

# SOI Camp 2021 – Graph Day

Competitive Programmer's Handbook by Antti Laaksonen

modified by Johannes Kapfhammer, February 2021



# Contents

<b>Preface</b>	<b>v</b>
<b>I Main Topics</b>	<b>1</b>
<b>1 Basics of graphs</b>	<b>3</b>
1.1 Graph terminology . . . . .	3
1.2 Graph representation . . . . .	7
<b>2 Graph traversal</b>	<b>11</b>
2.1 Depth-first search . . . . .	11
2.2 Breadth-first search . . . . .	13
2.3 Applications . . . . .	15
<b>3 Shortest paths</b>	<b>17</b>
3.1 Dijkstra's algorithm . . . . .	17
<b>4 Tree algorithms</b>	<b>21</b>
4.1 Tree traversal . . . . .	22
4.2 Diameter . . . . .	23
4.3 All longest paths . . . . .	25
4.4 Binary trees . . . . .	27
<b>5 Topological sorting</b>	<b>29</b>
<b>II Advanced topics</b>	<b>33</b>
<b>6 Spanning trees</b>	<b>35</b>
6.1 Kruskal's algorithm . . . . .	36
6.2 Union-find structure . . . . .	39
6.3 Prim's algorithm . . . . .	41
<b>7 Strong connectivity</b>	<b>43</b>
7.1 Kosaraju's algorithm . . . . .	44
<b>8 Tree queries</b>	<b>47</b>
8.1 Finding ancestors . . . . .	47
8.2 Subtrees and paths . . . . .	48

8.3	Lowest common ancestor . . . . .	51
8.4	Offline algorithms . . . . .	54
<b>9</b>	<b>Paths and circuits</b>	<b>57</b>
9.1	Eulerian paths . . . . .	57
9.2	Hamiltonian paths . . . . .	61
9.3	De Bruijn sequences . . . . .	62
9.4	Knight's tours . . . . .	63
	<b>Bibliography</b>	<b>65</b>

# Preface

This script is based on the Competitive Programmer's Handbook by Antti Laaksonen.

It contains the topics relevant for the graph day of the SOI Camp 2021. Most of the code was modified slightly, and also some minor adjustments were made on the text.



# **Part I**

## **Main Topics**





# Chapter 1

## Basics of graphs

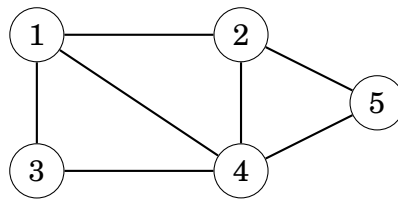
Many programming problems can be solved by modeling the problem as a graph problem and using an appropriate graph algorithm. A typical example of a graph is a network of roads and cities in a country. Sometimes, though, the graph is hidden in the problem and it may be difficult to detect it.

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.

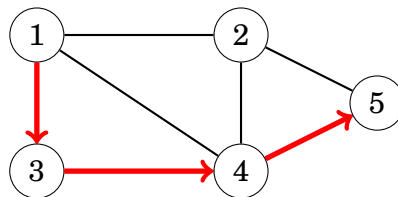
### 1.1 Graph terminology

A **graph** consists of **nodes** and **edges**. In this book, the variable  $n$  denotes the number of nodes in a graph, and the variable  $m$  denotes the number of edges. The nodes are numbered using integers  $1, 2, \dots, n$ .

For example, the following graph consists of 5 nodes and 7 edges:



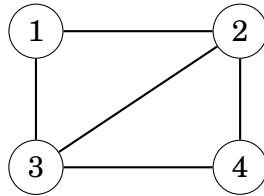
A **path** leads from node  $a$  to node  $b$  through edges of the graph. The **length** of a path is the number of edges in it. For example, the above graph contains a path  $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$  of length 3 from node 1 to node 5:



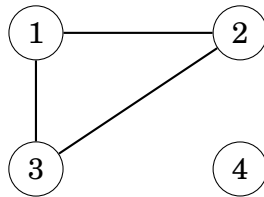
A path is a **cycle** if the first and last node is the same. For example, the above graph contains a cycle  $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$ . A path is **simple** if each node appears at most once in the path.

## Connectivity

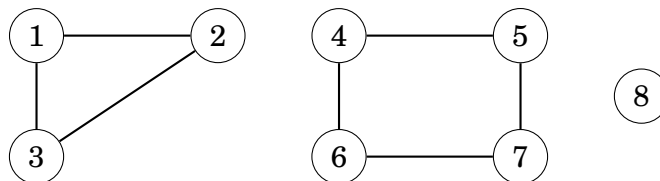
A graph is **connected** if there is a path between any two nodes. For example, the following graph is connected:



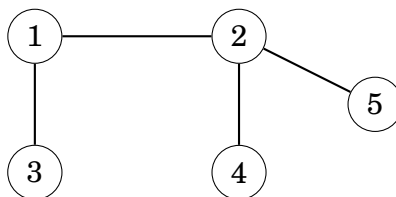
The following graph is not connected, because it is not possible to get from node 4 to any other node:



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}.

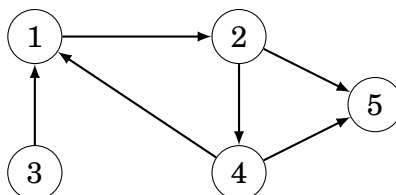


A **tree** is a connected graph that consists of  $n$  nodes and  $n - 1$  edges. There is a unique path between any two nodes of a tree. For example, the following graph is a tree:



## Edge directions

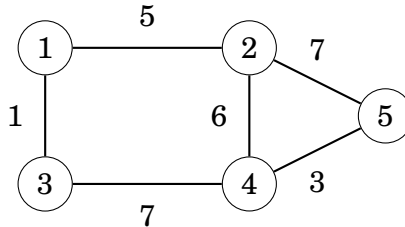
A graph is **directed** if the edges can be traversed in one direction only. For example, the following graph is directed:



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

## Edge weights

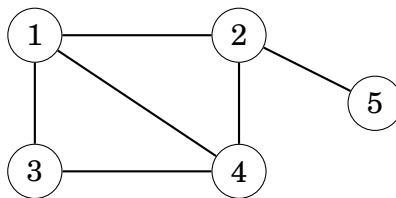
In a **weighted** graph, each edge is assigned a **weight**. The weights are often interpreted as edge lengths. For example, the following graph is weighted:



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  $1 \rightarrow 2 \rightarrow 5$  is 12, and the length of the path  $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$  is 11. The latter path is the **shortest** path from node 1 to node 5.

## Neighbors and degrees

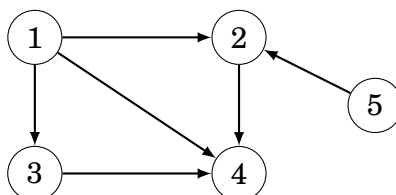
Two nodes are **neighbors** or **adjacent** if there is an edge between them. The **degree** of a node is the number of its neighbors. For example, in the following graph, the neighbors of node 2 are 1, 4 and 5, so its degree is 3.



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.

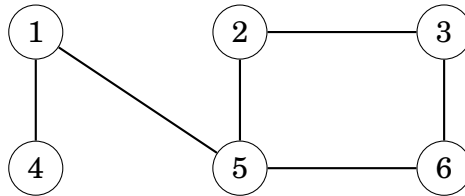
In a directed graph, the **indegree** of a node is the number of edges that end at the node, and the **outdegree** of a node is the number of edges that start at the node. For example, in the following graph, the indegree of node 2 is 2, and the outdegree of node 2 is 1.



