Competitive Programmer's Handbook

(modified for 02105 at DTU)

Antti Laaksonen[1]

Draft February 28, 2022

[1]This contains parts of chapter 11, 12, 13, and 15 from "Competitive Programmer's Handbook". The chapters have been adapted to use in the course 02105 at Technical University of Denmark by Inge Li Gørtz and Christian Fuglsang Mikkelsen. The C++ code has been changed to Java, and minor changes have been made.

Chapter 11

Basics of graphs

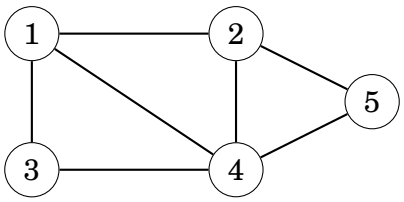
Many programming problems can be solved by modeling the problem as a graph problem and using an appropriate graph algorithm. A typical example of a graph is a network of roads and cities in a country. Sometimes, though, the graph is hidden in the problem and it may be difficult to detect it.

This part of the book discusses graph algorithms, especially focusing on topics that are important in competitive programming. In this chapter, we go through concepts related to graphs, and study different ways to represent graphs in algorithms.

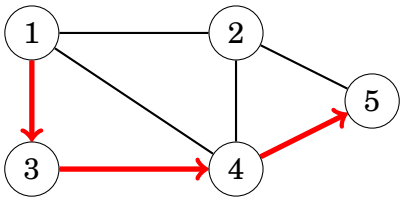
11.1 Graph terminology

A **graph** consists of **nodes** and **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, the following graph consists of 5 nodes and 7 edges:



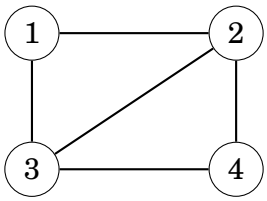
A **path** leads from node a to node b through edges of the graph. The **length** of a path is the number of edges in it. For example, the above graph contains a path $1 \to 3 \to 4 \to 5$ of length 3 from node 1 to node 5:



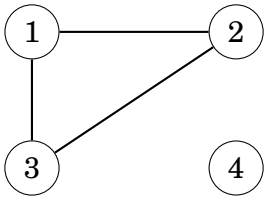
A path is a **cycle** if the first and last node is the same. For example, the above graph contains a cycle $1 \to 3 \to 4 \to 1$. A path is **simple** if each node appears at most once in the path.

Connectivity

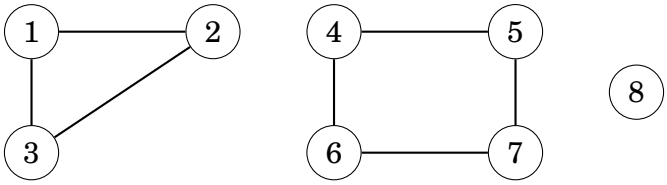
A graph is **connected** if there is a path between any two nodes. For example, the following graph is connected:



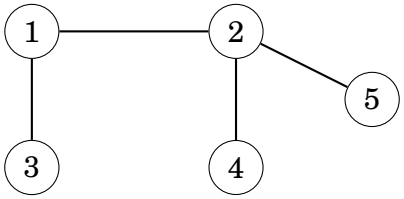
The following graph is not connected, because it is not possible to get from node 4 to any other node:



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

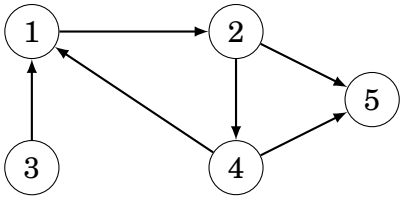


A **tree** is a connected graph that consists of n nodes and $n-1$ edges. There is a unique path between any two nodes of a tree. For example, the following graph is a tree:



Edge directions

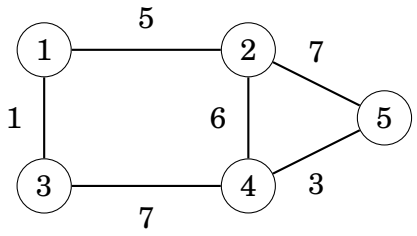
A graph is **directed** if the edges can be traversed in one direction only. For example, the following graph is directed:



The above graph contains a path $3 \to 1 \to 2 \to 5$ from node 3 to node 5, but there is no path from node 5 to node 3.

Edge weights

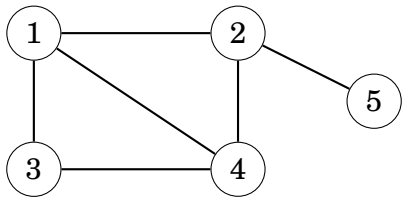
In a **weighted** graph, each edge is assigned a **weight**. The weights are often interpreted as edge lengths. For example, the following graph is weighted:



The length of a path in a weighted graph is the sum of the edge weights on the path. For example, in the above graph, the length of the path $1 \to 2 \to 5$ is 12, and the length of the path $1 \to 3 \to 4 \to 5$ is 11. The latter path is the **shortest** path from node 1 to node 5.

Neighbors and degrees

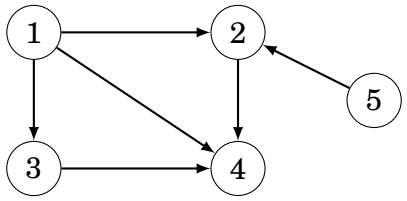
Two nodes are **neighbors** or **adjacent** if there is an edge between them. The **degree** of a node is the number of its neighbors. For example, in the following graph, the neighbors of node 2 are 1, 4 and 5, so its degree is 3.



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.

