Part II

Graph algorithms

Chapter 11

Basics of graphs

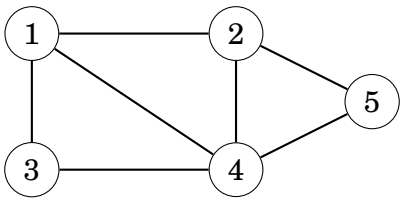
Many programming problems can be solved by modeling the problem as a graph problem and using an appropriate graph algorithm. A typical example of a graph is a network of roads and cities in a country. Sometimes, though, the graph is hidden in the problem and it may be difficult to detect it.

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.

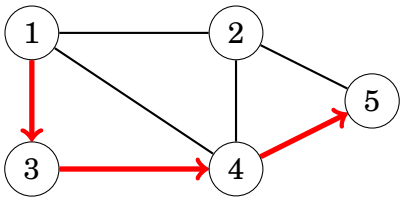
Graph terminology

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:



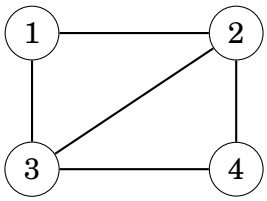
A **path** leads from node a to node b through edges of the graph. The **length** of a path is the number of edges in it. For example, the above graph contains a path $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ of length 3 from node 1 to node 5:



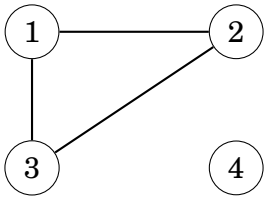
A path is a **cycle** if the first and last node is the same. For example, the above graph contains a cycle $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$. A path is **simple** if each node appears at most once in the path.

Connectivity

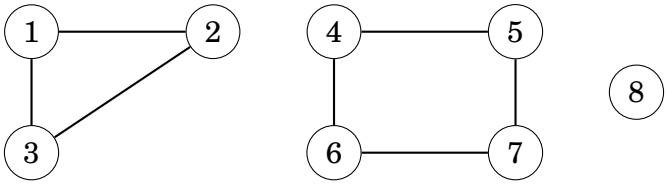
A graph is **connected** if there is a path between any two nodes. For example, the following graph is connected:



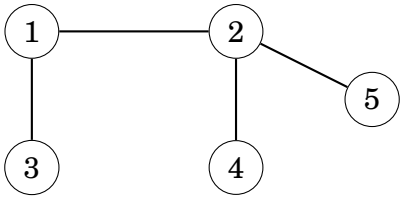
The following graph is not connected, because it is not possible to get from node 4 to any other node:



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 $\{8\}$.

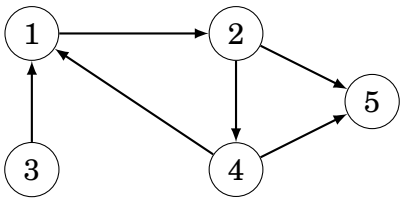


A **tree** is a connected graph that consists of n nodes and $n-1$ edges. There is a unique path between any two nodes of a tree. For example, the following graph is a tree:



Edge directions

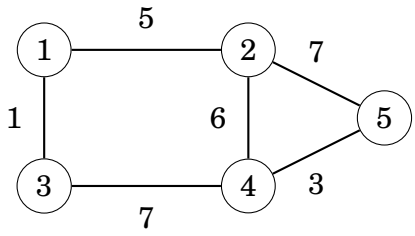
A graph is **directed** if the edges can be traversed in one direction only. For example, the following graph is directed:



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

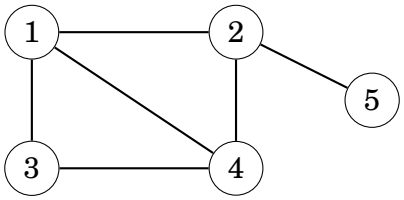
In a **weighted** graph, each edge is assigned a **weight**. The weights are often interpreted as edge lengths. For example, the following graph is weighted:



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

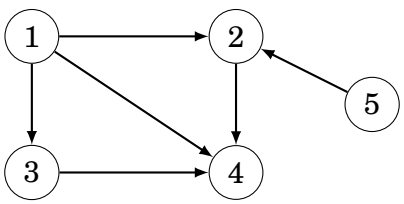
Two nodes are **neighbors** or **adjacent** if there is an edge between them. The **degree** of a node is the number of its neighbors. For example, in the following graph, the neighbors of node 2 are 1, 4 and 5, so its degree is 3.



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.

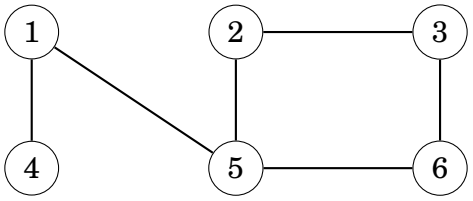
In a directed graph, the **indegree** of a node is the number of edges that end at the node, and the **outdegree** of a node is the number of edges that start at the node. For example, in the following graph, the indegree of node 2 is 2, and the outdegree of node 2 is 1.



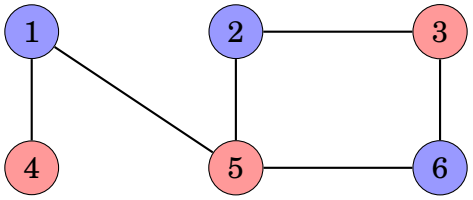
Colorings

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

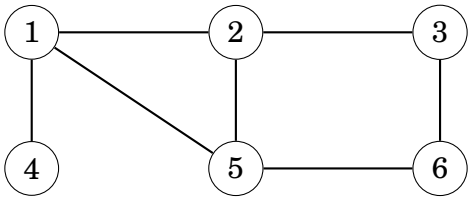
A graph is **bipartite** if it is possible to color it using two colors. It turns out that a graph is bipartite exactly when it does not contain a cycle with an odd number of edges. For example, the graph



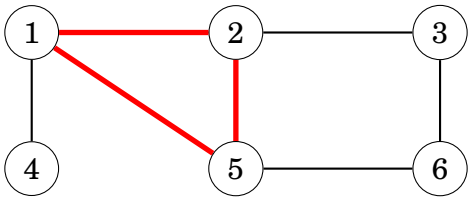
is bipartite, because it can be colored as follows:



However, the graph

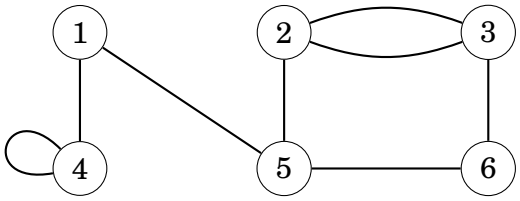


is not bipartite, because it is not possible to color the following cycle of three nodes using two colors:



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:



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.

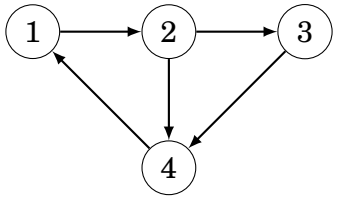
Adjacency list representation

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 vectors as follows:

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

The constant N is chosen so that all adjacency lists can be stored. For example, the graph



can be stored as follows:

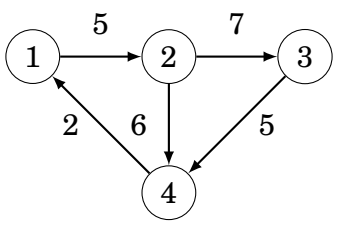
```
adj[1].push_back(2);
adj[2].push_back(3);
adj[2].push_back(4);
adj[3].push_back(4);
adj[4].push_back(1);
```

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



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});
```

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:

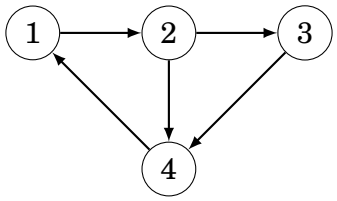
```
for (auto u : adj[s]) {
    // process node u
}
```

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[N][N];
```

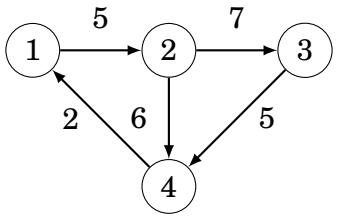
where each value `adj`$[a][b]$ indicates whether the graph contains an edge from node a to node b. If the edge is included in the graph, then `adj`$[a][b] = 1$, and otherwise `adj`$[a][b] = 0$. For example, the graph



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.

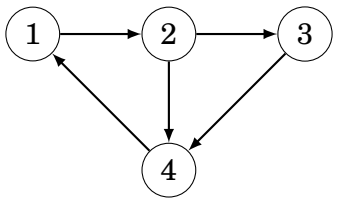
Edge list representation

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 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



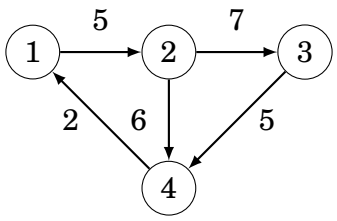
can be represented as follows:

```
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



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});
```

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

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.

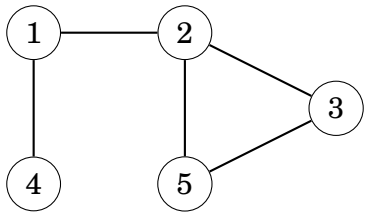
Depth-first search

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.

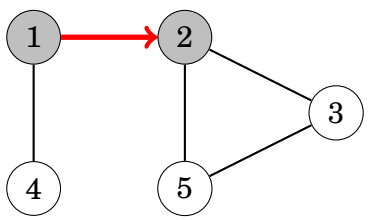
Example

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

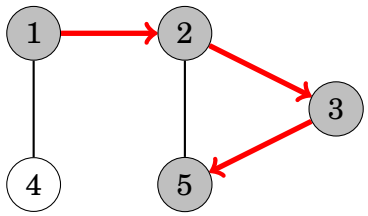


We may begin the search at any node of the graph; now we will begin the search at node 1.

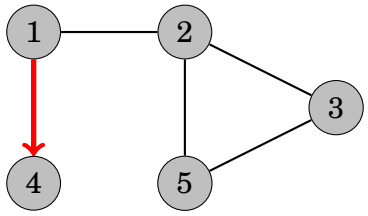
The search first proceeds to node 2:



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

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

and also maintains an array

```
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);
    }
}
```

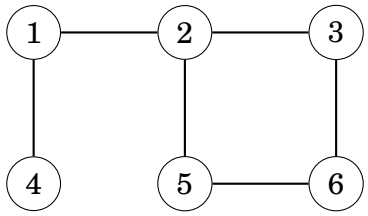
Breadth-first search

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.

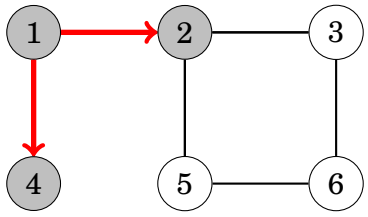
Breadth-first search goes through the nodes one level after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.

Example

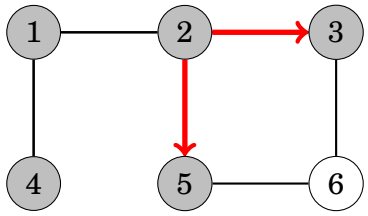
Let us consider how breadth-first search processes the following graph:



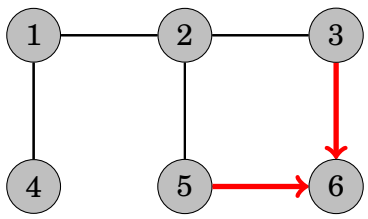
Suppose that the search begins at node 1. First, we process all nodes that can be reached from node 1 using a single edge:



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:

```
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:

```
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);
    }
}
```

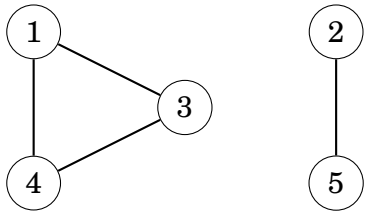
Applications

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.

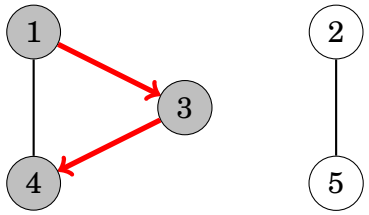
Connectivity check

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



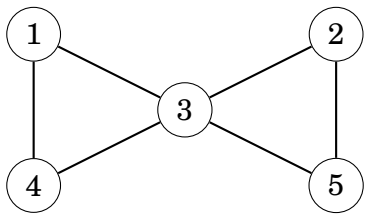
a depth-first search from node 1 visits the following nodes:



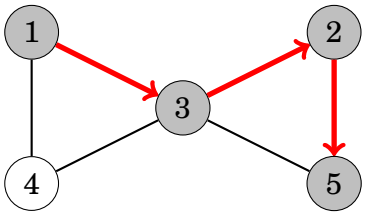
Since the search did not visit all the nodes, we can conclude that the graph is not connected. In a similar way, we can also find all connected components of a graph by iterating through the nodes and always starting a new depth-first search if the current node does not belong to any component yet.

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, $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

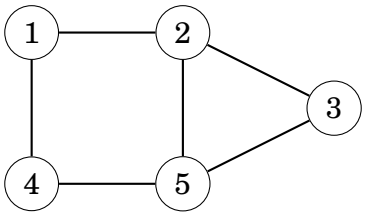
Another way to find out whether a graph contains a cycle is to simply calculate the number of nodes and edges in every component. If a component contains c nodes and no cycle, it must contain exactly $c-1$ edges (so it has to be a tree). If there are c or more edges, the component surely contains a cycle.

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.

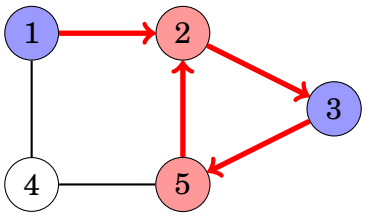
The idea is to color the starting node blue, all its neighbors red, all their neighbors blue, and so on. If at some point of the search we notice that two adjacent nodes have the same color, this means that the graph is not bipartite. Otherwise the graph is bipartite and one coloring has been found.

For example, the graph



BFS is better here

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



We notice that the color or 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.

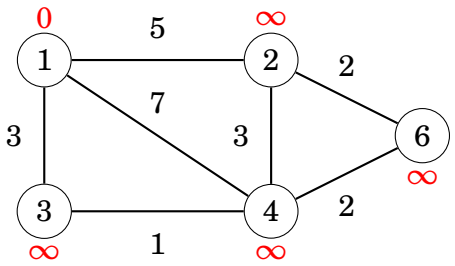
Bellman–Ford algorithm

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

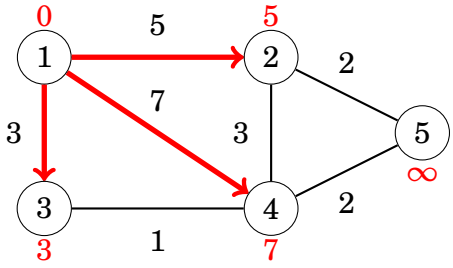
Let us consider how the Bellman–Ford algorithm works in the following graph:



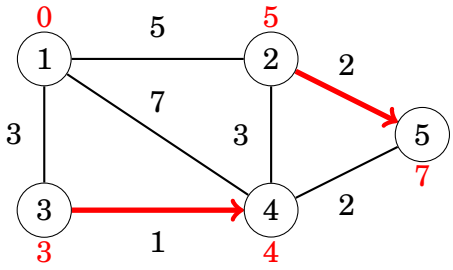
[1]The algorithm is named after R. E. Bellman and L. R. Ford who published it independently in 1958 and 1956, respectively [5, 24].

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.

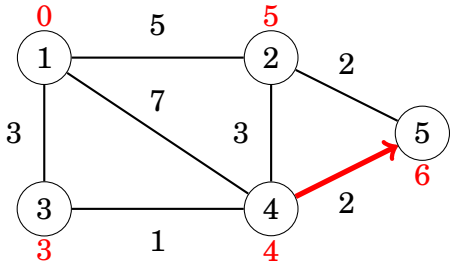
The algorithm searches for edges that reduce distances. First, all edges from node 1 reduce distances:



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

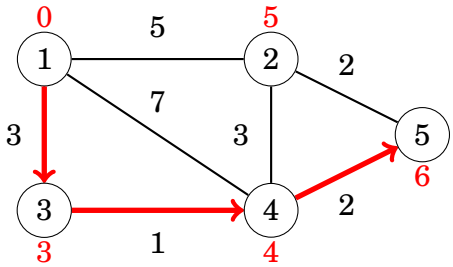


Finally, there is one more change:



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.

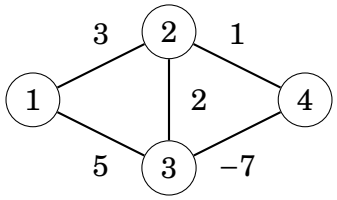
```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a]+w);
    }
}
```

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.

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") [20] 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.

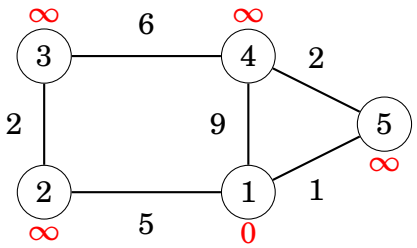
Dijkstra's algorithm

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:

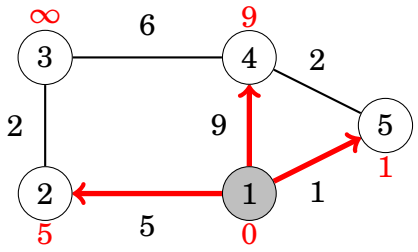


Like in the Bellman–Ford algorithm, initially the distance to the starting node is 0 and the distance to all other nodes is infinite.

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.

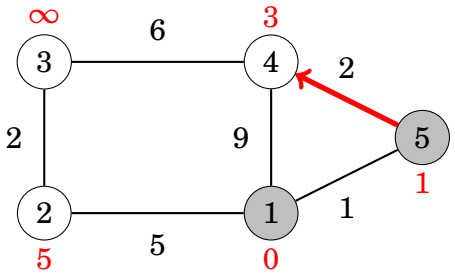
[2] E. W. Dijkstra published the algorithm in 1959 [14]; however, his original paper does not mention how to implement the algorithm efficiently.

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

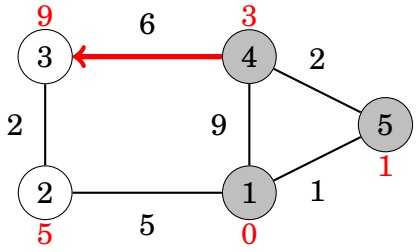


In this case, the edges from node 1 reduced the distances of nodes 2, 4 and 5, whose distances are now 5, 9 and 1.

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

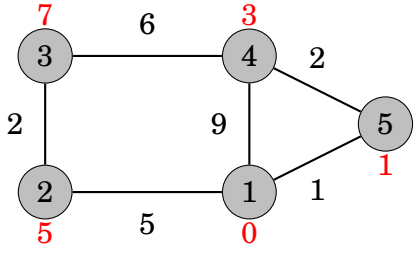


After this, the next node is node 4, which reduces the distance to node 3 to 9:



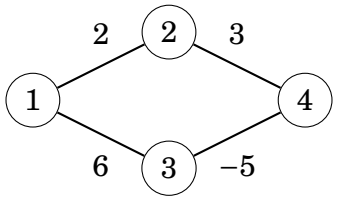
A remarkable property in Dijkstra's algorithm is that whenever a node is selected, its distance is final. For example, at this point of the algorithm, the distances 0, 1 and 3 are the final distances to nodes 1, 5 and 4.

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.

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.