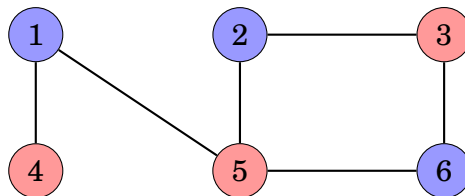
## Colorings

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

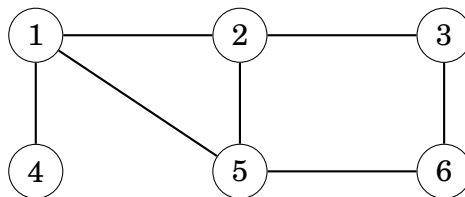
A graph is **bipartite** if it is possible to color it using two colors. It turns out that a graph is bipartite exactly when it does not contain a cycle with an odd number of edges. For example, the graph



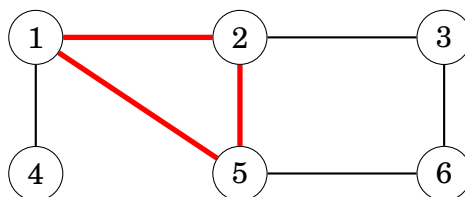
is bipartite, because it can be colored as follows:



However, the graph

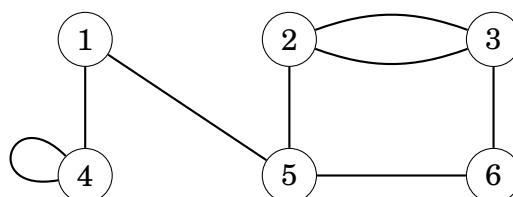


is not bipartite, because it is not possible to color the following cycle of three nodes using two colors:



## 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:



## 1.2 Graph representation

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

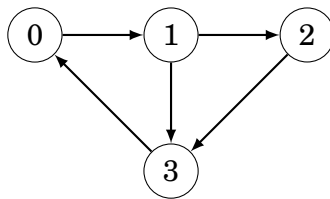
### Adjacency list representation

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 a vector of vectors as follows:

```
vector<vector<int>>> g;
```

For example, the graph



can be stored as follows:

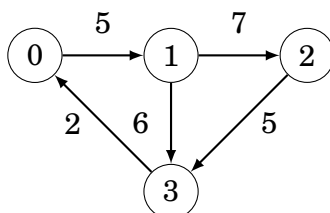
```
g.assign(4, {}); // g now consists of 4 empty arrays
g[0].push_back(1);
g[1].push_back(2);
g[1].push_back(3);
g[2].push_back(3);
g[3].push_back(0);
```

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

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:

```
g.assign(4, {});
g[0].push_back({1,5});
g[1].push_back({2,7});
g[1].push_back({3,6});
g[2].push_back({3,5});
g[3].push_back({0,2});
```

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  $s$ :

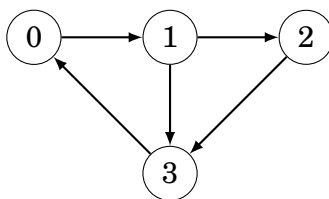
```
for (auto u : g[s]) {
    // process node u
}
```

## 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

```
vector<vector<int>> adj;
adj.assign(n, vector<int>(n, 0));
```

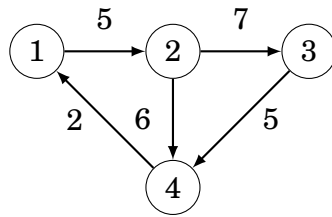
where each value  $\text{adj}[a][b]$  indicates whether the graph contains an edge from node  $a$  to node  $b$ . If the edge is included in the graph, then  $\text{adj}[a][b] = 1$ , and otherwise  $\text{adj}[a][b] = 0$ . For example, the graph



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.

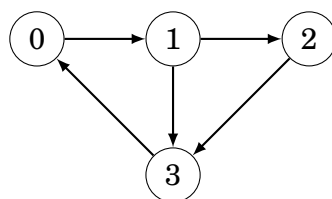
## Edge list representation

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



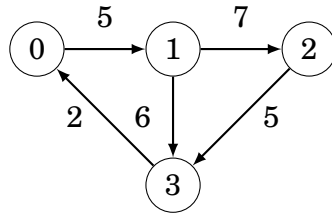
can be represented as follows:

```
edges.push_back({0,2});
edges.push_back({1,3});
edges.push_back({1,4});
edges.push_back({2,4});
edges.push_back({3,1});
```

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<sup>1</sup>:

```
edges.push_back({0,2,5});  
edges.push_back({1,3,7});  
edges.push_back({1,4,6});  
edges.push_back({2,4,5});  
edges.push_back({3,1,2});
```

---

<sup>1</sup>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 2

## 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.

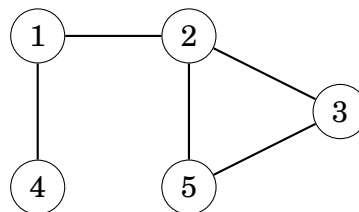
### 2.1 Depth-first search

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

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

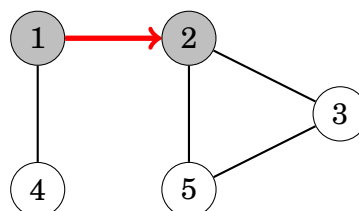
#### Example

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

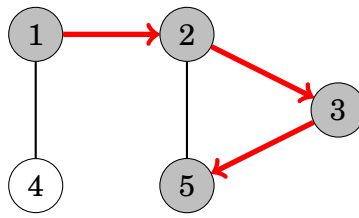


We may begin the search at any node of the graph; now we will begin the search at node 1.

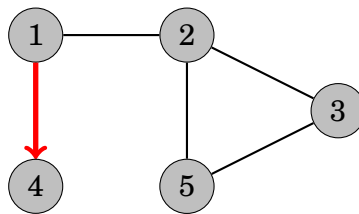
The search first proceeds to node 2:



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<vector<int>>> g;
```

and also maintains an array

```
vector<bool> visited;
```

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 : g[s]) {
        dfs(u);
    }
}
```

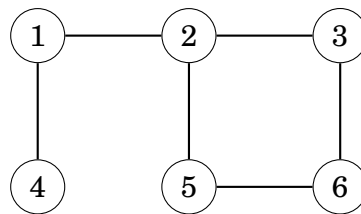
## 2.2 Breadth-first search

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

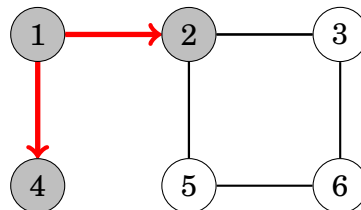
Breadth-first search goes through the nodes one level after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.

### Example

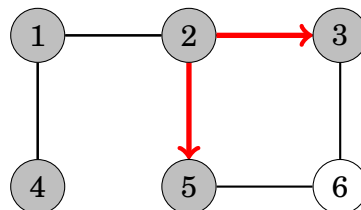
Let us consider how breadth-first search processes the following graph:



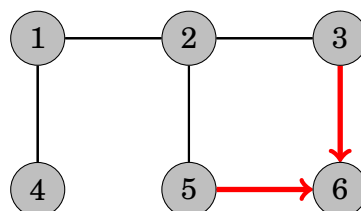
Suppose that the search begins at node 1. First, we process all nodes that can be reached from node 1 using a single edge:



After this, we proceed to nodes 3 and 5:



Finally, we visit node 6:



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

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

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

## Implementation

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

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

```
queue<int> q;
vector<bool> visited(n);
vector<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 : g[s]) {
        if (visited[u]) continue;
        visited[u] = true;
        distance[u] = distance[s]+1;
        q.push(u);
    }
}
```

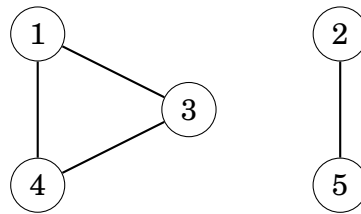
## 2.3 Applications

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

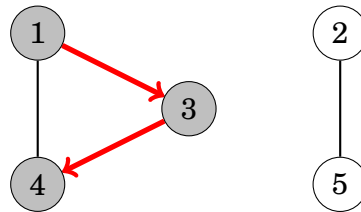
### Connectivity check

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



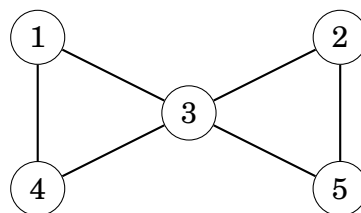
a depth-first search from node 1 visits the following nodes:



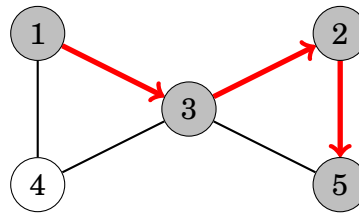
Since the search did not visit all the nodes, we can conclude that the graph is not connected. In a similar way, we can also find all connected components of a graph by iterating through the nodes and always starting a new depth-first search if the current node does not belong to any component yet.

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

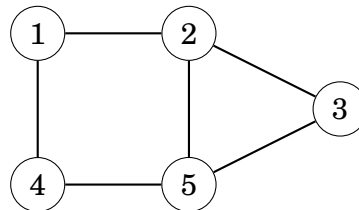
Another way to find out whether a graph contains a cycle is to simply calculate the number of nodes and edges in every component. If a component contains  $c$  nodes and no cycle, it must contain exactly  $c - 1$  edges (so it has to be a tree). If there are  $c$  or more edges, the component surely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

## Bipartiteness check

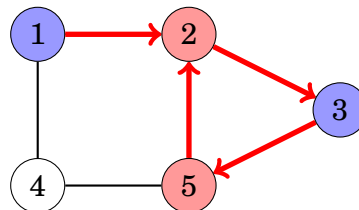
A graph is bipartite if its nodes can be colored using two colors so that there are no adjacent nodes with the same color. It is surprisingly easy to check if a graph is bipartite using graph traversal algorithms.