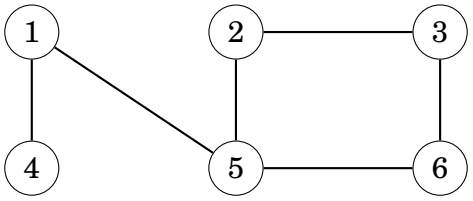
In a directed graph, the **indegree** of a node is the number of edges that end at the node, and the **outdegree** of a node is the number of edges that start at the node. For example, in the following graph, the indegree of node 2 is 2, and the outdegree of node 2 is 1.



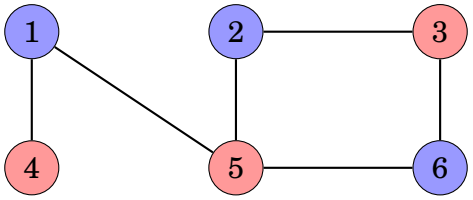
Colorings

In a **coloring** of a graph, each node is assigned a color so that no adjacent nodes have the same color.

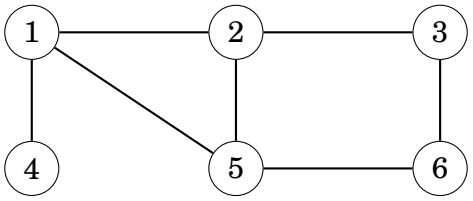
A graph is **bipartite** if it is possible to color it using two colors. It turns out that a graph is bipartite exactly when it does not contain a cycle with an odd number of edges. For example, the graph



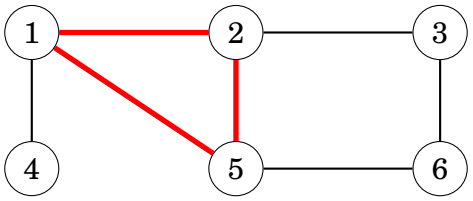
is bipartite, because it can be colored as follows:



However, the graph

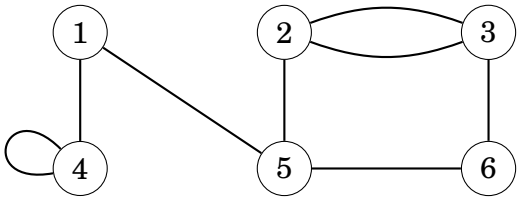


is not bipartite, because it is not possible to color the following cycle of three nodes using two colors:



Simplicity

A graph is **simple** if no edge starts and ends at the same node, and there are no multiple edges between two nodes. Often we assume that graphs are simple. For example, the following graph is *not* simple:



11.2 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 common representations.

Adjacency list representation

In the adjacency list representation, each node x in 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.

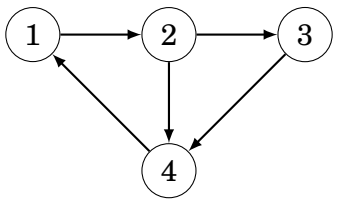
A convenient way to store the adjacency lists is to declare an array of array-lists as follows[1]:

```
ArrayList<Integer>[] adj = new ArrayList[N];
```

Where-after we initialize each of the lists in the array as follows:

```
for (int i = 0; i < N; i++)
    adj[i] = new ArrayList<>();
```

The constant N is chosen so that all adjacency lists can be stored. Typically, it is enough to have $N = n + 1$. For example, the graph



can be stored as follows:

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

```
ArrayList<Pair>[] adj = new ArrayList[N];
```

[1] You will likely get a compiler warning about type safety using the shown declaration. This is suppressed by writing `@SuppressWarnings("unchecked")` on the line above.

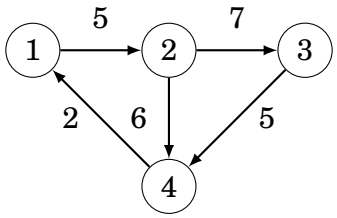
Where `Pair` is a class used to maintain two values of our choosing in a single container. The class can be declared above the main method as follows:

```
public static class Pair {

    public int first;
    public int second;

    public Pair(int first, int second) {
        this.first = first;
        this.second = second;
    }
}
```

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



can be stored as follows:

```
adj[1].add(new Pair(2, 5));
adj[2].add(new Pair(3, 7));
adj[2].add(new Pair(4, 6));
adj[3].add(new Pair(4, 5));
adj[4].add(new Pair(1, 2));
```

The benefit of using adjacency lists is that 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:

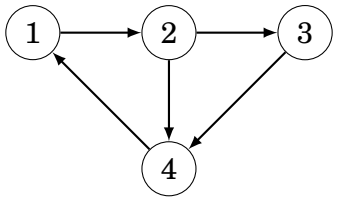
```
for (int u : adj[s]) {
    // process node u
}
```

Adjacency matrix representation

An **adjacency matrix** is a two-dimensional array that indicates which edges the 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 = new int[N][N];
```

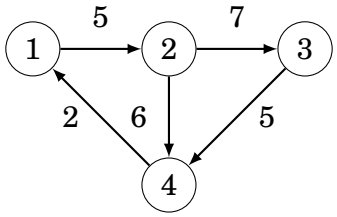
where each value adj[a][b] indicates whether the graph contains an edge from node a to node b. If the edge is included in the graph, then adj[a][b] = 1, and otherwise adj[a][b] = 0. For example, the graph



can be represented as follows:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	1	0	0	0

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



corresponds to the following matrix:

	1	2	3	4
1	0	5	0	0
2	0	0	7	6
3	0	0	0	5
4	2	0	0	0

The drawback of the adjacency matrix representation is that the 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.

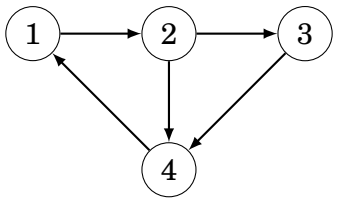
Edge list representation

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 edges of the graph and it is not needed to find edges that start at a given node.

The edge list can be stored in an array-list