```
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});
        }
    }
}
```

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 using negative distances, we can directly use the default priority queue[3]. Also 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.

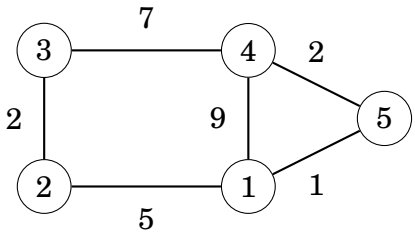
Floyd–Warshall algorithm

The **Floyd–Warshall algorithm**[4] 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:



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

[3] Of course, we could also declare the priority queue as in Chapter 4.5 and use positive distances, but the implementation would be a bit longer.

[4] The algorithm is named after R. W. Floyd and S. Warshall who published it independently in 1962 [23, 70].

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

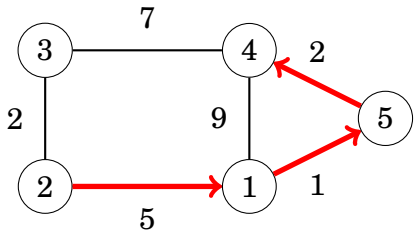
On the third round, node 3 is the new intermediate round. There is a new path between nodes 2 and 4:

	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:

```
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]) 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] = 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.

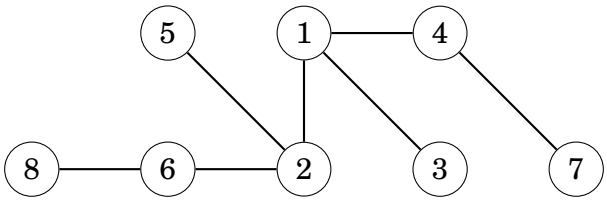
Since the implementation of the Floyd–Warshall algorithm is simple, the algorithm can be a good choice even if it is only needed to find a single shortest path in the graph. However, the algorithm can only be used when the graph is so small that a cubic time complexity is fast enough.

Chapter 14

Tree algorithms

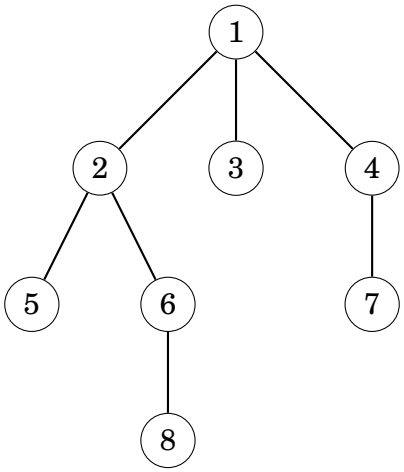
A **tree** is a connected, acyclic graph that consists of n nodes and $n-1$ edges. Removing any edge from a tree divides it into two components, and adding any edge to a tree creates a cycle. Moreover, there is always a unique path between any two nodes of a tree.

For example, the following tree consists of 8 nodes and 7 edges:



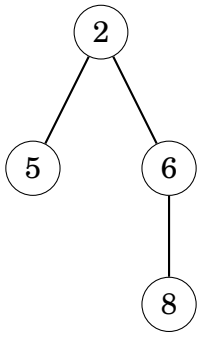
The **leaves** of a tree are the nodes with degree 1, i.e., with only one neighbor. For example, the leaves of the above tree are nodes 3, 5, 7 and 8.

In a **rooted** tree, one of the nodes is appointed the **root** of the tree, and all other nodes are placed underneath the root. For example, in the following tree, node 1 is the root node.



In a rooted tree, the **children** of a node are its lower neighbors, and the **parent** of a node is its upper neighbor. Each node has exactly one parent, except for the root that does not have a parent. For example, in the above tree, the children of node 2 are nodes 5 and 6, and its parent is node 1.

The structure of a rooted tree is *recursive*: each node of the tree acts as the root of a **subtree** that contains the node itself and all nodes that are in the subtrees of its children. For example, in the above tree, the subtree of node 2 consists of nodes 2, 5, 6 and 8:



Tree traversal

General graph traversal algorithms can be used to traverse the nodes of a tree. However, the traversal of a tree is easier to implement than that of a general graph, because there are no cycles in the tree and it is not possible to reach a node from multiple directions.

The typical way to traverse a tree is to start a depth-first search at an arbitrary node. The following recursive function can be used:

```
void dfs(int s, int e) {
    // process node s
    for (auto u : adj[s]) {
        if (u != e) dfs(u, s);
    }
}
```

visited bool check is not needed, because each vertex can be approached from 1 side only

The function is given two parameters: the current node s and the previous node e. The purpose of the parameter e is to make sure that the search only moves to nodes that have not been visited yet.

The following function call starts the search at node x:

```
dfs(x, 0);
```

In the first call $e = 0$, because there is no previous node, and it is allowed to proceed to any direction in the tree.

Dynamic programming

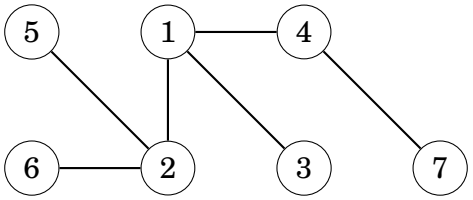
Dynamic programming can be used to calculate some information during a tree traversal. Using dynamic programming, we can, for example, calculate in $O(n)$ time for each node of a rooted tree the number of nodes in its subtree or the length of the longest path from the node to a leaf.

As an example, let us calculate for each node s a value count[s]: the number of nodes in its subtree. The subtree contains the node itself and all nodes in the subtrees of its children, so we can calculate the number of nodes recursively using the following code:

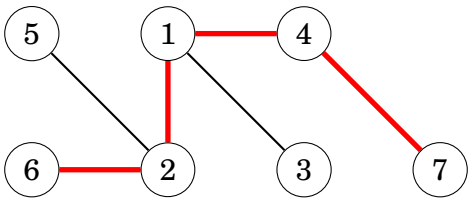
```
void dfs(int s, int e) {
    count[s] = 1;
    for (auto u : adj[s]) {
        if (u == e) continue;
        dfs(u, s);
        count[s] += count[u];
    }
}
```

Diameter

The **diameter** of a tree is the maximum length of a path between two nodes. For example, consider the following tree:



The diameter of this tree is 4, which corresponds to the following path:



Note that there may be several maximum-length paths. In the above path, we could replace node 6 with node 5 to obtain another path with length 4.

Next we will discuss two $O(n)$ time algorithms for calculating the diameter of a tree. The first algorithm is based on dynamic programming, and the second algorithm uses two depth-first searches.

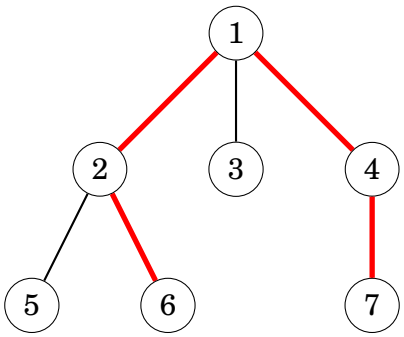
Algorithm 1

A general way to approach many tree problems is to first root the tree arbitrarily. After this, we can try to solve the problem separately for each subtree. Our first algorithm for calculating the diameter is based on this idea.

An important observation is that every path in a rooted tree has a *highest point*: the highest node that belongs to the path. Thus, we can calculate for each

node the length of the longest path whose highest point is the node. One of those paths corresponds to the diameter of the tree.

For example, in the following tree, node 1 is the highest point on the path that corresponds to the diameter:



We calculate for each node x two values:

- $\texttt{toLeaf}(x)$: the maximum length of a path from x to any leaf
- $\texttt{maxLength}(x)$: the maximum length of a path whose highest point is x

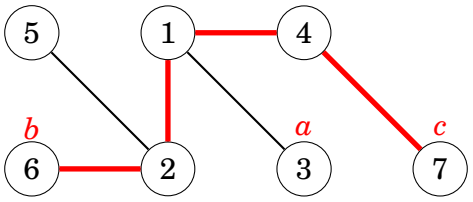
For example, in the above tree, $\texttt{toLeaf}(1) = 2$, because there is a path $1 \to 2 \to 6$, and $\texttt{maxLength}(1) = 4$, because there is a path $6 \to 2 \to 1 \to 4 \to 7$. In this case, $\texttt{maxLength}(1)$ equals the diameter.

Dynamic programming can be used to calculate the above values for all nodes in $O(n)$ time. First, to calculate $\texttt{toLeaf}(x)$, we go through the children of x, choose a child c with maximum $\texttt{toLeaf}(c)$ and add one to this value. Then, to calculate $\texttt{maxLength}(x)$, we choose two distinct children a and b such that the sum $\texttt{toLeaf}(a) + \texttt{toLeaf}(b)$ is maximum and add two to this sum.

Algorithm 2

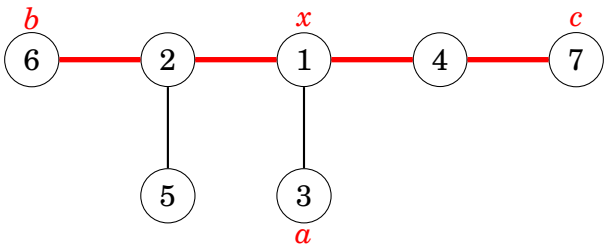
Another efficient way to calculate the diameter of a tree is based on two depth-first searches. First, we choose an arbitrary node a in the tree and find the farthest node b from a. Then, we find the farthest node c from b. The diameter of the tree is the distance between b and c.

In the following graph, a, b and c could be:



This is an elegant method, but why does it work?

It helps to draw the tree differently so that the path that corresponds to the diameter is horizontal, and all other nodes hang from it:

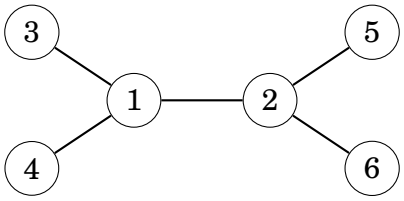


Node x indicates the place where the path from node a joins the path that corresponds to the diameter. The farthest node from a is node b, node c or some other node that is at least as far from node x. Thus, this node is always a valid choice for an endpoint of a path that corresponds to the diameter.

All longest paths

Our next problem is to calculate for every node in the tree the maximum length of a path that begins at the node. This can be seen as a generalization of the tree diameter problem, because the largest of those lengths equals the diameter of the tree. Also this problem can be solved in $O(n)$ time.

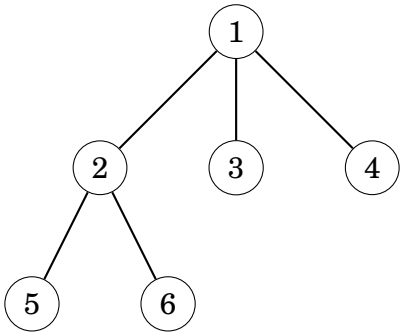
As an example, consider the following tree:



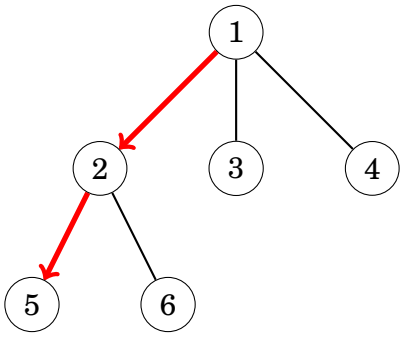
Let $\texttt{maxLength}(x)$ denote the maximum length of a path that begins at node x. For example, in the above tree, $\texttt{maxLength}(4) = 3$, because there is a path $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$. Here is a complete table of the values:

node x	1	2	3	4	5	6
$\texttt{maxLength}(x)$	2	2	3	3	3	3

Also in this problem, a good starting point for solving the problem is to root the tree arbitrarily:

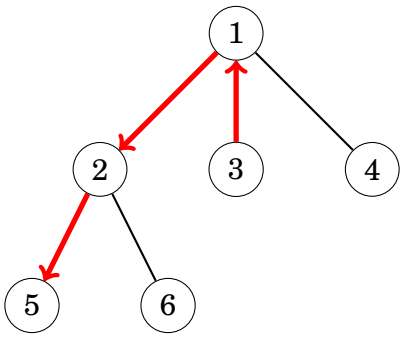


The first part of the problem is to calculate for every node x the maximum length of a path that goes through a child of x. For example, the longest path from node 1 goes through its child 2:

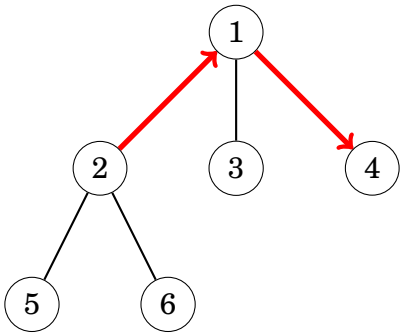


This part is easy to solve in $O(n)$ time, because we can use dynamic programming as we have done previously.