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



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



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

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

Note that in the general case, it is difficult to find out if the nodes in a graph can be colored using  $k$  colors so that no adjacent nodes have the same color. Even when  $k = 3$ , no efficient algorithm is known but the problem is NP-hard.

# Chapter 3

## Shortest paths

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

In an unweighted graph, the length of a path equals the number of its edges, and we can simply use breadth-first search to find a shortest path. However, in this chapter we focus on weighted graphs where more sophisticated algorithms are needed for finding shortest paths.

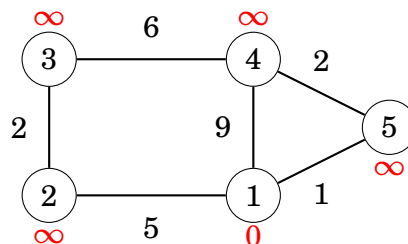
### 3.1 Dijkstra's algorithm

**Dijkstra's algorithm**<sup>1</sup> finds shortest paths from the starting node to all nodes of the graph. Dijkstra's algorithm is very efficient and can be used for processing large graphs. However, the algorithm requires that there are no negative weight edges in the graph.

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:



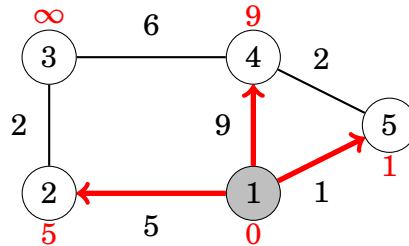
---

<sup>1</sup>E. W. Dijkstra published the algorithm in 1959 [14]; however, his original paper does not mention how to implement the algorithm efficiently.

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

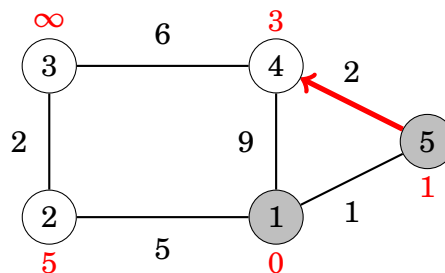
At each step, Dijkstra's algorithm selects a node that has not been processed yet and whose distance is as small as possible. The first such node is node 1 with distance 0.

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

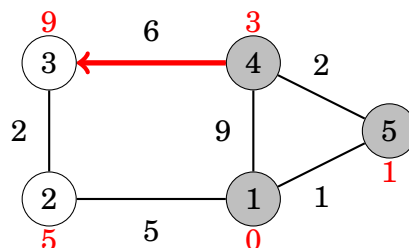


In this case, the edges from node 1 reduced the distances of nodes 2, 4 and 5, whose distances are now 5, 9 and 1.

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



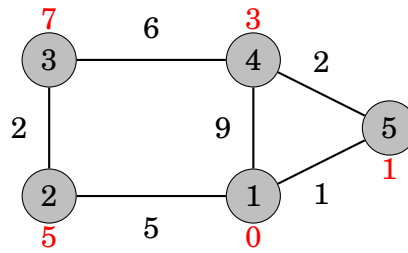
After this, the next node is node 4, which reduces the distance to node 3 to 9:



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.

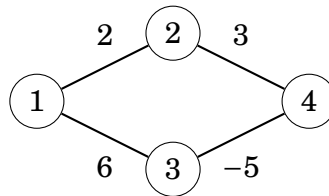
After this, the algorithm processes the two remaining nodes, and the final distances are as follows:





## 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  $1 \rightarrow 3 \rightarrow 4$  and its length is 1. However, Dijkstra's algorithm finds the path  $1 \rightarrow 2 \rightarrow 4$  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  $g[v]$  contains a pair  $(w, \text{cost})$  always when there is an edge from node  $v$  to node  $w$  with weight  $\text{cost}$ .

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  $pq$  contains pairs of the form  $(d, x)$ , meaning that the current distance to node  $x$  is  $d$ .

The array  $\text{distance}$  contains the distance to each node. Initially, the distance is 0 to start and  $-1$  to all other nodes. We use  $-1$  as invalid value to denote that the node has not been reached yet.

```
vector<int> distance(n, -1);
priority_queue<pair<int, int>,
               vector<pair<int, int>>,
               greater<pair<int, int>>> pq;
distance[start] = 0;
```

```

pq.emplace(0, start);
while (!pq.empty()) {
    auto [d, v] = q.top();
    q.pop();
    if (distance[v] != -1) continue;
    distance[v] = d;
    for (auto [w, cost] : g[v])
        q.emplace(d + cost, w);
}

```

Note that the type of the priority queue is not `priority_queue<pair<int, int>` but instead `priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>`. This is because in C++, a priority queue by default puts the *largest* element on top, so we reverse the ordering by changing the comparison operator from `less` (the default) to `greater` (which does the opposite).

In case you forget, you can look up the syntax for the priority queue in the C++ cheatsheet linked on the camp page.

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.

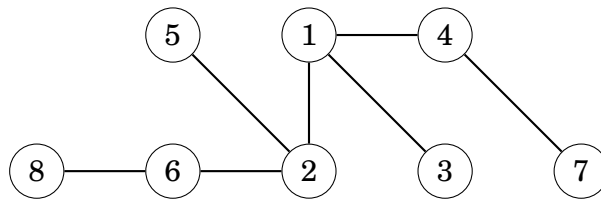
The time complexity of the above implementation is  $O(n + m \log m)$ , because the algorithm goes through all nodes of the graph and adds for each edge at most one distance to the priority queue.

# Chapter 4

## Tree algorithms

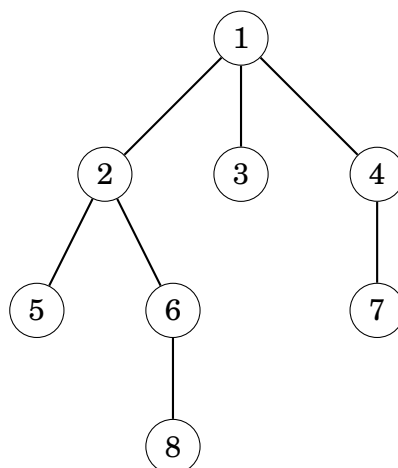
A **tree** is a connected, acyclic graph that consists of  $n$  nodes and  $n - 1$  edges. Removing any edge from a tree divides it into two components, and adding any edge to a tree creates a cycle. Moreover, there is always a unique path between any two nodes of a tree.

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



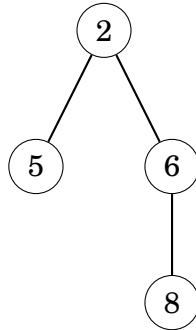
The **leaves** of a tree are the nodes with degree 1, i.e., with only one neighbor. For example, the leaves of the above tree are nodes 3, 5, 7 and 8.

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:



## 4.1 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 v, int p) {  
    for (auto w : g[v])  
        if (w != p)  
            dfs(w, v);  
}
```

The function is given two parameters: the current node  $v$  and the previous node  $p$ . The purpose of the parameter  $p$  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, -1);
```

In the first call  $p = -1$ , because there is no previous node, and it is allowed to proceed to any direction in the tree.

### Storing Information

We can calculate some information during a tree traversal and store that for later use. 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.

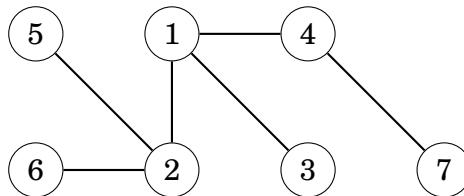
As an example, let us calculate for each node  $v$  a value `subtreesize[v]`: the number of nodes in its subtree. The subtree contains the node itself and all

nodes in the subtrees of its children, so we can calculate the number of nodes recursively using the following code:

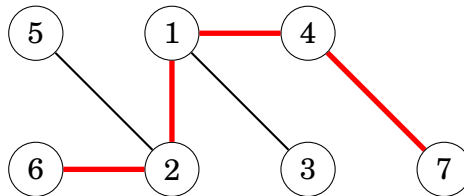
```
void dfs(int v, int p) {
    subtreesize[v] = 1;
    for (auto w : g[v]) {
        if (w == p) continue;
        dfs(w, v);
        subtreesize[v] += subtreesize[w];
    }
}
```

## 4.2 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.

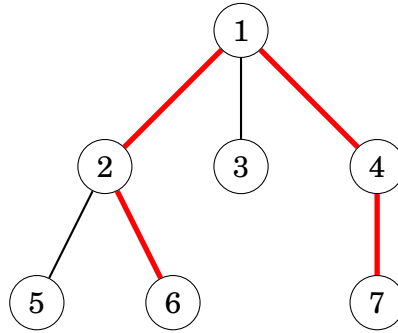
Next we will discuss two  $O(n)$  time algorithms for calculating the diameter of a tree. The first algorithm is based on the previous idea of storing information, and the second algorithm uses two depth-first searches.

