Graph Algorithms

7

Many programming problems can be solved by considering the situation as a graph and using an appropriate graph algorithm. In this chapter, we will learn the basics of graphs and a selection of important graph algorithms.

Section 7.1 discusses graph terminology and data structures that can be used to represent graphs in algorithms.

Section 7.2 introduces two fundamental graph traversal algorithms. Depth-first search is a simple way to visit all nodes that can be reached from a starting node, and breadth-first search visits the nodes in increasing order of their distance from the starting node.

Section 7.3 presents algorithms for finding shortest paths in weighted graphs. The Bellman–Ford algorithm is a simple algorithm that finds shortest paths from a starting node to all other nodes. Dijkstra's algorithm is a more efficient algorithm which requires that all edge weights are nonnegative. The Floyd–Warshall algorithm determines shortest paths between all node pairs of a graph.

Section 7.4 explores special properties of directed acyclic graphs. We will learn how to construct a topological sort and how to use dynamic programming to efficiently process such graphs.

Section 7.5 focuses on successor graphs where each node has a unique successor. We will discuss an efficient way to find successors of nodes and Floyd's algorithm for cycle detection.

Section 7.6 presents Kruskal's and Prim's algorithms for constructing minimum spanning trees. Kruskal's algorithm is based on an efficient union-find structure which has also other uses in algorithm design.

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate
Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_7

7.1 Basics of Graphs

In this section, we first go through terminology which is used when discussing graphs and their properties. After this, we focus on data structures that can be used to represent graphs in algorithm programming.

7.1.1 Graph Terminology

A *graph* consists of *nodes* (also called *vertices*) that are connected with *edges*. In this book, the variable n denotes the number of nodes in a graph, and the variable m denotes the number of edges. The nodes are numbered using integers $1, 2, \ldots, n$. For example, Fig. 7.1 shows a graph with 5 nodes and 7 edges.

A *path* leads from a node to another node through the edges of the graph. The *length* of a path is the number of edges in it. For example, Fig. 7.2 shows a path $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ of length 3 from node 1 to node 5. A *cycle* is a path where the first and last node is the same. For example, Fig. 7.3 shows a cycle $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$.

A graph is *connected* if there is a path between any two nodes. In Fig. 7.4, the left graph is connected, but the right graph is not connected, because it is not possible to get from node 4 to any other node.

The connected parts of a graph are called its *components*. For example, the graph in Fig. 7.5 has three components: $\{1, 2, 3\}$, $\{4, 5, 6, 7\}$, and $\{8\}$.

A *tree* is a connected graph that does not contain cycles. Figure 7.6 shows an example of a graph that is a tree.

In a *directed* graph, the edges can be traversed in one direction only. Figure 7.7 shows an example of a directed graph. This graph contains a path $3 \rightarrow 1 \rightarrow 2 \rightarrow 5$ from node 3 to node 5, but there is no path from node 5 to node 3.

In a *weighted* graph, each edge is assigned a *weight*. The weights are often interpreted as edge lengths, and the length of a path is the sum of its edge weights. For example, the graph in Fig. 7.8 is weighted, and the length of the path $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ is $1 + 7 + 3 = 11$. This is the *shortest* path from node 1 to node 5.

Two nodes are *neighbors* or *adjacent* if there is an edge between them. The *degree* of a node is the number of its neighbors. Figure 7.9 shows the degree of each node

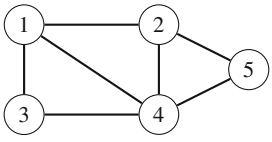


Fig. 7.1 A graph with 5 nodes and 7 edges

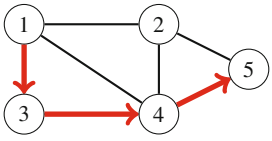


Fig. 7.2 A path from node 1 to node 5

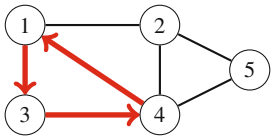


Fig. 7.3 A cycle of three nodes

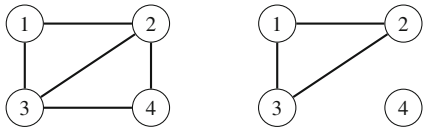


Fig. 7.4 The left graph is connected, the right graph is not

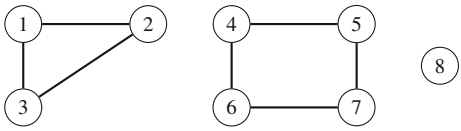


Fig. 7.5 A graph with three components

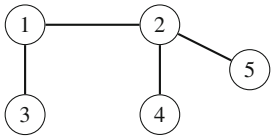


Fig. 7.6 A tree

of a graph. For example, the degree of node 2 is 3, because its neighbors are 1, 4, and 5.

The sum of degrees in a graph is always $2m$, where m is the number of edges, because each edge increases the degree of exactly two nodes by one. For this reason, the sum of degrees is always even. A graph is *regular* if the degree of every node is a constant d. A graph is *complete* if the degree of every node is $n - 1$, i.e., the graph contains all possible edges between the nodes.

In a directed graph, the *indegree* of a node is the number of edges that end at the node, and the *outdegree* of a node is the number of edges that start at the node.

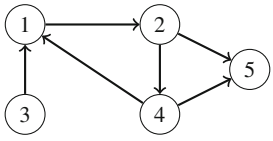


Fig. 7.7 A directed graph

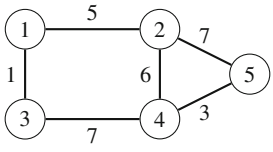


Fig. 7.8 A weighted graph

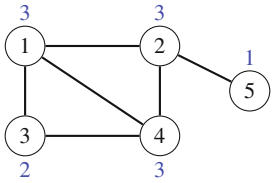


Fig. 7.9 Degrees of nodes

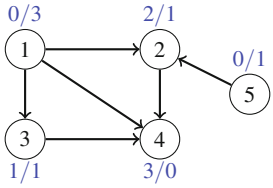


Fig. 7.10 Indegrees and outdegrees

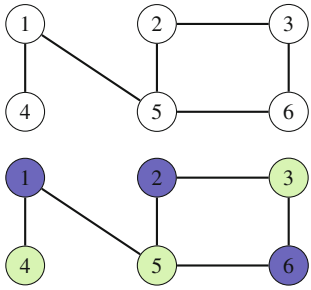


Fig. 7.11 A bipartite graph and its coloring

Figure 7.10 shows the indegree and outdegree of each node of a graph. For example, node 2 has indegree 2 and outdegree 1.

A graph is *bipartite* if it is possible to color its nodes using two colors in such a way that no adjacent nodes have the same color. It turns out that a graph is bipartite exactly when it does not have a cycle with an odd number of edges. For example, Fig. 7.11 shows a bipartite graph and its coloring.

7.1.2 Graph Representation

There are several ways to represent graphs in algorithms. The choice of a data structure depends on the size of the graph and the way the algorithm processes it. Next we will go through three popular representations.

