figure 4.7).

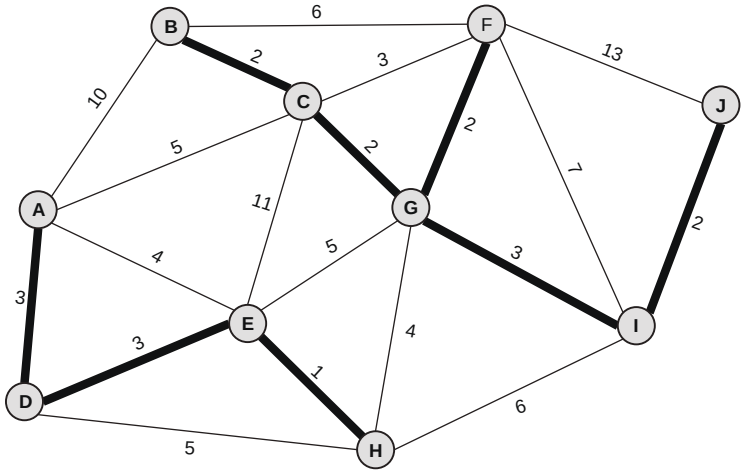


Figure 4.7 Solving the muddy city problem using Kruskal's algorithm—step 3

4 The next-shortest edges are A-E and G-H with lengths 4. A-E cannot be chosen because it forms a cycle, so the process finishes with the edge G-H. The minimum spanning tree has been found (figure 4.8).

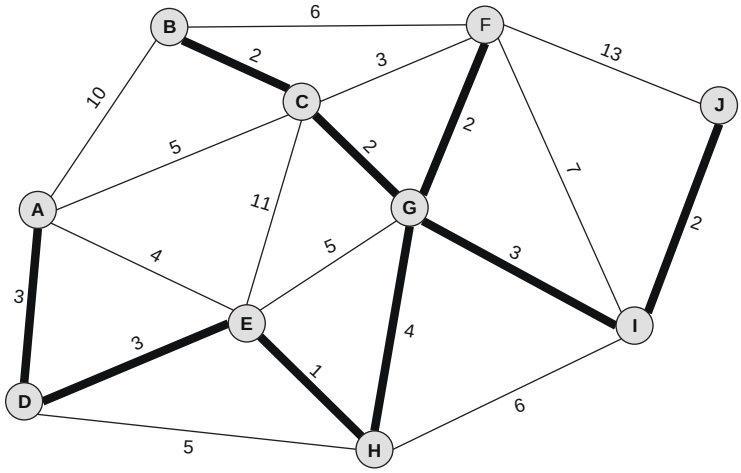


Figure 4.8 Solving the muddy city problem using Kruskal's algorithm—step 4

Figure 4.9 shows the final solution with all nodes in the graph connected.

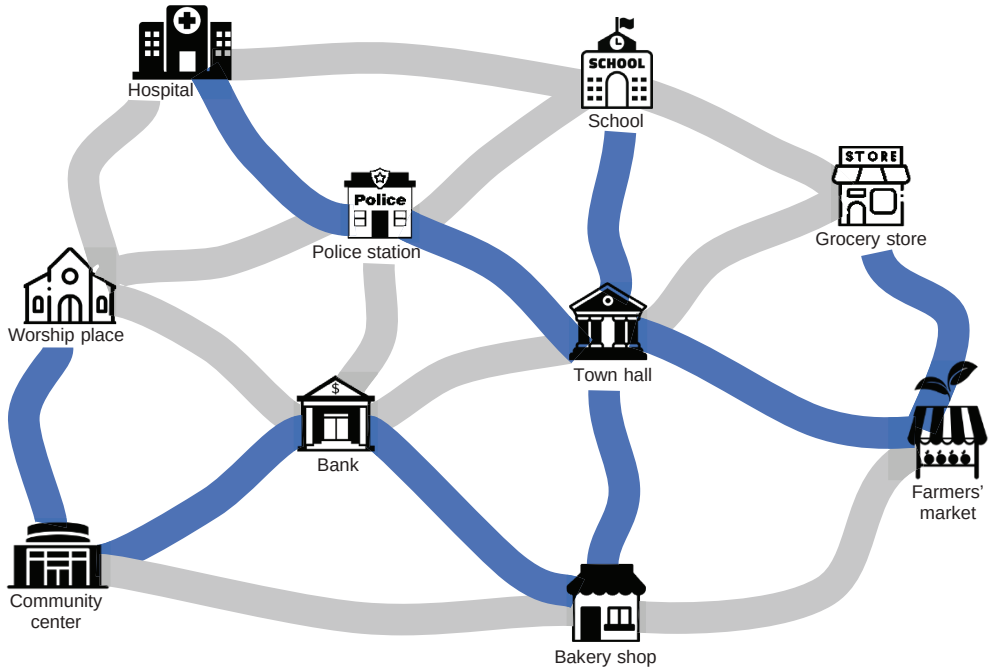


Figure 4.9 Solving the muddy city problem using Kruskal's algorithm. The algorithm adds edges to the final tree by ascending weight order, ignoring edges that will form a cycle.

This algorithm can be implemented easily in Python by using NetworkX's `find_cycle()` and `is_connected()` methods, which determine if an edge is a viable candidate for the MST, as well as the overall algorithm's termination condition, respectively. For the purposes of visual presentation, I've also used `spring_layout()` for the positions of the nodes and edges of the graph. The `spring_layout()` method uses a random number generator internally to generate these positions, and we can pass a seed (which allows a deterministic generation of so-called "pseudo-random" numbers) to guarantee a specific layout on each execution. Try modifying the seed parameter, and see what happens.

Listing 4.1 Solving the muddy city problem using Kruskal's algorithm

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

G = nx.Graph()
G.add_nodes_from(["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"])

edges = [
    ("A", "B", {"weight": 10}),
    ("A", "C", {"weight": 5}),
    ("A", "D", {"weight": 3}),
    ("A", "E", {"weight": 4}),
    ("B", "C", {"weight": 2}),
    ("B", "F", {"weight": 6}),
    ("C", "E", {"weight": 11}),
    ("C", "F", {"weight": 3}),
    ("C", "G", {"weight": 2}),
    ("D", "E", {"weight": 3}),
    ("D", "H", {"weight": 5}),
    ("E", "G", {"weight": 5}),
    ("E", "H", {"weight": 1}),
    ("F", "G", {"weight": 2}),
    ("F", "I", {"weight": 7}),
    ("F", "J", {"weight": 13}),
    ("G", "H", {"weight": 4}),
    ("G", "I", {"weight": 3}),
    ("H", "I", {"weight": 6}),
    ("I", "J", {"weight": 2}),
]

G.add_edges_from(edges)
pos = nx.spring_layout(G, seed=74)

def Kruskal(G, attr = "weight"):
    edges = sorted(G.edges(data=True), key=lambda t: t[2].get(attr, 1))
    mst = nx.Graph()
    mst.add_nodes_from(G)
    for e in edges:
        mst.add_edges_from([e])
```

Create an undirected graph and populate it with nodes and edges.

Using a seed with the spring_layout method guarantees the same placement of nodes every time.

Sort edges by weight in ascending order.

```
        try:
            nx.find_cycle(mst)
            mst.remove_edge(e[0], e[1])
        except:
            if nx.is_connected(mst):
                break
            continue
    return mst
```

find_cycle raises an error if no cycles exist in the graph. We can try/catch this error to determine if adding a new edge creates a cycle.

The set of edges in mst is a spanning tree if the graph formed by those edges is connected.

As a continuation of listing 4.1, the following code snippet is used to generate an MST using Kruskal and to visualize the MST:

Call Kruskal's algorithm to generate the MST.

Add the labels.

Draw the edges.

```
MST = Kruskal(G).edges
labels = nx.get_edge_attributes(G, "weight")
nx.draw_networkx_edges(G, pos, list(MST), width=4, edge_color="red")
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
nx.draw_networkx(G, pos, with_labels=True, font_color="white")
plt.show()
```

Draw the MST.

As you can see in figure 4.10, our Python implementation of the muddy city problem produces the exact same results that we achieved using hand iteration. Node G, which is the town hall, becomes a sort of central hub for the town's transportation infrastructure, and the total cost of road construction is minimized. It's worth noting, however, that while MST minimizes the total cost of connecting all nodes, it doesn't usually produce the most "convenient" solution. Should someone wish to travel from the place of worship to the hospital, for example, the shortest achievable distance would be 7 (passing through the police station), while our road network requires a total distance of 15.

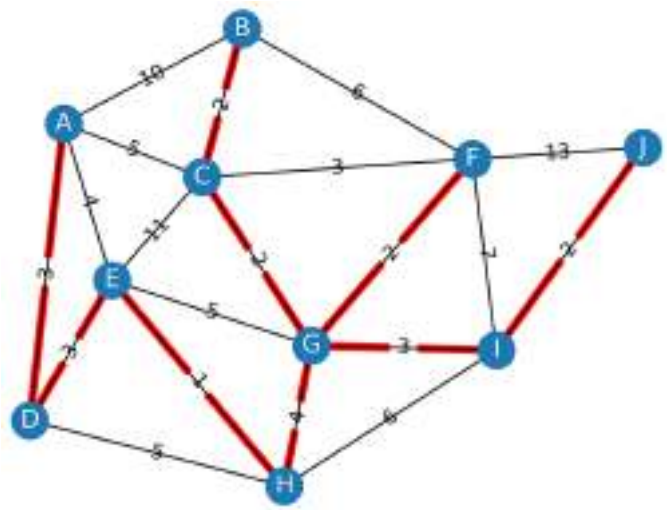


Figure 4.10 The muddy city problem solved using Kruskal's algorithm. The highlighted edges are part of the MST.

Let's apply this algorithm and this code to find the MST for all nodes in a search space surrounding the University of Toronto. Imagine that we have been tasked with installing new communications cables across the city, and we want to minimize the length of cable we use.

Listing 4.2 University of Toronto MST

```
import networkx as nx
import osmnx
import matplotlib.pyplot as plt
from optalgotools.algorithms.graph_search import Kruskal

reference = (43.661667, -79.395)
G = osmnx.graph_from_point(
    reference, dist=1000, clean_periphery=True, simplify=True, network_
type="drive"
)
fig, ax = osmnx.plot_graph(G)

undir_G = G.to_undirected()
sorted_edges = sorted(undir_G.edges(data=True), key=lambda t:
➥ t[2].get("length",1))

mst = Kruskal(G, sorted_edges=True, edges= sorted_edges,
➥ graph_type=nx.MultiDiGraph)

highlight_edges = ['b' if e in mst.edges else 'r' for e in G.edges]
edge_alphas = [1 if e in mst.edges else 0.25 for e in G.edges]
osmnx.plot_graph(
    G,
    edge_linewidth=2,
    edge_color=highlight_edges,
    edge_alpha=edge_alphas
)
plt.show()
```

Use the drive network to only focus on drivable roads. This prevents the code from building an MST with a combination of roads and sidewalks.

Get an undirected copy of the graph, and sort the road network edges by edge length.

Call the Kruskal algorithm from optalgotools, using a presorted list and specifying the graph type.

Visualize the MST by highlighting the included edges.

Figure 4.11 shows the resulting MST generated by Kruskal's algorithm. The road network graph in the figure may seem like one big connected component, but it isn't. There are one-way streets that seem to connect adjacent nodes on the graph, but in reality, they are not connected (you can go from A to B but not the reverse). We overcome this by converting the directed graph into an undirected graph using the `to_undirected` function in NetworkX.

The version of Kruskal's algorithm used in the listing is the same as was used for the muddy city problem. We're importing it from optalgotools to reduce the amount of code needed.

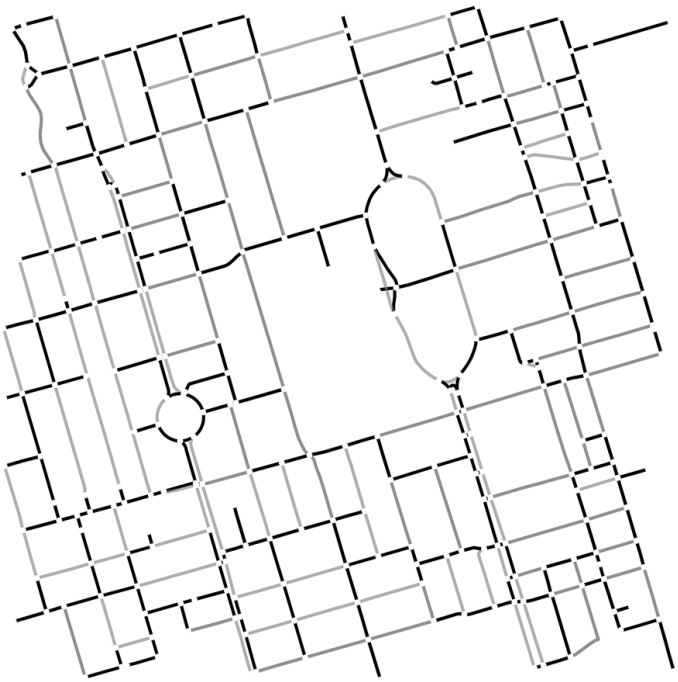


Figure 4.11 The MST generated by Kruskal's algorithm. All the edges included in the MST are highlighted. There are no cycles in the MST, and the total weight of the tree is minimized.

MSTs have a wide variety of applications in the real world, including network design, image segmentation, clustering, and facility location problems. MSTs are especially useful when dealing with problems concerning budgeting, such as planning networks, as they allow all nodes to be connected with a minimum total cost. As previously mentioned, informed search algorithms can be used to find MSTs as described in this section and with shortest path algorithms, which are discussed in the next section.

4.3 Shortest path algorithms

Informed search algorithms can be used to find the shortest path between two nodes by using knowledge about the problem (domain-specific knowledge) to prune the search. This knowledge, in the form of a heuristic function, gives an estimate of the distance to the goal. Examples of informed search algorithms include hill climbing, beam search, best-first, A*, and contraction hierarchies. The following subsections discuss these algorithms in detail.

4.3.1 Hill climbing algorithm

Assume that you are trying to climb to the top of a mountain in a dense fog. There is only one path up and down the mountain, but you aren't sure exactly where the peak is. Thus, you are only able to judge your progress by looking one step behind you and seeing if you've gone uphill or downhill since your last step. How can you know when you've reached the summit? A good guess would be when you're no longer going uphill!

Starting with a known (non-optimized) solution to a function or with an initial state, the hill climbing algorithm checks the neighbors of that solution and chooses the neighbor that is more optimized. This process is repeated until no better solution can be found, at which point the algorithm terminates.

The hill climbing algorithm is a local greedy search algorithm that tries to improve the efficiency of depth-first by incorporating domain-specific knowledge or heuristic information, so it can be considered as an informed depth-first algorithm. The hill climbing algorithm's pseudocode applied to graph search is shown in the algorithm 4.2 assuming minimization problem.

Algorithm 4.2 The hill climbing algorithm

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

Initialize current ← random route from source to destination
Initialize neighbours ← children of current

While min(neighbours) > current do
   Set current ← min(neighbours)
   Update neighbours ← children of current
Return current as the route from source to destination
```

The algorithm sorts the successors of a node (according to their heuristic values) before adding them to the list to be expanded. This algorithm demands very little in the way of memory and computational overhead, as it simply remembers the current successors as the current path it is working on. It's a non-exhaustive technique; it does not examine the entire tree, so its performance will be reasonably fast. However, while this algorithm works relatively well with convex problems, functions with multiple local maxima will often result in an answer that is not the global maximum. It also performs poorly when there are plateaus (a local set of solutions that are all similarly optimized).