```
ArrayList<Pair> edges = new ArrayList<>();
```

where each pair (a, b) denotes that there is an edge from node a to node b. Thus, the graph



can be represented as follows:

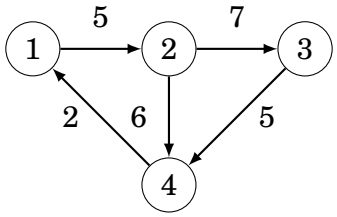
```
edges.add(new Pair(1, 2));
edges.add(new Pair(2, 3));
edges.add(new Pair(2, 4));
edges.add(new Pair(3, 4));
edges.add(new Pair(4, 1));
```

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

```
ArrayList<Tuple> edges = new ArrayList<>();
```

Where `Tuple` maintains three values. It is declared similarly to `Pair`, but has an *extra* value named `third` in the constructor and as a member.

Each element in the above edge 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



can be represented as follows:

```
edges.add(new Tuple(1, 2, 5));
edges.add(new Tuple(2, 3, 7));
edges.add(new Tuple(2, 4, 6));
edges.add(new Tuple(3, 4, 5));
edges.add(new Tuple(4, 1, 2));
```

Chapter 12

Graph traversal

This chapter 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.

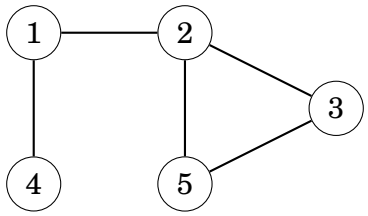
12.1 Depth-first search

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.

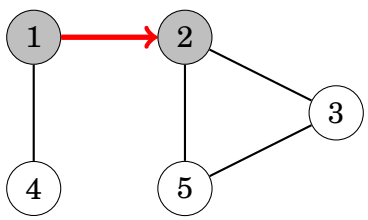
Example

Let us consider how depth-first search processes the following graph:

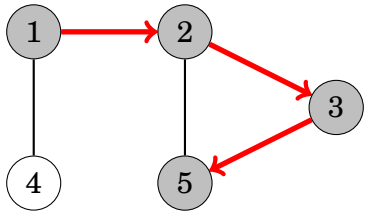


We may begin the search at any node of the graph; now we will begin the search at node 1.

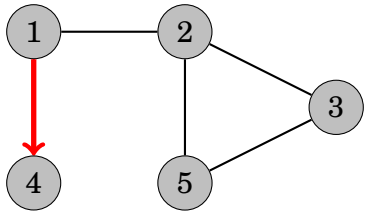
The search first proceeds to node 2:



After this, nodes 3 and 5 will be visited:



The neighbors of node 5 are 2 and 3, but the search has already visited both of them, so it is time to return to the previous nodes. Also the neighbors of nodes 3 and 2 have been visited, so we next move from node 1 to node 4:



After this, the search terminates because it has visited all nodes.

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.

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

```
ArrayList<Integer>[] adj = new ArrayList[N];
```

and also maintains an array

```
boolean[] visited = new boolean[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:

```
public static void dfs(ArrayList<Integer>[] adj, boolean[] visited,
    ↪  int s) {
    if (visited[s]) return;
    visited[s] = true;
    // process node s
    for (int u : adj[s]) {
        dfs(adj, visited, u);
    }
}
```

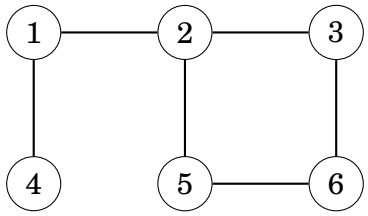
12.2 Breadth-first search

Breadth-first search (BFS) visits the nodes 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.

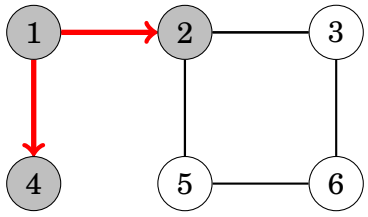
Breadth-first search goes through the nodes one level after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.

Example

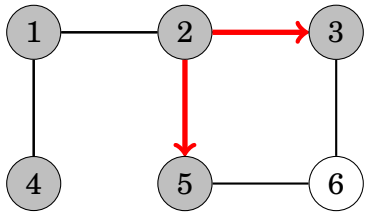
Let us consider how breadth-first search processes the following graph:



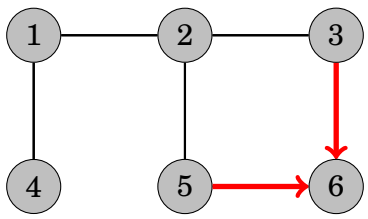
Suppose that the search begins at node 1. First, we process all nodes that can be reached from node 1 using a single edge:



After this, we proceed to nodes 3 and 5:



Finally, we visit node 6:



Now we have calculated the distances from the starting node to all nodes of the graph. The distances are as follows:

node	distance
1	0
2	1
3	2
4	1
5	2
6	3

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.