Adjacency Lists In the adjacency list representation, each node x of the graph is assigned an *adjacency list* that consists of nodes to which there is an edge from x. Adjacency lists are the most popular way to represent graphs, and most algorithms can be efficiently implemented using them.

A convenient way to store the adjacency lists is to declare an array of vectors as follows:

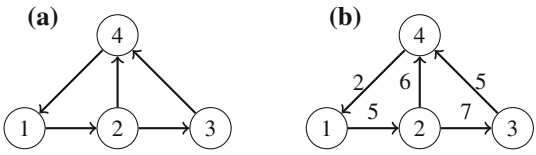


Fig. 7.12 Example graphs

```
vector<int> adj[N];
```

The constant N is chosen so that all adjacency lists can be stored. For example, the graph in Fig. 7.12a can be stored as follows:

```
adj[1].push_back(2);
adj[2].push_back(3);
adj[2].push_back(4);
adj[3].push_back(4);
adj[4].push_back(1);
```

If the graph is undirected, it can be stored in a similar way, but each edge is added in both directions.

For a weighted graph, the structure can be extended as follows:

```
vector<pair<int,int>> adj[N];
```

In this case, the adjacency list of node a contains the pair (b, w) always when there is an edge from node a to node b with weight w. For example, the graph in Fig. 7.12b can be stored as follows:

```
adj[1].push_back({2,5});
adj[2].push_back({3,7});
adj[2].push_back({4,6});
adj[3].push_back({4,5});
adj[4].push_back({1,2});
```

Using adjacency lists, we can efficiently find the nodes to which we can move from a given node through an edge. For example, the following loop goes through all nodes to which we can move from node s:

```
for (auto u : adj[s]) {
    // process node u
}
```

Adjacency Matrix An *adjacency matrix* indicates the edges that a graph contains. We can efficiently check from an adjacency matrix if there is an edge between two nodes. The matrix can be stored as an array

```
int adj[N][N];
```

where each value adj[a][b] indicates whether the graph contains an edge from node a to node b. If the edge is included in the graph, then adj[a][b] $= 1$, and otherwise adj[a][b] $= 0$. For example, the adjacency matrix for the graph in Fig. 7.12a is

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

If the graph is weighted, the adjacency matrix representation can be extended so that the matrix contains the weight of the edge if the edge exists. Using this representation, the graph in Fig. 7.12b corresponds to the following matrix:

$$\begin{bmatrix} 0 & 5 & 0 & 0 \\ 0 & 0 & 7 & 6 \\ 0 & 0 & 0 & 5 \\ 2 & 0 & 0 & 0 \end{bmatrix}$$

The drawback of the adjacency matrix representation is that an adjacency matrix contains n^2 elements, and usually most of them are zero. For this reason, the representation cannot be used if the graph is large.

Edge List An *edge list* contains all edges of a graph in some order. This is a convenient way to represent a graph if the algorithm processes all its edges, and it is not needed to find edges that start at a given node.

The edge list can be stored in a vector

```
vector<pair<int,int>> edges;
```

where each pair (a, b) denotes that there is an edge from node a to node b. Thus, the graph in Fig. 7.12a can be represented as follows:

```
edges.push_back({1,2});
edges.push_back({2,3});
edges.push_back({2,4});
edges.push_back({3,4});
edges.push_back({4,1});
```

If the graph is weighted, the structure can be extended as follows:

```
vector<tuple<int,int,int>> edges;
```

Each element in this list is of the form (a, b, w), which means that there is an edge from node a to node b with weight w. For example, the graph in Fig. 7.12b can be represented as follows[1]:

```
edges.push_back({1,2,5});
edges.push_back({2,3,7});
edges.push_back({2,4,6});
edges.push_back({3,4,5});
edges.push_back({4,1,2});
```

7.2 Graph Traversal

This section discusses two fundamental graph algorithms: depth-first search and breadth-first search. Both algorithms are given a starting node in the graph, and they visit all nodes that can be reached from the starting node. The difference in the algorithms is the order in which they visit the nodes.

7.2.1 Depth-First Search

Depth-first search (DFS) is a straightforward graph traversal technique. The algorithm begins at a starting node and proceeds to all other nodes that are reachable from the starting node using the edges of the graph.

Depth-first search always follows a single path in the graph as long as it finds new nodes. After this, it returns to previous nodes and begins to explore other parts of the graph. The algorithm keeps track of visited nodes, so that it processes each node only once.

Figure 7.13 shows how depth-first search processes a graph. The search can begin at any node of the graph; in this example we begin the search at node 1. First the search explores the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 5$, then returns back to node 1 and visits the remaining node 4.

Implementation Depth-first search can be conveniently implemented using recursion. The following function `dfs` begins a depth-first search at a given node. The function assumes that the graph is stored as adjacency lists in an array

```
vector<int> adj[N];
```

and also maintains an array

[1] In some older compilers, the function `make_tuple` must be used instead of the braces (e.g., `make_tuple(1,2,5)` instead of `{1,2,5}`).

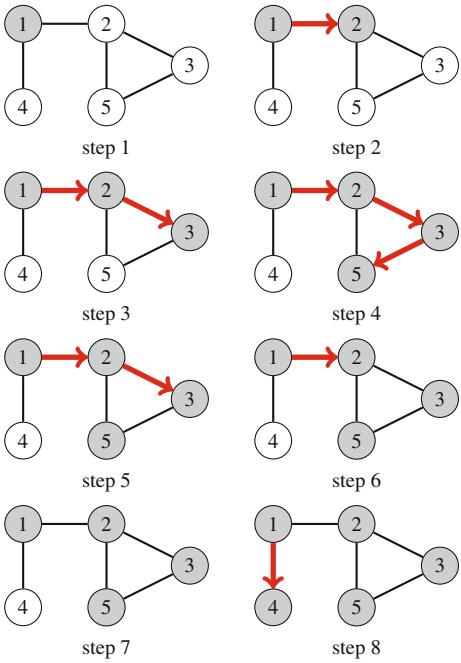


Fig. 7.13 Depth-first search

```
bool visited[N];
```

that keeps track of the visited nodes. Initially, each array value is `false`, and when the search arrives at node s, the value of `visited`[s] becomes `true`. The function can be implemented as follows:

```
void dfs(int s) {
    if (visited[s]) return;
    visited[s] = true;
    // process node s
    for (auto u: adj[s]) {
        dfs(u);
    }
}
```

The time complexity of depth-first search is $O(n+m)$ where n is the number of nodes and m is the number of edges, because the algorithm processes each node and edge once.

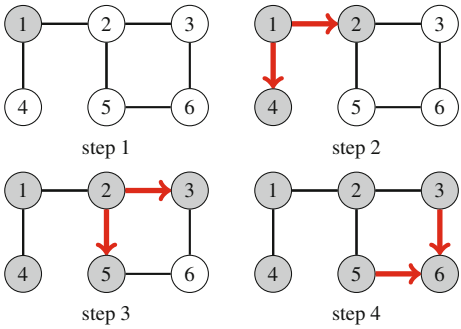


Fig. 7.14 Breadth-first search

7.2.2 Breadth-First Search

Breadth-first search (BFS) visits the nodes of a graph in increasing order of their distance from the starting node. Thus, we can calculate the distance from the starting node to all other nodes using breadth-first search. However, breadth-first search is more difficult to implement than depth-first search.