As shown in figure 4.12, depending on the initial state, the hill climbing algorithm may get stuck in local optima. Once it reaches the top of a hill, the algorithm will stop, since any new successor will be down the hill. This is analogous to climbing the mountain in the fog, reaching a smaller peak, and thinking that you've reached the main summit.

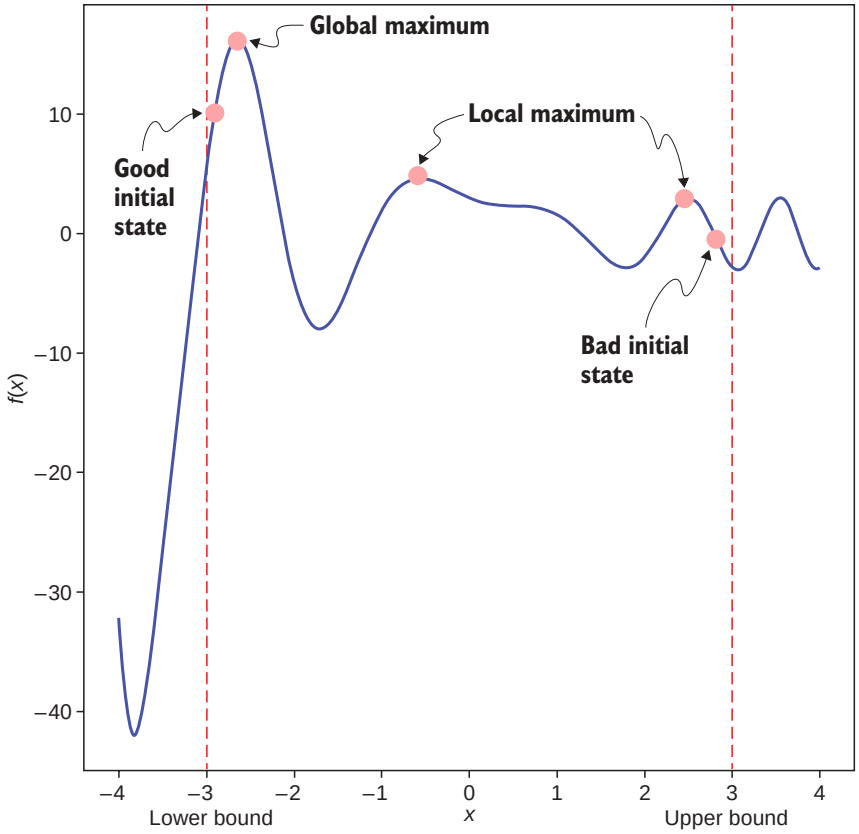


Figure 4.12 Depending on the initial state, the hill climbing algorithm may get stuck in local optima. Once it reaches a peak, the algorithm will stop, since any new successor will be down the hill.

Simple hill climbing, steepest-ascent hill climbing, stochastic hill climbing, and random-restart hill climbing are all variants of the hill climbing algorithm, as shown in figure 4.13.

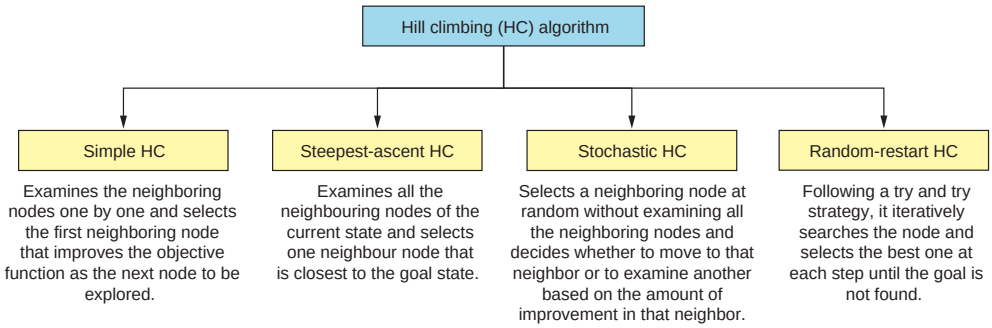


Figure 4.13 Variants of the hill climbing algorithm

Simple hill climbing examines the neighboring nodes one by one and selects the first neighboring node that optimizes the objective function as the next node to be explored. *Steepest-ascent or steepest-descent hill climbing* is a variation on the simple hill-climbing algorithm that first examines all the neighboring nodes of the current state and selects one neighbor node that is closest to the goal state. *Stochastic hill climbing* is a randomized version of a hill climbing algorithm that selects a neighboring node at random without examining all the neighboring nodes. This algorithm decides whether to move to that neighbor or to examine another based on the amount of improvement in that neighbor. Random-restart hill climbing or first-choice hill climbing follows a try-and-try strategy and iteratively searches the nodes and selects the best one at each step until the goal is found. If it gets stuck in a local maximum, it restarts the process from a new random initial state. Compared to the other hill climbing variants, this algorithm is better able to reach the destination if there are plateaus, local optima, and ridges.

Gradient descent algorithm

The *gradient descent algorithm* is widely used in machine learning to train models and make predictions. Gradient descent and hill climbing are two fundamentally different algorithms and cannot be confused with each other. Instead of climbing up a hill, gradient descent can be seen as hiking down to the bottom of a valley. Gradient descent is an iterative algorithm that looks at the slope of the local neighbors and moves in the direction with the steepest slope or the direction of negative gradient to optimize a continuous differentiable function. Alternatively, in the case of a maximization problem, *gradient ascent* moves in the direction with a positive gradient to optimize the objective function.

The gradient descent algorithm usually converges to a global minimum if the function is convex (i.e., if any local minimum is also a global minimum) and the learning rate is properly chosen. The hill climbing algorithm is a heuristic greedy algorithm that can easily get stuck in local optima. It is used mainly for discrete optimization problems, such as the traveling salesman, as it doesn't require the objective function to be differentiable.

Assume we have the simple graph shown in figure 4.14. The source node is S and the destination node is G.

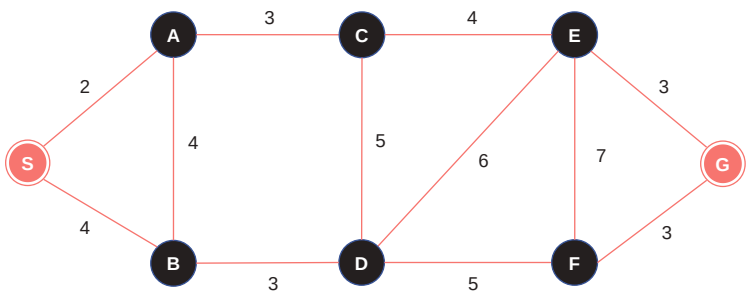


Figure 4.14 An 8 points of interest (POIs) road network in the form of a graph

This graph can be converted into a tree by finding a spanning tree that includes all the vertices of the original graph and that is connected and acyclic, as shown in figure 4.15.

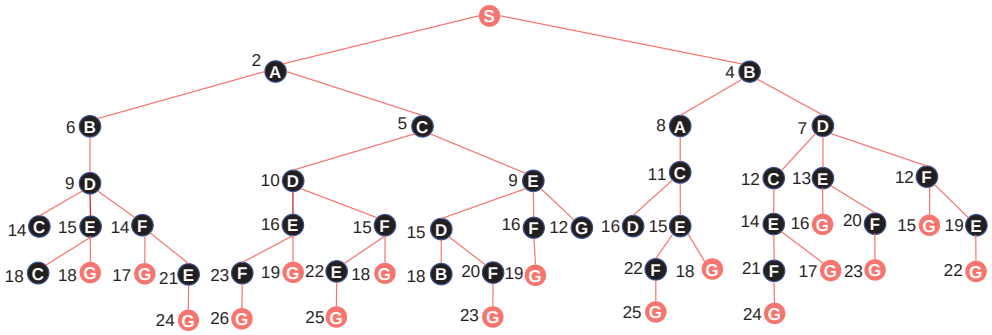


Figure 4.15 An 8 points of interest (POIs) road network in the form of a tree

As you can see, there are multiple ways to go from S to G, each with different costs. Following the hill climbing algorithm, the shortest path between S and G will be S→A→C→E→G, as illustrated in figure 4.16.

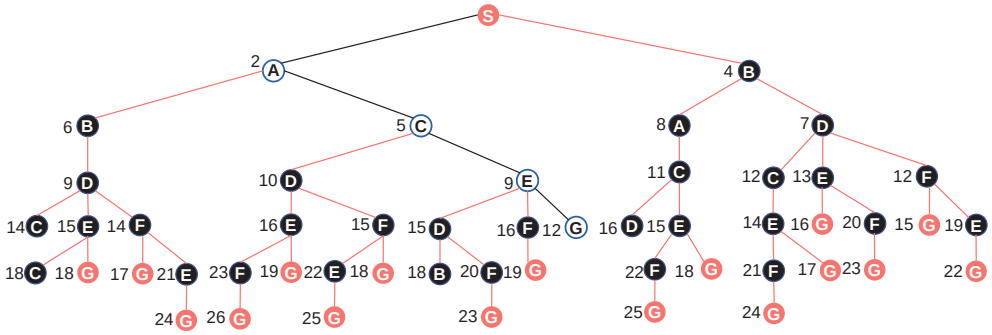


Figure 4.16 Shortest path between S and G using the hill climbing algorithm

Let's now use the 8-puzzle problem to illustrate the hill climbing approach. Figure 4.17 shows how the hill climbing search progresses, using the number of misplaced tiles, excluding the blank tile, as heuristic information $h(n)$. For example, in step 2, tiles 1, 4, 6, and 7 are wrongly placed, so $h = 4$. In step 3, tiles 1, and 4 are misplaced, so $h = 2$.

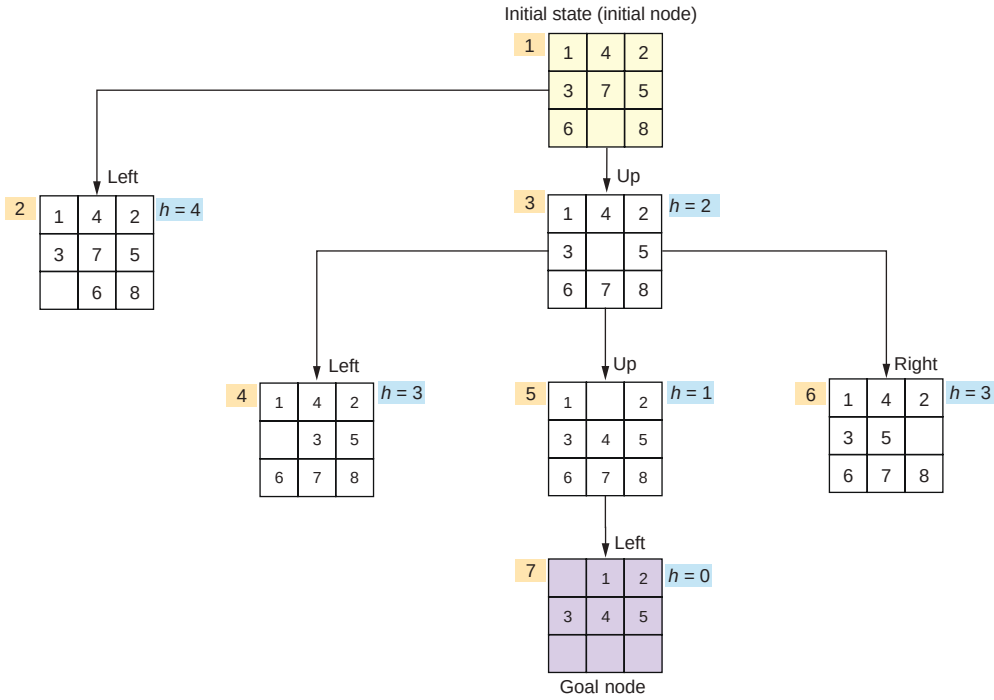


Figure 4.17 Using the hill climbing algorithm to solve the 8-puzzle problem. At each iteration, the algorithm explores neighboring states, looking for the minimum heuristic value.

Listing 4.3 shows a simple implementation of hill climbing in Python. The code selects nodes to explore by minimizing the heuristic value of the next node. A more complex version, involving generating shortest paths, will be presented at the end of this chapter.

Listing 4.3 Solving the 8-puzzle problem using hill climbing

```
import matplotlib.pyplot as plt

def Hill_Climbing(origin, destination, cost_fn):
    costs = [cost_fn(origin)]
    current = origin
    route = [current]
    neighbours = current.expand()
    shortest = min(neighbours, key=lambda s: cost_fn(s))
```

The neighbor nodes of the current state can be generated using expand().