Then, the second part of the problem is to calculate for every node x the maximum length of a path through its parent p. For example, the longest path from node 3 goes through its parent 1:



At first glance, it seems that we should choose the longest path from p. However, this *does not* always work, because the longest path from p may go through x. Here is an example of this situation:



Still, we can solve the second part in $O(n)$ time by storing *two* maximum lengths for each node x:

- $\texttt{maxLength}_1(x)$: the maximum length of a path from x
- $\texttt{maxLength}_2(x)$ the maximum length of a path from x in another direction than the first path

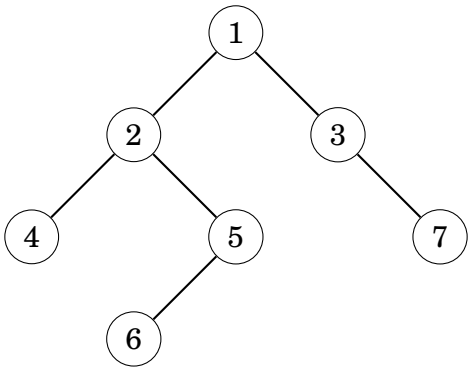
For example, in the above graph, $\texttt{maxLength}_1(1) = 2$ using the path $1 \to 2 \to 5$, and $\texttt{maxLength}_2(1) = 1$ using the path $1 \to 3$.

Finally, if the path that corresponds to $\texttt{maxLength}_1(p)$ goes through x, we conclude that the maximum length is $\texttt{maxLength}_2(p)+1$, and otherwise the maximum length is $\texttt{maxLength}_1(p)+1$.

Binary trees

A **binary tree** is a rooted tree where each node has a left and right subtree. It is possible that a subtree of a node is empty. Thus, every node in a binary tree has zero, one or two children.

For example, the following tree is a binary tree:



The nodes of a binary tree have three natural orderings that correspond to different ways to recursively traverse the tree:

- **pre-order**: first process the root, then traverse the left subtree, then traverse the right subtree
- **in-order**: first traverse the left subtree, then process the root, then traverse the right subtree
- **post-order**: first traverse the left subtree, then traverse the right subtree, then process the root

For the above tree, the nodes in pre-order are $[1,2,4,5,6,3,7]$, in in-order $[4,2,6,5,1,3,7]$ and in post-order $[4,6,5,2,7,3,1]$.

If we know the pre-order and in-order of a tree, we can reconstruct the exact structure of the tree. For example, the above tree is the only possible tree with pre-order $[1,2,4,5,6,3,7]$ and in-order $[4,2,6,5,1,3,7]$. In a similar way, the post-order and in-order also determine the structure of a tree.

However, the situation is different if we only know the pre-order and post-order of a tree. In this case, there may be more than one tree that match the orderings. For example, in both of the trees

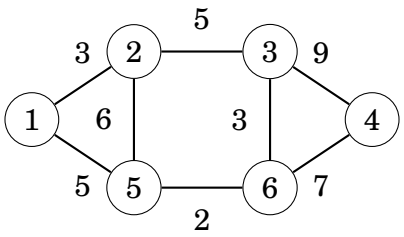
the pre-order is $[1,2]$ and the post-order is $[2,1]$, but the structures of the trees are different.

Chapter 15

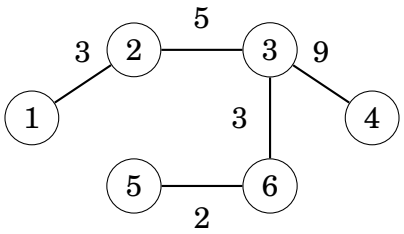
Spanning trees

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:

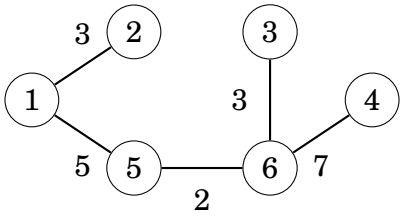


One spanning tree for the graph is as follows:

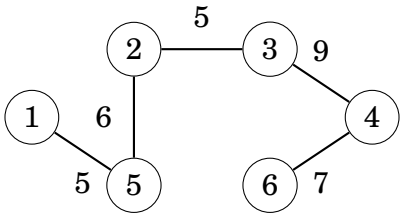


The weight of a spanning tree is the sum of its edge weights. For example, the weight of the above spanning tree is $3+5+9+3+2=22$.

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.

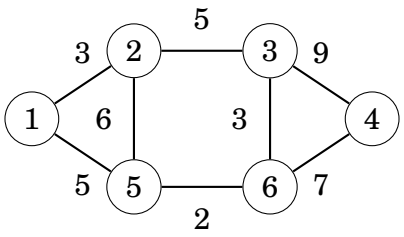
Kruskal's algorithm

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:



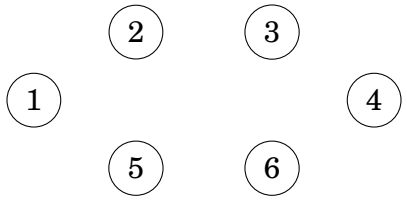
The first step of the algorithm is to sort the edges in increasing order of their weights. The result is the following list:

[1] The algorithm was published in 1956 by J. B. Kruskal [48].

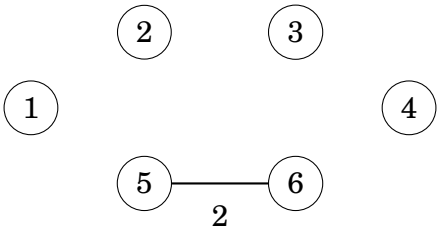
edge	weight
5–6	2
1–2	3
3–6	3
1–5	5
2–3	5
2–5	6
4–6	7
3–4	9

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

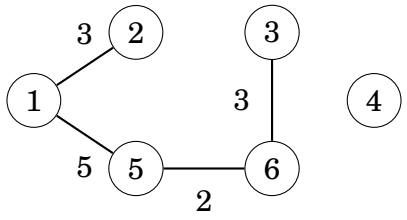
Initially, each node is in its own component:



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 $\{6\}$:



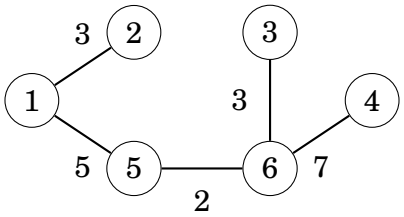
After this, the edges 1–2, 3–6 and 1–5 are added in a similar way:



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:

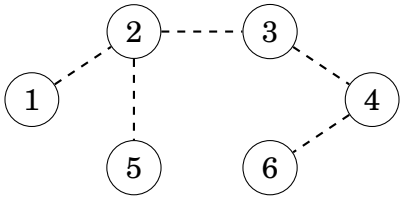


After this, the algorithm will not add any new edges, because the graph is connected and there is a path between any two nodes. The resulting graph is a minimum spanning tree with weight $2+3+3+5+7=20$.

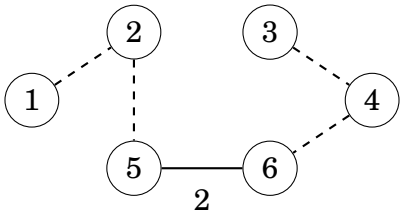
Why does this work?

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.

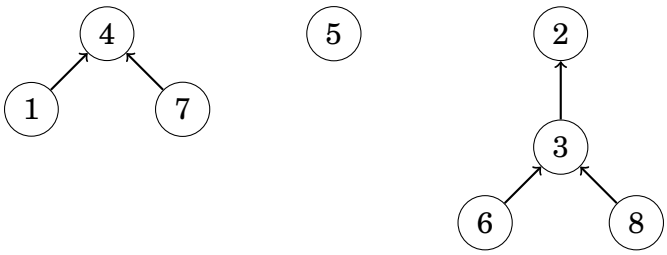
We will solve the problem using a union-find structure that implements both functions in $O(\log n)$ time. Thus, the time complexity of Kruskal's algorithm will be $O(m \log n)$ after sorting the edge list.

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

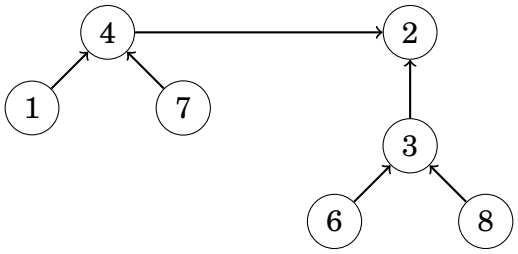
In a union-find structure, one element in each set is the representative of the set, and there is a chain from any other element of the set to the representative. For example, assume that the sets are $\{1,4,7\}$, $\{5\}$ and $\{2,3,6,8\}$:



[2]The structure presented here was introduced in 1971 by J. D. Hopcroft and J. D. Ullman [38]. Later, in 1975, R. E. Tarjan studied a more sophisticated variant of the structure [64] 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:

```
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.

```
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 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.

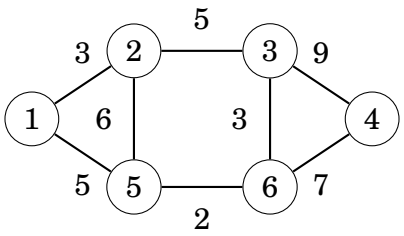
Prim's algorithm

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.

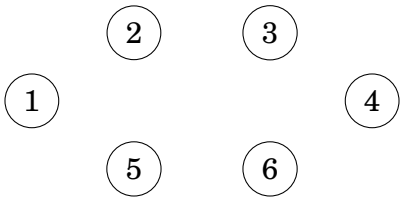
Example

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

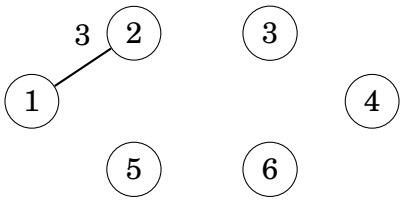


[3]The algorithm is named after R. C. Prim who published it in 1957 [54]. However, the same algorithm was discovered already in 1930 by V. Jarník.

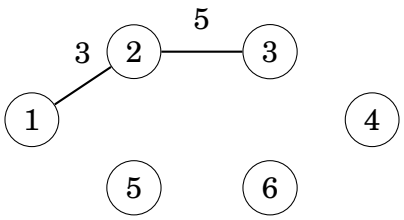
Initially, there are no edges between the nodes:



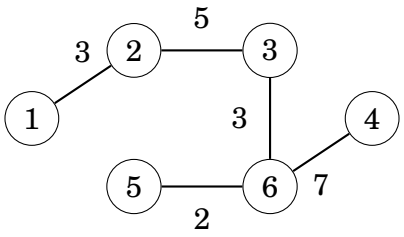
An arbitrary node can be the starting node, so let us choose node 1. First, we add node 2 that is connected by an edge of weight 3:



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.

Chapter 16

Directed graphs

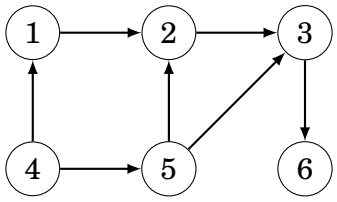
In this chapter, we focus on two classes of directed graphs:

- **Acyclic graphs**: There are no cycles in the graph, so there is no path from any node to itself[1].
- **Successor graphs**: The outdegree of each node is 1, so each node has a unique successor.

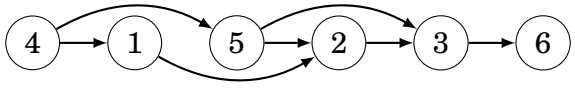
It turns out that in both cases, we can design efficient algorithms that are based on the special properties of the graphs.

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, for the graph



one topological sort is $[4,1,5,2,3,6]$:



An acyclic graph always has a topological sort. However, 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 contain a cycle, to construct a topological sort.

[1] Directed acyclic graphs are sometimes called DAGs.

Algorithm

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)

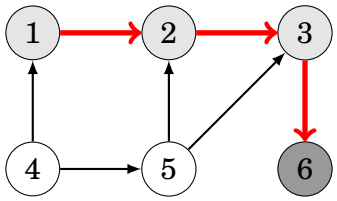
Initially, the state of each node is 0. When a search reaches a node for the first time, its state becomes 1. Finally, after all successors of the node have been processed, its state becomes 2.

If the graph contains a cycle, we will find this out 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.

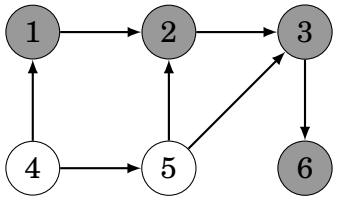
If the graph does not contain a cycle, we can construct a topological sort by adding each node to a list when the state of the node becomes 2. This list in reverse order is a topological sort.

Example 1

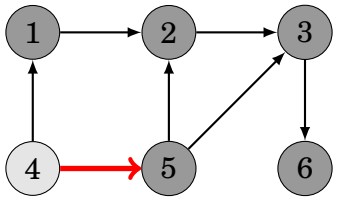
In the example graph, the search first proceeds from node 1 to node 6:



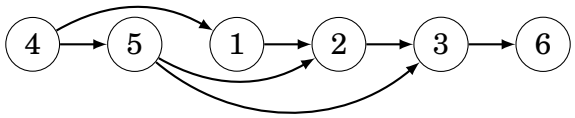
Now node 6 has been processed, so it is added to the list. After this, also nodes 3, 2 and 1 are added to the list:



At this point, the list is $[6,3,2,1]$. The next search begins at node 4:



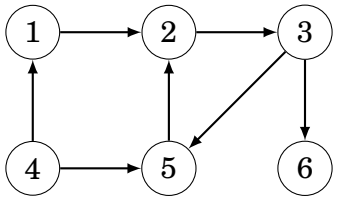
Thus, the final list is $[6,3,2,1,5,4]$. We have processed all nodes, so a topological sort has been found. The topological sort is the reverse list $[4,5,1,2,3,6]$:



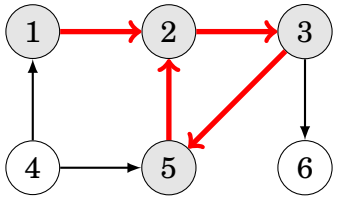
Note that a topological sort is not unique, and there can be several topological sorts for a graph.

Example 2

Let us now consider a graph for which we cannot construct a topological sort, because the graph contains a cycle:



The search proceeds as follows:



The search reaches node 2 whose state is 1, which means that the graph contains a cycle. In this example, there is a cycle $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$.

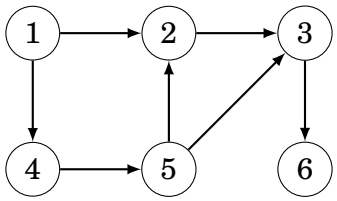
Dynamic programming

If a directed graph is acyclic, dynamic programming can be applied to it. For example, we can efficiently solve the following problems concerning paths from a starting node to an ending node:

- how many different paths are there?
- what is the shortest/longest path?
- what is the minimum/maximum number of edges in a path?
- which nodes certainly appear in any path?