Breadth-first search goes through the nodes one level after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.

Figure 7.14 shows how breadth-first search processes a graph. Suppose that the search begins at node 1. First the search visits nodes 2 and 4 with distance 1, then nodes 3 and 5 with distance 2, and finally node 6 with distance 3.

Implementation Breadth-first search is more difficult to implement than depth-first search, because the algorithm visits nodes in different parts of the graph. A typical implementation is based on a queue that contains nodes. At each step, the next node in the queue will be processed.

The following code assumes that the graph is stored as adjacency lists and maintains the following data structures:

```
queue<int> q;
bool visited[N];
int distance[N];
```

The queue `q` contains nodes to be processed in increasing order of their distance. New nodes are always added to the end of the queue, and the node at the beginning of the queue is the next node to be processed. The array `visited` indicates which nodes the search has already visited, and the array `distance` will contain the distances from the starting node to all nodes of the graph.

The search can be implemented as follows, starting at node x:

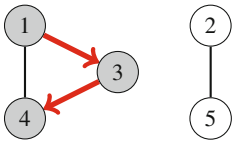


Fig. 7.15 Checking the connectivity of a graph

```
visited[x] = true;
distance[x] = 0;
q.push(x);
while (!q.empty()) {
    int s = q.front(); q.pop();
    // process node s
    for (auto u : adj[s]) {
        if (visited[u]) continue;
        visited[u] = true;
        distance[u] = distance[s]+1;
        q.push(u);
    }
}
```

Like in depth-first search, the time complexity of breadth-first search is $O(n+m)$, where n is the number of nodes and m is the number of edges.

7.2.3 Applications

Using the graph traversal algorithms, we can check many properties of graphs. Usually, both depth-first search and breadth-first search may be used, but in practice, depth-first search is a better choice, because it is easier to implement. In the applications described below we will assume that the graph is undirected.

Connectivity Check A graph is connected if there is a path between any two nodes of the graph. Thus, we can check if a graph is connected by starting at an arbitrary node and finding out if we can reach all other nodes.

For example, in Fig. 7.15, since a depth-first search from node 1 does not visit all the nodes, we can conclude that the graph is not connected. In a similar way, we can also find all connected components of a graph by iterating through the nodes and always starting a new depth-first search if the current node does not belong to any component yet.

Cycle Detection A graph contains a cycle if during a graph traversal, we find a node whose neighbor (other than the previous node in the current path) has already been visited. For example, in Fig. 7.16, a depth-first search from node 1 reveals that the graph contains a cycle. After moving from node 2 to node 5 we notice that the neighbor 3 of node 5 has already been visited. Thus, the graph contains a cycle that goes through node 3, for example, $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

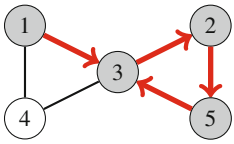


Fig. 7.16 Finding a cycle in a graph

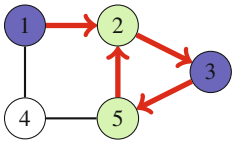


Fig. 7.17 A conflict when checking bipartiteness

Another way to determine if a graph contains a cycle is to simply calculate the number of nodes and edges in every component. If a component contains c nodes and no cycle, it must contain exactly $c - 1$ edges (so it has to be a tree). If there are c or more edges, the component surely contains a cycle.

Bipartiteness Check A graph is bipartite if its nodes can be colored using two colors so that there are no adjacent nodes with the same color. It is surprisingly easy to check if a graph is bipartite using graph traversal algorithms.

The idea is to pick two colors X and Y, color the starting node X, all its neighbors Y, all their neighbors X, and so on. If at some point of the search we notice that two adjacent nodes have the same color, this means that the graph is not bipartite. Otherwise the graph is bipartite and one coloring has been found.

For example, in Fig. 7.17, a depth-first search from node 1 shows that the graph is not bipartite, because we notice that both nodes 2 and 5 should have the same color, while they are adjacent nodes in the graph.

This algorithm always works, because when there are only two colors available, the color of the starting node in a component determines the colors of all other nodes in the component. It does not make any difference what the colors are.

Note that in the general case it is difficult to find out if the nodes in a graph can be colored using k colors so that no adjacent nodes have the same color. The problem is NP-hard already for $k = 3$.

7.3 Shortest Paths

Finding a shortest path between two nodes of a graph is an important problem that has many practical applications. For example, a natural problem related to a road network is to calculate the shortest possible length of a route between two cities, given the lengths of the roads.

In an unweighted graph, the length of a path equals the number of its edges, and we can simply use breadth-first search to find a shortest path. However, in this section

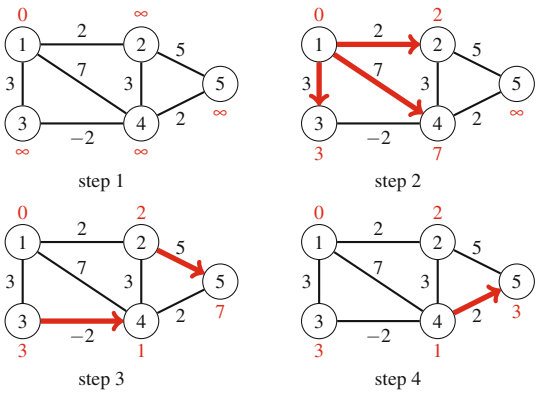


Fig. 7.18 The Bellman–Ford algorithm

we focus on weighted graphs where more sophisticated algorithms are needed for finding shortest paths.

7.3.1 Bellman–Ford Algorithm

The *Bellman–Ford algorithm* finds shortest paths from a starting node to all nodes of the graph. The algorithm can process all kinds of graphs, provided that the graph does not contain a cycle with negative length. If the graph contains a negative cycle, the algorithm can detect this.

The algorithm keeps track of distances from the starting node to all nodes of the graph. Initially, the distance to the starting node is 0 and the distance to any other node is infinite. The algorithm then reduces the distances by finding edges that shorten the paths until it is not possible to reduce any distance.

Figure 7.18 shows how the Bellman–Ford algorithm processes a graph. First, the algorithm reduces distances using the edges $1 \to 2$, $1 \to 3$ and $1 \to 4$, then using the edges $2 \to 5$ and $3 \to 4$, and finally using the edge $4 \to 5$. After this, no edge can be used to reduce distances, which means that the distances are final.

Implementation The implementation of the Bellman–Ford algorithm below determines the shortest distances from a node x to all nodes of the graph. The code assumes that the graph is stored as an edge list `edges` that consists of tuples of the form (a, b, w), meaning that there is an edge from node a to node b with weight w.

The algorithm consists of $n - 1$ rounds, and on each round the algorithm goes through all edges of the graph and attempts to reduce the distances. The algorithm constructs an array `distance` that will contain the distances from node x to all nodes. The constant `INF` denotes an infinite distance.