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:

```
ArrayDeque<Integer> q = new ArrayDeque<>();
boolean[] visited = new boolean[N];
int[] distance = new int[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:

```
visited[x] = true;
distance[x] = 0;
q.add(x);
while (!q.isEmpty()) {
    int s = q.poll();
    // process node s
    for (int u : adj[s]) {
        if (visited[u]) continue;
        visited[u] = true;
        distance[u] = distance[s] + 1;
        q.add(u);
    }
}
```

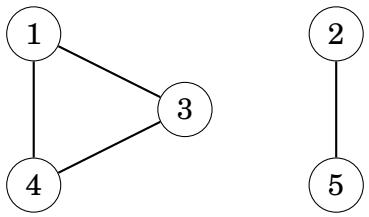
12.3 Applications

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 following applications we will assume that the graph is undirected.

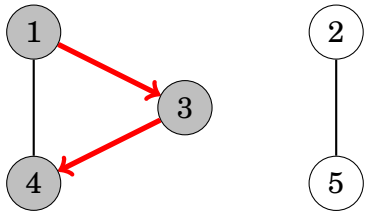
Connectivity check

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 the graph



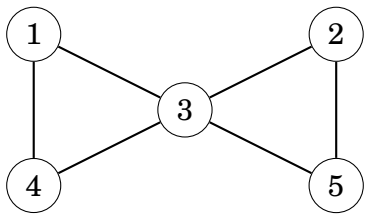
a depth-first search from node 1 visits the following nodes:



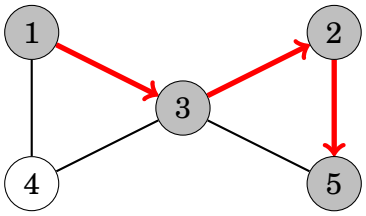
Since the search did not visit all the nodes, we can conclude that the graph is not connected. In a similar way, we can also find all connected components of a graph by iterating through the nodes and always starting a new depth-first search if the current node does not belong to any component yet.

Finding cycles

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, the graph



contains two cycles and we can find one of them as follows:



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, $3 \to 2 \to 5 \to 3$.

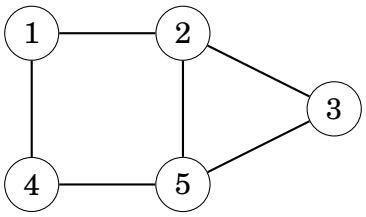
Another way to find out whether a graph contains a cycle is to simply calculate the number of nodes and edges in every component. If a component contains c nodes and no cycle, it must contain exactly $c - 1$ edges (so it has to be a tree). If there are c or more edges, the component surely contains a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

Bipartiteness check

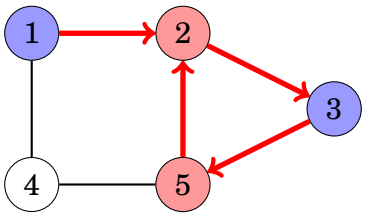
A graph is bipartite if its nodes can be colored using two colors so that there are no adjacent nodes with the same color. It is surprisingly easy to check if a graph is bipartite using graph traversal algorithms.

The idea is to color the starting node blue, all its neighbors red, all their neighbors blue, 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, the graph



is not bipartite, because a search from node 1 proceeds as follows:



We notice that the color of both nodes 2 and 5 is red, while they are adjacent nodes in the graph. Thus, the graph is not bipartite.

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 whether the starting node is red or blue.

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. Even when $k = 3$, no efficient algorithm is known but the problem is NP-hard.

Chapter 13

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

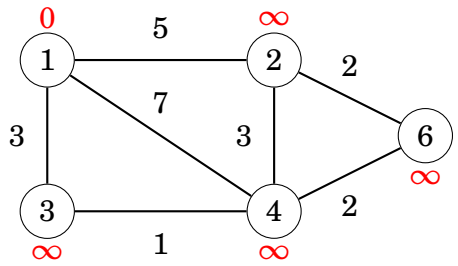
13.1 Bellman–Ford algorithm

The **Bellman–Ford algorithm**[1] 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 all other nodes in infinite. The algorithm reduces the distances by finding edges that shorten the paths until it is not possible to reduce any distance.

Example

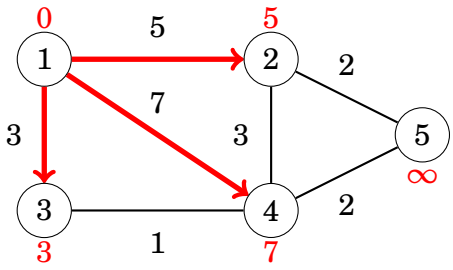
Let us consider how the Bellman–Ford algorithm works in the following graph:



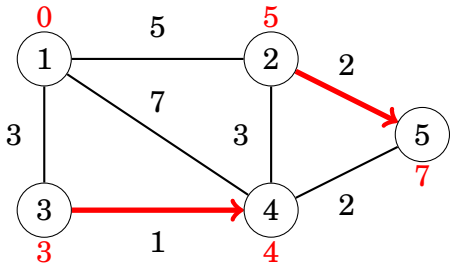
[1]The algorithm is named after R. E. Bellman and L. R. Ford who published it independently in 1958 and 1956, respectively [1, 5].

Each node of the graph is assigned a distance. Initially, the distance to the starting node is 0, and the distance to all other nodes is infinite.

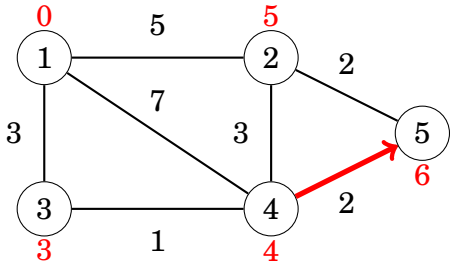
The algorithm searches for edges that reduce distances. First, all edges from node 1 reduce distances:



After this, edges $2 \to 5$ and $3 \to 4$ reduce distances:

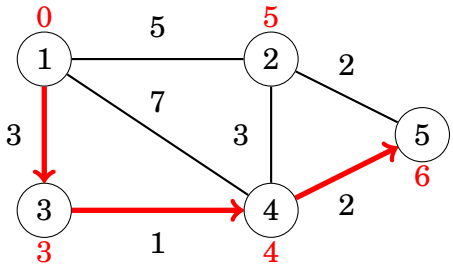


Finally, there is one more change:



After this, no edge can reduce any distance. This means that the distances are final, and we have successfully calculated the shortest distances from the starting node to all nodes of the graph.

For example, the shortest distance 3 from node 1 to node 5 corresponds to the following path:



Implementation

The following implementation of the Bellman–Ford algorithm 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 tries to reduce the distances. The algorithm constructs an array `distance` that will contain the distances from x to all nodes of the graph. The constant `INF` denotes an infinite distance.

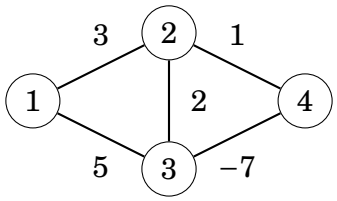
```
int[] distance = new int[N];
for (int i = 1; i <= n; i++)
    distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n - 1; i++) {
    for (Tuple e : edges) {
        int a = e.first, b = e.second, w = e.third;
        distance[b] = Math.min(distance[b], distance[a] + w);