Counting the number of paths

As an example, let us calculate the number of paths from node 1 to node 6 in the following graph:



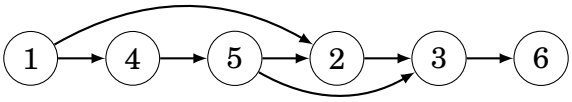
There are a total of three such paths:

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

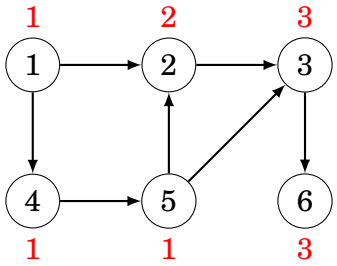
Let $\texttt{paths}(x)$ denote the number of paths from node 1 to node x. As a base case, $\texttt{paths}(1) = 1$. Then, to calculate other values of $\texttt{paths}(x)$, we may use the recursion

$$\texttt{paths}(x) = \texttt{paths}(a_1) + \texttt{paths}(a_2) + \cdots + \texttt{paths}(a_k)$$

where $a_1, a_2, \ldots, a_k$ are the nodes from which there is an edge to x. Since the graph is acyclic, the values of $\texttt{paths}(x)$ can be calculated in the order of a topological sort. A topological sort for the above graph is as follows:



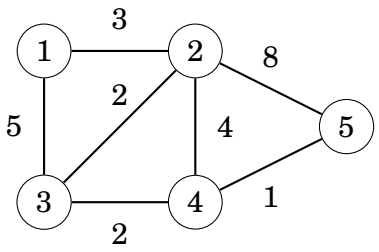
Hence, the numbers of paths are as follows:



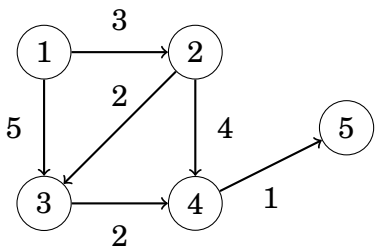
For example, to calculate the value of $\texttt{paths}(3)$, we can use the formula $\texttt{paths}(2) + \texttt{paths}(5)$, because there are edges from nodes 2 and 5 to node 3. Since $\texttt{paths}(2) = 2$ and $\texttt{paths}(5) = 1$, we conclude that $\texttt{paths}(3) = 3$.

Extending Dijkstra's algorithm

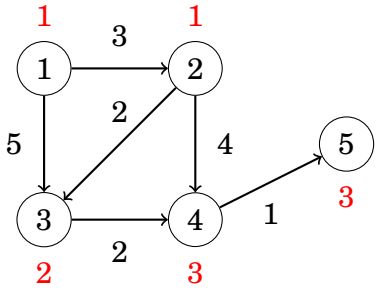
A by-product of Dijkstra's algorithm is a directed, acyclic graph that indicates for each node of the original graph the possible ways to reach the node using a shortest path from the starting node. Dynamic programming can be applied to that graph. For example, in the graph



the shortest paths from node 1 may use the following edges:



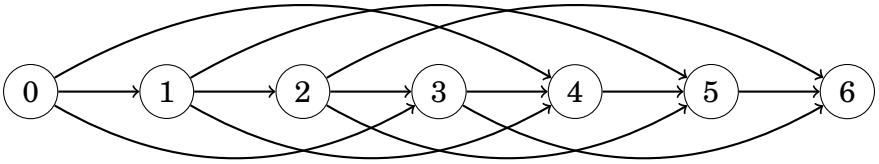
Now we can, for example, calculate the number of shortest paths from node 1 to node 5 using dynamic programming:



Representing problems as graphs

Actually, any dynamic programming problem can be represented as a directed, acyclic graph. In such a graph, each node corresponds to a dynamic programming state and the edges indicate how the states depend on each other.

As an example, consider the problem of forming a sum of money n using coins $\{c_1, c_2, \ldots, c_k\}$. In this problem, 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, for coins $\{1, 3, 4\}$ and $n = 6$, the graph is as follows:



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.

Successor paths

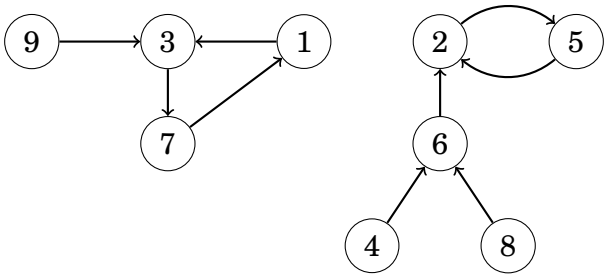
For the rest of the chapter, we will focus on **successor graphs**. In those graphs, the outdegree of each node is 1, i.e., exactly one edge starts at each node. A successor graph consists of one or more components, each of which contains one cycle and some paths that lead to it.

Successor graphs are sometimes called **functional graphs**. The reason for this is that any successor graph corresponds to a function that defines the edges of the graph. The parameter for the function is a node of the graph, and the function gives the successor of that node.

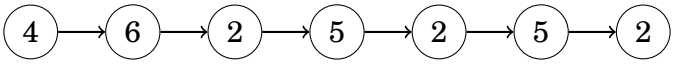
For example, the function

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

defines the following graph:



Since each node of a successor graph has a unique successor, we can also define a function 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 the above graph succ$(4,6) = 2$, because we will reach node 2 by walking 6 steps from node 4:



A straightforward way to calculate a value of 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 succ(x,k) can be calculated in only $O(\log k)$ time.

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

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

Precalculating the values takes $O(n \log u)$ time, because $O(\log u)$ values are calculated for each node. In the above 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
$\cdots$									

After this, any value of $\text{succ}(x,k)$ can be calculated by presenting the number of steps k as a sum of powers of two. For example, if we want to calculate the value of $\text{succ}(x,11)$, we first form the representation $11 = 8+2+1$. Using that,

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

For example, in the previous graph

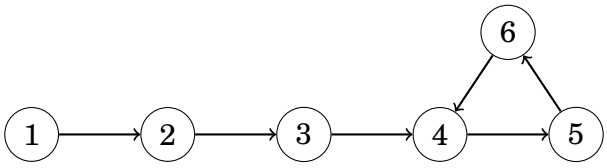
$$\text{succ}(4,11) = \text{succ}(\text{succ}(\text{succ}(4,8),2),1) = 5.$$

Such a representation always consists of $O(\log k)$ parts, so calculating a value of $\text{succ}(x,k)$ takes $O(\log k)$ time.

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 the graph



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. This is an important improvement if n is large. Next we will discuss Floyd's algorithm that achieves these properties.