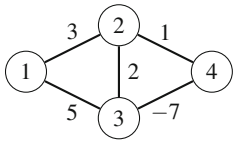


Fig. 7.19 A graph with a negative cycle

```
for (int i = 1; i <= n; i++) {
    distance[i] = INF;
}
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

The time complexity of the algorithm is $O(nm)$, because the algorithm consists of $n-1$ rounds and iterates through all m edges during a round. If there are no negative cycles in the graph, all distances are final after $n-1$ rounds, because each shortest path can contain at most $n-1$ edges.

There are several ways to optimize the algorithm in practice. First, the final distances can usually be found earlier than after $n-1$ rounds, so we can simply stop the algorithm if no distance can be reduced during a round. A more advanced variant is the *SPFA algorithm* ("Shortest Path Faster Algorithm" [8]) which maintains a queue of nodes that might be used for reducing the distances. Only the nodes in the queue will be processed, which often yields a more efficient search.

Negative Cycles The Bellman–Ford algorithm can also be used to check if the graph contains a cycle with negative length. In this case, any path that contains the cycle can be shortened infinitely many times, so the concept of a shortest path is not meaningful. For example, the graph in Fig. 7.19 contains a negative cycle $2 \to 3 \to 4 \to 2$ with length -4.

A negative cycle can be detected using the Bellman–Ford algorithm by running the algorithm for n rounds. If the last round reduces any distance, the graph contains a negative cycle. Note that this algorithm can be used to search for a negative cycle in the entire graph regardless of the starting node.

7.3.2 Dijkstra's Algorithm

Dijkstra's algorithm finds shortest paths from the starting node to all nodes of the graph, like the Bellman–Ford algorithm. The benefit of Dijkstra's algorithm is that it

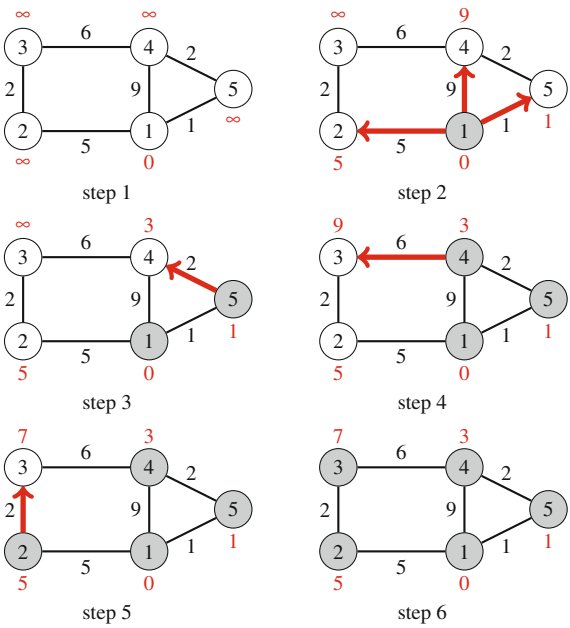


Fig. 7.20 Dijkstra's algorithm

is more efficient and can be used for processing large graphs. However, the algorithm requires that there are no negative weight edges in the graph.

Like the Bellman–Ford algorithm, Dijkstra's algorithm maintains distances to the nodes and reduces them during the search. At each step, Dijkstra's algorithm selects a node that has not been processed yet and whose distance is as small as possible. Then, the algorithm goes through all edges that start at the node and reduces the distances using them. Dijkstra's algorithm is efficient, because it only processes each edge in the graph once, using the fact that there are no negative edges.

Figure 7.20 shows how Dijkstra's algorithm processes a graph. Like in the Bellman–Ford algorithm, the initial distance to all nodes, except for the starting node, is infinite. The algorithm processes the nodes in the order 1, 5, 4, 2, 3, and at each node reduces distances using edges that start at the node. Note that the distance to a node never changes after processing the node.

Implementation An efficient implementation of Dijkstra's algorithm requires that we can efficiently find the minimum-distance node that has not been processed. An appropriate data structure for this is a priority queue that contains the remaining nodes ordered by their distances. Using a priority queue, the next node to be processed can be retrieved in logarithmic time.

A typical textbook implementation of Dijkstra's algorithm uses a priority queue that has an operation for modifying a value in the queue. This allows us to have a single instance of each node in the queue and update its distance when needed. However, standard library priority queues do not provide such an operation, and a somewhat different implementation is usually used in competitive programming.

The idea is to add a new instance of a node to the priority queue always when its distance changes.

Our implementation of Dijkstra's algorithm calculates the minimum distances from a node x to all other nodes of the graph. The graph is stored as adjacency lists so that `adj[a]` contains a pair (b, w) always when there is an edge from node a to node b with weight w. The priority queue

```
priority_queue<pair<int,int>> q;
```

contains pairs of the form $(-d, x)$, meaning that the current distance to node x is d. The array `distance` contains the distance to each node, and the array `processed` indicates whether a node has been processed.

Note that the priority queue contains *negative* distances to nodes. The reason for this is that the default version of the C++ priority queue finds maximum elements, while we want to find minimum elements. By exploiting negative distances, we can directly use the default priority queue.[2] Also note that while there may be several instances of a node in the priority queue, only the instance with the minimum distance will be processed.

The implementation is as follows:

```
for (int i = 1; i <= n; i++) {
    distance[i] = INF;
}
distance[x] = 0;
q.push({0,x});
while (!q.empty()) {
    int a = q.top().second; q.pop();
    if (processed[a]) continue;
    processed[a] = true;
    for (auto u : adj[a]) {
        int b = u.first, w = u.second;
        if (distance[a]+w < distance[b]) {
            distance[b] = distance[a]+w;
            q.push({-distance[b],b});
        }
    }
}
```

The time complexity of the above implementation is $O(n + m \log m)$, because the algorithm goes through all nodes of the graph and adds for each edge at most one distance to the priority queue.

Negative Edges The efficiency of Dijkstra's algorithm is based on the fact that the graph does not have negative edges. However, if the graph has a negative edge, the

[2] Of course, we could also declare the priority queue as in Sect. 5.2.3 and use positive distances, but the implementation would be longer.

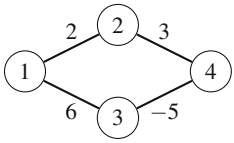


Fig. 7.21 A graph where Dijkstra's algorithm fails

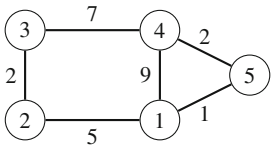


Fig. 7.22 An input for the Floyd–Warshall algorithm

algorithm may give incorrect results. As an example, consider the graph in Fig. 7.21. The shortest path from node 1 to node 4 is $1 \rightarrow 3 \rightarrow 4$ and its length is 1. However, Dijkstra's algorithm incorrectly finds the path $1 \rightarrow 2 \rightarrow 4$ by greedily following minimum weight edges.

7.3.3 Floyd–Warshall Algorithm

The *Floyd–Warshall algorithm* provides an alternative way to approach the problem of finding shortest paths. Unlike the other algorithms in this chapter, it finds shortest paths between all node pairs of the graph in a single run.

The algorithm maintains a matrix that contains distances between the nodes. The initial matrix is directly constructed based on the adjacency matrix of the graph. Then, the algorithm consists of consecutive rounds, and on each round, it selects a new node that can act as an intermediate node in paths from now on, and reduces distances using this node.

Let us simulate the Floyd–Warshall algorithm for the graph in Fig. 7.22. In this case, the initial matrix is as follows:

$$\begin{bmatrix} 0 & 5 & \infty & 9 & 1 \\ 5 & 0 & 2 & \infty & \infty \\ \infty & 2 & 0 & 7 & \infty \\ 9 & \infty & 7 & 0 & 2 \\ 1 & \infty & \infty & 2 & 0 \end{bmatrix}$$

On the first round, node 1 is the new intermediate node. There is a new path between nodes 2 and 4 with length 14, because node 1 connects them. There is also a new path between nodes 2 and 5 with length 6.

$$\begin{bmatrix} 0 & 5 & \infty & 9 & 1 \\ 5 & 0 & 2 & \mathbf{14} & \mathbf{6} \\ \infty & 2 & 0 & 7 & \infty \\ 9 & \mathbf{14} & 7 & 0 & 2 \\ 1 & \mathbf{6} & \infty & 2 & 0 \end{bmatrix}$$

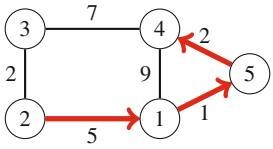


Fig. 7.23 A shortest path from node 2 to node 4

On the second round, node 2 is the new intermediate node. This creates new paths between nodes 1 and 3 and between nodes 3 and 5:

$$\begin{bmatrix} 0 & 5 & \mathbf{7} & 9 & 1 \\ 5 & 0 & 2 & 14 & 6 \\ \mathbf{7} & 2 & 0 & 7 & \mathbf{8} \\ 9 & 14 & 7 & 0 & 2 \\ 1 & 6 & \mathbf{8} & 2 & 0 \end{bmatrix}$$

The algorithm continues like this, until all nodes have been appointed intermediate nodes. After the algorithm has finished, the matrix contains the minimum distances between any two nodes:

$$\begin{bmatrix} 0 & 5 & 7 & 3 & 1 \\ 5 & 0 & 2 & 8 & 6 \\ 7 & 2 & 0 & 7 & 8 \\ 3 & 8 & 7 & 0 & 2 \\ 1 & 6 & 8 & 2 & 0 \end{bmatrix}$$

For example, the matrix tells us that the shortest distance between nodes 2 and 4 is 8. This corresponds to the path in Fig. 7.23.

Implementation The Floyd–Warshall algorithm is particularly easy to implement. The implementation below constructs a distance matrix where `dist`[a][b] denotes the shortest distance between nodes a and b. First, the algorithm initializes `dist` using the adjacency matrix `adj` of the graph:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (i == j) dist[i][j] = 0;
        else if (adj[i][j]) dist[i][j] = adj[i][j];
        else dist[i][j] = INF;
    }
}
```

After this, the shortest distances can be found as follows:

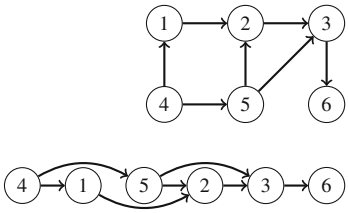


Fig. 7.24 A graph and a topological sort

```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            dist[i][j] = min(dist[i][j],dist[i][k]+dist[k][j]);
        }
    }
}
```

The time complexity of the algorithm is $O(n^3)$, because it contains three nested loops that go through the nodes of the graph.

Since the implementation of the Floyd–Warshall algorithm is simple, the algorithm can be a good choice even if it is only needed to find a *single* shortest path in the graph. However, the algorithm can only be used when the graph is so small that a cubic time complexity is fast enough.

7.4 Directed Acyclic Graphs

An important class of graphs are *directed acyclic graphs*, also called *DAGs*. Such graphs do not contain cycles, and many problems are easier to solve if we may assume that this is the case. In particular, we can always construct a topological sort for the graph and then apply dynamic programming.

7.4.1 Topological Sorting

A *topological sort* is an ordering of the nodes of a directed graph such that if there is a path from node a to node b, then node a appears before node b in the ordering. For example, in Fig. 7.24, one possible topological sort is [4, 1, 5, 2, 3, 6].

A directed graph has a topological sort exactly when it is acyclic. If the graph contains a cycle, it is not possible to form a topological sort, because no node of the cycle can appear before the other nodes of the cycle in the ordering. It turns out that depth-first search can be used to both check if a directed graph contains a cycle and, if it does not, to construct a topological sort.

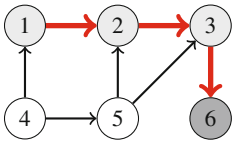


Fig. 7.25 The first search adds nodes 6, 3, 2, and 1 to the list

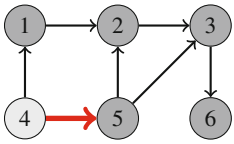


Fig. 7.26 The second search adds nodes 5 and 4 to the list

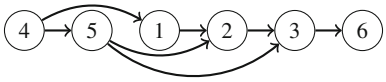


Fig. 7.27 The final topological sort

The idea is to go through the nodes of the graph and always begin a depth-first search at the current node if it has not been processed yet. During the searches, the nodes have three possible states:

- state 0: the node has not been processed (white)
- state 1: the node is under processing (light gray)
- state 2: the node has been processed (dark gray)

Initially, the state of each node is 0. When a search reaches a node for the first time, its state becomes 1. Finally, after all edges from the node have been processed, its state becomes 2.

If the graph contains a cycle, we will discover this during the search, because sooner or later we will arrive at a node whose state is 1. In this case, it is not possible to construct a topological sort. If the graph does not contain a cycle, we can construct a topological sort by adding each node to a list when its state becomes 2. Finally, we reverse the list and get a topological sort for the graph.

Now we are ready to construct a topological sort for our example graph. The first search (Fig. 7.25) proceeds from node 1 to node 6, and adds nodes 6, 3, 2, and 1 to the list. Then, the second search (Fig. 7.26) proceeds from node 4 to node 5 and adds nodes 5 and 4 to the list. The final reversed list is [4, 5, 1, 2, 3, 6], which corresponds to a topological sort (Fig. 7.27). Note that a topological sort is not unique; there can be several topological sorts for a graph.

Figure 7.28 shows a graph that does not have a topological sort. During the search, we reach node 2 whose state is 1, which means that the graph contains a cycle. Indeed, there is a cycle $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$.

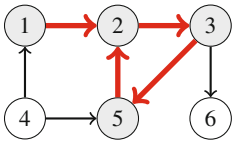


Fig. 7.28 This graph does not have a topological sort, because it contains a cycle

7.4.2 Dynamic Programming

Using dynamic programming, we can efficiently answer many questions regarding paths in directed acyclic graphs. Examples of such questions are:

- What is the shortest/longest path from node a to node b?
- How many different paths are there?
- What is the minimum/maximum number of edges in a path?
- Which nodes appear in every possible path?