The "closest" neighbor is the one with the lowest cost function (i.e., the fewest misplaced tiles).

```
    costs.append(cost_fn(shortest))
    route.append(shortest)

    while cost_fn(shortest) < cost_fn(current):
        current = shortest
        neighbours = current.expand()
        shortest = min(neighbours, key=lambda s: cost_fn(s))
        costs.append(cost_fn(shortest))
        route.append(shortest)
        if shortest == destination:
            break

    return route, costs

def misplaced_tiles(state: State):
    flat = state.flatten()
    goal = range(len(flat))
    return sum([0 if goal[i] == flat[i] else 1 for i in range(len(flat))])
```

Terminate the algorithm if the goal state has been reached.

Calculate and return the number of misplaced tiles that are not in their goal positions.

NOTE The State class and visualize function are defined in the complete listing, available in the book's GitHub repo.

The following code snippet defines the initial and goal states of the puzzle and uses hill climbing to solve the puzzle:

```
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, costs = Hill_Climbing(init_state, goal_state,
    ➥ misplaced_tiles)
    plt.xticks(range(len(costs)))
    plt.ylabel("Misplaced tiles")
    plt.title("Hill climbing: 8 Puzzle")
    plt.plot(costs)
    visualize(solution)
```

Check if there is even a solution.

Solve the puzzle using hill climbing.

Plot the search progress, and visualize the solution.

The output is shown in figure 4.18.

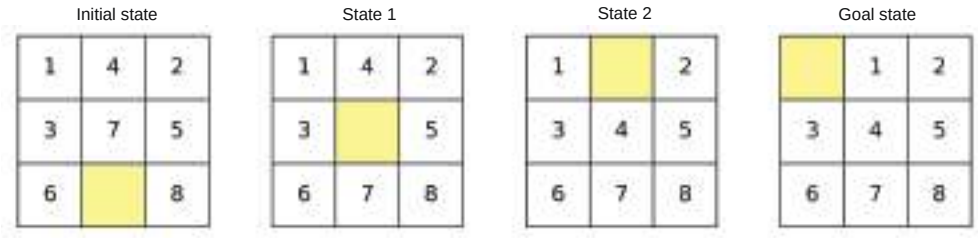


Figure 4.18 States of the hill climbing solution for the 8-puzzle problem. Each subsequent state is selected by minimizing its cost compared to its neighbors. As the 8-puzzle problem has not only a well-defined but also an achievable goal state, the algorithm's termination condition (the goal being reached) coincides with the "peak" of the hill (which in this case is a valley, as it is a minimization problem).

As the 8-puzzle problem uses a heuristic as a cost, it essentially becomes a minimization problem. This implementation differs from the standard hill climbing in that, as a solution can always be found, and the graph is fully connected (you can transition from any state to another state through some combination of tile movements), the algorithm is guaranteed to find the optimal solution eventually. More complex problems will often generate solutions that are near-optimal.

4.3.2 Beam search algorithm

The *beam search algorithm* tries to minimize the memory requirements of the breadth-first algorithm, so it can be seen as an informed breadth-first algorithm. While hill climbing maintains a single best state throughout the run, beam search keeps w states in memory, where w is the beam width. At each iteration, it generates the neighbors for each of the states and puts them into a pool with the states from the original beam. It then selects the best w states from the pool at each level to become the new beam, and the rest of the states are discarded. This process then repeats. The algorithm expands only the first w promising nodes at each level.

This is a non-exhaustive search, but it is also a hazardous process, because a goal state might be missed. As this is a local search algorithm, it is also susceptible to getting stuck at a local optima. A beam search with w equal to the number of nodes in each level is the same as a BFS. Because there is the risk that a state that could lead to the optimal solution might be discarded, beam searches are incomplete (they may not terminate with the solution).

Algorithm 4.3 shows the pseudocode for the beam search algorithm applied to a graph search.

Algorithm 4.3 The beam search algorithm

```
Inputs: A source node, a destination node, and beam width w
Output: A route from source to destination

Initialize Seen ← nil
Initialize beam ← random w routes from source to destination
```

```
Add beam to seen
Initialize pool ← children of routes in the beam with consideration of seen +
beam
Initialize last_beam ← nil
While beam is not last_beam do
   Update last_beam ← beam
   Update beam ← the best w routes from pool
   If last_beam == beam then break
   Add beam to seen
   Update pool ← children of routes in the beam + beam

Return optimal route in beam
```

In section 2.3.1, you saw that BFS has an exponential complexity of $O(b^d)$, where b represents the maximum branching factor for each node and d is the depth one must expand to reach the goal. In the case of beam search, we only explore $w \times b$ nodes at any depth, saving many unneeded nodes compared to BFS. However, finding the best states or routes requires sorting, which is time-consuming if $w \times b$ is a huge number. A beam threshold can be used to handle this problem, where the best node is selected based on the heuristic function $h(n)$ within a certain threshold, and all the nodes outside this are pruned away.

Revisiting the simple routing problem with 8 points of interest (figure 4.14) and following the beam search algorithm with $w = 2$, the shortest path between S and G will be S-A-C-E-G, as illustrated in figure 4.19.

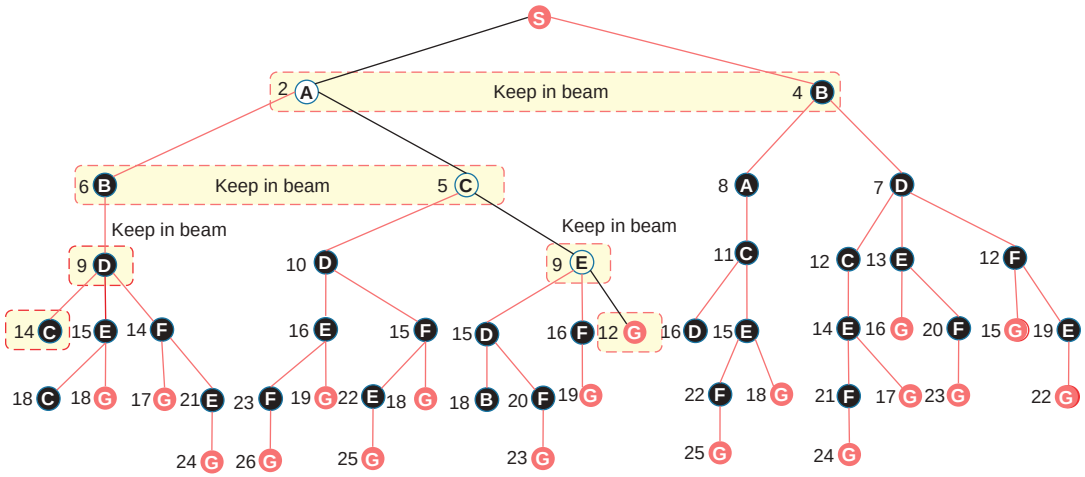


Figure 4.19 The shortest path between S and G using a beam search algorithm. With a beam width $w = 2$, two states are kept in the beam at each iteration. After generating the neighbors of each element in the beam, only the best w beams are kept.

The following listing shows a basic implementation of beam search used for a simple routing problem. See the full code in the GitHub repo to see how the graph is initialized, as well as how the visualization is generated (it is quite similar to that of listing 4.1).

Listing 4.4 Simple routing with beam search

```
import matplotlib.pyplot as plt
import networkx as nx
import heapq
from optalgotools.structures import Node
from optalgotools.routing import cost

G = nx.Graph()
G.add_nodes_from(["A", "B", "C", "D", "E", "F", "G", "S"])
edges = [
    ("A", "B", {"weight": 4}),
    ("A", "C", {"weight": 3}),
    ("A", "S", {"weight": 2}),
    ("B", "D", {"weight": 3}),
    ("B", "S", {"weight": 4}),
    ("C", "E", {"weight": 4}),
    ("C", "D", {"weight": 5}),
    ("D", "E", {"weight": 6}),
    ("D", "F", {"weight": 5}),
    ("E", "F", {"weight": 7}),
    ("E", "G", {"weight": 3}),
    ("F", "G", {"weight": 3}),
]
G.add_edges_from(edges)
G=G.to_directed()

def Beam_Search(G, origin, destination, cost_fn, w=2, expand_kwargs=[],
➥ cost_kwargs=[]):
    seen = set()
    seen.add(origin)
    last_beam = None
    pool = set(origin.expand(**expand_kwargs))
    beam = []
    while beam != last_beam:
        last_beam = beam
        beam = heapq.nsmallest(
            w, pool, key=lambda node: cost_fn(G, node.path(),
            ➥ **cost_kwargs))
        current = beam.pop(0)
        seen.add(current)
        pool.remove(current)
        children = set(current.expand(**expand_kwargs))
        for child in children:
            if child in seen: next
            else:    #D
                if child == destination:
                    return child.path()
                beam.append(child)
        pool.update(beam)
    return None
```

Create a directed graph.

Get the neighbors of the node using the origin class's expand() method, passing any necessary arguments.

Prune the pool down to only the best k paths, passing any necessary arguments to the cost function.

Child routes are generated for each route by adding one extra node to the route. For each of these new routes, they are rejected (already explored), added to the beam (and then to the pool), or accepted (because they reach the destination).

None is returned if a path cannot be found.

This function can be called with the following example parameters:

```
result = Beam_Search(
    G,
    Node(G, "S"),
    Node(G, "G"),
    cost,
    expand_kwargs={"attr_name": "weight"},
    cost_kwargs={"attr_name": "weight"},
)
```

Visualizing the output of this algorithm produces the graph in figure 4.20, where the highlighted line represents the solution path from S to G.

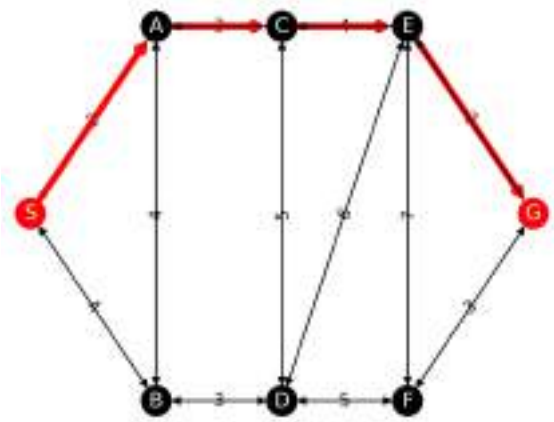


Figure 4.20 Solution using a beam width of *w* = 2. The highlighted line represents the solution path.

As you will see in a later section, when applying beam search to a real-life routing problem, generating the children in a beam search can become very complicated and time consuming.

4.3.3 A* search algorithm

The *A** (pronounced A-star) algorithm is an informed search algorithm widely used in pathfinding and graph traversal. This algorithm is a special case of the best-first algorithm. Best-first search is a kind of mixed depth- and breadth-first search that expands the most desirable unexpanded node based on either the cost to reach the node or an estimate or heuristic value of the cost to reach the goal from that node. The A* algorithm takes into account both the cost to reach the node and the estimated cost to reach the goal in order to find the optimal solution.

The pseudocode for A* is shown in algorithm 4.4.

Algorithm 4.4 A* algorithm

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

Initialize A* heuristic ← sum of straight-line distance to source and
destination
Initialize PQ ← min heap according to A* heuristic
```

```
Initialize frontier ← a PQ initialized with source
Initialize explored ← empty
Initialize found ← False

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

Return route
```

A* search tries to reduce the total number of states explored by incorporating both the actual cost and a heuristic estimate of the cost to get to the goal from a given state. The driving force behind A* is the selection of a new vertex (or node) to explore based on the lowest value. The value of the evaluation function $f(n)$ is computed using the following formula:

$$f(n) = g(n) + h(n) \tag{4.1}$$

In equation 4.1, $g(n)$ is the actual cost of the partial path already traveled from the source node S to node n. The heuristic information $h(n)$ can be the straight-line distance between node n and destination node G, or some other function. When $h(n) = 0$ for all nodes, A* will behave like a uniform-cost search (UCS), which was explained in section 3.4.2, so the node with the lowest cost will be expanded regardless of the estimated cost to reach the goal.

In *weighted A**, a constant weight is added to the heuristic function as follows:

$$f(n) = g(n) + w \times h(n) \tag{4.2}$$

To increase the importance of $h(n)$, w should be greater than 1. A dynamic weight $w(n)$ can be also used. The choice of the heuristic information is critical to the search results. The heuristic information $h(n)$ is admissible if and only if $h(n)$ is less than the actual cost to reach the goal state from n for every node n. This means that admissible heuristics never overestimate the cost to reach the goal and can lead to optimal solutions only when the heuristic function is close to the true remaining distance.

The A* heuristic algorithm operates by choosing the next vertex for exploration in a *greedy* manner, prioritizing nodes based on the value of the heuristic function. As the sum of the distance to the origin and destination is minimized when n lies on a straight line from S to G, this heuristic prioritizes nodes that are closer to the straight-line distance from origin to destination.

To better understand the A* procedure, let's consider the simple problem of finding the shortest path between a source node S and a goal node G in an 8 points of interest

(POIs) road network. This is the same POI graph from figure 4.14 but with heuristic values added for each node. An example of heuristic information is the straight-line distance to the goal as shown above each vertex in figure 4.21.

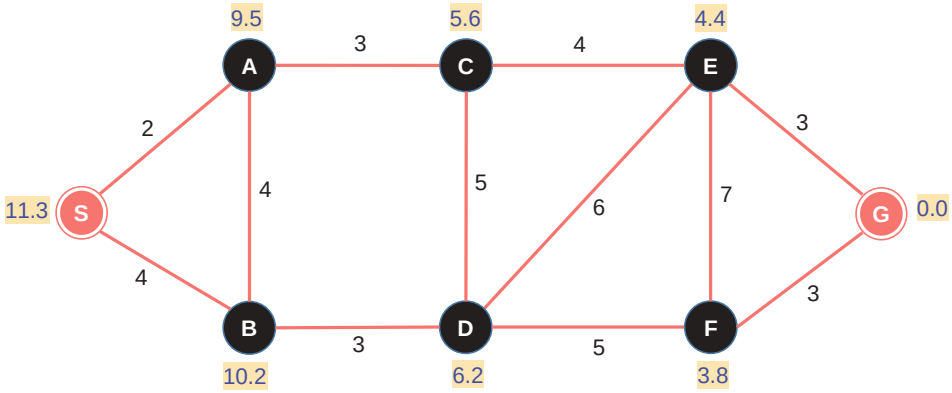
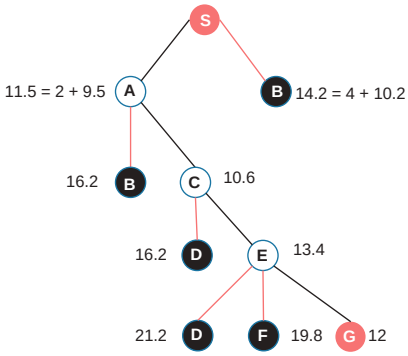


Figure 4.21 An 8 POIs road network in the form of a graph with heuristic information (straight-line distance to the goal) shown above each vertex

Figure 4.22 shows the steps for using A* to find the shortest path from S to G.



Nose	*g*(*n*)	*h*(*n*)	*f*(*n*) = *g*(*n*) + *h*(*n*)	Expand?
S→A	2	9.5	11.5	Yes
S→B	4	10.2	14.2	No
S→A→B	2 + 4 = 6	10.2	16.2	No
S→A→C	2 + 3 = 5	5.6	10.6	Yes
S→A→C→D	2 + 3 + 5 = 10	6.2	16.2	No
S→A→C→E	2 + 3 + 4 = 9	4.4	13.4	Yes
S→A→C→E→D	2 + 3 + 4 + 6 = 15	6.2	21.2	No
S→A→C→E→F	2 + 3 + 4 + 7 = 16	3.8	19.8	Yes
S→A→C→E→G	2 + 3 + 4 + 3 = 12	0	12	Goal reached

Figure 4.22 The A* steps to find the shortest path between the source node S and goal node G in an 8 POIs road network. The sum of the already incurred costs and the distance to the goal is used as the heuristic value to determine whether to expand a certain node.

This algorithm may look complex since it seems to need to store incomplete paths and their lengths at various places. However, using a recursive best-first search implementation can solve this problem in an elegant way without the need for explicit path storing. The quality of the lower-bound goal distance from each node greatly influences the timing complexity of the algorithm. The closer the given lower bound is to the true distance, the shorter the execution time.

We can apply the A* algorithm to the simple routing problem as follows (see the book's GitHub repo for the full code, containing graph initialization and visualization).

Listing 4.5 Simple routing using A* search