Floyd's algorithm

Floyd's algorithm[2] walks forward in the graph using two pointers a and b. Both pointers begin at a node x that is the starting node of the graph. 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++;
}
```

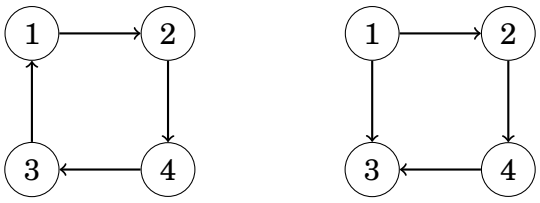
[2] The idea of the algorithm is mentioned in [46] and attributed to R. W. Floyd; however, it is not known if Floyd actually discovered the algorithm.

Chapter 17

Strong connectivity

In a directed graph, the edges can be traversed in one direction only, so even if the graph is connected, this does not guarantee that there would be a path from a node to another node. For this reason, it is meaningful to define a new concept that requires more than connectivity.

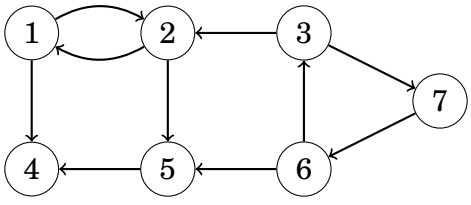
A graph is **strongly connected** if there is a path from any node to all other nodes in the graph. For example, in the following picture, the left graph is strongly connected while the right graph is not.



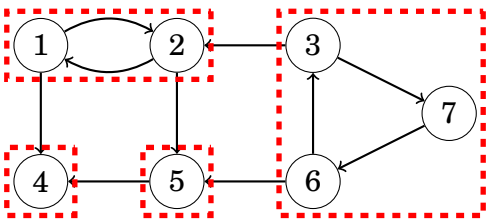
The right graph is not strongly connected because, for example, there is no path from node 2 to node 1.

The **strongly connected components** of a graph divide the graph into strongly connected parts that are as large as possible. The strongly connected components form an acyclic **component graph** that represents the deep structure of the original graph.

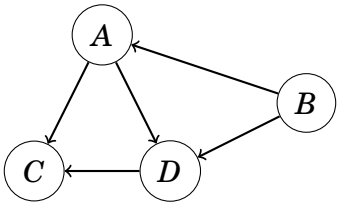
For example, for the graph



the strongly connected components are as follows:



The corresponding component graph is as follows:



The components are $A = \{1,2\}$, $B = \{3,6,7\}$, $C = \{4\}$ and $D = \{5\}$.

A component graph is an acyclic, directed graph, so it is easier to process than the original graph. Since the graph does not contain cycles, we can always construct a topological sort and use dynamic programming techniques like those presented in Chapter 16.

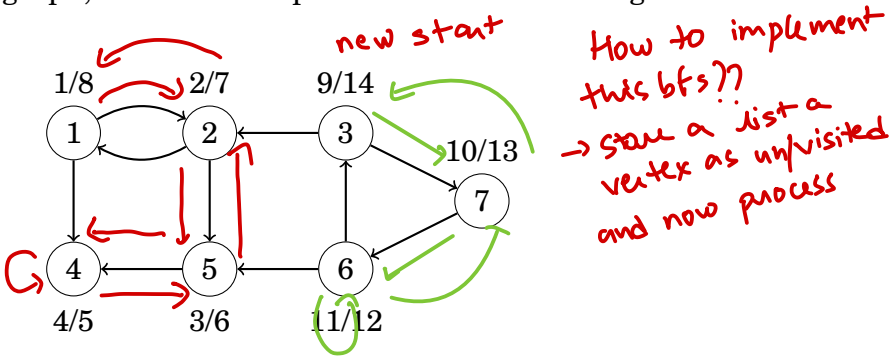
Kosaraju's algorithm

Kosaraju's algorithm[1] is an efficient method for finding the strongly connected components of a directed graph. The algorithm performs two depth-first searches: the first search constructs a list of nodes according to the structure of the graph, and the second search forms the strongly connected components.

Search 1

The first phase of Kosaraju's algorithm constructs a list of nodes in the order in which a depth-first search processes them. The algorithm goes through the nodes, and begins a depth-first search at each unprocessed node. Each node will be added to the list after it has been processed.

In the example graph, the nodes are processed in the following order:



The notation x/y means that processing the node started at time x and finished at time y. Thus, the corresponding list is as follows:

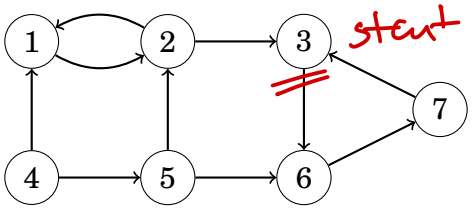
[1] According to [1], S. R. Kosaraju invented this algorithm in 1978 but did not publish it. In 1981, the same algorithm was rediscovered and published by M. Sharir [57].

node	processing time
4	5
5	6
2	7
1	8
6	12
7	13
3	14

Search 2

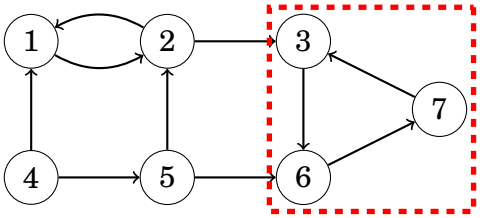
The second phase of the algorithm forms the strongly connected components of the graph. First, the algorithm reverses every edge in the graph. This guarantees that during the second search, we will always find strongly connected components that do not have extra nodes.

After reversing the edges, the example graph is as follows:



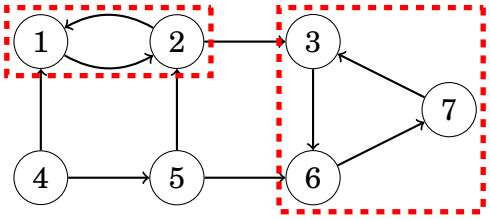
After this, the algorithm goes through the list of nodes created by the first search, in *reverse* order. If a node does not belong to a component, the algorithm creates a new component and starts a depth-first search that adds all new nodes found during the search to the new component.

In the example graph, the first component begins at node 3:

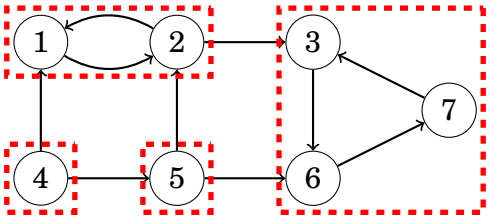


Note that since all edges are reversed, the component does not "leak" to other parts in the graph.

The next nodes in the list are nodes 7 and 6, but they already belong to a component, so the next new component begins at node 1:



Finally, the algorithm processes nodes 5 and 4 that create the remaining strongly connected components:



The time complexity of the algorithm is $O(n+m)$, because the algorithm performs two depth-first searches.

2SAT problem

Strong connectivity is also linked with the **2SAT problem**[2]. In this problem, we are given a logical formula

$$(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge \cdots \wedge (a_m \vee b_m),$$

where each a_i and b_i is either a logical variable ($x_1, x_2, \ldots, x_n$) or a negation of a logical variable ($\neg x_1, \neg x_2, \ldots, \neg x_n$). The symbols "$\wedge$" and "$\vee$" denote logical operators "and" and "or". Our task is to assign each variable a value so that the formula is true, or state that this is not possible.

For example, the formula

$$L_1 = (x_2 \vee \neg x_1) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_4)$$

is true when the variables are assigned as follows:

$$\begin{cases} x_1 = \text{false} \\ x_2 = \text{false} \\ x_3 = \text{true} \\ x_4 = \text{true} \end{cases}$$

[2]The algorithm presented here was introduced in [4]. There is also another well-known linear-time algorithm [19] that is based on backtracking.

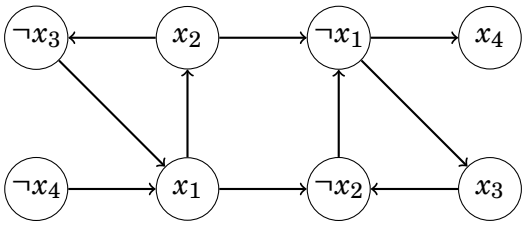
However, the formula

$$L_2 = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$$

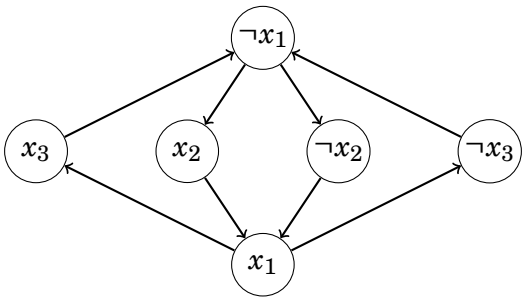
is always false, regardless of how we assign the values. The reason for this is that we cannot choose a value for x_1 without creating a contradiction. If x_1 is false, both x_2 and $\neg x_2$ should be true which is impossible, and if x_1 is true, both x_3 and $\neg x_3$ should be true which is also impossible.

The 2SAT problem can be represented as a graph whose nodes correspond to variables x_i and negations $\neg x_i$, and edges determine the connections between the variables. Each pair $(a_i \vee b_i)$ generates two edges: $\neg a_i \to b_i$ and $\neg b_i \to a_i$. This means that if a_i does not hold, b_i must hold, and vice versa.

The graph for the formula L_1 is:



And the graph for the formula L_2 is:



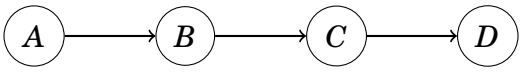
The structure of the graph tells us whether it is possible to assign the values of the variables so that the formula is true. It turns out that this can be done exactly when there are no nodes x_i and $\neg x_i$ such that both nodes belong to the same strongly connected component. If there are such nodes, the graph contains a path from x_i to $\neg x_i$ and also a path from $\neg x_i$ to x_i, so both x_i and $\neg x_i$ should be true which is not possible.

In the graph of the formula L_1 there are no nodes x_i and $\neg x_i$ such that both nodes belong to the same strongly connected component, so a solution exists. In the graph of the formula L_2 all nodes belong to the same strongly connected component, so a solution does not exist.

If a solution exists, the values for the variables can be found by going through the nodes of the component graph in a reverse topological sort order. At each step, we process a component that does not contain edges that lead to an unprocessed component. If the variables in the component have not been assigned values, their values will be determined according to the values in the component, and if

they already have values, they remain unchanged. The process continues until each variable has been assigned a value.

The component graph for the formula L_1 is as follows:



The components are $A = \{\neg x_4\}$, $B = \{x_1, x_2, \neg x_3\}$, $C = \{\neg x_1, \neg x_2, x_3\}$ and $D = \{x_4\}$. When constructing the solution, we first process the component D where x_4 becomes true. After this, we process the component C where x_1 and x_2 become false and x_3 becomes true. All variables have been assigned values, so the remaining components A and B do not change the variables.

Note that this method works, because the graph has a special structure: if there are paths from node x_i to node x_j and from node x_j to node $\neg x_j$, then node x_i never becomes true. The reason for this is that there is also a path from node $\neg x_j$ to node $\neg x_i$, and both x_i and x_j become false.

A more difficult problem is the **3SAT problem**, where each part of the formula is of the form $(a_i \vee b_i \vee c_i)$. This problem is NP-hard, so no efficient algorithm for solving the problem is known.

Chapter 18

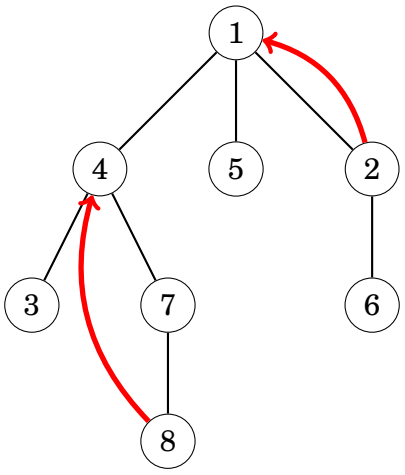
Tree queries

This chapter discusses techniques for processing queries on subtrees and paths of a rooted tree. For example, such queries are:

- what is the kth ancestor of a node? log k
- what is the sum of values in the subtree of a node? dfs
- what is the sum of values on a path between two nodes?
- what is the lowest common ancestor of two nodes?

Finding ancestors

The kth **ancestor** of a node x in a rooted tree is the node that we will reach if we move k levels up from x. Let $\texttt{ancestor}(x,k)$ denote the kth ancestor of a node x (or 0 if there is no such an ancestor). For example, in the following tree, $\texttt{ancestor}(2,1) = 1$ and $\texttt{ancestor}(8,2) = 4$.



An easy way to calculate any value of $\texttt{ancestor}(x,k)$ is to perform a sequence of k moves in the tree. However, the time complexity of this method is $O(k)$, which may be slow, because a tree of n nodes may have a chain of n nodes.

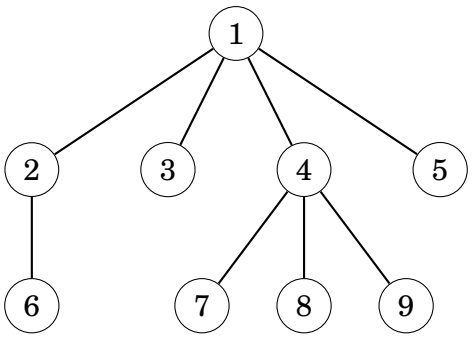
Fortunately, using a technique similar to that used in Chapter 16.3, any value of $\texttt{ancestor}(x,k)$ can be efficiently calculated in $O(\log k)$ time after preprocessing. The idea is to precalculate all values $\texttt{ancestor}(x,k)$ where $k \leq n$ is a power of two. For example, the values for the above tree are as follows:

x	1	2	3	4	5	6	7	8
$\texttt{ancestor}(x,1)$	0	1	4	1	1	2	4	7
$\texttt{ancestor}(x,2)$	0	0	1	0	0	1	1	4
$\texttt{ancestor}(x,4)$	0	0	0	0	0	0	0	0
$\cdots$								

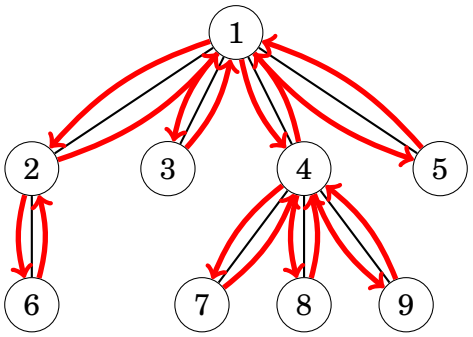
The preprocessing takes $O(n \log n)$ time, because $O(\log n)$ values are calculated for each node. After this, any value of $\texttt{ancestor}(x,k)$ can be calculated in $O(\log k)$ time by representing k as a sum where each term is a power of two.