Note that many of the above problems are difficult to solve or not well-defined for general graphs.

As an example, consider the problem of calculating the number of paths from node a to node b. Let $\texttt{paths}(x)$ denote the number of paths from node a to node x. As a base case, $\texttt{paths}(a) = 1$. Then, to calculate other values of $\texttt{paths}(x)$, we can use the recursive formula

$$\texttt{paths}(x) = \texttt{paths}(s_1) + \texttt{paths}(s_2) + \cdots + \texttt{paths}(s_k),$$

where $s_1, s_2, \ldots, s_k$ are the nodes from which there is an edge to x. Since the graph is acyclic, the values of `paths` can be calculated in the order of a topological sort.

Figure 7.29 shows the values of `paths` in an example scenario where we want to calculate the number of paths from node 1 to node 6. For example,

$$\texttt{paths}(6) = \texttt{paths}(2) + \texttt{paths}(3),$$

because the edges that end at node 6 are $2 \to 6$ and $3 \to 6$. Since $\texttt{paths}(2) = 2$ and $\texttt{paths}(3) = 2$, we conclude that $\texttt{paths}(6) = 4$. The paths are as follows:

- $1 \to 2 \to 3 \to 6$
- $1 \to 2 \to 6$
- $1 \to 4 \to 5 \to 2 \to 3 \to 6$
- $1 \to 4 \to 5 \to 2 \to 6$

Processing Shortest Paths Dynamic programming can also be used to answer questions regarding *shortest* paths in general (not necessarily acyclic) graphs. Namely, if we know minimum distances from a starting node to other nodes (e.g., after using Dijkstra's algorithm), we can easily create a directed acyclic *shortest paths graph*

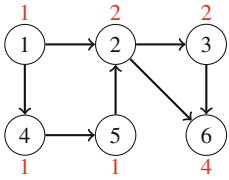


Fig. 7.29 Calculating the number of paths from node 1 to node 6

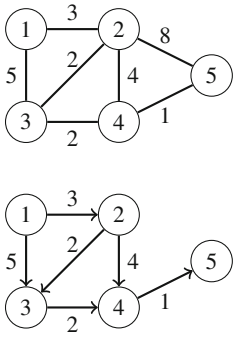


Fig. 7.30 A graph and its shortest paths graph

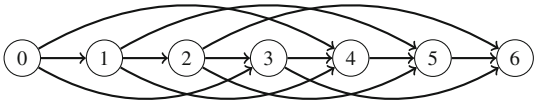


Fig. 7.31 Coin problem as a directed acyclic graph

that indicates for each node the possible ways to reach the node using a shortest path from the starting node. For example, Fig. 7.30 shows a graph and the corresponding shortest paths graph.

Coin Problem Revisited In fact, *any* dynamic programming problem can be represented as a directed acyclic graph where each node corresponds to a dynamic programming state and the edges indicate how the states depend on each other.

For example, consider the problem of forming a sum of money n using coins $\{c_1, c_2, \ldots, c_k\}$ (Sect. 6.1.1). In this scenario, we can construct a graph where each node corresponds to a sum of money, and the edges show how the coins can be chosen. For example, Fig. 7.31 shows the graph for the coins $\{1, 3, 4\}$ and $n = 6$. Using this representation, the shortest path from node 0 to node n corresponds to a solution with the minimum number of coins, and the total number of paths from node 0 to node n equals the total number of solutions.

7.5 Successor Graphs

Another special class of directed graphs are *successor graphs*. In those graphs, the outdegree of each node is 1, i.e., each node has a unique *successor*. A successor

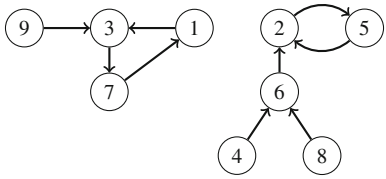


Fig. 7.32 A successor graph

Fig. 7.33 Walking in a successor graph

graph consists of one or more components, each of which contains one cycle and some paths that lead to it.

Successor graphs are sometimes called *functional graphs*, because any successor graph corresponds to a function $\texttt{succ}(x)$ that defines the edges of the graph. The parameter x is a node of the graph, and the function gives the successor of the node. For example, the function

x	1	2	3	4	5	6	7	8	9
$\texttt{succ}(x)$	3	5	7	6	2	2	1	6	3

defines the graph in Fig. 7.32.

7.5.1 Finding Successors

Since each node of a successor graph has a unique successor, we can also define a function $\texttt{succ}(x, k)$ that gives the node that we will reach if we begin at node x and walk k steps forward. For example, in our example graph $\texttt{succ}(4, 6) = 2$, because we will reach node 2 by walking 6 steps from node 4 (Fig. 7.33).

A straightforward way to calculate a value of $\texttt{succ}(x, k)$ is to start at node x and walk k steps forward, which takes $O(k)$ time. However, using preprocessing, any value of $\texttt{succ}(x, k)$ can be calculated in only $O(\log k)$ time.

Let u denote the maximum number of steps we will ever walk. The idea is to precalculate all values of $\texttt{succ}(x, k)$ where k is a power of two and at most u. This can be efficiently done, because we can use the following recurrence:

$$\texttt{succ}(x, k) = \begin{cases} \texttt{succ}(x) & k = 1 \\ \texttt{succ}(\texttt{succ}(x, k/2), k/2) & k > 1 \end{cases}$$

The idea is that a path of length k that begins at node x can be divided into two paths of length $k/2$. Precalculating all values of $\texttt{succ}(x, k)$ where k is a power of two and at most u takes $O(n \log u)$ time, because $O(\log u)$ values are calculated for each node. In our example graph, the first values are as follows:

x	1	2	3	4	5	6	7	8	9
succ$(x, 1)$	3	5	7	6	2	2	1	6	3
succ$(x, 2)$	7	2	1	2	5	5	3	2	7
succ$(x, 4)$	3	2	7	2	5	5	1	2	3
succ$(x, 8)$	7	2	1	2	5	5	3	2	7
...									

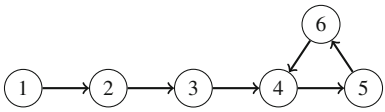


Fig. 7.34 A cycle in a successor graph

After the precalculation, any value of succ(x, k) can be calculated by presenting k as a sum of powers of two. Such a representation always consists of $O(\log k)$ parts, so calculating a value of succ(x, k) takes $O(\log k)$ time. For example, if we want to calculate the value of succ$(x, 11)$, we use the formula

$$\texttt{succ}(x, 11) = \texttt{succ}(\texttt{succ}(\texttt{succ}(x, 8), 2), 1).$$

In our example graph,

$$\texttt{succ}(4, 11) = \texttt{succ}(\texttt{succ}(\texttt{succ}(4, 8), 2), 1) = 5.$$

7.5.2 Cycle Detection

Consider a successor graph that only contains a path that ends in a cycle. We may ask the following questions: if we begin our walk at the starting node, what is the first node in the cycle and how many nodes does the cycle contain? For example, in Fig. 7.34, we begin our walk at node 1, the first node that belongs to the cycle is node 4, and the cycle consists of three nodes (4, 5, and 6).