### 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:

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

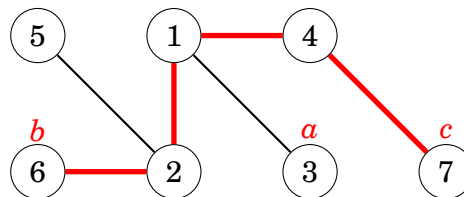
For example, in the above tree,  $\text{toLeaf}(1) = 2$ , because there is a path  $1 \rightarrow 2 \rightarrow 6$ , and  $\text{maxLength}(1) = 4$ , because there is a path  $6 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 7$ . In this case,  $\text{maxLength}(1)$  equals the diameter.

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

## Algorithm 2

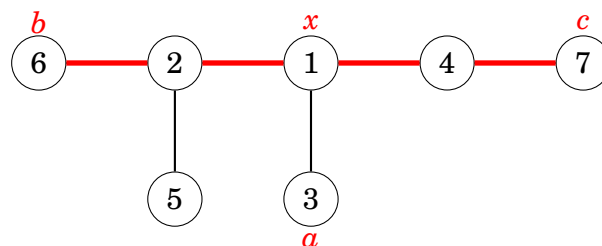
Another efficient way to calculate the diameter of a tree is based on two depth-first searches. First, we choose an arbitrary node  $a$  in the tree and find the farthest node  $b$  from  $a$ . Then, we find the farthest node  $c$  from  $b$ . The diameter of the tree is the distance between  $b$  and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

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:

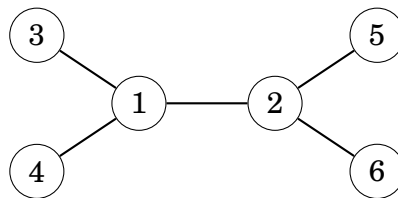


Node  $x$  indicates the place where the path from node  $a$  joins the path that corresponds to the diameter. The farthest node from  $a$  is node  $b$ , node  $c$  or some other node that is at least as far from node  $x$ . Thus, this node is always a valid choice for an endpoint of a path that corresponds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

## 4.3 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.

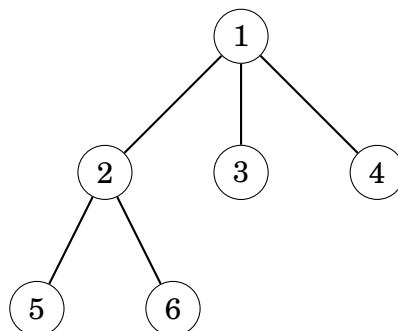
As an example, consider the following tree:



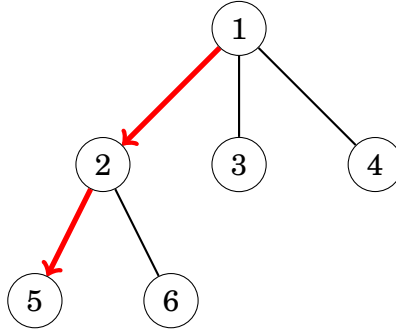
Let  $\text{maxLength}(x)$  denote the maximum length of a path that begins at node  $x$ . For example, in the above tree,  $\text{maxLength}(4) = 3$ , because there is a path  $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$ . Here is a complete table of the values:

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

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

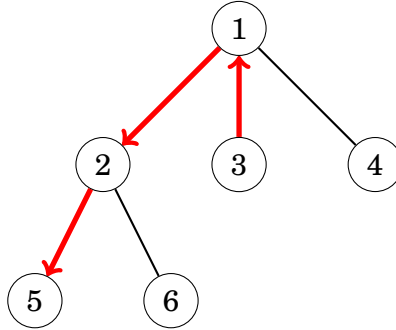


The first part of the problem is to calculate for every node  $x$  the maximum length of a path that goes through a child of  $x$ . For example, the longest path from node 1 goes through its child 2:

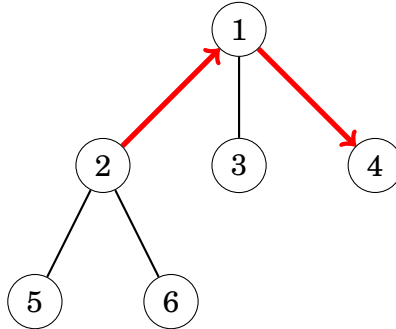


This part is easy to solve in  $O(n)$  time, because we can use a similar technique to what we have done previously.

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  $x$ :

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

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

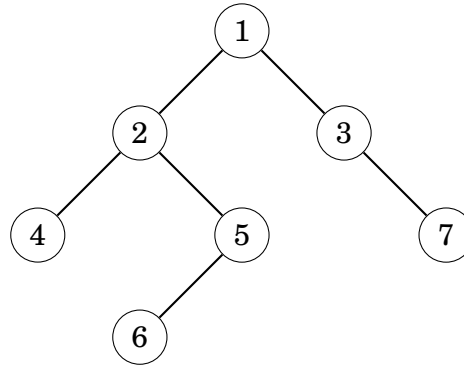
Finally, if the path that corresponds to  $\text{maxLength}_1(p)$  goes through  $x$ , we conclude that the maximum length is  $\text{maxLength}_2(p) + 1$ , and otherwise the maximum length is  $\text{maxLength}_1(p) + 1$ .



## 4.4 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



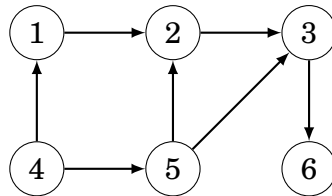
the pre-order is [1,2] and the post-order is [2,1], but the structures of the trees are different.



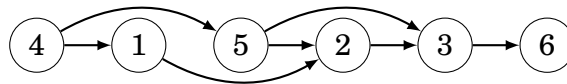
# Chapter 5

## 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.

### 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)

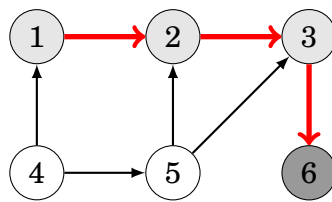
Initially, the state of each node is 0. When a search reaches a node for the first time, its state becomes 1. Finally, after all successors of the node have been processed, its state becomes 2.

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.

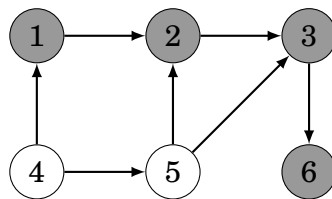
If the graph does not contain a cycle, we can construct a topological sort by adding each node to a list when the state of the node becomes 2. This list in reverse order is a topological sort.

## Example 1

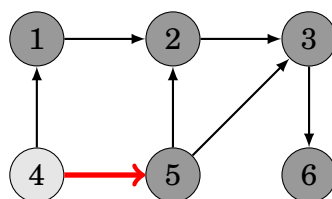
In the example graph, the search first proceeds from node 1 to node 6:



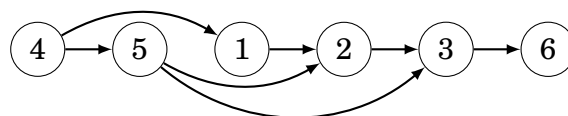
Now node 6 has been processed, so it is added to the list. After this, also nodes 3, 2 and 1 are added to the list:



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



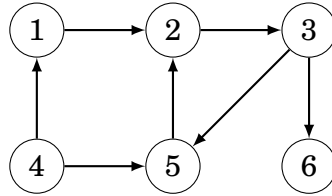
Thus, the final list is [6,3,2,1,5,4]. We have processed all nodes, so a topological sort has been found. The topological sort is the reverse list [4,5,1,2,3,6]:



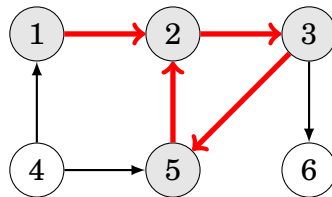
Note that a topological sort is not unique, and there can be several topological sorts for a graph.