    }
}
```

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.

In practice, the final distances can usually be found faster than in $n-1$ rounds. Thus, a possible way to make the algorithm more efficient is to stop the algorithm if no distance can be reduced during a round.

Negative cycles

The Bellman–Ford algorithm can also be used to check if the graph contains a cycle with negative length. For example, the graph



contains a negative cycle $2 \to 3 \to 4 \to 2$ with length -4.

If the graph contains a negative cycle, we can shorten infinitely many times any path that contains the cycle by repeating the cycle again and again. Thus, the concept of a shortest path is not meaningful in this situation.

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 whole graph regardless of the starting node.

SPFA algorithm

The **SPFA algorithm** ("Shortest Path Faster Algorithm") [3] is a variant of the Bellman–Ford algorithm, that is often more efficient than the original algorithm. The SPFA algorithm does not go through all the edges on each round, but instead, it chooses the edges to be examined in a more intelligent way.

The algorithm maintains a queue of nodes that might be used for reducing the distances. First, the algorithm adds the starting node x to the queue. Then, the algorithm always processes the first node in the queue, and when an edge $a \to b$ reduces a distance, node b is added to the queue.

The efficiency of the SPFA algorithm depends on the structure of the graph: the algorithm is often efficient, but its worst case time complexity is still $O(nm)$ and it is possible to create inputs that make the algorithm as slow as the original Bellman–Ford algorithm.

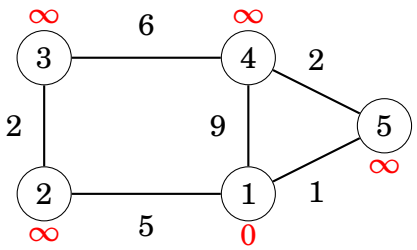
13.2 Dijkstra's algorithm

Dijkstra's algorithm[2] finds shortest paths from the starting node to all nodes of the graph, like the Bellman–Ford algorithm. The benefit of Dijsktra's algorithm is that it 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. Dijkstra's algorithm is efficient, because it only processes each edge in the graph once, using the fact that there are no negative edges.

Example

Let us consider how Dijkstra's algorithm works in the following graph when the starting node is node 1:

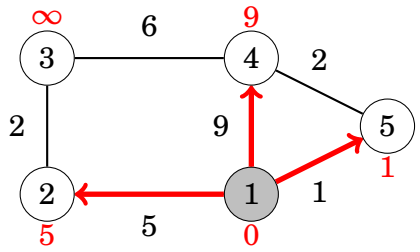


Like in the Bellman–Ford algorithm, initially the distance to the starting node is 0 and the distance to all other nodes is infinite.

[2] E. W. Dijkstra published the algorithm in 1959 [2]; however, his original paper does not mention how to implement the algorithm efficiently.

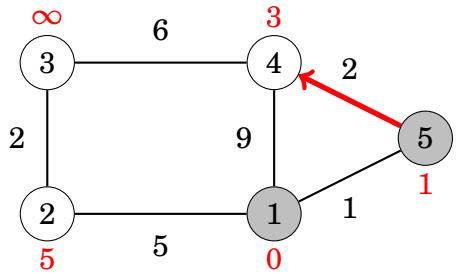
At each step, Dijkstra's algorithm selects a node that has not been processed yet and whose distance is as small as possible. The first such node is node 1 with distance 0.

When a node is selected, the algorithm goes through all edges that start at the node and reduces the distances using them:

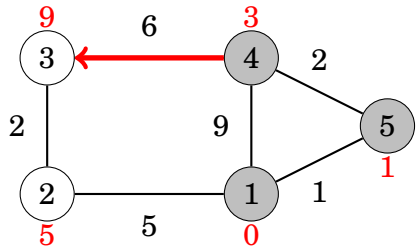


In this case, the edges from node 1 reduced the distances of nodes 2, 4 and 5, whose distances are now 5, 9 and 1.

The next node to be processed is node 5 with distance 1. This reduces the distance to node 4 from 9 to 3:

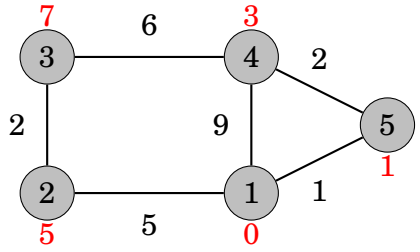


After this, the next node is node 4, which reduces the distance to node 3 to 9:



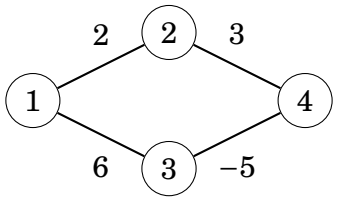
A remarkable property in Dijkstra's algorithm is that whenever a node is selected, its distance is final. For example, at this point of the algorithm, the distances 0, 1 and 3 are the final distances to nodes 1, 5 and 4.

After this, the algorithm processes the two remaining nodes, and the final distances are as follows:



Negative edges

The efficiency of Dijkstra's algorithm is based on the fact that the graph does not contain negative edges. If there is a negative edge, the algorithm may give incorrect results. As an example, consider the following graph:



The shortest path from node 1 to node 4 is $1 \to 3 \to 4$ and its length is 1. However, Dijkstra's algorithm finds the path $1 \to 2 \to 4$ by following the minimum weight edges. The algorithm does not take into account that on the other path, the weight -5 compensates the previous large weight 6.

Implementation

The following implementation of Dijkstra's algorithm calculates the minimum distances from a node x to 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.

An efficient implementation of Dijkstra's algorithm requires that it is possible to efficiently find the minimum distance node that has not been processed. An appropriate data structure for this is a priority queue that contains the nodes ordered by their distances. Using a priority queue, the next node to be processed can be retrieved in logarithmic time. We declare a priority queue of pairs as follows:

```
PriorityQueue<Pair> q = new PriorityQueue<>((a, b) -> {
    return Integer.compare(a.first, b.first);
});
```

The ordering of the elements is determined by the comparator declared in the constructor. In the above, the elements are ordered according to the ascending order of the first element in the pair.

In the following code, the priority queue `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. Initially the distance is 0 to x and ∞ to all other nodes.

```
int[] distance = new int[N];
boolean[] processed = new boolean[N];
for (int i = 1; i <= n; i++)
    distance[i] = INF;
distance[x] = 0;
q.add(new Pair(0, x));
while (!q.isEmpty()) {
    int a = q.poll().second;
    if (processed[a]) continue;
    processed[a] = true;
    for (Pair u : adj[a]) {
        int b = u.first, w = u.second;
        if (distance[a] + w < distance[b]) {
            distance[b] = distance[a] + w;
            q.add(new Pair(distance[b], b));
        }
    }
}
```

Note that there may be several instances of the same node in the priority queue; however, only the instance with the minimum distance will be processed.

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.

13.3 Floyd–Warshall algorithm

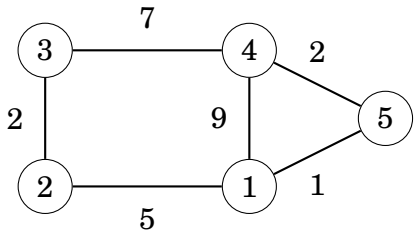
The **Floyd–Warshall algorithm**[3] provides an alternative way to approach the problem of finding shortest paths. Unlike the other algorithms of this chapter, it finds all shortest paths between the nodes in a single run.

The algorithm maintains a two-dimensional array that contains distances between the nodes. First, distances are calculated only using direct edges between the nodes, and after this, the algorithm reduces distances by using intermediate nodes in paths.

Example

Let us consider how the Floyd–Warshall algorithm works in the following graph:

[3]The algorithm is named after R. W. Floyd and S. Warshall who published it independently in 1962 [4, 10].



Initially, the distance from each node to itself is 0, and the distance between nodes a and b is x if there is an edge between nodes a and b with weight x. All other distances are infinite.

In this graph, the initial array is as follows:

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

The algorithm consists of consecutive rounds. On each round, the algorithm selects a new node that can act as an intermediate node in paths from now on, and distances are reduced using this node.

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.

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	**14**	**6**
3	∞	2	0	7	∞
4	9	**14**	7	0	2
5	1	**6**	∞	2	0

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:

	1	2	3	4	5
1	0	5	**7**	9	1
2	5	0	2	14	6
3	**7**	2	0	7	**8**
4	9	14	7	0	2
5	1	6	**8**	2	0

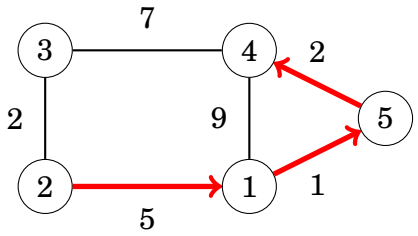
On the third round, node 3 is the new intermediate round. There is a new path between nodes 2 and 4:

	1	2	3	4	5
1	0	5	7	9	1
2	5	0	2	**9**	6
3	7	2	0	7	8
4	9	**9**	7	0	2
5	1	6	8	2	0

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

	1	2	3	4	5
1	0	5	7	3	1
2	5	0	2	8	6
3	7	2	0	7	8
4	3	8	7	0	2
5	1	6	8	2	0

For example, the array tells us that the shortest distance between nodes 2 and 4 is 8. This corresponds to the following path:



Implementation

The advantage of the Floyd–Warshall algorithm that it is easy to implement. The following code constructs a distance matrix where `distance`[a][b] is the shortest distance between nodes a and b. First, the algorithm initializes `distance` using the adjacency matrix `adj` of the graph:

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

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

```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            distance[i][j] = Math.min(distance[i][j],
                                      distance[i][k] + distance[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.

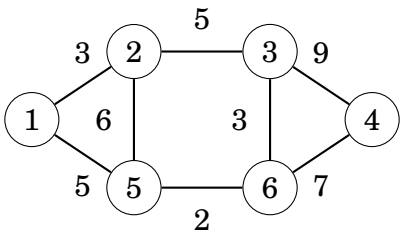
Since the implementation of the Floyd–Warshall algorithm is simple, the algorithm can be a good choice even if it is only needed to find a single shortest path in the graph. However, the algorithm can only be used when the graph is so small that a cubic time complexity is fast enough.

Chapter 15

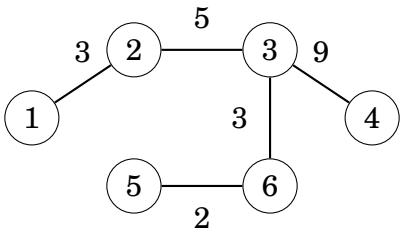
Spanning trees

A **spanning tree** of a graph consists of all nodes of the graph and some of the edges of the graph so that there is a path between any two nodes. Like trees in general, spanning trees are connected and acyclic. Usually there are several ways to construct a spanning tree.