A simple way to detect the cycle is to walk in the graph and keep track of all nodes that have been visited. Once a node is visited for the second time, we can conclude that the node is the first node in the cycle. This method works in $O(n)$ time and also uses $O(n)$ memory. However, there are better algorithms for cycle detection. The time complexity of such algorithms is still $O(n)$, but they only use $O(1)$ memory, which may be an important improvement if n is large.

One such algorithm is *Floyd's algorithm*, which walks in the graph using two pointers a and b. Both pointers begin at the starting node x. Then, on each turn, the pointer a walks one step forward and the pointer b walks two steps forward. The process continues until the pointers meet each other:

```
a = succ(x);
b = succ(succ(x));
while (a != b) {
    a = succ(a);
    b = succ(succ(b));
}
```

At this point, the pointer a has walked k steps and the pointer b has walked $2k$ steps, so the length of the cycle divides k. Thus, the first node that belongs to the cycle can be found by moving the pointer a to node x and advancing the pointers step by step until they meet again.

```
a = x;
while (a != b) {
    a = succ(a);
    b = succ(b);
}
first = a;
```

After this, the length of the cycle can be calculated as follows:

```
b = succ(a);
length = 1;
while (a != b) {
    b = succ(b);
    length++;
}
```

7.6 Minimum Spanning Trees

A *spanning tree* contains all nodes of a graph and some of its edges so that there is a path between any two nodes. Like trees in general, spanning trees are connected and acyclic. The *weight* of a spanning tree is the sum of its edge weights. For example, Fig. 7.35 shows a graph and one of its spanning tree. The weight of this spanning tree is $3 + 5 + 9 + 3 + 2 = 22$.

A *minimum spanning tree* is a spanning tree whose weight is as small as possible. Figure 7.36 shows a minimum spanning tree for our example graph with weight 20. In a similar way, a *maximum spanning tree* is a spanning tree whose weight is as large as possible. Figure 7.37 shows a maximum spanning tree for our example graph with weight 32. Note that a graph may have several minimum and maximum spanning trees, so the trees are not unique.

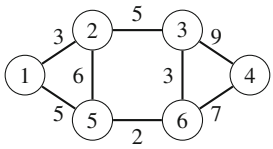


Fig. 7.35 A graph and a spanning tree

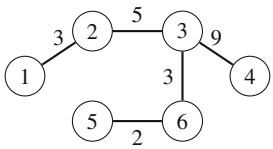


Fig. 7.36 A minimum spanning tree with weight 20

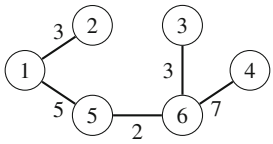
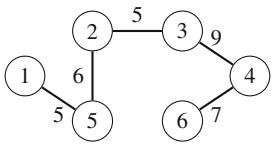


Fig. 7.37 A maximum spanning tree with weight 32



It turns out that several greedy methods can be used to construct minimum and maximum spanning trees. This section discusses two algorithms that process the edges of the graph ordered by their weights. We focus on finding minimum spanning trees, but the same algorithms can also find maximum spanning trees by processing the edges in reverse order.

7.6.1 Kruskal's Algorithm

Kruskal's algorithm builds a minimum spanning tree by greedily adding edges to the graph. The initial spanning tree only contains the nodes of the graph and does not contain any edges. Then the algorithm goes through the edges ordered by their weights and always adds an edge to the graph if it does not create a cycle.

The algorithm maintains the components of the graph. Initially, each node of the graph belongs to a separate component. Always when an edge is added to the graph, two components are joined. Finally, all nodes belong to the same component, and a minimum spanning tree has been found.

As an example, let us construct a minimum spanning tree for our example graph (Fig. 7.35). The first step is to sort the edges in increasing order of their weights:

edge	weight
5–6	2
1–2	3
3–6	3
1–5	5
2–3	5
2–5	6
4–6	7
3–4	9

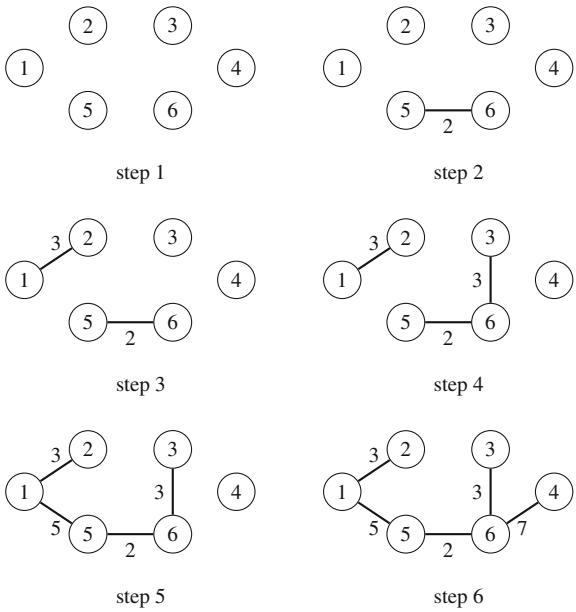


Fig. 7.38 Kruskal's algorithm

Then, we go through the list and add each edge to the graph if it joins two separate components. Figure 7.38 shows the steps of the algorithm. Initially, each node belongs to its own component. Then, the first edges on the list (5–6, 1–2, 3–6, and 1–5) are added to the graph. After this, the next edge would be 2–3, but this edge is not added, because it would create a cycle. The same applies to edge 2–5. Finally, the edge 4–6 is added, and the minimum spanning tree is ready.

Why Does This Work? It is a good question *why* Kruskal's algorithm works. Why does the greedy strategy guarantee that we will find a minimum spanning tree?

Let us see what happens if the minimum weight edge of the graph is *not* included in the spanning tree. For example, suppose that a minimum spanning tree of our example graph would not contain the minimum weight edge 5–6. We do not know the exact structure of such a spanning tree, but in any case it has to contain some edges. Assume that the tree would look like the tree in Fig. 7.39.

However, it is not possible that the tree in Fig. 7.39 would be a minimum spanning tree, because we can remove an edge from the tree and replace it with the minimum

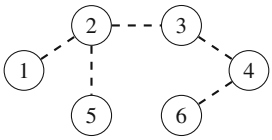


Fig. 7.39 A hypothetical minimum spanning tree

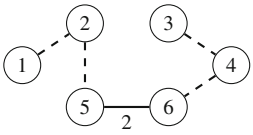


Fig. 7.40 Including the edge 5–6 reduces the weight of the spanning tree

weight edge 5–6. This produces a spanning tree whose weight is *smaller*, shown in Fig. 7.40.

For this reason, it is always optimal to include the minimum weight edge in the tree to produce a minimum spanning tree. Using a similar argument, we can show that it is also optimal to add the next edge in weight order to the tree, and so on. Hence, Kruskal's algorithm always produces a minimum spanning tree.