## 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  $2 \rightarrow 3 \rightarrow 5 \rightarrow 2$ .



# **Part II**

## **Advanced topics**



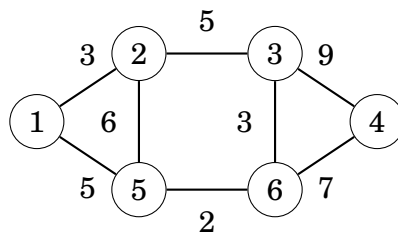


# Chapter 6

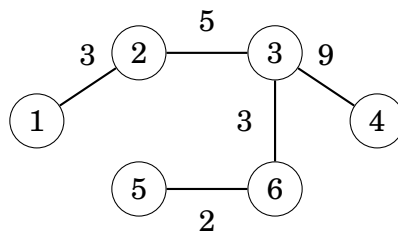
## Spanning trees

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:

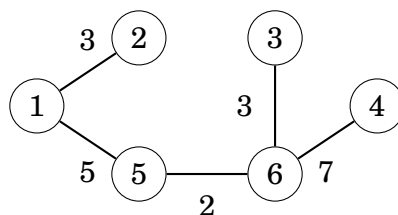


One spanning tree for the graph is as follows:

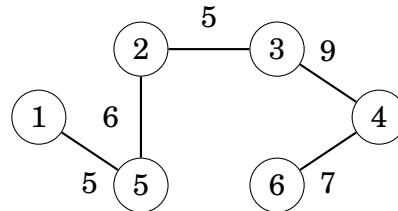


The weight of a spanning tree is the sum of its edge weights. For example, the weight of the above spanning tree is  $3 + 5 + 9 + 3 + 2 = 22$ .

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.

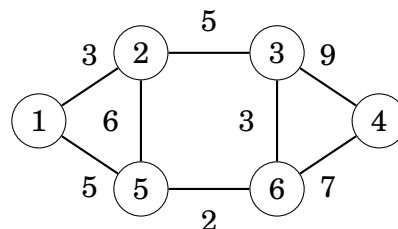
## 6.1 Kruskal's algorithm

In **Kruskal's algorithm**<sup>1</sup>, 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:



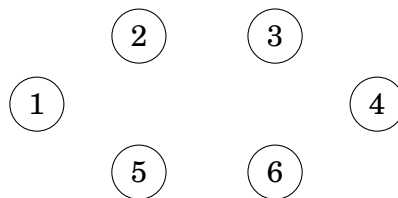
The first step of the algorithm is to sort the edges in increasing order of their weights. The result is the following list:

<sup>1</sup>The algorithm was published in 1956 by J. B. Kruskal [48].

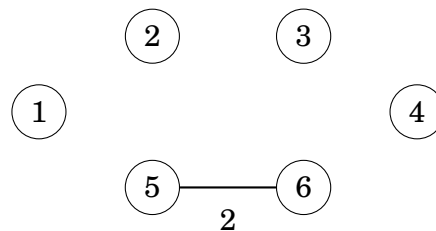
edge	weight
5-6	2
1-2	3
3-6	3
1-5	5
2-3	5
2-5	6
4-6	7
3-4	9

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

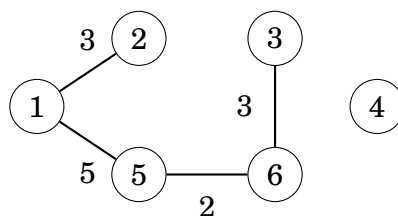
Initially, each node is in its own component:



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}:



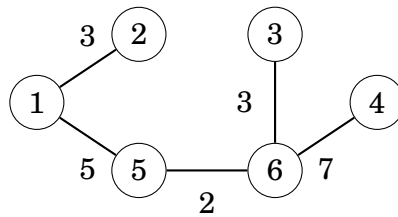
After this, the edges 1-2, 3-6 and 1-5 are added in a similar way:



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:

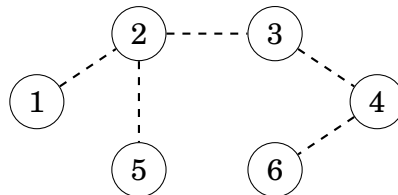


After this, the algorithm will not add any new edges, because the graph is connected and there is a path between any two nodes. The resulting graph is a minimum spanning tree with weight  $2 + 3 + 3 + 5 + 7 = 20$ .

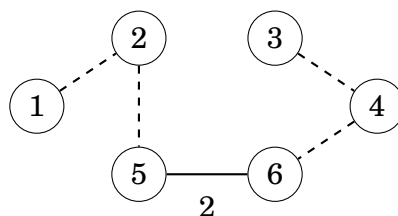
## Why does this work?

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.

## 6.2 Union-find structure

A **union-find structure** maintains a collection of sets. The sets are disjoint, so no element belongs to more than one set. Two  $O(\log n)$  time operations are supported: the `unite` operation joins two sets, and the `find` operation finds the representative of the set that contains a given element<sup>2</sup>.

### 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  $\{2, 3, 6, 8\}$ :

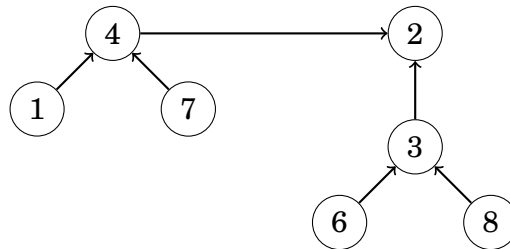


---

<sup>2</sup>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.

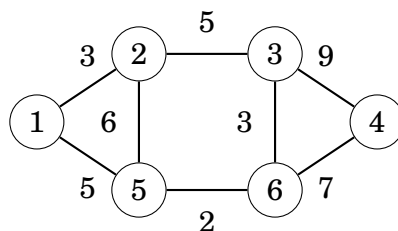
## 6.3 Prim's algorithm

**Prim's algorithm**<sup>3</sup> 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.

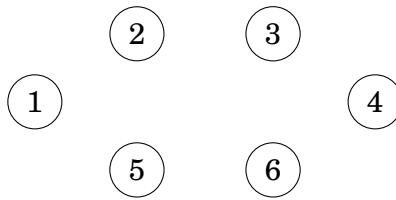
### Example

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

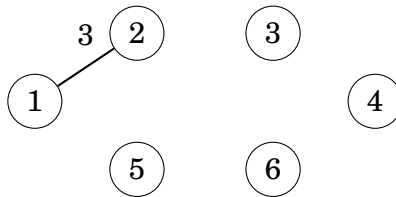


<sup>3</sup>The algorithm is named after R. C. Prim who published it in 1957 [54]. However, the same algorithm was discovered already in 1930 by V. Jarník.

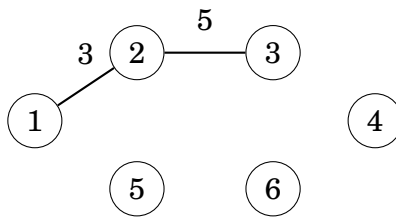
Initially, there are no edges between the nodes:



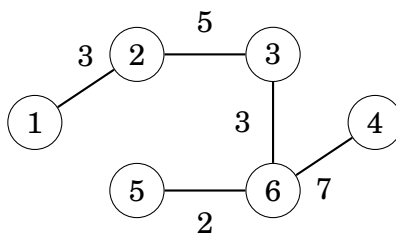
An arbitrary node can be the starting node, so let us choose node 1. First, we add node 2 that is connected by an edge of weight 3:



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.

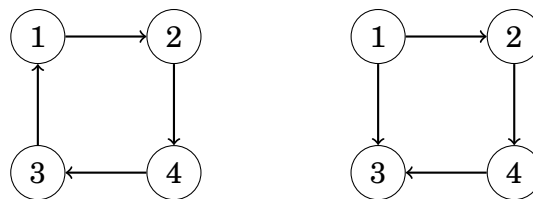


# Chapter 7

## 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.

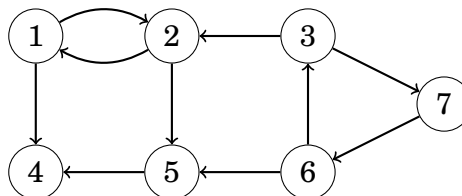
A graph is **strongly connected** if there is a path from any node to all other nodes in the graph. For example, in the following picture, the left graph is strongly connected while the right graph is not.