For example, consider the following graph:

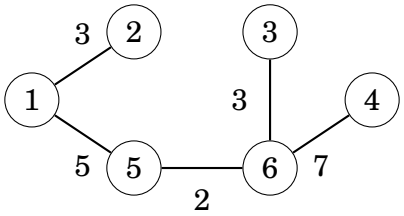


One spanning tree for the graph is as follows:

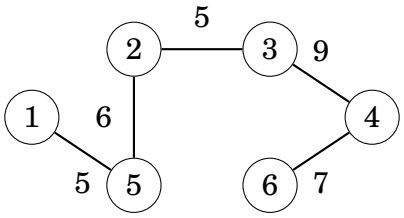


The weight of a spanning tree is the sum of its edge weights. For example, the weight of the above spanning tree is $3+5+9+3+2=22$.

A **minimum spanning tree** is a spanning tree whose weight is as small as possible. The weight of a minimum spanning tree for the example graph is 20, and such a tree can be constructed as follows:



In a similar way, a **maximum spanning tree** is a spanning tree whose weight is as large as possible. The weight of a maximum spanning tree for the example graph is 32:



Note that a graph may have several minimum and maximum spanning trees, so the trees are not unique.

It turns out that several greedy methods can be used to construct minimum and maximum spanning trees. In this chapter, we discuss 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 find maximum spanning trees by processing the edges in reverse order.

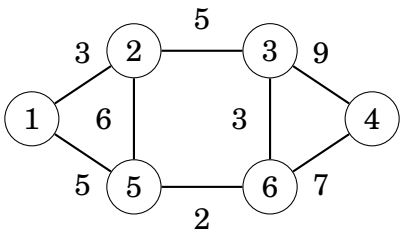
15.1 Kruskal's algorithm

In **Kruskal's algorithm**[1], 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 tree if it does not create a cycle.

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

Example

Let us consider how Kruskal's algorithm processes the following graph:



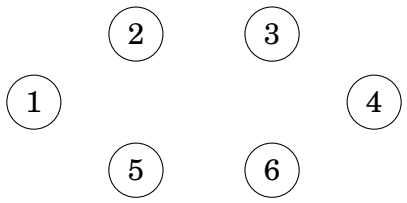
The first step of the algorithm is to sort the edges in increasing order of their weights. The result is the following list:

[1]The algorithm was published in 1956 by J. B. Kruskal [7].

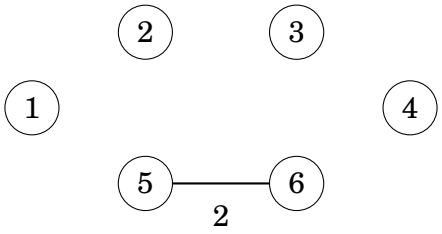
edge	weight
5–6	2
1–2	3
3–6	3
1–5	5
2–3	5
2–5	6
4–6	7
3–4	9

After this, the algorithm goes through the list and adds each edge to the tree if it joins two separate components.

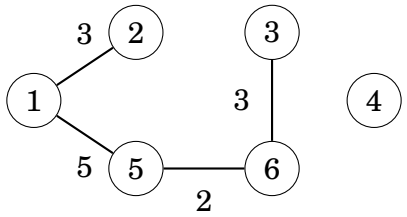
Initially, each node is in its own component:



The first edge to be added to the tree is the edge 5–6 that creates a component $\{5,6\}$ by joining the components $\{5\}$ and $\{6\}$:



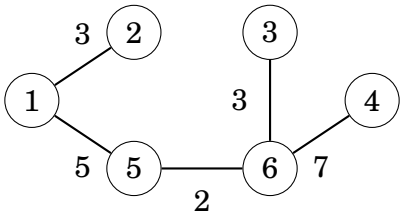
After this, the edges 1–2, 3–6 and 1–5 are added in a similar way:



After those steps, most components have been joined and there are two components in the tree: $\{1,2,3,5,6\}$ and $\{4\}$.

The next edge in the list is the edge 2–3, but it will not be included in the tree, because nodes 2 and 3 are already in the same component. For the same reason, the edge 2–5 will not be included in the tree.

Finally, the edge 4–6 will be included in the tree:

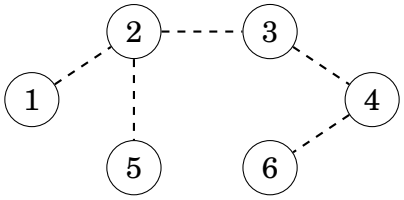


After this, the algorithm will not add any new edges, because the graph is connected and there is a path between any two nodes. The resulting graph is a minimum spanning tree with weight $2+3+3+5+7=20$.

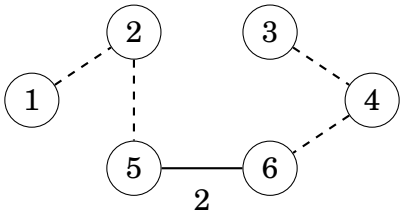
Why does this work?

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 spanning tree for the previous 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 be as follows:



However, it is not possible that the above tree would be a minimum spanning tree for the graph. The reason for this is that we can remove an edge from the tree and replace it with the minimum weight edge 5–6. This produces a spanning tree whose weight is *smaller*:



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 works correctly and 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.