Implementation When implementing Kruskal's algorithm, it is convenient to use the edge list representation of the graph. The first phase of the algorithm sorts the edges in the list in $O(m \log m)$ time. After this, the second phase of the algorithm builds the minimum spanning tree as follows:

```
for (...) {
  if (!same(a,b)) unite(a,b);
}
```

The loop goes through the edges in the list and always processes an edge (a, b) where a and b are two nodes. Two functions are needed: the function `same` determines if a and b are in the same component, and the function `unite` joins the components that contain a and b.

The problem is how to efficiently implement the functions `same` and `unite`. One possibility is to implement the function `same` as a graph traversal and check if we can get from node a to node b. However, the time complexity of such a function would be $O(n+m)$ and the resulting algorithm would be slow, because the function `same` will be called for each edge in the graph.

We will solve the problem using a union-find structure that implements both functions in $O(\log n)$ time. Thus, the time complexity of Kruskal's algorithm will be $O(m \log n)$ after sorting the edge list.

7.6.2 Union-Find Structure

A *union-find structure* maintains a collection of sets. The sets are disjoint, so no element belongs to more than one set. Two $O(\log n)$ time operations are supported:

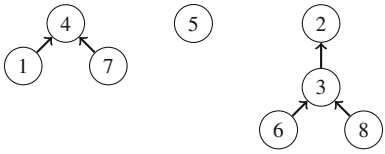


Fig. 7.41 A union-find structure with three sets

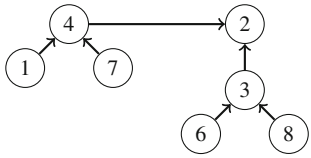


Fig. 7.42 Joining two sets into a single set

the `unite` operation joins two sets, and the `find` operation finds the representative of the set that contains a given element.

In a union-find structure, one element in each set is the representative of the set, and there is a path from any other element of the set to the representative. For example, assume that the sets are {1, 4, 7}, {5} and {2, 3, 6, 8}. Figure 7.41 shows one way to represent these sets.

In this case the representatives of the sets are 4, 5, and 2. We can find the representative of any element by following the path that begins at the element. For example, the element 2 is the representative for the element 6, because we follow the path $6 \rightarrow 3 \rightarrow 2$. Two elements belong to the same set exactly when their representatives are the same.

To join two sets, the representative of one set is connected to the representative of the other set. For example, Fig. 7.42 shows a possible way to join the sets {1, 4, 7} and {2, 3, 6, 8}. From this on, the element 2 is the representative for the entire set and the old representative 4 points to the element 2.

The efficiency of the union-find structure depends on how the sets are joined. It turns out that we can follow a simple strategy: always connect the representative of the *smaller* set to the representative of the *larger* set (or if the sets are of equal size, we can make an arbitrary choice). Using this strategy, the length of any path will be $O(\log n)$, so we can find the representative of any element efficiently by following the corresponding path.

Implementation The union-find structure can be conveniently implemented using arrays. In the following implementation, the array `link` indicates for each element the next element in the path, or the element itself if it is a representative, and the array `size` indicates for each representative the size of the corresponding set.

Initially, each element belongs to a separate set:

```
for (int i = 1; i <= n; i++) link[i] = i;
for (int i = 1; i <= n; i++) size[i] = 1;
```

The function `find` returns the representative for an element x. The representative can be found by following the path that begins at x.

```
int find(int x) {
    while (x != link[x]) x = link[x];
    return x;
}
```

The function `same` checks whether elements a and b belong to the same set. This can easily be done by using the function `find`:

```
bool same(int a, int b) {
    return find(a) == find(b);
}
```

The function `unite` joins the sets that contain elements a and b (the elements have to be in different sets). The function first finds the representatives of the sets and then connects the smaller set to the larger set.

```
void unite(int a, int b) {
    a = find(a);
    b = find(b);
    if (size[a] < size[b]) swap(a,b);
    size[a] += size[b];
    link[b] = a;
}
```

The time complexity of the function `find` is $O(\log n)$ assuming that the length of each path is $O(\log n)$. In this case, the functions `same` and `unite` also work in $O(\log n)$ time. The function `unite` makes sure that the length of each path is $O(\log n)$ by connecting the smaller set to the larger set.

Path Compression Here is an alternative way to implement the `find` operation:

```
int find(int x) {
    if (x == link[x]) return x;
    return link[x] = find(link[x]);
}
```

This function uses *path compression*: each element in the path will directly point to its representative after the operation. It can be shown that using this function, the union-find operations work in amortized $O(\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann function which grows very slowly (it is almost a constant). However, path compression cannot be used in some applications of the union-find structure, such as in the dynamic connectivity algorithm (Sect. 15.5.4).

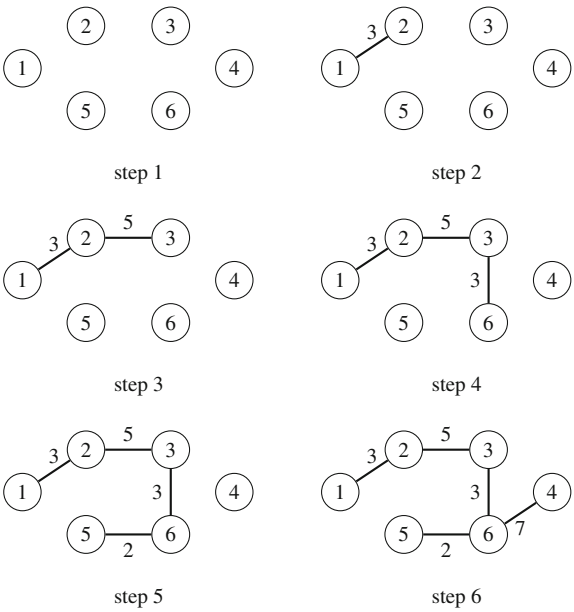


Fig. 7.43 Prim's algorithm

7.6.3 Prim's Algorithm

Prim's algorithm is an alternative method for constructing minimum spanning trees. The algorithm first adds an arbitrary node to the tree, and then always chooses a minimum weight edge that adds a new node to the tree. Finally, all nodes have been added and a minimum spanning tree has been found.

Prim's algorithm resembles Dijkstra's algorithm. The difference is that Dijkstra's algorithm always selects a node whose distance from the starting node is minimum, but Prim's algorithm simply selects a node that can be added to the tree using a minimum weight edge.

As an example, Fig. 7.43 shows how Prim's algorithm constructs a minimum spanning tree for our example graph, assuming that the starting node is node 1.

Like Dijkstra's algorithm, Prim's algorithm can be efficiently implemented using a priority queue. The priority queue should contain all nodes that can be connected to the current component using a single edge, in increasing order of the weights of the corresponding edges.

The time complexity of Prim's algorithm is $O(n + m \log m)$ that equals the time complexity of Dijkstra's algorithm. In practice, Prim's and Kruskal's algorithms are both efficient, and the choice of the algorithm is a matter of taste. Still, most competitive programmers use Kruskal's algorithm.