```
import matplotlib.pyplot as plt
from optalgotools.structures import Node
from optalgotools.routing import cost
import networkx as nx
import heapq

def A_Star(
    G, origin, destination, heuristic_fn, cost_fn, cost_kwargs=[],
    ➥ expand_kwargs=[]
):
    toDestination = heuristic_fn(G, origin, destination)
    toOrigin = {}
    route = []
    frontier = list()
    frontier.append(origin)
    toOrigin[origin] = 0
    explored = set()
    found = False
    while frontier and not found:
        node = min(frontier, key=lambda node: toOrigin[node] +
        ➥ toDestination[node])
        frontier.remove(node)
        explored.add(node)
        for child in node.expand(**expand_kwargs):
            if child not in explored and child not in frontier:
                if child == destination:
                    route = child.path()
                    found = True
                    continue
                frontier.append(child)
                toOrigin[child] = cost_fn(G, child.path(), **cost_kwargs)
    return route
```

Choose a node based on its heuristic value.

Expand the node's children, adding them to the frontier or terminating if the destination is found.

Add the toOrigin value for each node on the fly.

The implementation of A* in listing 4.5 doesn't use a "real" A* heuristic algorithm for two reasons:

- The straight line or haversine distance from any node to the destination cannot be readily determined, as we only have edge weights and no real spatial data (coordinates) to situate each node. To get around this, I created a function called `dummy_astar_heuristic` that returns static, arbitrarily generated distances for the purposes of this example.
- The distance from the origin to any node (straight line or otherwise) cannot be determined ahead of time for the same reasons as in the previous point. Thus, we use the traveled distance (i.e., the cost from the origin to the node as far as has been explored), and we update that value as the algorithm discovers new nodes. Later in this chapter, we will see how working with geographic data (such as road networks) allows us to capture this information beforehand.

`A_Star` can be called as follows, with some example parameters:

```
result = A_Star(
    G,
    Node(G, "S"),
    Node(G, "G"),
    dummy_astar_heuristic,
    cost,
    expand_kwargs={"attr_name": "weight"},
    cost_kwargs={"attr_name": "weight"},)
```

This gives us the same result as with beam search and hill climbing: a path of S-A-C-E-G.

Haversine distance

The haversine formula is used to calculate the geographic distance between two points on earth, given their longitudes and latitudes, based on a mean spherical earth radius. This distance is also known as the great-circle distance and is calculated using the following formula:

$$d = R \times C \text{ and } C = 2 \times \text{atan2}\left(\sqrt{a}, \sqrt{1-a}\right) \quad \textbf{4.3}$$

In the preceding equation, $a = \sin^2(\Delta lat/2) + \cos(lat1) \times \cos(lat2) \times \sin^2(\Delta lon/2)$, R is the earth radius (6,371 km or 3,691 miles), and d is the final distance between the two points. The following figure shows the haversine distance between Los Angeles, USA (34.0522° N, 118.2437° W) and Madrid, Spain (40.4168° N, 3.7038° W).

Haversine distance between Los Angeles and Madrid

The following Python code can be used to calculate the haversine distance:

```
!pip install haversine
from haversine import haversine
```

Install the Haversine package, and import the haversine function.

```
LA = (34.052235, -118.243683)
```

Set coordinates of two points in (latitude, longitude) format.

```
Madrid = (40.416775, -3.703790)

distance = haversine(LA, Madrid)
print(distance)
```

Set coordinates of two points in (latitude, longitude) format.

Calculate the distance in kilometers.

The implementation of the A* heuristic in optalgotools defaults to calculating distances as if the earth were flat. For local searches, this yields the best results. If the size of the search area is larger, it is better to calculate distance by passing `optalgotools.utilities.haversine_distance` into the `measuring_dist` parameter, which considers the curvature of the earth.

4.3.4 Hierarchical approaches

When facing routing problems at larger scales, such as those involving entire countries or graphs with millions of nodes, it is simply implausible to use basic approaches like Dijkstra's. In the previous chapter, you saw that bidirectional Dijkstra's gives a two times speedup compared to Dijkstra's algorithm. However, much faster routing algorithms are needed for interactive applications like navigation apps. One way to achieve this is to precompute certain routes and cache them on servers so that response times to user queries are reasonable. Another method involves pruning the search space. Hierarchical search algorithms prune the search space by generating admissible heuristics that abstract the search space.

> **NOTE** For more details about the general approaches of hierarchical methods, see Leighton, Wheeler, and Holte, "Faster optimal and suboptimal hierarchical search" [1].

Highway hierarchies involve the assignment of hierarchy "levels" to each road in a road network graph. This distinguishes the type of road segment (e.g., residential roads, national roads, highways). This is further supplemented by relevant data such as the maximum designated driving speed as well as the number of turns in the road. After the heuristics are generated for the graph, the data is passed through a modified search function (bidirectional Dijkstra's, A*, etc.) that considers the distance to the destination and the potential expansion node type. Highway hierarchy algorithms will generally consider highways as viable expansion nodes when they are further away from the target and will start to include national roads, and finally residential streets, as they near the destination. During the trip, less important roads merge with more important roads (e.g., residential roads merge with national roads, and national roads merge with highways). This allows us to avoid exploring millions of nodes.

Take, for example, a long-distance driving trip from New York to Miami. In the beginning, you will need to navigate local roads toward a highway or interstate. In the middle section of the trip, you will drive exclusively on the interstate or highway. Nearing your

destination, you will leave the interstate and once again take local roads. While this approach makes sense, there are some disadvantages. First, the algorithm overlooks what kind of roads humans prefer to drive on. While a highway might make sense for a given route, the user may prefer to take local roads (such as when driving to a friend's house who lives nearby). Second, highway hierarchies do not consider factors such as traffic, which fluctuates often and adds significant cost to an "optimal" route. You can learn more about highway hierarchies in Sanders and Schultes' article "Highway hierarchies hasten exact shortest path queries" [2].

The contraction hierarchies (CH) algorithm is another hierarchical approach. It is a speed-up technique that improves the performance of shortest-path computations by pruning the search space based on the concept of node contraction. For example, for an 80 mile single-source single-destination shortest path search query, the bidirectional Dijkstra's algorithm explores 220,000 nodes, unidirectional A* explores 50,000 nodes, and bidirectional A* improves on those by exploring about 25,000 nodes. Contraction hierarchies solve the same problem by exploring only about 600 nodes. This makes CH much faster than Dijkstra's, bidirectional Dijkstra's, and A*.

> **NOTE** Contraction hierarchies were introduced in Geisberger et al.'s 2008 "Contraction hierarchies: Faster and simpler hierarchical routing in road networks" article [3]. The 80 mile single-source single-destination shortest path search query is discussed on the *GraphHopper* blog (http://mng.bz/n142).

The CH algorithm encompasses two main phases:

1. The *preprocessing phase* is where nodes and edges are categorized based on some notion of importance. Important nodes can be major cities, major intersections, bridges connecting the two sides of a city, or points of interest that shortest paths go through. Each node is contracted based on the level of importance from least important to most important. During the contraction process, a set of shortcut edges is added to the graph to preserve shortest paths.
2. The *query phase* is where a bidirectional Dijkstra's search (or any other search) is run on the preprocessed graph, considering only increasingly important edges. This results in selectively ignoring less important nodes, and overall improving querying speed.

It's worth noting that the CH algorithm is mainly a preprocessing algorithm, which means that it is used before querying the shortest path. This preprocessing phase takes some time, but once it's done, the query phase is very fast. The algorithm can handle large graphs and can be used for various types of graphs, not only road networks. Let's dive into both phases in further detail.

THE CH PREPROCESSING PHASE

The preprocessing phase takes as input the original graph, and it returns an augmented graph and node order to be used during the query phase.

Assume a weighted directed graph $G = (V,E)$. The nodes of this graph are ordered based on node importance. In the case of road networks, node importance can be based on road type: residential roads, national roads, and motorways or highways. The basic intuition here is that closer to the source or target, we usually consider residential roads; far away from the source or target, national roads are considered; and even further away from the source or the target, it makes sense to consider highways. Some other heuristics that affect the node importance include the maximum speed, toll rates, the number of turns, etc.

Once the node order is determined, the vertex set or nodes are ordered by importance: $V = \{1,2,3\ldots,n\}$. Nodes are contracted or removed in this order using the following procedure:

```
for each pair (u,v)and (v,w)of edges:
   if <u,v,w> is a unique shortest path then
      add shortcut(u,w) with weight ω(<u,v,w>)or ω(<u,w>)+ω(<v,w>)
```

As illustrated in figure 4.23, node v can be contracted from graph G. If necessary, a shortcut or edge with a cost of 5 should be added to ensure that the shortest distance between u and w is preserved or remains the same, even after v has been contracted. Contracting a node v means replacing the shortest paths going through v with shortcuts. The new graph is called an *overlay graph* or an *augmented graph* (i.e., a graph with an augmented set of edges). This graph contains the same set of vertices as the initial graph and all the edges, plus all the added edges (shortcuts) used to preserve shortest distances in the original graph.

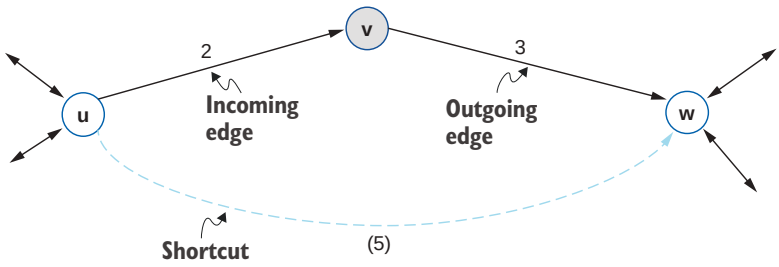


Figure 4.23 Node contraction operation—the number in brackets denotes the cost of the added shortcut.

When contracting node v, no shortcut is needed if there is a path P between u and w with $w(P) <= w(<u,v,w>)$. This path is called a *witness path*, as shown in figure 4.24.

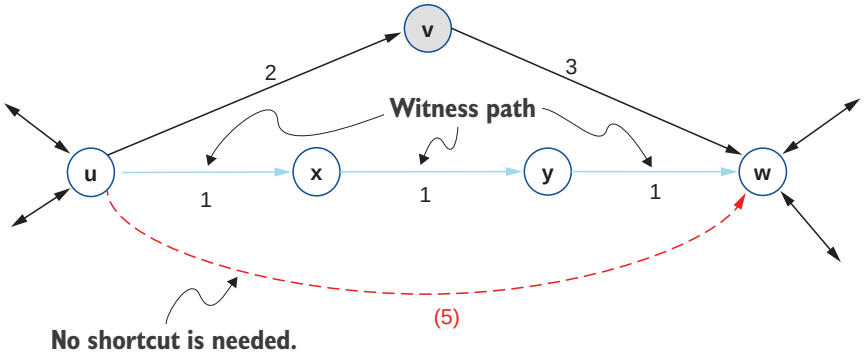


Figure 4.24 Witness path—there is another path from *u* to *w* that is shorter, so no shortcut is needed when contracting *v*.

During the CH preprocessing phase, since the nodes are ordered based on importance, a node can be iteratively contracted, and an additional shortcut arc can be added to preserve short distances and to form an augmented graph. We end up with a contraction hierarchy, with one overlay graph for each individual node. This preprocessing is done offline, and the augmented graph is used later during the query phase.

Let's consider a simple graph with four nodes. Figure 4.25 shows the steps of the contraction process. We will contract each node following the order of importance from the least to the most important node (i.e., following the importance or hierarchy level from 1 to n). This process will form shortcuts, which will allow us to search the graph much faster, as we can ignore nodes that have been pruned. The initial graph is shown in figure 4.25a:

1. By contracting the least important node, node 1, nothing happens, as the shortest path between the neighboring nodes 2 and 4 does not pass by node 1 (figure 4.25b).
2. Moving forward and contracting the next most important node, node 2, we have now changed the shortest path for 1→3, 1→4, and 3→4. We can encode these shortest paths by creating new edges (shortcuts). The numbers in brackets denote the costs of the added shortcuts (figure 4.25c).
3. Contracting node 3 does not cause any change, as there is a shorter path between nodes 2 and 4 that does not pass by node 3 (figure 4.25d).
4. We do not need to contract node 4, as it is the last node in the graph (figure 4.25e).

The generated overlay graph after the contraction process is shown in figure 4.25f. The nodes are ordered based on importance.

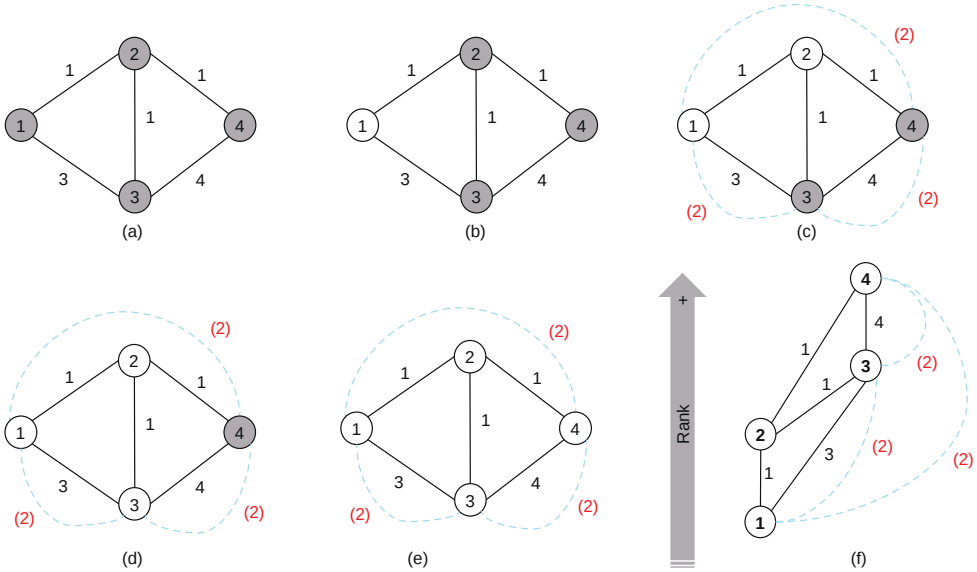


Figure 4.25 An example of the CH preprocessing phase

The order of contraction does not affect the success of CH, but it will affect the preprocessing and query time. Some contraction ordering systems minimize the number of shortcuts added in the augmented graph and thus the overall running time.

To begin, we need to use some notion of importance and keep all the nodes in a priority queue by decreasing importance. Edge difference, lazy updates, the number of contracted neighbors, and shortcut cover (all explained shortly) are examples of importance criteria. The *importance* of each node in the graph is its *priority*. This metric guides the order in which nodes are contracted. This *priority* term is dynamic and is continuously updated as nodes are contracted. The typical importance criteria include

- *Lazy updates*—The priority of the node on top of the priority queue (i.e., the node with the smallest priority) is updated before it is removed. If this node is still on top after the update, it will be contracted. Otherwise, the new topmost node will be processed in the same way.
- *Edge difference* (ED)—The ED of a node is the number of edges that need to be added versus the number of edges to be removed. We want to minimize the number of edges added to the augmented graph. For a node v in a graph, assume that
  - $\text{in}(v)$ is the incoming degree (i.e., the number of edges coming into a node)
  - $\text{out}(v)$ is the outgoing degree (i.e., the number of outgoing edges emanating from a node)

- deg(v) is the total degree of the node, which is the sum of its in and out degrees so deg(v) = in(v) + out(v)
- add(v) is the number of added shortcuts
- ED(v) is the edge difference after contracting node v, and it's given by ED(v) = add(v) – deg(v)

The next two figures show how the edge difference is calculated and used to choose between contracting an edge node like A (figure 4.26) and a hub node like E (figure 4.27)

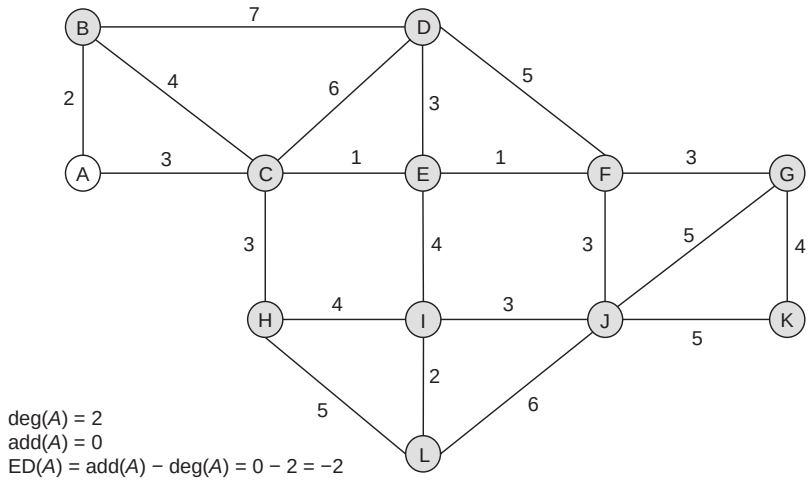


Figure 4.26 Edge difference in the case of edge node *A*

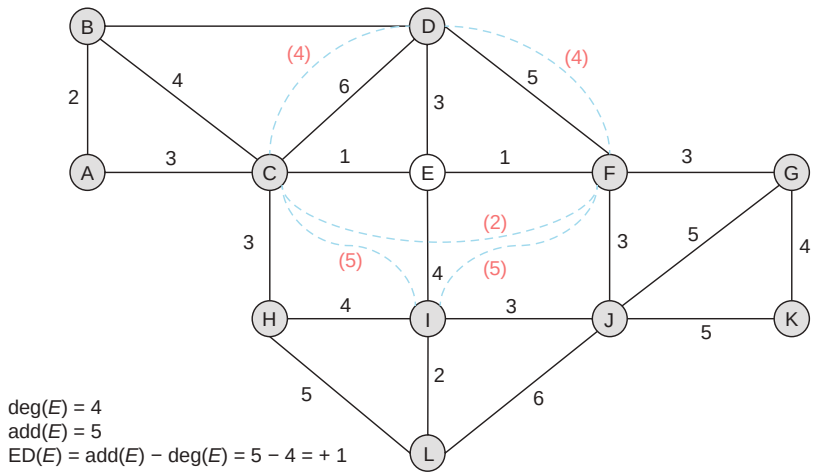


Figure 4.27 Edge difference in case of a hub node. The numbers in brackets denote the cost of the added shortcuts.

- *Number of contracted neighbors*—This reflects how nodes are spread across the map. It is better to avoid contracting all nodes in a small region of the graph and to ensure uniformity during the contraction process. We first contract the node with the smallest number of contracted neighbors.
- *Shortcut cover*—This method approximates how unavoidable the node is (e.g., a bridge connecting two parts of a city across a river). It represents the number of neighbors of a node, and thus how many shortcuts we'll need to create to or from them after contracting the node, because they're unavoidable nodes. Nodes with a smaller number of shortcut covers are contracted first.

The priority of a node estimates the attractiveness of contracting the node and can be a weighted linear combination of the previously described importance criteria, such as edge difference, number of contracted neighbors, and shortcut cover. The least important node is extracted in each iteration. The contraction process may affect the importance of a node, so we need to recompute this node importance. The newly updated importance is then compared with the node on the top of the priority queue (with the lowest importance) to decide whether or not this node needs to be contracted. The node with the smallest updated importance is always contracted.