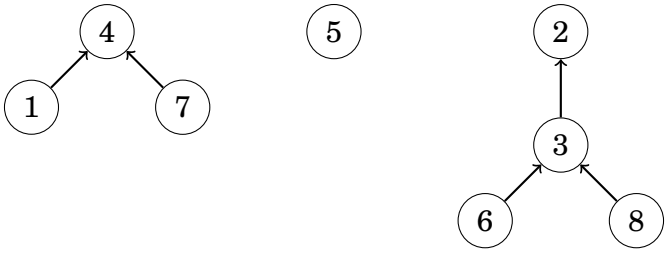
We will solve the problem using a union-find structure that implements both functions in $O(\log n)$ time. Thus, the time complexity of Kruskal's algorithm will be $O(m \log n)$ after sorting the edge list.

15.2 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: the `unite` operation joins two sets, and the `find` operation finds the representative of the set that contains a given element[2].

Structure

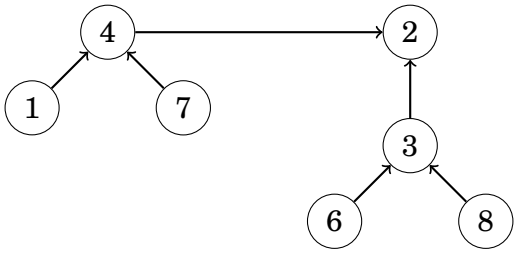
In a union-find structure, one element in each set is the representative of the set, and there is a chain from any other element of the set to the representative. For example, assume that the sets are $\{1,4,7\}$, $\{5\}$ and $\{2,3,6,8\}$:



[2]The structure presented here was introduced in 1971 by J. D. Hopcroft and J. D. Ullman [6]. Later, in 1975, R. E. Tarjan studied a more sophisticated variant of the structure [9] that is discussed in many algorithm textbooks nowadays.

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

Two sets can be joined by connecting the representative of one set to the representative of the other set. For example, the sets $\{1,4,7\}$ and $\{2,3,6,8\}$ can be joined as follows:



The resulting set contains the elements $\{1,2,3,4,6,7,8\}$. 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 chain will be $O(\log n)$, so we can find the representative of any element efficiently by following the corresponding chain.

Implementation

The union-find structure can be implemented using arrays. In the following implementation, the array `link` contains for each element the next element in the chain 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:

```
int[] link = new int[n + 1];
int[] size = new int[n + 1];
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 chain that begins at x. Note that we must include `link` as a parameter to ensure that it is usable by the method.

```
public static int find(int[] link, 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`:

```
public static boolean same(int[] link, int a, int b) {
    return find(link, a) == find(link, 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.

```
public static void unite(int[] link, int[] size, int a, int b) {
    a = find(link, a);
    b = find(link, b);
    if (size[a] > size[b]) {
        size[a] += size[b];
        link[b] = a;
    } else {
        size[b] += size[a];
        link[a] = b;
    }
}
```

The time complexity of the function `find` is $O(\log n)$ assuming that the length of each chain 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 chain is $O(\log n)$ by connecting the smaller set to the larger set.

15.3 Prim's algorithm

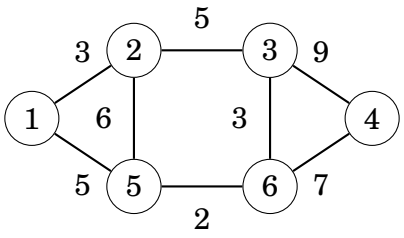
Prim's algorithm[3] is an alternative method for finding a minimum spanning tree. The algorithm first adds an arbitrary node to the tree. After this, the algorithm always chooses a minimum-weight edge that adds a new node to the tree. Finally, all nodes have been added to the tree and a minimum spanning tree has been found.

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

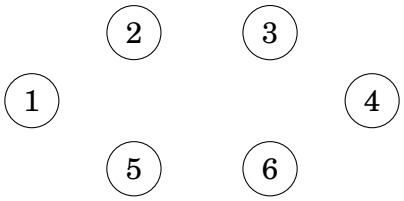
Example

Let us consider how Prim's algorithm works in the following graph:

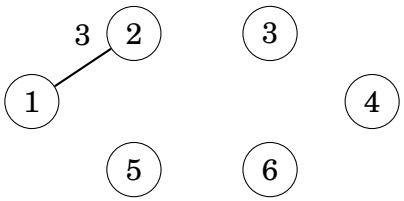
[3]The algorithm is named after R. C. Prim who published it in 1957 [8]. However, the same algorithm was discovered already in 1930 by V. Jarník.



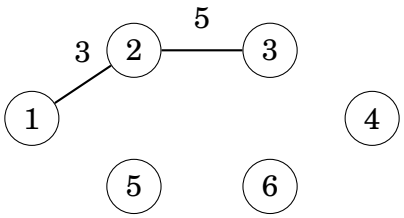
Initially, there are no edges between the nodes:



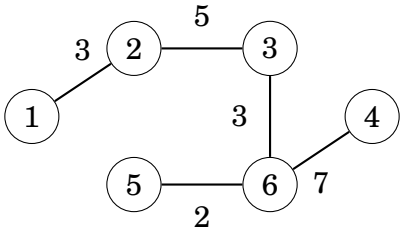
An arbitrary node can be the starting node, so let us choose node 1. First, we add node 2 that is connected by an edge of weight 3:



After this, there are two edges with weight 5, so we can add either node 3 or node 5 to the tree. Let us add node 3 first:



The process continues until all nodes have been included in the tree:



Implementation

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.

Bibliography

[1] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[3] D. Fanding. A faster algorithm for shortest-path – SPFA. *Journal of Southwest Jiaotong University*, 2, 1994.

[4] R. W. Floyd Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[5] L. R. Ford. Network flow theory. RAND Corporation, Santa Monica, California, 1956.

[6] J. E. Hopcroft and J. D. Ullman. A linear list merging algorithm. Technical report, Cornell University, 1971.

[7] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[8] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.

[9] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[10] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.