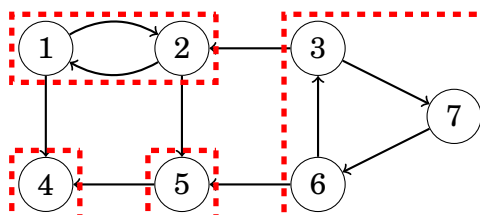
The right graph is not strongly connected because, for example, there is no path from node 2 to node 1.

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.

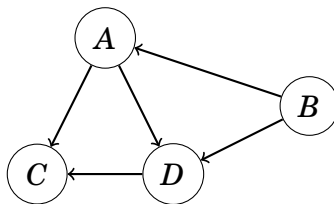
For example, for the graph



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.

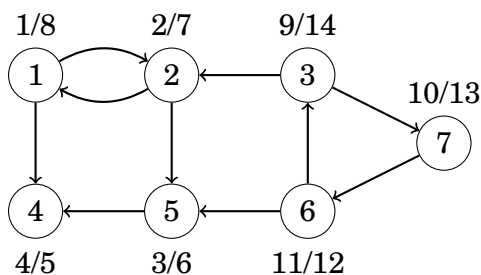
## 7.1 Kosaraju's algorithm

**Kosaraju's algorithm**<sup>1</sup> 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:

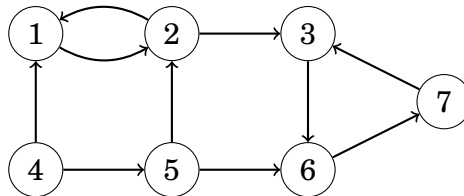
<sup>1</sup>According to [1], S. R. Kosaraju invented this algorithm in 1978 but did not publish it. In 1981, the same algorithm was rediscovered and published by M. Sharir [57].

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

## Search 2

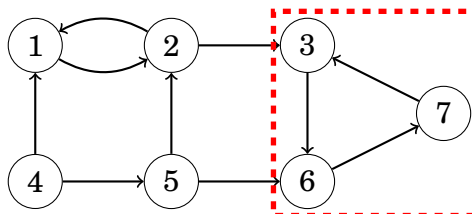
The second phase of the algorithm forms the strongly connected components of the graph. First, the algorithm reverses every edge in the graph. This guarantees that during the second search, we will always find strongly connected components that do not have extra nodes.

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



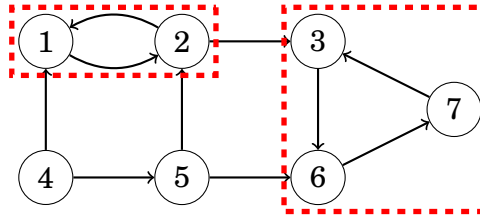
After this, the algorithm goes through the list of nodes created by the first search, in *reverse* order. If a node does not belong to a component, the algorithm creates a new component and starts a depth-first search that adds all new nodes found during the search to the new component.

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

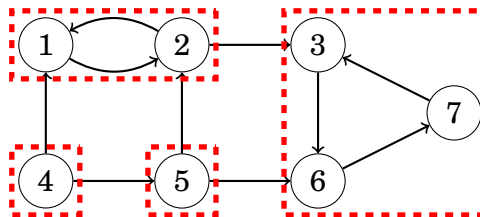


Note that since all edges are reversed, the component does not "leak" to other parts in the graph.

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.

# Chapter 8

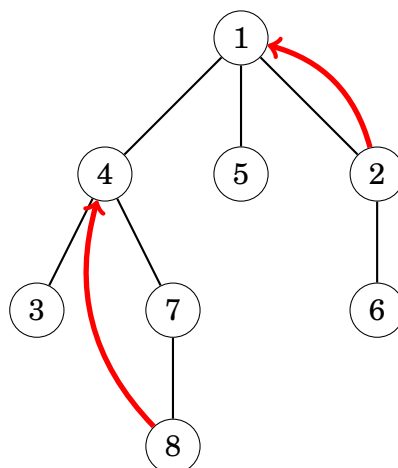
## Tree queries

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

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

### 8.1 Finding ancestors

The  $k$ th **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  $\text{ancestor}(x, k)$  denote the  $k$ th ancestor of a node  $x$  (or 0 if there is no such an ancestor). For example, in the following tree,  $\text{ancestor}(2, 1) = 1$  and  $\text{ancestor}(8, 2) = 4$ .



An easy way to calculate any value of  $\text{ancestor}(x, k)$  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.

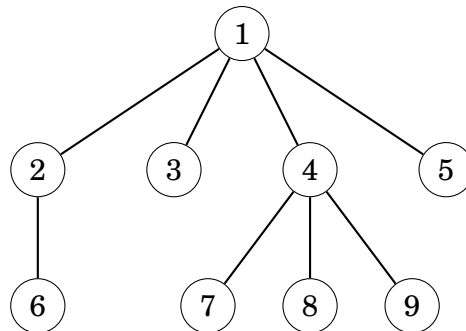
Fortunately, using a technique similar to that used in Chapter 16.3 (of the full book), any value of  $\text{ancestor}(x, k)$  can be efficiently calculated in  $O(\log k)$  time after preprocessing. The idea is to precalculate all values  $\text{ancestor}(x, k)$  where  $k \leq n$  is a power of two.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

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

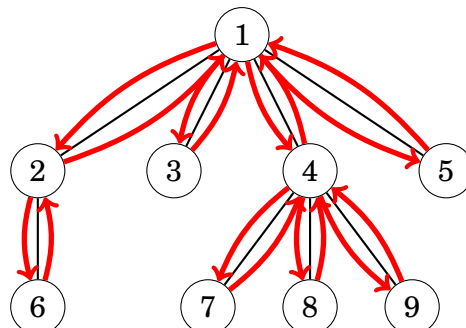
The preprocessing takes  $O(n \log n)$  time, because  $O(\log n)$  values are calculated for each node. After this, any value of  $\text{ancestor}(x, k)$  can be calculated in  $O(\log k)$  time by representing  $k$  as a sum where each term is a power of two.

## 8.2 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

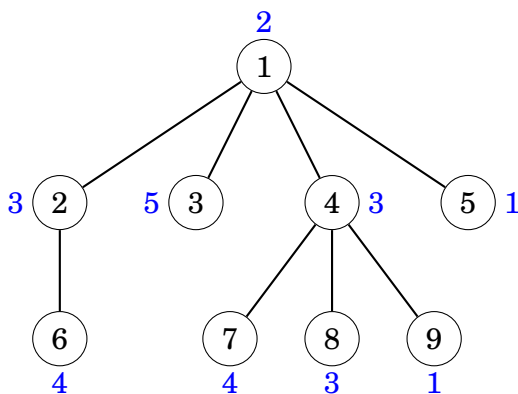
Each subtree of a tree corresponds to a subarray of the tree traversal array such that the first element of the subarray is the root node. For example, the following subarray contains the nodes of the subtree of node 4:

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

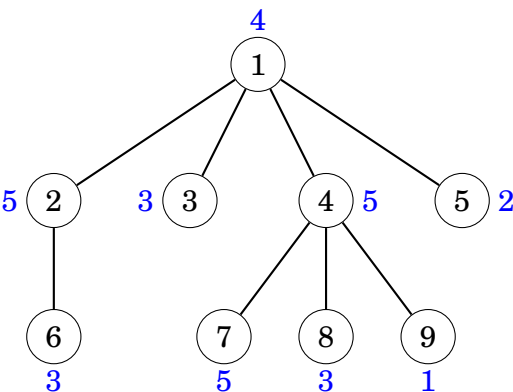
To answer the queries efficiently, it suffices to store the values of the nodes in a binary indexed or segment tree. After this, we can both update a value and calculate the sum of values in  $O(\log n)$  time.

# 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

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

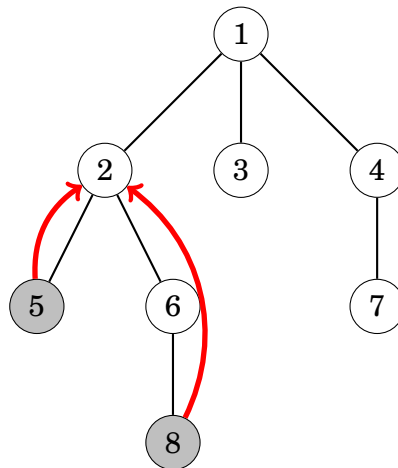
Thus, to support both the operations, we should be able to increase all values in a range and retrieve a single value. This can be done in  $O(\log n)$  time using a binary indexed or segment tree (see Chapter 9.4).