THE CH QUERY PHASE

During the CH query phase, we apply bidirectional Dijkstra's to find the shortest path between the source and the target, as follows (figure 4.28):

- Dijkstra's algorithm from the source only considers edges u,v where level(u) > level(v), so you only want to relax nodes with a higher level than the node you have relaxed at that iteration. This is called the *upward graph*. In the context of Dijkstra's algorithm, *relaxing* a node refers to the process of updating the estimated distance or cost to reach that node from a source node by considering shorter paths through neighboring nodes. This helps refine the estimate of the shortest path to the node from the source.
- Dijkstra's algorithm from the target only considers edges u,v where level(u) < level(v), so you only want to relax nodes with a lower level than the node you have relaxed at that iteration. This is called the *downward graph*.

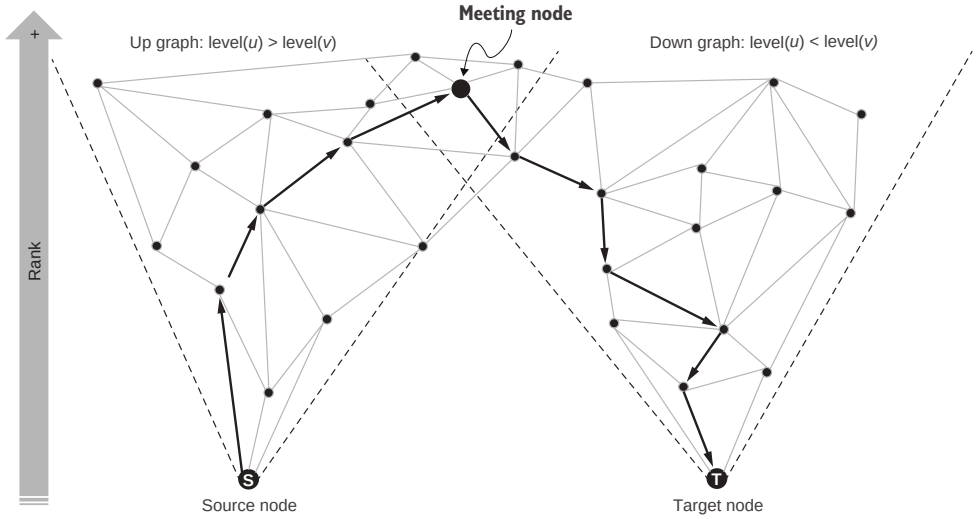


Figure 4.28 CH query phase

A CH EXAMPLE

Consider the following network with an arbitrary node ordering (figure 4.29). The numbers in the circles are the order in which the nodes will be contracted. The numbers on the edges represent the costs.

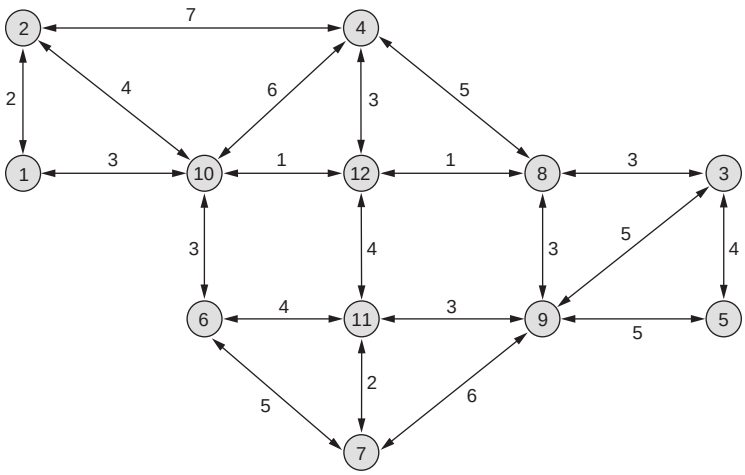


Figure 4.29 A CH example

Let's run the CH algorithm on this graph to get the shortest path between two nodes in this road network. The following steps show how to apply the CH algorithm:

1 *Contracting node 1*—There's no need to add a shortcut, as we do not lose a shortest path (figure 4.30).

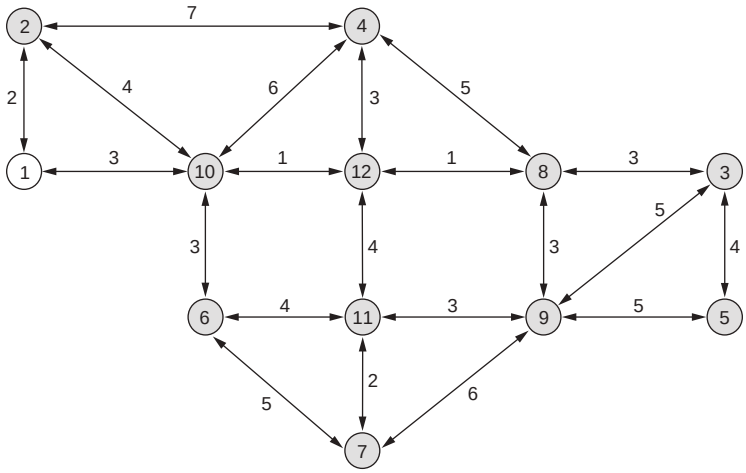


Figure 4.30 Graph contraction using an arbitrary node ranking—contracting node 1

2 *Contracting node 2*—There's no need to add a shortcut, as we do not lose a shortest path (figure 4.31).

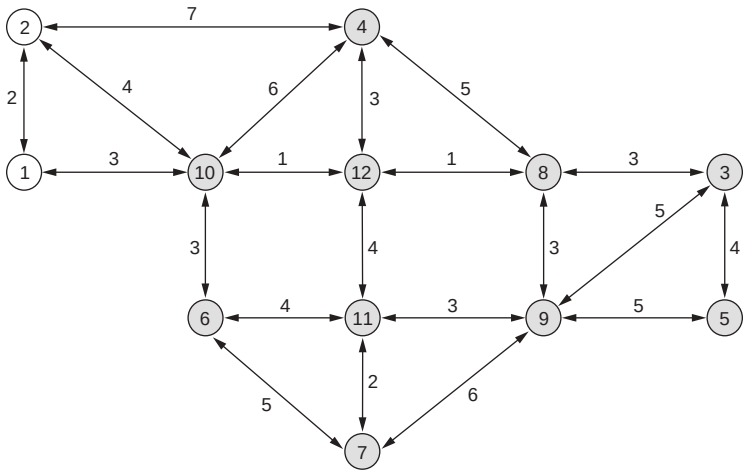


Figure 4.31 Graph contraction using an arbitrary node ranking—contracting node 2

3 *Contracting node 3*—A shortcut needs to be added to preserve the shortest path between 8 and 5, as there is no witness path. The cost of the added arc is 7 (figure 4.32).

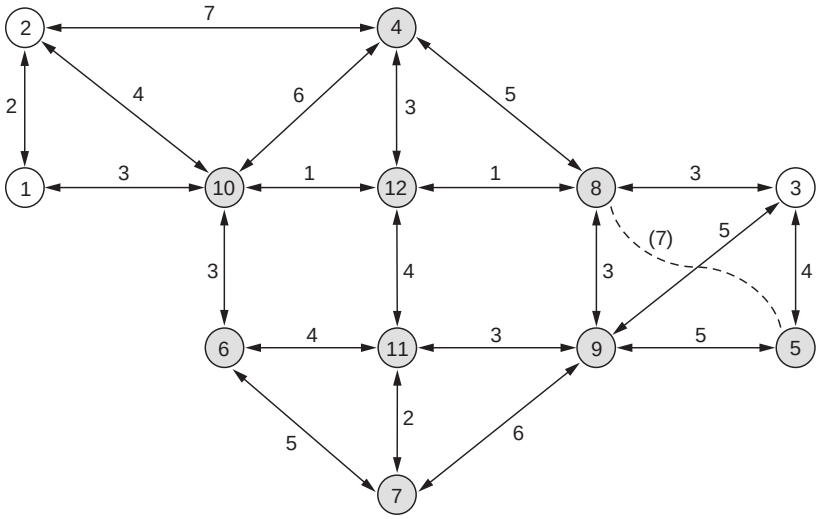


Figure 4.32 Graph contraction using an arbitrary node ranking—contracting node 3

4 *Contracting node 4*—No shortcuts need to be added (figure 4.33).

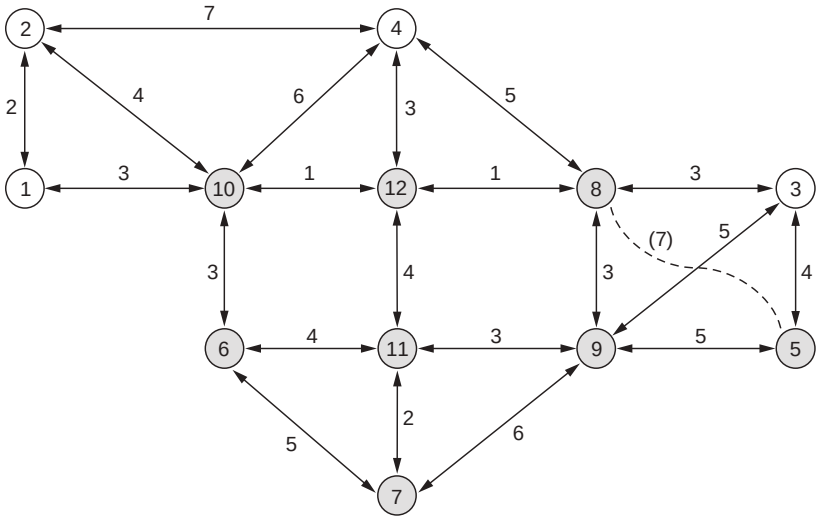


Figure 4.33 Graph contraction using an arbitrary node ranking—contracting node 4

5 *Contracting node 5*—No shortcuts need to be added (figure 4.34).

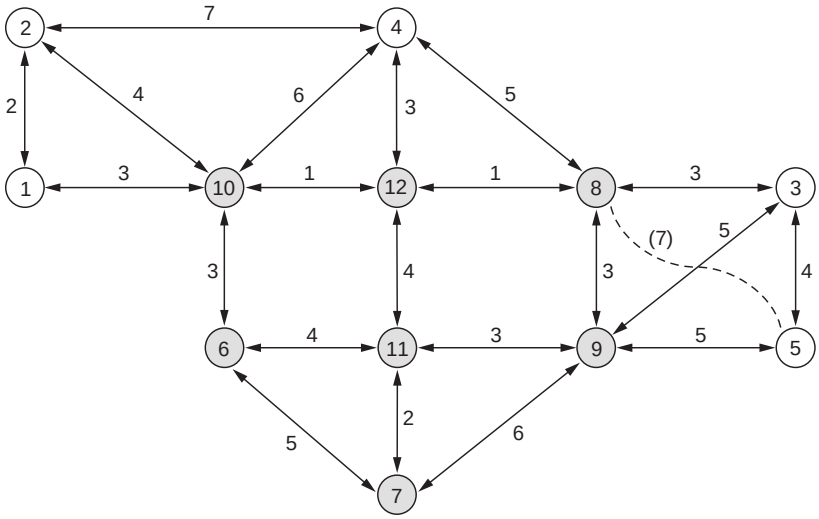


Figure 4.34 Graph contraction using an arbitrary node ranking—contracting node 5

6 *Contracting node 6*—No shortcuts need to be added, as there is a witness path between 7 and 10, which is 7-11-12-10 (figure 4.35).

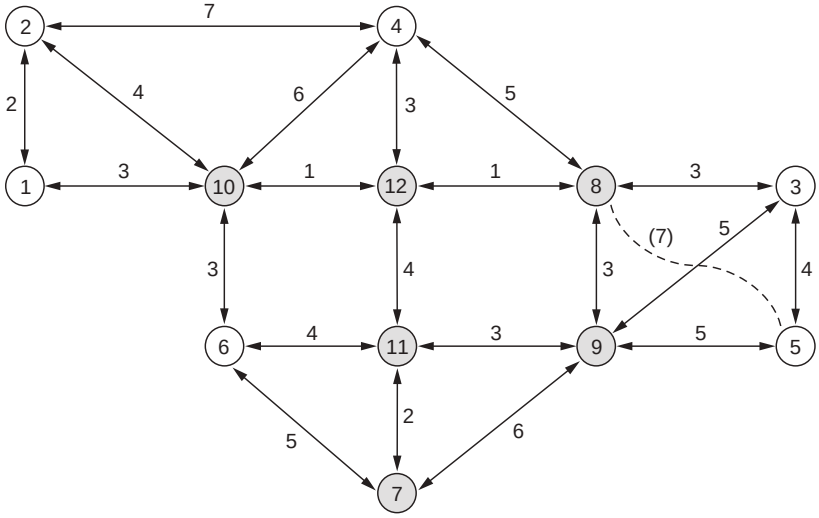


Figure 4.35 Graph contraction using an arbitrary node ranking—contracting node 6

7 *Contracting node 7*—No shortcuts need to be added (figure 4.36).

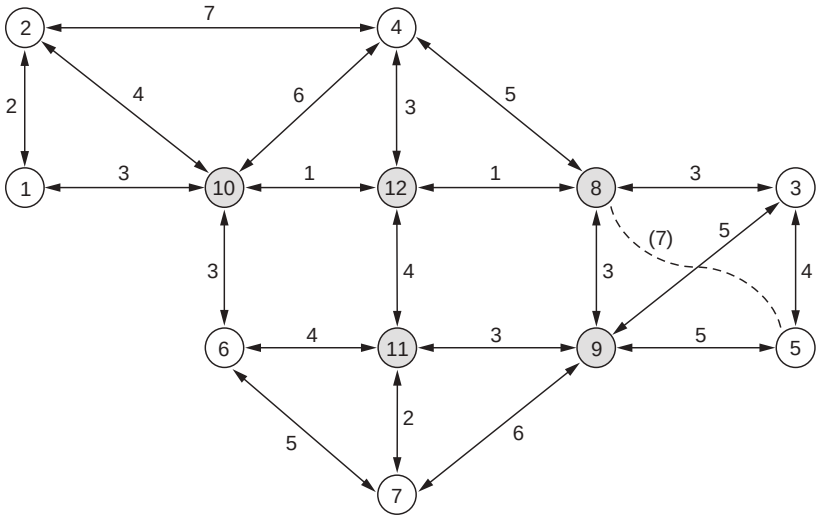


Figure 4.36 Graph contraction using an arbitrary node ranking—contracting node 7

8 *Contracting node 8*—A shortcut needs to be added to preserve the shortest path between 9 and 12 (figure 4.37).

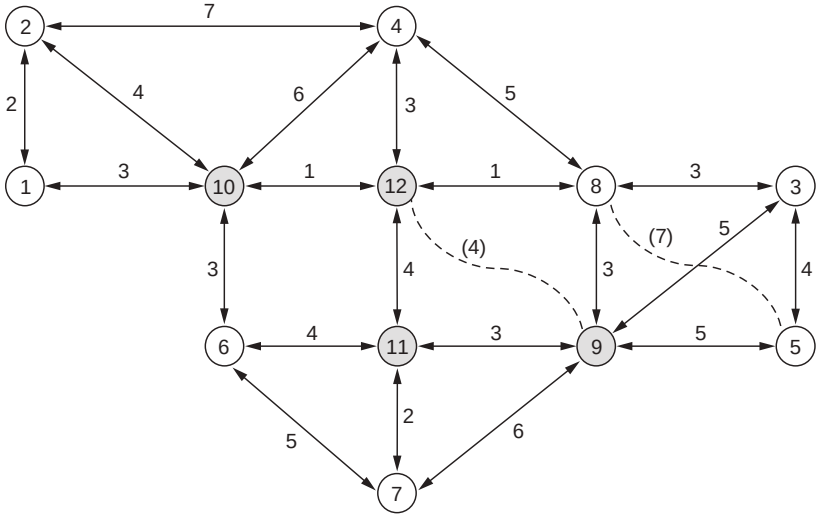


Figure 4.37 Graph contraction using an arbitrary node ranking—contracting node 8

9 *Contracting node 9*—No shortcuts need to be added (figure 4.38).

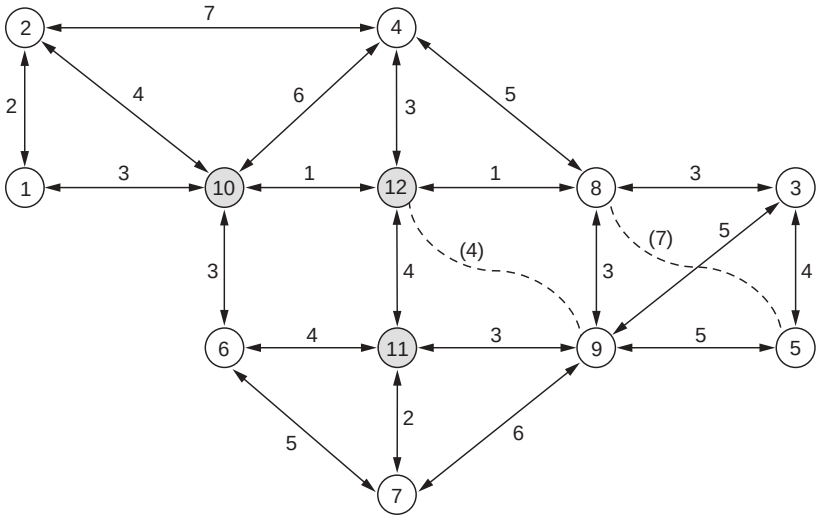


Figure 4.38 Graph contraction using an arbitrary node ranking—contracting node 9

10 *Contracting node 10*—No shortcuts need to be added (figure 4.39).

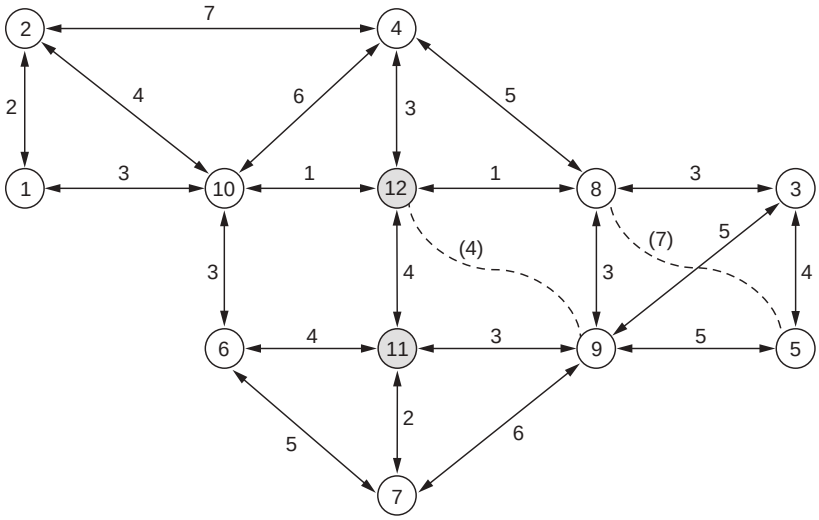


Figure 4.39 Graph contraction using an arbitrary node ranking—contracting node 10

11 *Contracting node 11*—No shortcuts need to be added (figure 4.40).

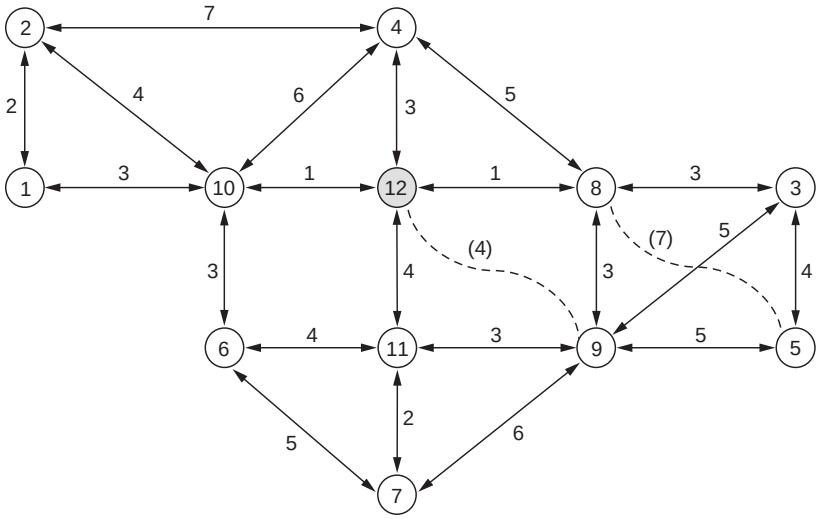


Figure 4.40 Graph contraction using an arbitrary node ranking—contracting node 11

12 *Contracting node 12*—No shortcuts need to be added (figure 4.41).

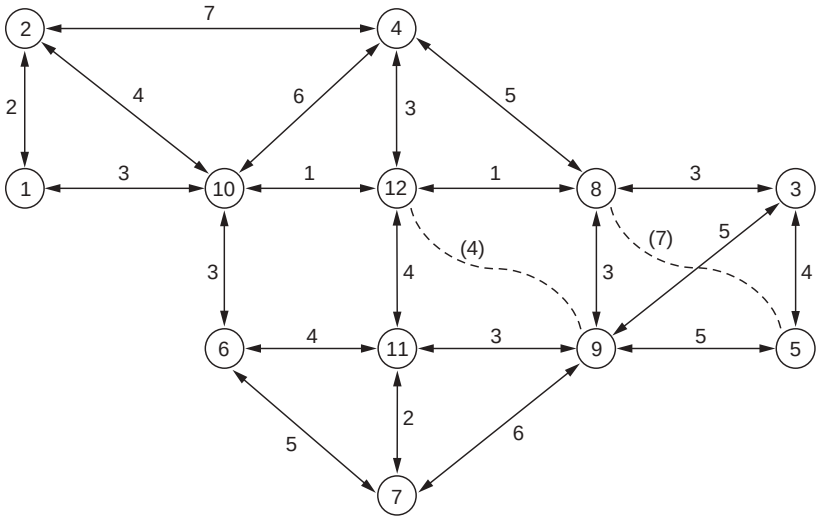


Figure 4.41 Graph contraction using an arbitrary node ranking—contracting node 12

The contracted graph can now be queried using a bidirectional Dijkstra's search. In the following figures, the numbers in brackets denote the cost of the added shortcut.

The upward graph in figure 4.42 shows the forward Dijkstra's search from the source to the target. The solid lines represent the visited edges, and the bold solid lines represent the shortest path between the source node and the meeting node.

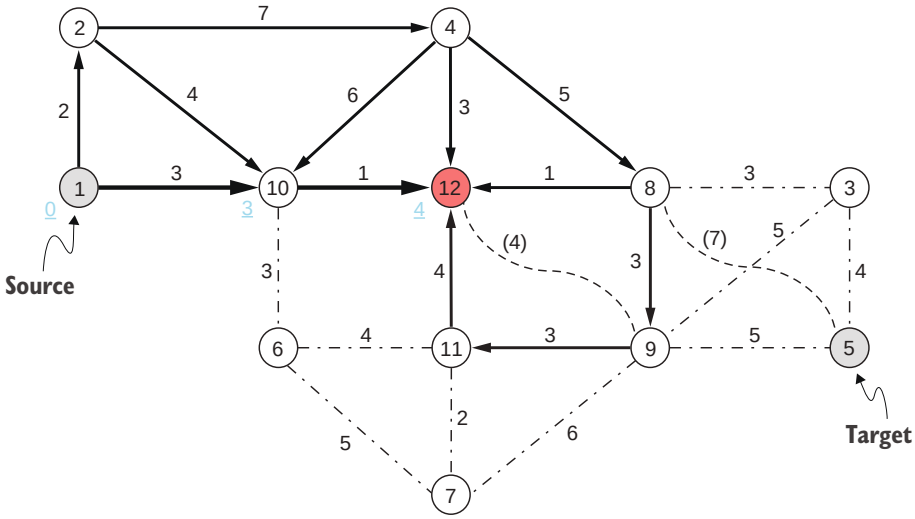


Figure 4.42 Solving a road network problem using the CH algorithm—upward graph

The downward graph in figure 4.43 shows the backward Dijkstra's search from the target to the source. The solid lines represent the visited edges, and the bold solid lines represent the shortest path between the target node and the meeting node.

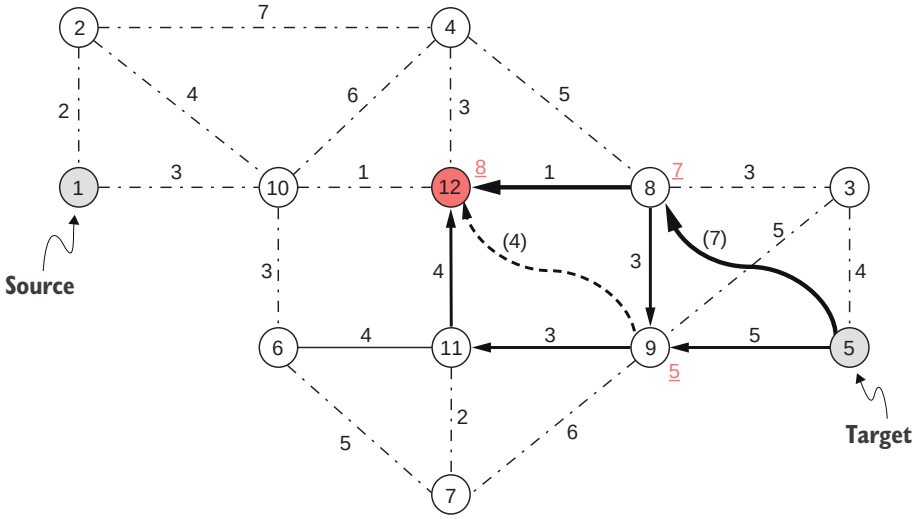


Figure 4.43 Solving a road network problem using the CH algorithm—downward graph

The minimum is at node 12 (4 + 8 = 12), so node 12 is the meeting point (figure 4.44).

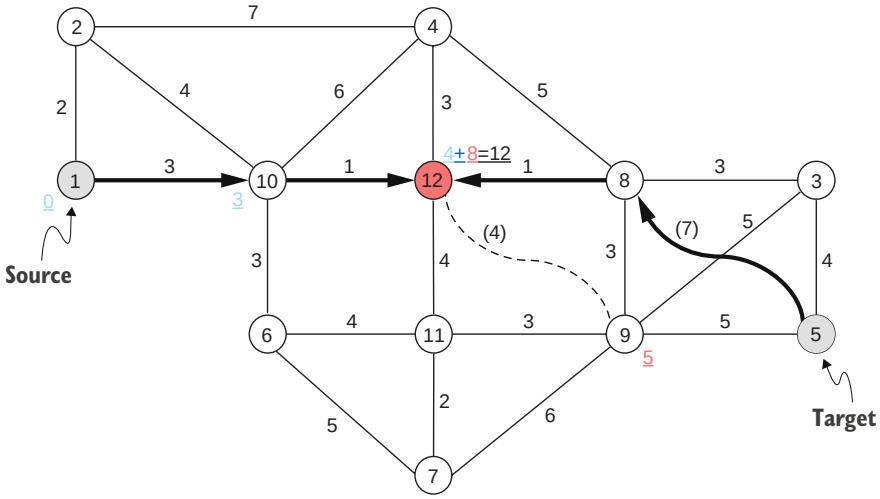


Figure 4.44 Solving a road network problem using the CH algorithm—meeting point

The shortest path will be 1-10-12-8-5. However, this path contains a shortcut (5-8). The actual arc (8-3-5) needs to be unpacked according to the shortcut pointer (node 3) stored during the contraction process. The actual shortest path is 1-10-12-8-3-4 with a cost of 12 (figure 4.45).

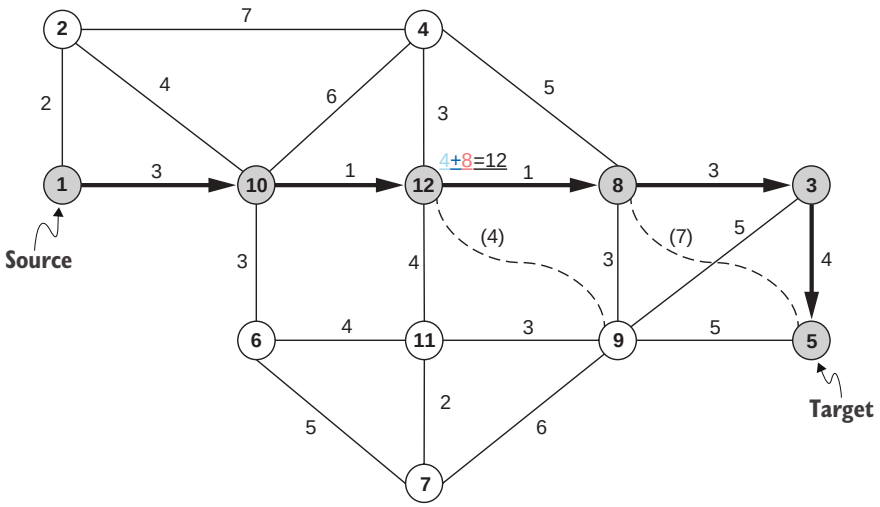


Figure 4.45 Solving a road network problem using the CH algorithm—shortest path

Listing 4.6 shows the implementation in Python. Note that the code for graph initialization has been omitted here, as it is similar to previous examples, but it can be viewed in the full listing in the book's GitHub repo. Likewise, the code for graph visualization is also in the full listing.

Listing 4.6 Contraction hierarchy with predetermined node order