Subtrees and paths

A **tree traversal array** contains the nodes of a rooted tree in the order in which a depth-first search from the root node visits them. For example, in the tree



a depth-first search proceeds as follows:



Hence, the corresponding tree traversal array is as follows:

1	2	6	3	4	7	8	9	5

Subtree queries

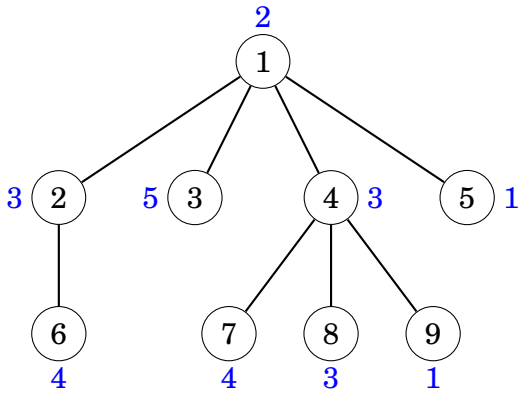
Each subtree of a tree corresponds to a subarray of the tree traversal array such that the first element of the subarray is the root node. For example, the following subarray contains the nodes of the subtree of node 4:

1	2	6	3	4	7	8	9	5

Using this fact, we can efficiently process queries that are related to subtrees of a tree. As an example, consider a problem where each node is assigned a value, and our task is to support the following queries:

- update the value of a node
- calculate the sum of values in the subtree of a node

Consider the following tree where the blue numbers are the values of the nodes. For example, the sum of the subtree of node 4 is $3+4+3+1=11$.



The idea is to construct a tree traversal array that contains three values for each node: the identifier of the node, the size of the subtree, and the value of the node. For example, the array for the above tree is as follows:

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
node value	2	3	4	5	3	4	3	1	1

Using this array, we can calculate the sum of values in any subtree by first finding out the size of the subtree and then the values of the corresponding nodes. For example, the values in the subtree of node 4 can be found as follows:

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
node value	2	3	4	5	3	4	3	1	1

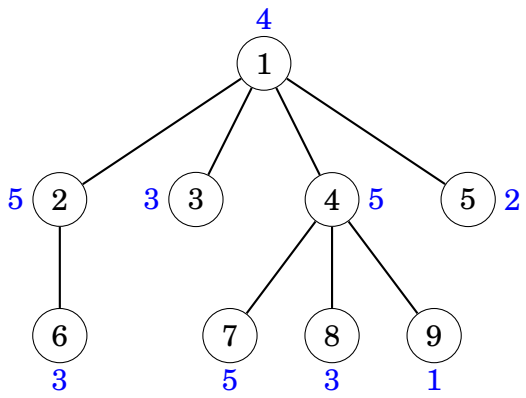
To answer the queries efficiently, it suffices to store the values of the nodes in a binary indexed or segment tree. After this, we can both update a value and calculate the sum of values in $O(\log n)$ time.

Path queries

Using a tree traversal array, we can also efficiently calculate sums of values on paths from the root node to any node of the tree. Consider a problem where our task is to support the following queries:

- change the value of a node
- calculate the sum of values on a path from the root to a node

For example, in the following tree, the sum of values from the root node to node 7 is $4+5+5=14$:



We can solve this problem like before, but now each value in the last row of the array is the sum of values on a path from the root to the node. For example, the following array corresponds to the above tree:

node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
path sum	4	9	12	7	9	14	12	10	6

← dfs again

When the value of a node increases by x, the sums of all nodes in its subtree increase by x. For example, if the value of node 4 increases by 1, the array changes as follows:

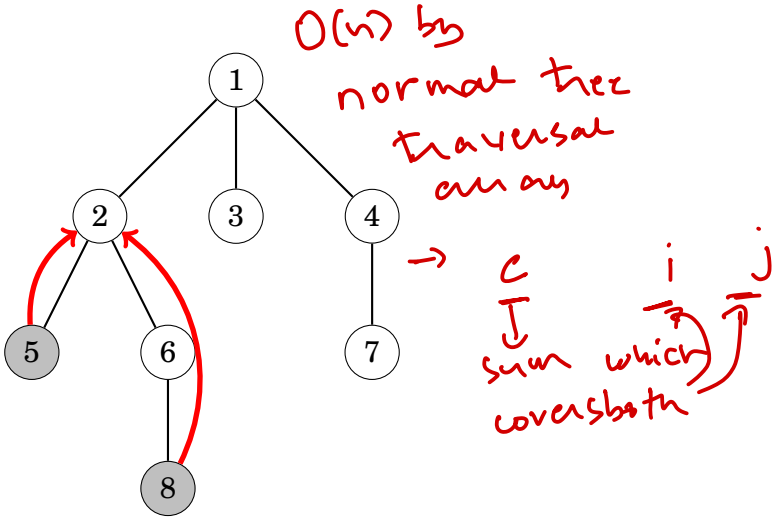
node id	1	2	6	3	4	7	8	9	5
subtree size	9	2	1	1	4	1	1	1	1
path sum	4	9	12	7	10	15	13	11	6

Thus, to support both the operations, we should be able to increase all values in a range and retrieve a single value. This can be done in $O(\log n)$ time using a binary indexed or segment tree (see Chapter 9.4).

Lowest common ancestor

The **lowest common ancestor** of two nodes of a rooted tree is the lowest node whose subtree contains both the nodes. A typical problem is to efficiently process queries that ask to find the lowest common ancestor of two nodes.

For example, in the following tree, the lowest common ancestor of nodes 5 and 8 is node 2:



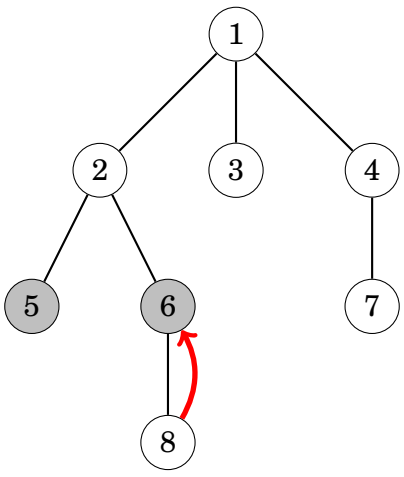
Next we will discuss two efficient techniques for finding the lowest common ancestor of two nodes.

Method 1

One way to solve the problem is to use the fact that we can efficiently find the kth ancestor of any node in the tree. Using this, we can divide the problem of finding the lowest common ancestor into two parts.

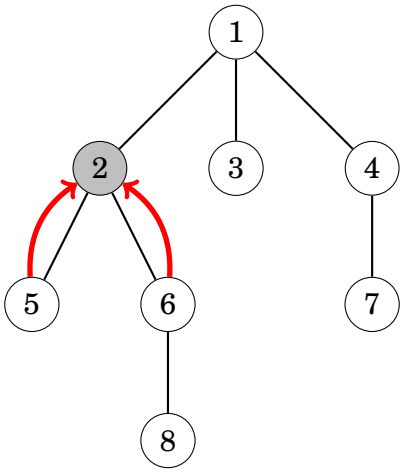
We use two pointers that initially point to the two nodes whose lowest common ancestor we should find. First, we move one of the pointers upwards so that both pointers point to nodes at the same level.

In the example scenario, we move the second pointer one level up so that it points to node 6 which is at the same level with node 5:



After this, we determine the minimum number of steps needed to move both pointers upwards so that they will point to the same node. The node to which the pointers point after this is the lowest common ancestor.

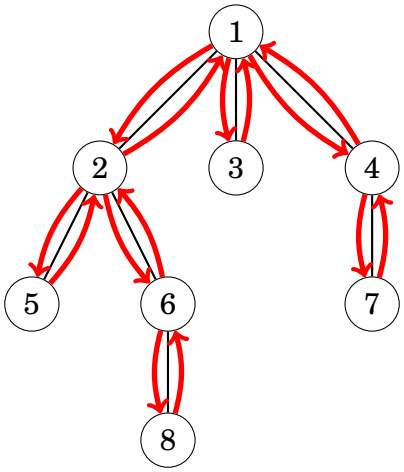
In the example scenario, it suffices to move both pointers one step upwards to node 2, which is the lowest common ancestor:



Since both parts of the algorithm can be performed in $O(\log n)$ time using precomputed information, we can find the lowest common ancestor of any two nodes in $O(\log n)$ time.

Method 2

Another way to solve the problem is based on a tree traversal array[1]. Once again, the idea is to traverse the nodes using a depth-first search:



However, we use a different tree traversal array than before: we add each node to the array *always* when the depth-first search walks through the node, and not only at the first visit. Hence, a node that has k children appears $k+1$ times in the array and there are a total of $2n-1$ nodes in the array.

[1] This lowest common ancestor algorithm was presented in [7]. This technique is sometimes called the **Euler tour technique** [66].

We store two values in the array: the identifier of the node and the depth of the node in the tree. The following array corresponds to the above tree:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
node id	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
depth	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1

Now we can find the lowest common ancestor of nodes a and b by finding the node with the *minimum* depth between nodes a and b in the array. For example, the lowest common ancestor of nodes 5 and 8 can be found as follows:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
node id	1	2	5	2	6	8	6	2	1	3	1	4	7	4	1
depth	1	2	3	2	3	4	3	2	1	2	1	2	3	2	1
				↑											

Node 5 is at position 2, node 8 is at position 5, and the node with minimum depth between positions 2...5 is node 2 at position 3 whose depth is 2. Thus, the lowest common ancestor of nodes 5 and 8 is node 2.

Thus, to find the lowest common ancestor of two nodes it suffices to process a range minimum query. Since the array is static, we can process such queries in $O(1)$ time after an $O(n \log n)$ time preprocessing.

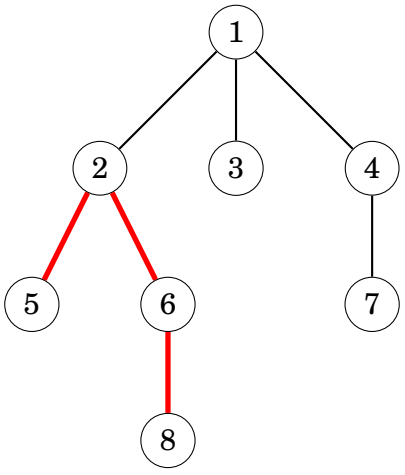
Distances of nodes

The distance between nodes a and b equals the length of the path from a to b. It turns out that the problem of calculating the distance between nodes reduces to finding their lowest common ancestor.

First, we root the tree arbitrarily. After this, the distance of nodes a and b can be calculated using the formula

$$\mathrm{depth}(a) + \mathrm{depth}(b) - 2 \cdot \mathrm{depth}(c),$$

where c is the lowest common ancestor of a and b and $\mathrm{depth}(s)$ denotes the depth of node s. For example, consider the distance of nodes 5 and 8:



The lowest common ancestor of nodes 5 and 8 is node 2. The depths of the nodes are $\texttt{depth}(5) = 3$, $\texttt{depth}(8) = 4$ and $\texttt{depth}(2) = 2$, so the distance between nodes 5 and 8 is $3 + 4 - 2 \cdot 2 = 3$.

Offline algorithms

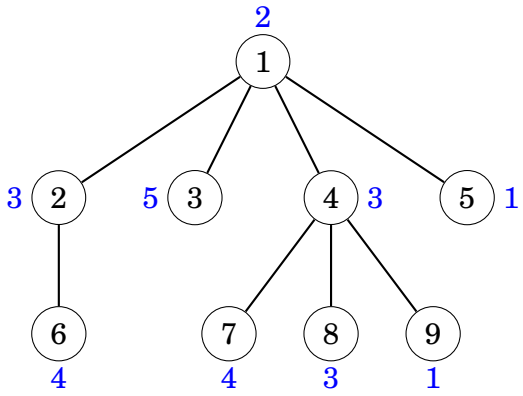
So far, we have discussed *online* algorithms for tree queries. Those algorithms are able to process queries one after another so that each query is answered before receiving the next query.

However, in many problems, the online property is not necessary. In this section, we focus on *offline* algorithms. Those algorithms are given a set of queries which can be answered in any order. It is often easier to design an offline algorithm compared to an online algorithm.

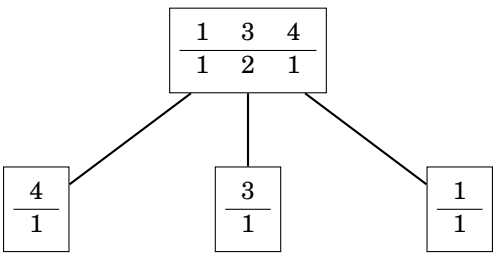
Merging data structures

One method to construct an offline algorithm is to perform a depth-first tree traversal and maintain data structures in nodes. At each node s, we create a data structure $\texttt{d}[s]$ that is based on the data structures of the children of s. Then, using this data structure, all queries related to s are processed.

As an example, consider the following problem: We are given a tree where each node has some value. Our task is to process queries of the form "calculate the number of nodes with value x in the subtree of node s". For example, in the following tree, the subtree of node 4 contains two nodes whose value is 3.



In this problem, we can use map structures to answer the queries. For example, the maps for node 4 and its children are as follows:



If we create such a data structure for each node, we can easily process all given queries, because we can handle all queries related to a node immediately after creating its data structure. For example, the above map structure for node 4 tells us that its subtree contains two nodes whose value is 3.

However, it would be too slow to create all data structures from scratch. Instead, at each node s, we create an initial data structure $\mathsf{d}[s]$ that only contains the value of s. After this, we go through the children of s and *merge* $\mathsf{d}[s]$ and all data structures $\mathsf{d}[u]$ where u is a child of s.

For example, in the above tree, the map for node 4 is created by merging the following maps:

Here the first map is the initial data structure for node 4, and the other three maps correspond to nodes 7, 8 and 9.

The merging at node s can be done as follows: We go through the children of s and at each child u merge $\mathsf{d}[s]$ and $\mathsf{d}[u]$. We always copy the contents from $\mathsf{d}[u]$ to $\mathsf{d}[s]$. However, before this, we *swap* the contents of $\mathsf{d}[s]$ and $\mathsf{d}[u]$ if $\mathsf{d}[s]$ is smaller than $\mathsf{d}[u]$. By doing this, each value is copied only $O(\log n)$ times during the tree traversal, which ensures that the algorithm is efficient.

To swap the contents of two data structures a and b efficiently, we can just use the following code:

```
swap(a,b);
```

It is guaranteed that the above code works in constant time when a and b are C++ standard library data structures.

Lowest common ancestors

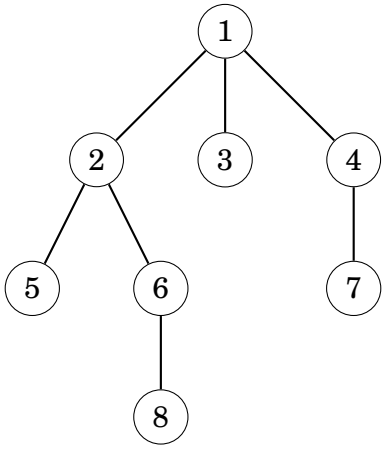
There is also an offline algorithm for processing a set of lowest common ancestor queries[2]. The algorithm is based on the union-find data structure (see Chapter 15.2), and the benefit of the algorithm is that it is easier to implement than the algorithms discussed earlier in this chapter.

The algorithm is given as input a set of pairs of nodes, and it determines for each such pair the lowest common ancestor of the nodes. The algorithm performs a depth-first tree traversal and maintains disjoint sets of nodes. Initially, each node belongs to a separate set. For each set, we also store the highest node in the tree that belongs to the set.

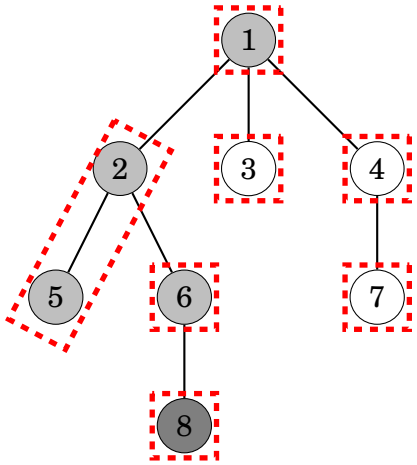
When the algorithm visits a node x, it goes through all nodes y such that the lowest common ancestor of x and y has to be found. If y has already been visited, the algorithm reports that the lowest common ancestor of x and y is the highest node in the set of y. Then, after processing node x, the algorithm joins the sets of x and its parent.

[2] This algorithm was published by R. E. Tarjan in 1979 [65].

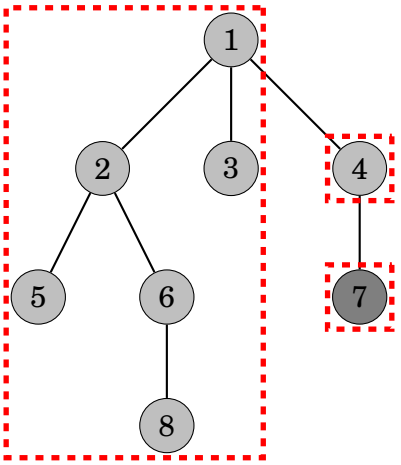
For example, suppose that we want to find the lowest common ancestors of node pairs $(5,8)$ and $(2,7)$ in the following tree:



In the following trees, gray nodes denote visited nodes and dashed groups of nodes belong to the same set. When the algorithm visits node 8, it notices that node 5 has been visited and the highest node in its set is 2. Thus, the lowest common ancestor of nodes 5 and 8 is 2:



Later, when visiting node 7, the algorithm determines that the lowest common ancestor of nodes 2 and 7 is 1:



Chapter 19

Paths and circuits

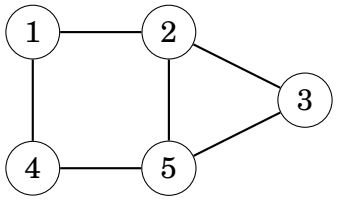
This chapter focuses on two types of paths in graphs:

- An **Eulerian path** is a path that goes through each edge exactly once.
- A **Hamiltonian path** is a path that visits each node exactly once.

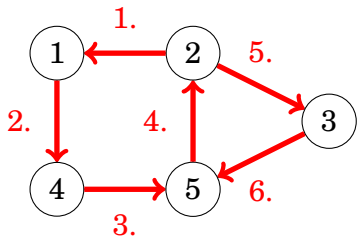
While Eulerian and Hamiltonian paths look like similar concepts at first glance, the computational problems related to them are very different. It turns out that there is a simple rule that determines whether a graph contains an Eulerian path, and there is also an efficient algorithm to find such a path if it exists. On the contrary, checking the existence of a Hamiltonian path is a NP-hard problem, and no efficient algorithm is known for solving the problem.

Eulerian paths

An **Eulerian path**[1] is a path that goes exactly once through each edge of the graph. For example, the graph

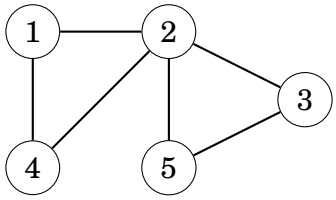


has an Eulerian path from node 2 to node 5:

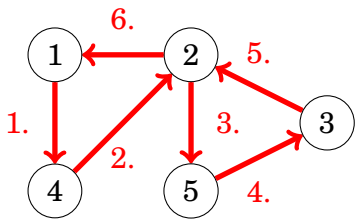


[1] L. Euler studied such paths in 1736 when he solved the famous Königsberg bridge problem. This was the birth of graph theory.

An **Eulerian circuit** is an Eulerian path that starts and ends at the same node. For example, the graph



has an Eulerian circuit that starts and ends at node 1:



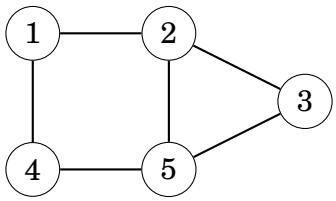
Existence

The existence of Eulerian paths and circuits depends on the degrees of the nodes. First, an undirected graph has an Eulerian path exactly when all the edges belong to the same connected component and

- the degree of each node is even *or*
- the degree of exactly two nodes is odd, and the degree of all other nodes is even.

In the first case, each Eulerian path is also an Eulerian circuit. In the second case, the odd-degree nodes are the starting and ending nodes of an Eulerian path which is not an Eulerian circuit.

For example, in the graph



nodes 1, 3 and 4 have a degree of 2, and nodes 2 and 5 have a degree of 3. Exactly two nodes have an odd degree, so there is an Eulerian path between nodes 2 and 5, but the graph does not contain an Eulerian circuit.

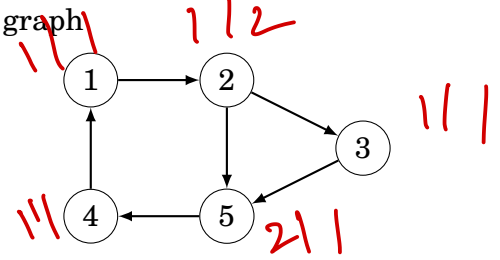
In a directed graph, we focus on indegrees and outdegrees of the nodes. A directed graph contains an Eulerian path exactly when all the edges belong to the same connected component and

- in each node, the indegree equals the outdegree, *or*

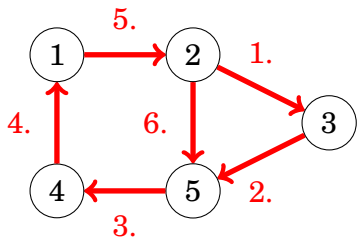
- in one node, the indegree is one larger than the outdegree, in another node, the outdegree is one larger than the indegree, and in all other nodes, the indegree equals the outdegree.

In the first case, each Eulerian path is also an Eulerian circuit, and in the second case, the graph contains an Eulerian path that begins at the node whose outdegree is larger and ends at the node whose indegree is larger.

For example, in the graph



nodes 1, 3 and 4 have both indegree 1 and outdegree 1, node 2 has indegree 1 and outdegree 2, and node 5 has indegree 2 and outdegree 1. Hence, the graph contains an Eulerian path from node 2 to node 5:



Hierholzer's algorithm

Hierholzer's algorithm[2] is an efficient method for constructing an Eulerian circuit. The algorithm consists of several rounds, each of which adds new edges to the circuit. Of course, we assume that the graph contains an Eulerian circuit; otherwise Hierholzer's algorithm cannot find it.

First, the algorithm constructs a circuit that contains some (not necessarily all) of the edges of the graph. After this, the algorithm extends the circuit step by step by adding subcircuits to it. The process continues until all edges have been added to the circuit.

The algorithm extends the circuit by always finding a node x that belongs to the circuit but has an outgoing edge that is not included in the circuit. The algorithm constructs a new path from node x that only contains edges that are not yet in the circuit. Sooner or later, the path will return to node x, which creates a subcircuit.

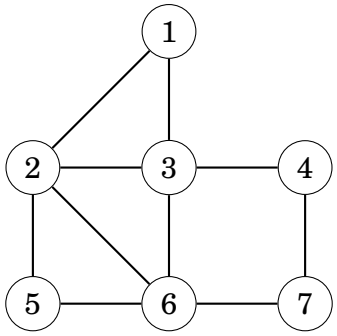
If the graph only contains an Eulerian path, we can still use Hierholzer's algorithm to find it by adding an extra edge to the graph and removing the edge after the circuit has been constructed. For example, in an undirected graph, we add the extra edge between the two odd-degree nodes.

Next we will see how Hierholzer's algorithm constructs an Eulerian circuit for an undirected graph.

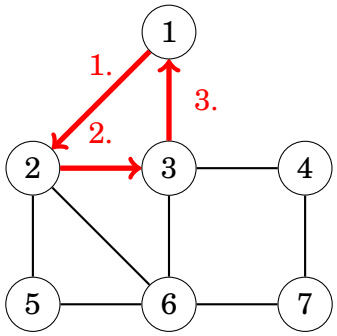
[2] The algorithm was published in 1873 after Hierholzer's death [35].

Example

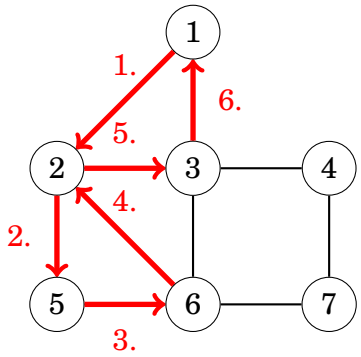
Let us consider the following graph:



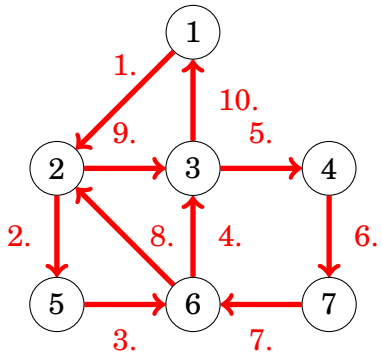
Suppose that the algorithm first creates a circuit that begins at node 1. A possible circuit is $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$:



After this, the algorithm adds the subcircuit $2 \rightarrow 5 \rightarrow 6 \rightarrow 2$ to the circuit:



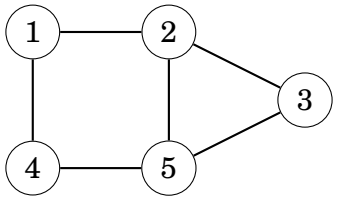
Finally, the algorithm adds the subcircuit $6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 6$ to the circuit:



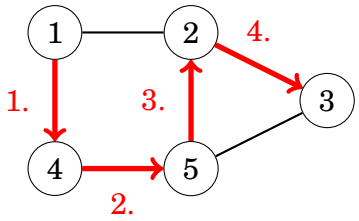
Now all edges are included in the circuit, so we have successfully constructed an Eulerian circuit.

Hamiltonian paths

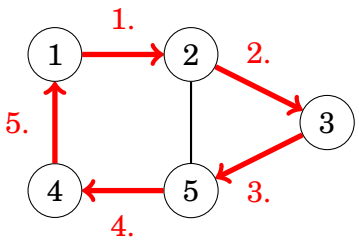
A **Hamiltonian path** is a path that visits each node of the graph exactly once. For example, the graph



contains a Hamiltonian path from node 1 to node 3:



If a Hamiltonian path begins and ends at the same node, it is called a **Hamiltonian circuit**. The graph above also has an Hamiltonian circuit that begins and ends at node 1:



Existence

No efficient method is known for testing if a graph contains a Hamiltonian path, and the problem is NP-hard. Still, in some special cases, we can be certain that a graph contains a Hamiltonian path.

A simple observation is that if the graph is complete, i.e., there is an edge between all pairs of nodes, it also contains a Hamiltonian path. Also stronger results have been achieved:

- **Dirac's theorem**: If the degree of each node is at least $n/2$, the graph contains a Hamiltonian path.
- **Ore's theorem**: If the sum of degrees of each non-adjacent pair of nodes is at least n, the graph contains a Hamiltonian path.

A common property in these theorems and other results is that they guarantee the existence of a Hamiltonian path if the graph has *a large number* of edges. This makes sense, because the more edges the graph contains, the more possibilities there is to construct a Hamiltonian path.

Construction

Since there is no efficient way to check if a Hamiltonian path exists, it is clear that there is also no method to efficiently construct the path, because otherwise we could just try to construct the path and see whether it exists.

A simple way to search for a Hamiltonian path is to use a backtracking algorithm that goes through all possible ways to construct the path. The time complexity of such an algorithm is at least $O(n!)$, because there are $n!$ different ways to choose the order of n nodes.

A more efficient solution is based on dynamic programming (see Chapter 10.5). The idea is to calculate values of a function `possible`(S,x), where S is a subset of nodes and x is one of the nodes. The function indicates whether there is a Hamiltonian path that visits the nodes of S and ends at node x. It is possible to implement this solution in $O(2^n n^2)$ time.

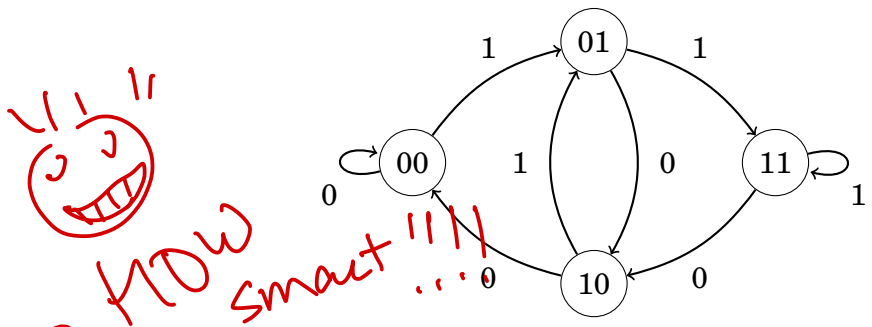
De Bruijn sequences

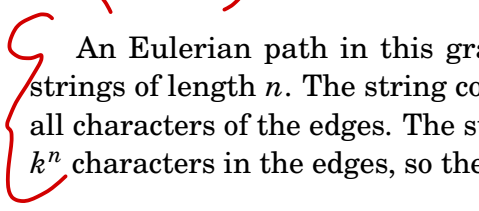
A **De Bruijn sequence** is a string that contains every string of length n exactly once as a substring, for a fixed alphabet of k characters. The length of such a string is $k^n + n - 1$ characters. For example, when $n = 3$ and $k = 2$, an example of a De Bruijn sequence is

$$0001011100.$$

The substrings of this string are all combinations of three bits: 000, 001, 010, 011, 100, 101, 110 and 111.

It turns out that each De Bruijn sequence corresponds to an Eulerian path in a graph. The idea is to construct a graph where each node contains a string of $n-1$ characters and each edge adds one character to the string. The following graph corresponds to the above scenario:





An Eulerian path in this graph corresponds to a string that contains all strings of length n. The string contains the characters of the starting node and all characters of the edges. The starting node has $n-1$ characters and there are k^n characters in the edges, so the length of the string is $k^n + n - 1$.

Knight's tours

A **knight's tour** is a sequence of moves of a knight on an $n \times n$ chessboard following the rules of chess such that the knight visits each square exactly once. A knight's tour is called a *closed* tour if the knight finally returns to the starting square and otherwise it is called an *open* tour.

For example, here is an open knight's tour on a 5×5 board:

1	4	11	16	25
12	17	2	5	10
3	20	7	24	15
18	13	22	9	6
21	8	19	14	23

A knight's tour corresponds to a Hamiltonian path in a graph whose nodes represent the squares of the board, and two nodes are connected with an edge if a knight can move between the squares according to the rules of chess.

A natural way to construct a knight's tour is to use backtracking. The search can be made more efficient by using *heuristics* that attempt to guide the knight so that a complete tour will be found quickly.

Warnsdorf's rule

Warnsdorf's rule is a simple and effective heuristic for finding a knight's tour[3]. Using the rule, it is possible to efficiently construct a tour even on a large board. The idea is to always move the knight so that it ends up in a square where the number of possible moves is as *small* as possible.

For example, in the following situation, there are five possible squares to which the knight can move (squares $a \ldots e$):

1				a
		2		
b				e
	c		d	

In this situation, Warnsdorf's rule moves the knight to square a, because after this choice, there is only a single possible move. The other choices would move the knight to squares where there would be three moves available.

[3] This heuristic was proposed in Warnsdorf's book [69] in 1823. There are also polynomial algorithms for finding knight's tours [52], but they are more complicated.

Chapter 20

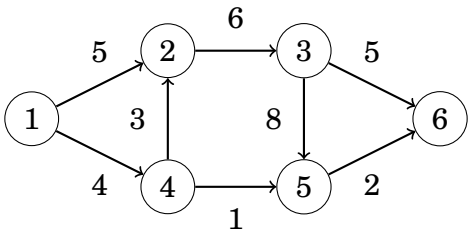
Flows and cuts

In this chapter, we focus on the following two problems:

- **Finding a maximum flow**: What is the maximum amount of flow we can send from a node to another node?
- **Finding a minimum cut**: What is a minimum-weight set of edges that separates two nodes of the graph?

The input for both these problems is a directed, weighted graph that contains two special nodes: the *source* is a node with no incoming edges, and the *sink* is a node with no outgoing edges.

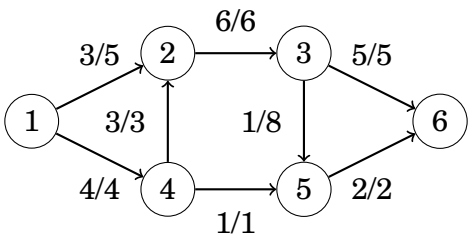
As an example, we will use the following graph where node 1 is the source and node 6 is the sink:



Maximum flow

In the **maximum flow** problem, our task is to send as much flow as possible from the source to the sink. The weight of each edge is a capacity that restricts the flow that can go through the edge. In each intermediate node, the incoming and outgoing flow has to be equal.

For example, the maximum size of a flow in the example graph is 7. The following picture shows how we can route the flow:

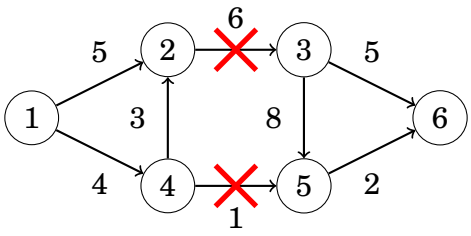


The notation v/k means that a flow of v units is routed through an edge whose capacity is k units. The size of the flow is 7, because the source sends $3+4$ units of flow and the sink receives $5+2$ units of flow. It is easy see that this flow is maximum, because the total capacity of the edges leading to the sink is 7.

Minimum cut

In the **minimum cut** problem, our task is to remove a set of edges from the graph such that there will be no path from the source to the sink after the removal and the total weight of the removed edges is minimum.

The minimum size of a cut in the example graph is 7. It suffices to remove the edges $2 \to 3$ and $4 \to 5$:



After removing the edges, there will be no path from the source to the sink. The size of the cut is 7, because the weights of the removed edges are 6 and 1. The cut is minimum, because there is no valid way to remove edges from the graph such that their total weight would be less than 7.

It is not a coincidence that the maximum size of a flow and the minimum size of a cut are the same in the above example. It turns out that a maximum flow and a minimum cut are *always* equally large, so the concepts are two sides of the same coin.

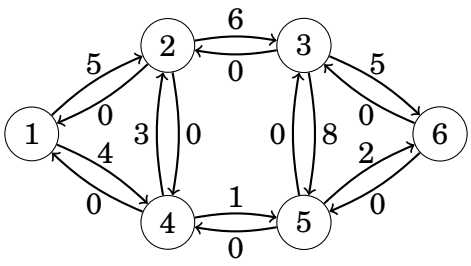
Next we will discuss the Ford–Fulkerson algorithm that can be used to find the maximum flow and minimum cut of a graph. The algorithm also helps us to understand *why* they are equally large.

Ford–Fulkerson algorithm

The **Ford–Fulkerson algorithm** [25] finds the maximum flow in a graph. The algorithm begins with an empty flow, and at each step finds a path from the source to the sink that generates more flow. Finally, when the algorithm cannot increase the flow anymore, the maximum flow has been found.