## 8.3 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:



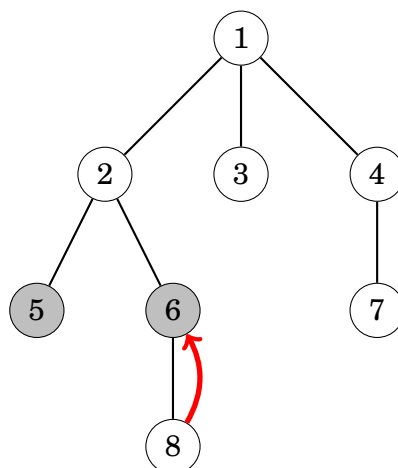
Next we will discuss two efficient techniques for finding the lowest common ancestor of two nodes.

### Method 1

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

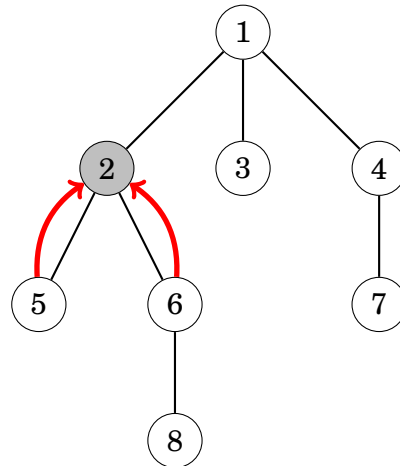
We use two pointers that initially point to the two nodes whose lowest common ancestor we should find. First, we move one of the pointers upwards so that both pointers point to nodes at the same level.

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.

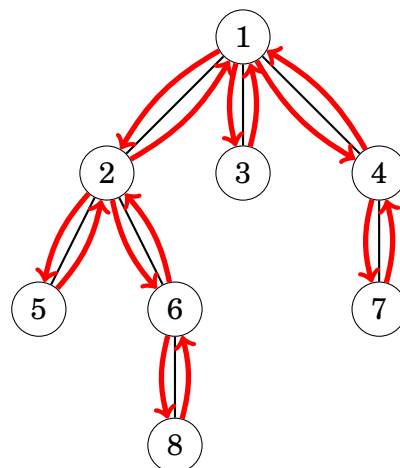
In the example scenario, it suffices to move both pointers one step upwards to node 2, which is the lowest common ancestor:



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<sup>1</sup>. 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.

<sup>1</sup>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.

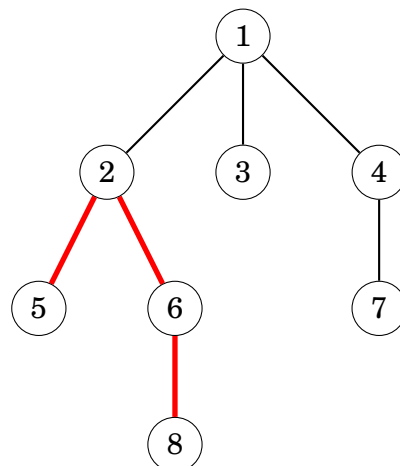
## Distances of nodes

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

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

where  $c$  is the lowest common ancestor of  $a$  and  $b$  and  $\text{depth}(s)$  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  $\text{depth}(5) = 3$ ,  $\text{depth}(8) = 4$  and  $\text{depth}(2) = 2$ , so the distance between nodes 5 and 8 is  $3 + 4 - 2 \cdot 2 = 3$ .

## 8.4 Offline algorithms

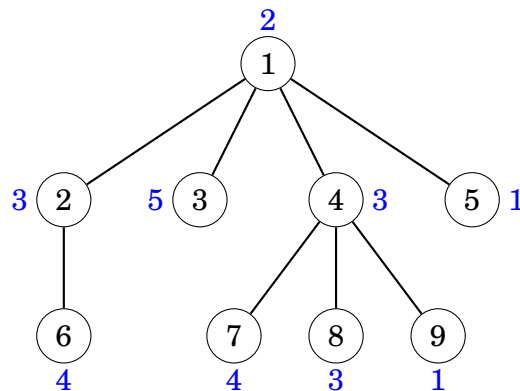
So far, we have discussed *online* algorithms for tree queries. Those algorithms are able to process queries one after another so that each query is answered before receiving the next query.

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.

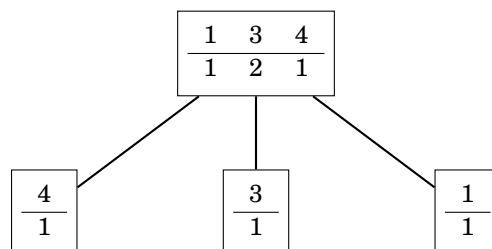
### Merging data structures

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  $d[s]$  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  $d[s]$  that only contains the value of  $s$ . After this, we go through the children of  $s$  and *merge*  $d[s]$  and all data structures  $d[u]$  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:

$\frac{3}{1}$	$\frac{4}{1}$	$\frac{3}{1}$	$\frac{1}{1}$
---------------	---------------	---------------	---------------

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  $d[s]$  and  $d[u]$ . We always copy the contents from  $d[u]$  to  $d[s]$ . However, before this, we *swap* the contents of  $d[s]$  and  $d[u]$  if  $d[s]$  is smaller than  $d[u]$ . 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

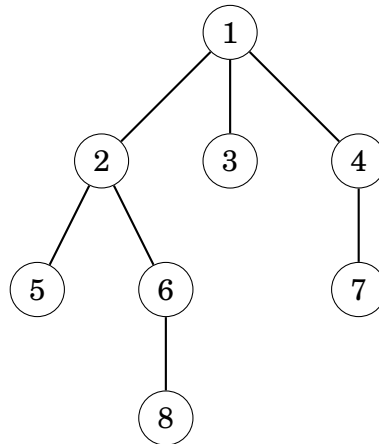
There is also an offline algorithm for processing a set of lowest common ancestor queries<sup>2</sup>. The algorithm is based on the union-find data structure (see Chapter 15.2), and the benefit of the algorithm is that it is easier to implement than the algorithms discussed earlier in this chapter.

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.

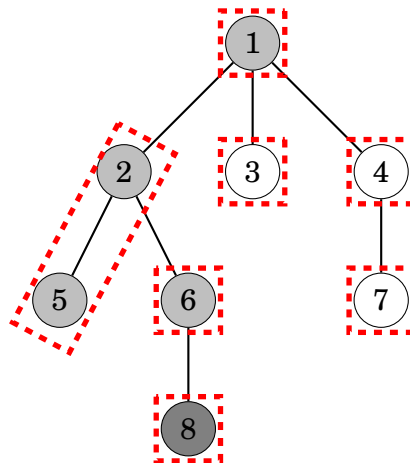
When the algorithm visits a node  $x$ , it goes through all nodes  $y$  such that the lowest common ancestor of  $x$  and  $y$  has to be found. If  $y$  has already been visited, the algorithm reports that the lowest common ancestor of  $x$  and  $y$  is the highest node in the set of  $y$ . Then, after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

<sup>2</sup>This algorithm was published by R. E. Tarjan in 1979 [65].

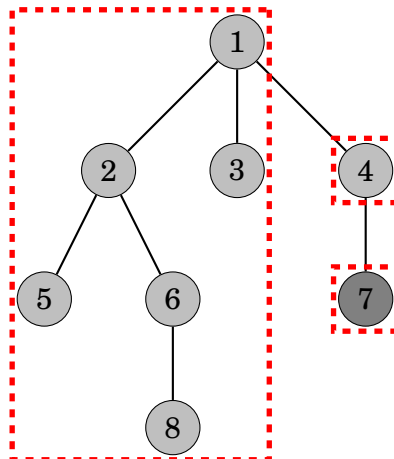
For example, suppose that we want to find the lowest common ancestors of node pairs (5,8) and (2,7) in the following tree:



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 9

## Paths and circuits

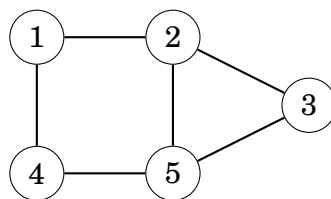
This chapter focuses on two types of paths in graphs:

- 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.

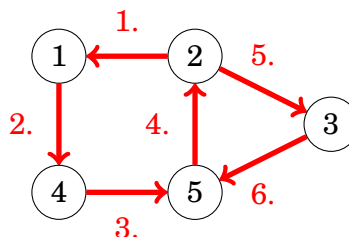
While Eulerian and Hamiltonian paths look like similar concepts at first glance, the computational problems related to them are very different. It turns out that there is a simple rule that determines whether a graph contains an Eulerian path, and there is also an efficient algorithm to find such a path 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.