```
import networkx as nx

shortcuts = {}
shortest_paths = dict(nx.all_pairs_dijkstra_path_length(G,
➥ weight="weight"))
current_G = G.copy()
for node in G.nodes:
    current_G.remove_node(node)
    current_shortest_paths = dict(
        nx.all_pairs_dijkstra_path_length(current_G, weight="weight")
    )
    for u in current_shortest_paths:
        if u == node:
            continue
        SP_contracted = current_shortest_paths[u]
        SP_original = shortest_paths[u]
        for v in SP_contracted:
            if u == v or node == v:
                continue
            if (
                SP_contracted[v] != SP_original[v]
                and G.has_edge(node, u)
                and G.has_edge(node, v)
            ):
                G.add_edge(u, v, weight=SP_original[v],contracted=True)
                shortcuts[(u,v)] = node
```

Copy the main graph so that the nodes are only removed from the copy, not the main graph.

Contract the node by removing it from the copied graph.

Recalculate the shortest path matrix, now with the node contracted.

Add a shortcut edge to replace the changed shortest path, and keep track of it so we can uncontract it when querying later.

You will notice that the preceding code creates two shortcut edges for each contraction, one from u to v, and one in reverse from v to u. As we are using an undirected graph, this duplication has no effect, since the edge (u, v) is the same as the edge (v, u).

Querying the generated graph requires a simple modified bidirectional Dijkstra's search, where neighbor nodes are disqualified for expansion if they are lower in the hierarchy than the current node. For the purposes of this book, we will use `networkx.algorithms.shortest_paths.weighted.bidirectional_dijkstra`, with a slight change (only nodes of higher hierarchy than the current node can be explored). As a continuation of listing 4.6, the following code snippet shows the querying process. The full code for the modified algorithm can be found in listing 4.6 in the book's GitHub repo:

Run bidirectional Dijkstra's using NetworkX.

```
sln = bidirectional_dijkstra(G, 1, 5, hierarchy, weight="weight")

uncontracted_route = [sln.result[0]]
for u, v in zip(sln.result[:-1], sln.result[1:]): ]]
    if (u, v) in shortcuts: ]]
        uncontracted_route.append(shortcuts[(u, v)])
    uncontracted_route.append(v)
```

Unpack any edges that are marked as contracted, and generate the unpacked route.

The preceding code will generate an unpacked route that can be visualized as in figure 4.46.

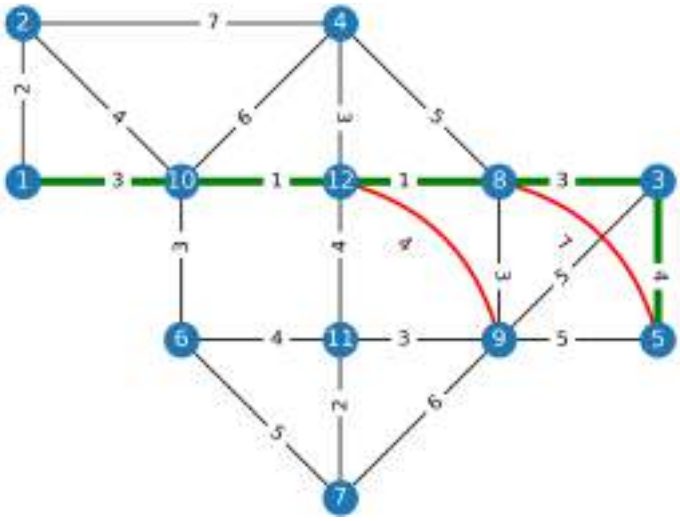


Figure 4.46 The solution path after unpacking the contracted edges. The original route returned by bidirectional Dijkstra's passes through the contracted edge (8,5), which is then unpacked into (8,3) and (3,5).

Contraction hierarchies expend a great deal of processing time on the preprocessing phase, but a correctly pruned graph (i.e., where the node contraction order is good) allows for much faster queries. While the small reduction in search space is negligible on a graph with 21 nodes, some graphs can be pruned up to 40%, resulting in significant cost and time savings when querying. In the example from listing 4.6, the search from node 1 to node 5 explores 11 nodes, as compared to the original 16 nodes in a normal bidirectional Dijkstra's. That's almost a 33% reduction!

4.4 Applying informed search to a routing problem

Let's look again at the University of Toronto routing problem introduced in section 3.5. We need to find the shortest path from the King Edward VII Equestrian statue at Queen's Park to the Bahen Centre for Information Technology. The search space is represented by a road network in which the intersections and points of interest (including the origin and destination) are nodes, and edges are used to represent road segments with weight (e.g., distance, travel time, fuel consumption, number of turns, etc.). Let's look at how we can find the shortest path using the informed search algorithms discussed in this chapter.

4.4.1 Hill climbing for routing

Listing 4.7 uses two functions from `optalgotools.routing` that generate random and child routes. While the actual HC algorithm is deterministic, the randomized initial route means that different results can be achieved over different runs. To counter this, we'll use a higher n value, which allows a broader diversity of children routes, so an optimal (or near optimal) solution will more likely be achieved.

Listing 4.7 U of T routing using the hill climbing algorithm