The algorithm uses a special representation of the graph where each original edge has a reverse edge in another direction. The weight of each edge indicates how much more flow we could route through it. At the beginning of the algorithm, the weight of each original edge equals the capacity of the edge and the weight of each reverse edge is zero.

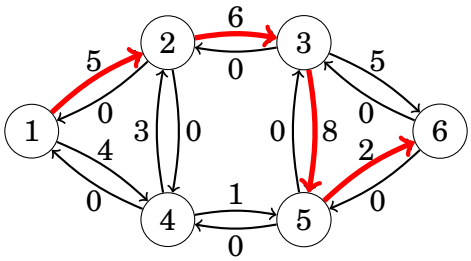
The new representation for the example graph is as follows:



Algorithm description

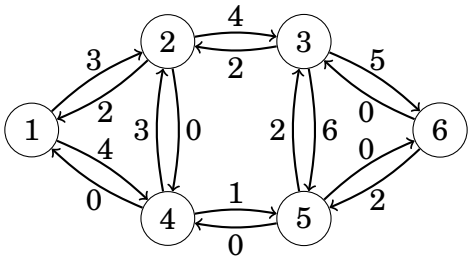
The Ford–Fulkerson algorithm consists of several rounds. On each round, the algorithm finds a path from the source to the sink such that each edge on the path has a positive weight. If there is more than one possible path available, we can choose any of them.

For example, suppose we choose the following path:



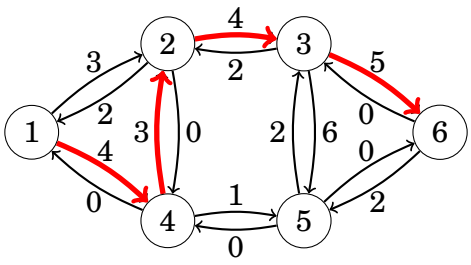
After choosing the path, the flow increases by x units, where x is the smallest edge weight on the path. In addition, the weight of each edge on the path decreases by x and the weight of each reverse edge increases by x.

In the above path, the weights of the edges are 5, 6, 8 and 2. The smallest weight is 2, so the flow increases by 2 and the new graph is as follows:



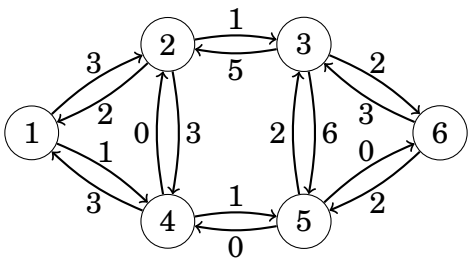
The idea is that increasing the flow decreases the amount of flow that can go through the edges in the future. On the other hand, it is possible to cancel flow later using the reverse edges of the graph if it turns out that it would be beneficial to route the flow in another way.

The algorithm increases the flow as long as there is a path from the source to the sink through positive-weight edges. In the present example, our next path can be as follows:

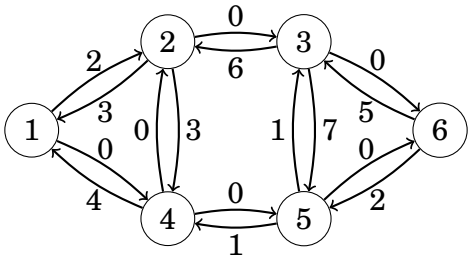


The minimum edge weight on this path is 3, so the path increases the flow by 3, and the total flow after processing the path is 5.

The new graph will be as follows:



We still need two more rounds before reaching the maximum flow. For example, we can choose the paths $1 \to 2 \to 3 \to 6$ and $1 \to 4 \to 5 \to 3 \to 6$. Both paths increase the flow by 1, and the final graph is as follows:



It is not possible to increase the flow anymore, because there is no path from the source to the sink with positive edge weights. Hence, the algorithm terminates and the maximum flow is 7.

Finding paths

The Ford–Fulkerson algorithm does not specify how we should choose the paths that increase the flow. In any case, the algorithm will terminate sooner or later and correctly find the maximum flow. However, the efficiency of the algorithm depends on the way the paths are chosen.

A simple way to find paths is to use depth-first search. Usually, this works well, but in the worst case, each path only increases the flow by 1 and the algorithm is slow. Fortunately, we can avoid this situation by using one of the following techniques:

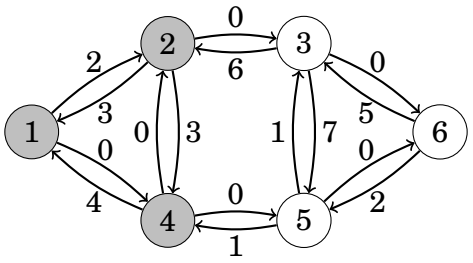
The **Edmonds–Karp algorithm** [18] chooses each path so that the number of edges on the path is as small as possible. This can be done by using breadth-first search instead of depth-first search for finding paths. It can be proven that this guarantees that the flow increases quickly, and the time complexity of the algorithm is $O(m^2 n)$.

The **scaling algorithm** [2] uses depth-first search to find paths where each edge weight is at least a threshold value. Initially, the threshold value is some large number, for example the sum of all edge weights of the graph. Always when a path cannot be found, the threshold value is divided by 2. The time complexity of the algorithm is $O(m^2 \log c)$, where c is the initial threshold value.

In practice, the scaling algorithm is easier to implement, because depth-first search can be used for finding paths. Both algorithms are efficient enough for problems that typically appear in programming contests.

Minimum cuts

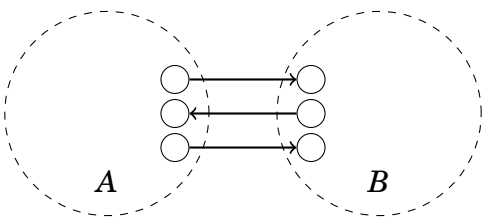
It turns out that once the Ford–Fulkerson algorithm has found a maximum flow, it has also determined a minimum cut. Let A be the set of nodes that can be reached from the source using positive-weight edges. In the example graph, A contains nodes 1, 2 and 4:



Now the minimum cut consists of the edges of the original graph that start at some node in A, end at some node outside A, and whose capacity is fully used in the maximum flow. In the above graph, such edges are $2 \to 3$ and $4 \to 5$, that correspond to the minimum cut $6 + 1 = 7$.

Why is the flow produced by the algorithm maximum and why is the cut minimum? The reason is that a graph cannot contain a flow whose size is larger than the weight of any cut of the graph. Hence, always when a flow and a cut are equally large, they are a maximum flow and a minimum cut.

Let us consider any cut of the graph such that the source belongs to A, the sink belongs to B and there are some edges between the sets:



The size of the cut is the sum of the edges that go from A to B. This is an upper bound for the flow in the graph, because the flow has to proceed from A to B. Thus, the size of a maximum flow is smaller than or equal to the size of any cut in the graph.

On the other hand, the Ford–Fulkerson algorithm produces a flow whose size is *exactly* as large as the size of a cut in the graph. Thus, the flow has to be a maximum flow and the cut has to be a minimum cut.

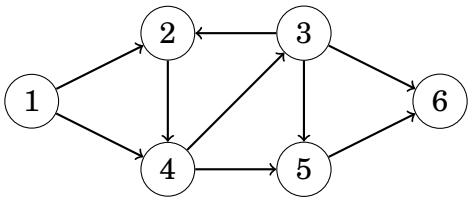
Disjoint paths

Many graph problems can be solved by reducing them to the maximum flow problem. Our first example of such a problem is as follows: we are given a directed graph with a source and a sink, and our task is to find the maximum number of disjoint paths from the source to the sink.

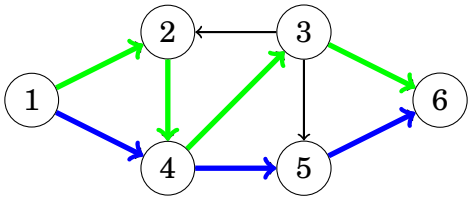
Edge-disjoint paths

We will first focus on the problem of finding the maximum number of **edge-disjoint paths** from the source to the sink. This means that we should construct a set of paths such that each edge appears in at most one path.

For example, consider the following graph:



In this graph, the maximum number of edge-disjoint paths is 2. We can choose the paths $1 \to 2 \to 4 \to 3 \to 6$ and $1 \to 4 \to 5 \to 6$ as follows:



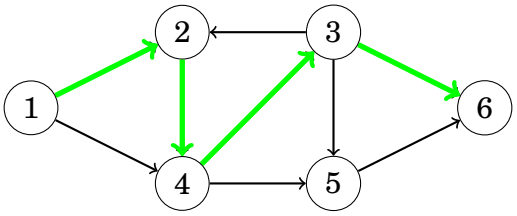
It turns out that the maximum number of edge-disjoint paths equals the maximum flow of the graph, assuming that the capacity of each edge is one. After the maximum flow has been constructed, the edge-disjoint paths can be found greedily by following paths from the source to the sink.

Node-disjoint paths

Let us now consider another problem: finding the maximum number of **node-disjoint paths** from the source to the sink. In this problem, every node, except

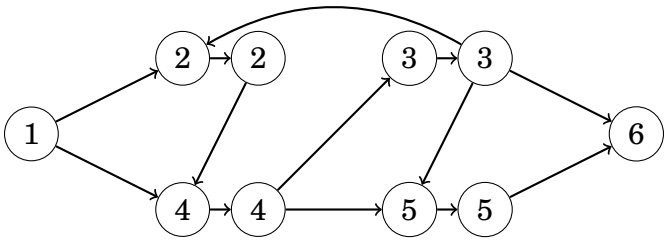
for the source and sink, may appear in at most one path. The number of node-disjoint paths may be smaller than the number of edge-disjoint paths.

For example, in the previous graph, the maximum number of node-disjoint paths is 1:

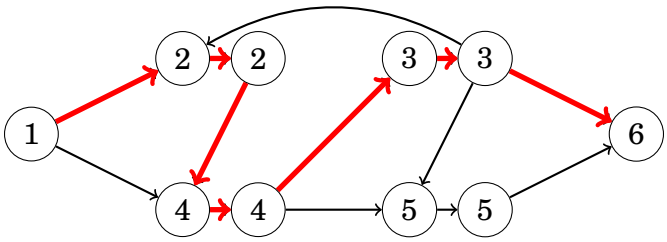


We can reduce also this problem to the maximum flow problem. Since each node can appear in at most one path, we have to limit the flow that goes through the nodes. A standard method for this is to divide each node into two nodes such that the first node has the incoming edges of the original node, the second node has the outgoing edges of the original node, and there is a new edge from the first node to the second node.

In our example, the graph becomes as follows:



The maximum flow for the graph is as follows:



Thus, the maximum number of node-disjoint paths from the source to the sink is 1.

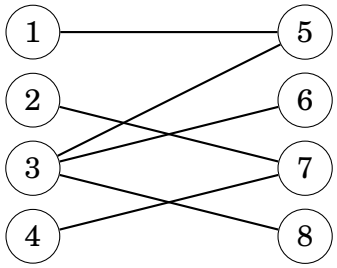
Maximum matchings

The **maximum matching** problem asks to find a maximum-size set of node pairs in an undirected graph such that each pair is connected with an edge and each node belongs to at most one pair.

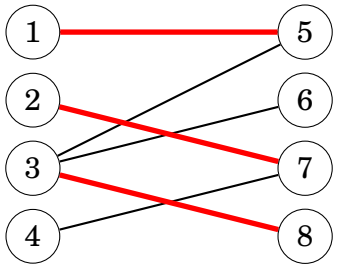
There are polynomial algorithms for finding maximum matchings in general graphs [17], but such algorithms are complex and rarely seen in programming contests. However, in bipartite graphs, the maximum matching problem is much easier to solve, because we can reduce it to the maximum flow problem.

Finding maximum matchings

The nodes of a bipartite graph can be always divided into two groups such that all edges of the graph go from the left group to the right group. For example, in the following bipartite graph, the groups are $\{1,2,3,4\}$ and $\{5,6,7,8\}$.

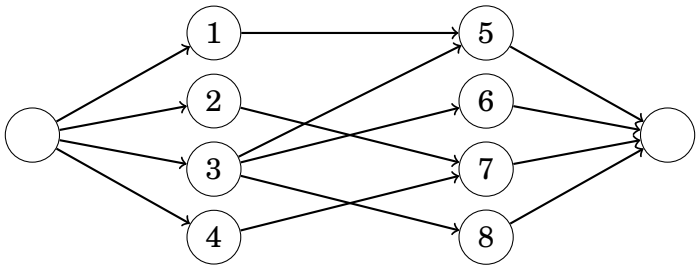


The size of a maximum matching of this graph is 3:

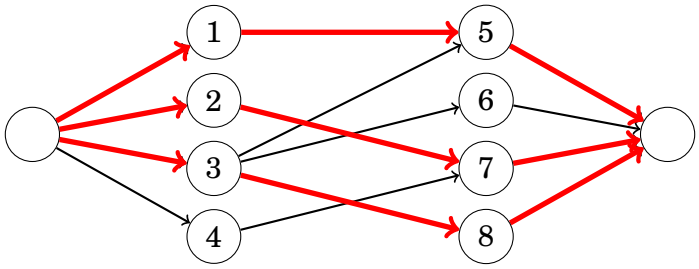


We can reduce the bipartite maximum matching problem to the maximum flow problem by adding two new nodes to the graph: a source and a sink. We also add edges from the source to each left node and from each right node to the sink. After this, the size of a maximum flow in the graph equals the size of a maximum matching in the original graph.

For example, the reduction for the above graph is as follows:



The maximum flow of this graph is as follows:

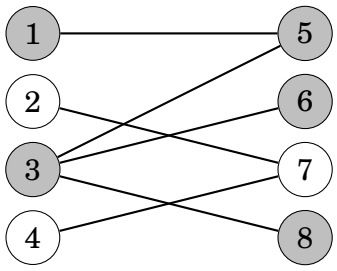


Hall's theorem

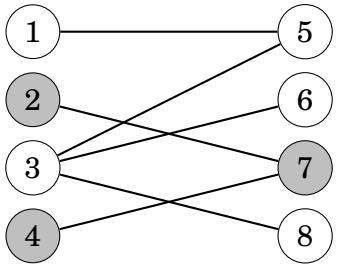
Hall's theorem can be used to find out whether a bipartite graph has a matching that contains all left or right nodes. If the number of left and right nodes is the same, Hall's theorem tells us if it is possible to construct a **perfect matching** that contains all nodes of the graph.

Assume that we want to find a matching that contains all left nodes. Let X be any set of left nodes and let $f(X)$ be the set of their neighbors. According to Hall's theorem, a matching that contains all left nodes exists exactly when for each X, the condition $|X| \leq |f(X)|$ holds.

Let us study Hall's theorem in the example graph. First, let $X = \{1,3\}$ which yields $f(X) = \{5,6,8\}$:



The condition of Hall's theorem holds, because $|X| = 2$ and $|f(X)| = 3$. Next, let $X = \{2,4\}$ which yields $f(X) = \{7\}$:



In this case, $|X| = 2$ and $|f(X)| = 1$, so the condition of Hall's theorem does not hold. This means that it is not possible to form a perfect matching for the graph. This result is not surprising, because we already know that the maximum matching of the graph is 3 and not 4.

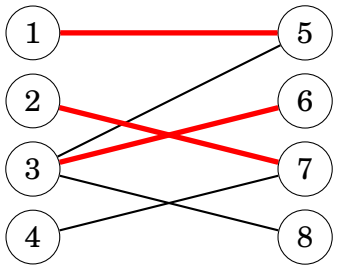
If the condition of Hall's theorem does not hold, the set X provides an explanation *why* we cannot form such a matching. Since X contains more nodes than $f(X)$, there are no pairs for all nodes in X. For example, in the above graph, both nodes 2 and 4 should be connected with node 7 which is not possible.

Kőnig's theorem

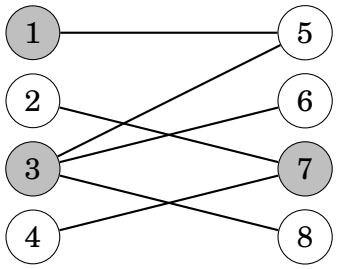
A **minimum node cover** of a graph is a minimum set of nodes such that each edge of the graph has at least one endpoint in the set. In a general graph, finding a minimum node cover is a NP-hard problem. However, if the graph is bipartite, **Kőnig's theorem** tells us that the size of a minimum node cover and the size

of a maximum matching are always equal. Thus, we can calculate the size of a minimum node cover using a maximum flow algorithm.

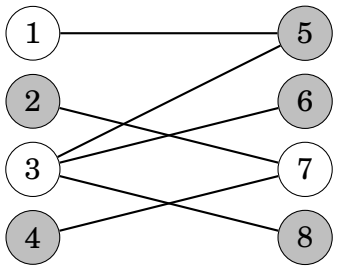
Let us consider the following graph with a maximum matching of size 3:



Now Kőnig's theorem tells us that the size of a minimum node cover is also 3. Such a cover can be constructed as follows:



The nodes that do *not* belong to a minimum node cover form a **maximum independent set**. This is the largest possible set of nodes such that no two nodes in the set are connected with an edge. Once again, finding a maximum independent set in a general graph is a NP-hard problem, but in a bipartite graph we can use Kőnig's theorem to solve the problem efficiently. In the example graph, the maximum independent set is as follows:

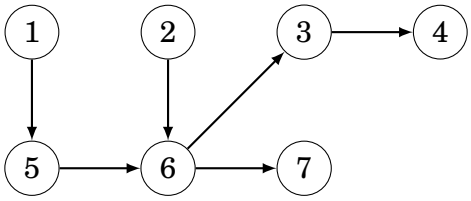


Path covers

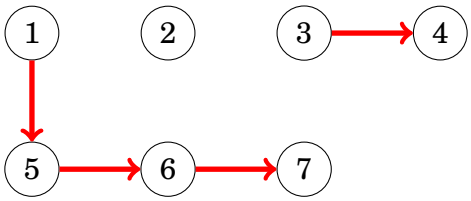
A **path cover** is a set of paths in a graph such that each node of the graph belongs to at least one path. It turns out that in directed, acyclic graphs, we can reduce the problem of finding a minimum path cover to the problem of finding a maximum flow in another graph.

Node-disjoint path cover

In a **node-disjoint path cover**, each node belongs to exactly one path. As an example, consider the following graph:



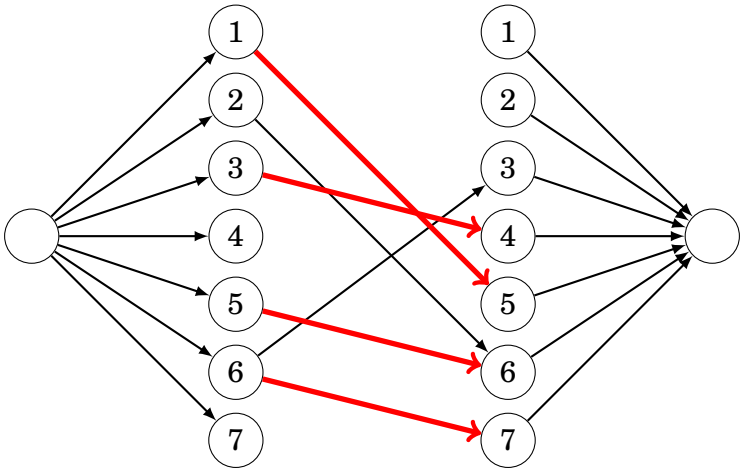
A minimum node-disjoint path cover of this graph consists of three paths. For example, we can choose the following paths:



Note that one of the paths only contains node 2, so it is possible that a path does not contain any edges.

We can find a minimum node-disjoint path cover by constructing a *matching graph* where each node of the original graph is represented by two nodes: a left node and a right node. There is an edge from a left node to a right node if there is such an edge in the original graph. In addition, the matching graph contains a source and a sink, and there are edges from the source to all left nodes and from all right nodes to the sink.

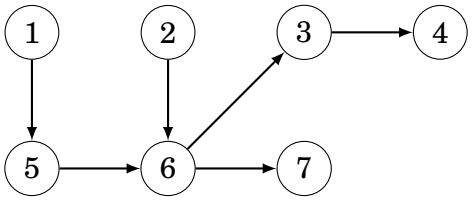
A maximum matching in the resulting graph corresponds to a minimum node-disjoint path cover in the original graph. For example, the following matching graph for the above graph contains a maximum matching of size 4:



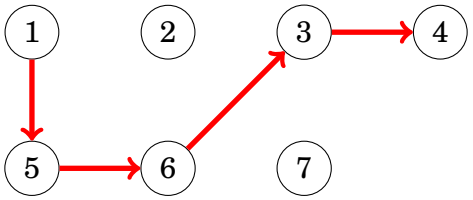
Each edge in the maximum matching of the matching graph corresponds to an edge in the minimum node-disjoint path cover of the original graph. Thus, the size of the minimum node-disjoint path cover is $n - c$, where n is the number of nodes in the original graph and c is the size of the maximum matching.

General path cover

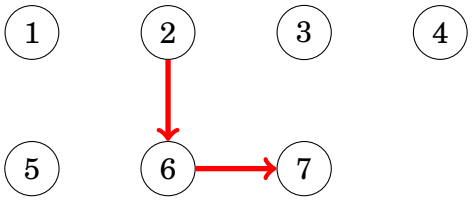
A **general path cover** is a path cover where a node can belong to more than one path. A minimum general path cover may be smaller than a minimum node-disjoint path cover, because a node can be used multiple times in paths. Consider again the following graph:



The minimum general path cover of this graph consists of two paths. For example, the first path may be as follows:

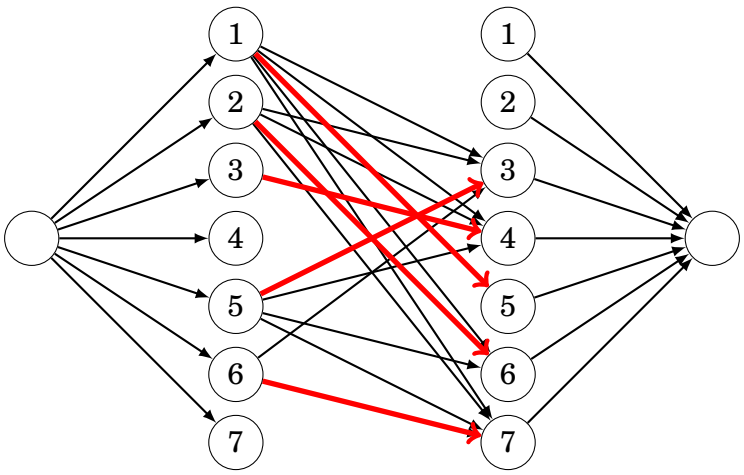


And the second path may be as follows:



A minimum general path cover can be found almost like a minimum node-disjoint path cover. It suffices to add some new edges to the matching graph so that there is an edge $a \rightarrow b$ always when there is a path from a to b in the original graph (possibly through several edges).

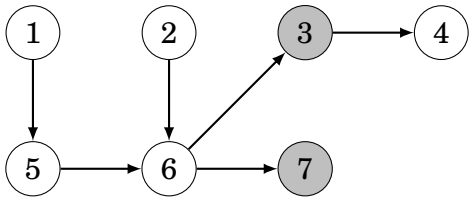
The matching graph for the above graph is as follows:



Dilworth's theorem

An **antichain** is a set of nodes of a graph such that there is no path from any node to another node using the edges of the graph. **Dilworth's theorem** states that in a directed acyclic graph, the size of a minimum general path cover equals the size of a maximum antichain.

For example, nodes 3 and 7 form an antichain in the following graph:



This is a maximum antichain, because it is not possible to construct any antichain that would contain three nodes. We have seen before that the size of a minimum general path cover of this graph consists of two paths.