```
def Hill_Climbing(G, origin, destination, n=20):
    time_start = process_time()
    costs = []

    current = randomized_search(G, origin.osmid, destination.osmid)
    costs.append(cost(G, current))

    neighbours = list(islice(get_child(G, current), n))
    space_required = getsizeof(neighbours)
    shortest = min(neighbours, key=lambda route: cost(G, route))

    while cost(G, shortest) < cost(G, current):
        current = shortest
        neighbours = list(islice(get_child(G, current), n))
        shortest = min(neighbours, key=lambda route: cost(G, route))
        costs.append(cost(G, current))

    route = current
    time_end = process_time()
    return Solution(route, time_end - time_start, space_required, costs)
```

Track time and costs for comparison.

Generate an initial route randomly.

Get k neighbors (children).

While the implementation in listing 4.7 is deterministic, the initial route is still randomized. That means it is possible to get different results across runs. Hill climbing will return some decent results, as there are few local optimal points in the route function. However, larger search spaces will naturally have more local maxima and plateaus, and the HC algorithm will get stuck quickly.

Figure 4.47 shows a final solution of 806.892 m, which happens to be the same as the result generated by Dijkstra's algorithm in chapter 3 (an optimal solution).

Figure 4.47 Shortest path solution generated using hill climbing. The solution shown here uses an *n* value of 100, which increases the total processing time but returns better and more consistent results.

4.4.2 Beam search for routing

A beam search for routing will follow much the same format as the HC search, with the exception that a "beam" of solutions is kept for comparison at each iteration. The full code for listing 4.8, with the graph initialization and visualization, is in the book's GitHub repo.

Listing 4.8 U of T routing using the beam search algorithm

```
def get_beam(G, beam, n=20):
    new_beam = []
    for route in beam:
        neighbours = list(islice(get_child(G, route), n))
        new_beam.extend(neighbours)
    return new_beam

def Beam_Search(G, origin, destination, k=10, n=20):
    start_time = process_time()
    seen = set()
    costs = []
    beam = [randomized_search(G, origin.osmid, destination.osmid) for _ in
    ➥  range(k)]

    for route in beam:
        seen.add(tuple(route))

    pool = []
    children = get_beam(G, beam, n)
    costs.append([cost(G, r) for r in beam])
    for r in children:
        if tuple(r) in seen:
            continue
        else:
            pool.append(r)
            seen.add(tuple(r))
    pool += beam
    space_required = getsizeof(pool)
    last_beam = None
    while beam != last_beam:
        last_beam = beam
        beam = heapq.nsmallest(k, pool, key=lambda r: cost(G, r))

        for route in beam:
            seen.add(tuple(route))

        pool = []
        children = get_beam(G, beam, n)
        costs.append([cost(G, r) for r in beam])
        for r in children:
            if tuple(r) in seen:
                continue
            else:
                pool.append(r)
                seen.add(tuple(r))
```

Generate child routes for each route in the beam.

Initialize empty sets to keep track of visited nodes and path costs.

The seen routes must be converted to a tuple so they are hashable and can be stored in a set.

Keep the k best routes at each iteration until generating new beams no longer finds better solutions.

```
        pool += beam
        space_required = (
            getsizeof(pool) if getsizeof(pool) > space_required else
            ➥ space_required
        )
    route = min(beam, key=lambda r: cost(G, r))
    end_time = process_time()
    return Solution(
        route, end_time - start_time, space_required, np.rot90(costs))
```

The final route is the best route in the last beam.

Return the final route, its cost, processing time, and space required.

Beam searches for routing are particularly costly, as they require multiple child routes to be generated for each beam. Like HC, generating more children results in a broader penetration of the search space, and thus is more likely to return a solution that is closer to or reaches the optimal solution. Figure 4.48 shows a final solution generated by beam search.

Figure 4.48 Shortest path using a beam search algorithm. This solution was generated using $k = 10$ and $n = 20$, which means that 20 routes were generated for each route in the beam, and the top 10 routes were kept for each new beam. Lower k and n values will improve processing time but reduce the likelihood of generating a near-optimal or optimal solution.

4.4.3 A* for routing

The next listing shows how we can use A* search to find the shortest route between two points of interest.

Listing 4.9 U of T routing using A*

```
import osmnx
from optalgotools.routing import (cost, draw_route, astar_heuristic)
from optalgotools.structures import Node
from optalgotools.algorithms.graph_search import A_Star
from optalgotools.utilities import haversine_distance

reference = (43.661667, -79.395)
```

Set up King's College Cir, Toronto, ON as a reference.

```
G = osmnx.graph_from_point(reference, dist=300, clean_periphery=True,
```

```
➥ simplify=True)
origin = (43.664527, -79.392442)
destination = (43.659659, -79.397669)

origin_id = osmnx.distance.nearest_nodes(G, origin[1], origin[0])
destination_id = osmnx.distance.nearest_nodes(G, destination[1],
➥ destination[0])

origin = Node(graph=G, osmid=origin_id)
destination = Node(graph=G, osmid=destination_id)

solution = A_Star(G, origin, destination, astar_heuristic,
➥ heuristic_kwargs={"measuring_dist": haversine_distance})
route = solution.result
print(f"Cost: {cost(G,route)} m")
print(f"Process time: {solution.time} s")
print(f"Space required: {solution.space} bytes")
print(f"Explored nodes: {solution.explored}")
draw_route(G,route)
```

Create a graph.

Set up the King Edward VII equestrian statue as the origin.

Set up the Bahen Centre for Information Technology at U of T as the destination.

Get the osmid of the nearest nodes to the points.

Convert the source and destination nodes to Node.

Find the shortest path using A*.

Print the cost, processing time, space required, and explored nodes, and draw the final route.

The optimality of the A* search depends on the heuristic used. In this case, the solution returned is not optimal, but the incredibly high processing speed achieved is more important for most applications. Figure 4.49 shows a final solution generated by A* search.

Figure 4.49 Shortest path using the A* algorithm. Better heuristic functions that closely approach the actual costs from any node to the goal will return better results.

4.4.4 Contraction hierarchies for routing

In order to run CH on the road network graph, we first need to rank the nodes by importance and then contract the graph. For this example, we are selecting edge difference (ED) as our measure of node importance.

Listing 4.10 U of T routing using CH

```
def edge_differences(G, sp):
    ed = {}
    degrees = dict(G.degree)
    for node in G.nodes:
        req_edges = 0
        neighbours = list(G.neighbors(node))

        if len(neighbours)==0: ed[node] = - degrees[node]

        for u, v in G.in_edges(node):
            for v, w in G.out_edges(node):
                if u == w: continue
                if v in sp[u][w]:
                    req_edges += 1

        ed[node] = req_edges - degrees[node]

    return dict(sorted(ed.items(), key=lambda x: x, reverse=True))
```

Some nodes are essentially dead ends, where they have no outbound edges. These nodes have an ED equal to their degree.

We can ignore two-way edges—an inbound edge and an outbound edge that originate and terminate at the same node.

The edge difference is the difference between edges that need to be added to the graph and the degree of the node.

Contracting the graph is as simple as adding an edge for every shortest path that gets altered by the contraction. The full code for graph contraction can be found in the book's GitHub repo. Contracted edges are marked with an attribute called *midpoint*, which stores the ID of the node that was contracted. Following a modified bidirectional Dijkstra's similar to that used in listing 4.6, the final route can be easily unpacked using the following snippet of code:

```
def unpack(G, u,v):
    u = int(u)
    v = int(v)
    if "midpoint" in G[u][v][0]:
        midpoint = G[u][v][0]["midpoint"]
        return unpack(G,u,midpoint) + unpack(G,midpoint, v)
    return [u]

route = []
for u,v in zip(solution.result[:-1], solution.result[1:]):
    route.extend(unpack(G,u,v))
route += [solution.result[-1]]
print(route)
```

For every midpoint unpacked, recursively unpack the resulting two edges, as some contracted edges may contain other contracted edges.

Unpack every node pair in the contracted route.

The GitHub repo also contains the full Python implementation of CH for routing. The route generated is identical to that shown by a normal bidirectional Dijkstra's algorithm (such as in chapter 3). If you will recall, running the normal bidirectional Dijkstra's in chapter 3 yielded a result where 282 nodes were explored during the

search. For our CH result, only 164 nodes were explored, which means more than a 40% reduction of search space! Thus, while the optimality of the algorithm remains unchanged, contraction hierarchies allow for much bigger spaces to be searched in a reasonable amount of time.

Table 4.1 compares the search algorithms discussed in this chapter when applied to the U of T routing problem. A similar table can be found in chapter 3 for blind search algorithms.

Table 4.1 Comparing informed search algorithms in terms of time and space complexities, where *b* is the branching factor, *w* is the beam width, *d* is the shallowest graph depth, *E* is the number of edges, and *V* is the number of vertices

Algorithm	Cost (meters)	Time (s)	Space (bytes)	Explored	Time complexity	Space complexity
Hill climbing	806.892	21.546	976	400 nodes	$O(\infty)$	$O(b)$
Beam search	825.929	44.797	1,664	800 nodes	$O(wd)$	$O(\text{wb})$
A* search	846.92	0.063	8,408	80 nodes	$O(b^d)$	$O(b^d)$
CH with bidirectional Dijkstra's	806.892	0.0469	72	164 nodes	$O(E + V \log V)$	$O(b^d/2)$

NOTE The time listed for CH with bidirectional Dijkstra's is only for querying. Remember that the preprocessing step is usually quite costly. In this case, contracting the road network of 404 nodes took around 24.03125 seconds.

While hill climbing and beam search produced respectable results, they were too costly in terms of time to be useful for larger graphs. A* gives the fastest results but a non-optimal heuristic function, and it required excessive space for the heuristic values, so it has its own disadvantages. CH with bidirectional Dijkstra's is the only algorithm in table 4.1 that guarantees optimality, but the costly preprocessing step may not be suitable for all applications.

When comparing search algorithms, it is important to be aware of the constraints for any given problem and to select an algorithm based on those constraints. For example, certain implementations of hill climbing may result in rapid exit conditions. If the goal is to maximize the number of problems solved (and if local maxima are an acceptable result), HC algorithms result in quick solutions that have some degree of optimality. On the other hand, preprocessing-heavy algorithms like CH offer incredibly low space costs (even more so when implemented with a bidirectional search), as well as rapid searches for guaranteed optimal solutions (if using Dijkstra's algorithm). For high-volume usage implementations where preprocessing is not a concern (e.g., Uber), contraction hierarchies are a viable choice. In fact, the osrm package used in this book is primarily based on an implementation of contraction hierarchies.

Pandana, a Python library for network analysis, uses CH to calculate shortest paths and fast travel accessibility metrics. In Pandana, the backend code for CH is in C++ but

can be accessed using Python. Pyrosm is another Python library for reading and parsing OpenStreetMap data. It is similar to OSMnx but faster, and it works with Pandana.

The next listing is a snippet of code that calculates the shortest distances to an amenity of interest in a selected city using the CH algorithm implemented in Pandana. The complete code is available in the book's GitHub repo.

Listing 4.11 Using CH to calculate the shortest distances to amenities

```
from pyrosm import OSM, get_data
import numpy as np
import matplotlib.pyplot as plt

osm = OSM(get_data("Toronto"))
nodes, edges = osm.get_network(network_type="driving", nodes=True)

hospitals = osm.get_pois({"amenity": ["hospital"]})

G = osm.to_graph(nodes, edges, graph_type='pandana')

hospitals['geometry'] = hospitals.centroid #E
hospitals = hospitals.dropna(subset=['lon', 'lat'])

G.precompute(1000)

G.set_pois(category='hospitals', maxdist=1000, maxitems=10,
➥ x_col=hospitals.lon, y_col=hospitals.lat)

nearest_five = G.nearest_pois(1000, "hospitals", num_pois=5)
```

Get data for the city, region, or country of interest.

Get nodes and edges from the road network with a "driving" type.

Create a network graph.

Get points of interest for a certain amenity in the city.

Precompute distances up to 1,000 meters.

Ensure all hospitals are represented as points.

Attach hospitals to the Pandana graph.

For each node, find the distances to the five closest hospitals up to 1,000 meters away.

In this example, OpenStreetMap is used to get data on the city of Toronto, and a subset is created to contain data on the city's hospitals. A Pandana object is then created, and the range queries are precomputed, given a horizon distance (e.g., 1,000 meters) to represent the reachable nodes within this distance. For each node in the network, we can find distances to the five closest hospitals up to 1,000 meters away using the fast CH algorithm implemented in Pandana.

In the next part of the book, we'll look at trajectory-based algorithms starting with the simulated annealing algorithm and then the tabu search algorithm. These algorithms improve local search and are less susceptible to getting stuck in local optima than the previously discussed greedy algorithms, which only accept improving moves.

Summary

- Informed search algorithms use domain-specific knowledge or heuristic information to streamline the search process while striving for optimal solutions or accepting near-optimal ones if necessary.
- Informed search algorithms can be used to solve minimum spanning tree (MST) problems and to find the shortest path between two nodes in a graph.

- The Borůvka algorithm, Jarník-Prim algorithm, and Kruskal algorithm are informed search algorithms for solving MST problems. An MST is a tree that contains the least weight among all the other spanning trees of a connected weighted graph. Kruskal's algorithm is a greedy algorithm that computes the MST for an undirected connected weighted graph by repeatedly adding the next shortest edge that doesn't produce a cycle.
- Hill climbing (HC), beam search, best-first search, the A* algorithm, and contraction hierarchies (CH) are examples of informed search algorithms that can be used to find the shortest path between two nodes.
- The HC algorithm is a local greedy search algorithm that tries to improve on the efficiency of depth-first by incorporating domain-specific knowledge or heuristic information.
- Beam search expands the most promising node within a limited predefined set defined by the beam width.
- Best-first search is a greedy algorithm that always expands the node that is closest to the goal node based on heuristic information only.
- The A* algorithm is a special case of a best-first algorithm that incorporates both the actual cost and a heuristic estimate of the cost to get to the goal from a given state.
- CH is a speed-up technique for improving the performance of pathfinding. During the preprocessing phase, each node is contracted in order of importance (from least important to most important), and shortcuts are added to preserve the shortest paths. Then the bidirectional Dijkstra's algorithm is applied to the resultant augmented graph to compute the shortest path between the source node and the target node.