### 9.1 Eulerian paths

An **Eulerian path**<sup>1</sup> is a path that goes exactly once through each edge of the graph. For example, the graph



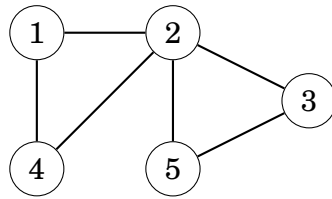
has an Eulerian path from node 2 to node 5:



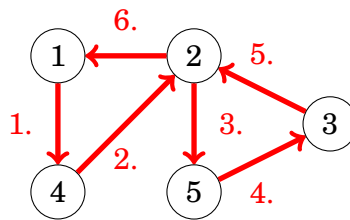
---

<sup>1</sup>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:



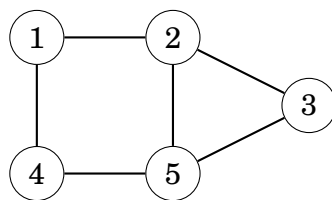
## Existence

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

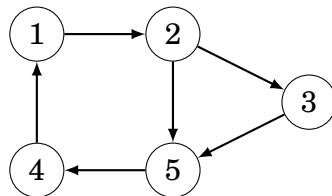
- in each node, the indegree equals the outdegree, *or*



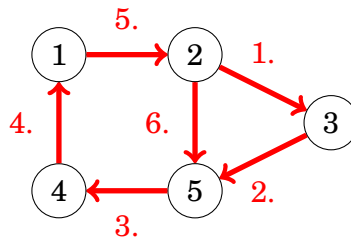
- in one node, the indegree is one larger than the outdegree, in another node, the outdegree is one larger than the indegree, and in all other nodes, the indegree equals the outdegree.

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**<sup>2</sup> 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.

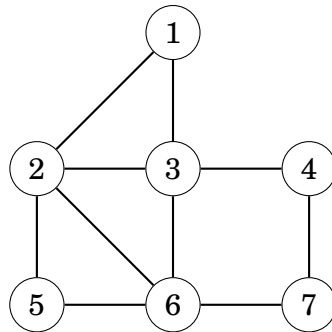
If the graph only contains an Eulerian path, we can still use Hierholzer's algorithm to find it by adding an extra edge to the graph and removing the edge after the circuit has been constructed. For example, in an undirected graph, we add the extra edge between the two odd-degree nodes.

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

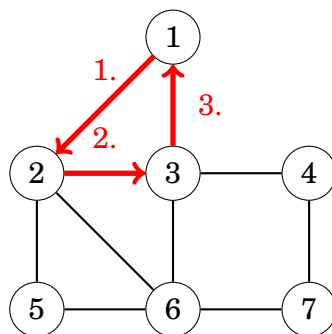
<sup>2</sup>The algorithm was published in 1873 after Hierholzer's death [35].

## Example

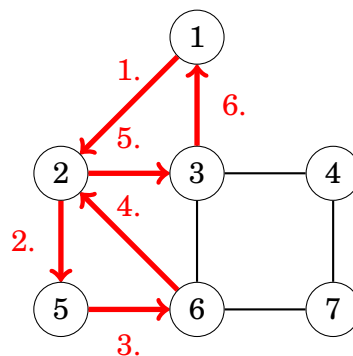
Let us consider the following graph:



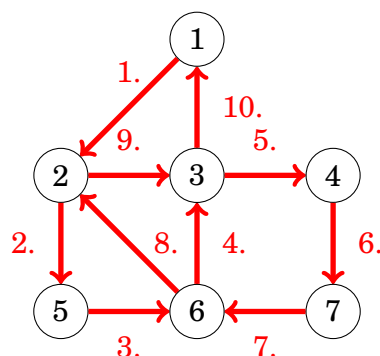
Suppose that the algorithm first creates a circuit that begins at node 1. A possible circuit is  $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ :



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



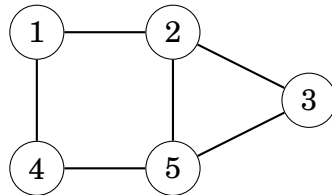
Finally, the algorithm adds the subcircuit  $6 \rightarrow 3 \rightarrow 4 \rightarrow 7 \rightarrow 6$  to the circuit:



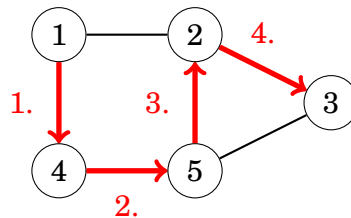
Now all edges are included in the circuit, so we have successfully constructed an Eulerian circuit.

## 9.2 Hamiltonian paths

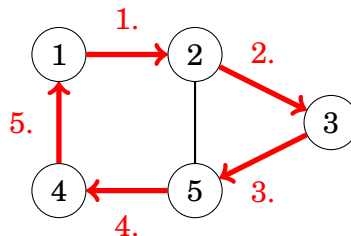
A **Hamiltonian path** is a path that visits each node of the graph exactly once. For example, the graph



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  $\text{possible}(S, x)$ , 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.

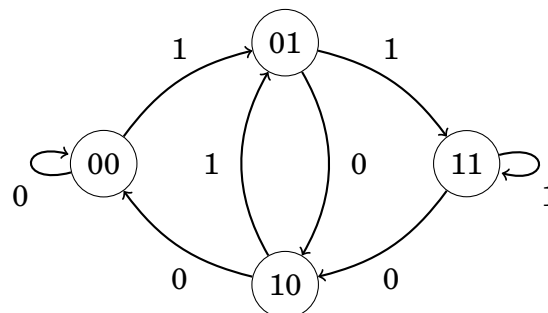
## 9.3 De Bruijn sequences

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  $k^n + n - 1$ .

## 9.4 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  $5 \times 5$  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<sup>3</sup>. 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  $a \dots e$ ):

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.

---

<sup>3</sup>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.



# Bibliography

- [1] A. V. Aho, J. E. Hopcroft and J. Ullman. *Data Structures and Algorithms*, Addison-Wesley, 1983.
- [2] R. K. Ahuja and J. B. Orlin. Distance directed augmenting path algorithms for maximum flow and parametric maximum flow problems. *Naval Research Logistics*, 38(3):413–430, 1991.
- [3] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.
- [4] B. Aspvall, M. F. Plass and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.
- [5] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
- [6] M. Beck, E. Pine, W. Tarrat and K. Y. Jensen. New integer representations as the sum of three cubes. *Mathematics of Computation*, 76(259):1683–1690, 2007.
- [7] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, 88–94, 2000.
- [8] J. Bentley. *Programming Pearls*. Addison-Wesley, 1999 (2nd edition).
- [9] J. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, C-29(7):571–577, 1980.
- [10] C. L. Bouton. Nim, a game with a complete mathematical theory. *Annals of Mathematics*, 3(1/4):35–39, 1901.
- [11] Croatian Open Competition in Informatics, <http://hsin.hr/coci/>
- [12] Codeforces: On "Mo's algorithm", <http://codeforces.com/blog/entry/20032>
- [13] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms*, MIT Press, 2009 (3rd edition).

- [14] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- [15] K. Diks et al. *Looking for a Challenge? The Ultimate Problem Set from the University of Warsaw Programming Competitions*, University of Warsaw, 2012.
- [16] M. Dima and R. Ceterchi. Efficient range minimum queries using binary indexed trees. *Olympiad in Informatics*, 9(1):39–44, 2015.
- [17] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.
- [18] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.
- [19] S. Even, A. Itai and A. Shamir. On the complexity of time table and multi-commodity flow problems. *16th Annual Symposium on Foundations of Computer Science*, 184–193, 1975.
- [20] D. Fanding. A faster algorithm for shortest-path – SPFA. *Journal of Southwest Jiaotong University*, 2, 1994.
- [21] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.
- [22] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Annual Symposium on Combinatorial Pattern Matching*, 36–48, 2006.
- [23] R. W. Floyd Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [24] L. R. Ford. Network flow theory. RAND Corporation, Santa Monica, California, 1956.
- [25] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.
- [26] R. Freivalds. Probabilistic machines can use less running time. In *IFIP congress*, 839–842, 1977.
- [27] F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, 296–303, 2014.
- [28] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.
- [29] Google Code Jam Statistics (2017), <https://www.go-hero.net/jam/17>



- [30] A. Grønlund and S. Pettie. Threesomes, degenerates, and love triangles. In *Proceedings of the 55th Annual Symposium on Foundations of Computer Science*, 621–630, 2014.
- [31] P. M. Grundy. Mathematics and games. *Eureka*, 2(5):6–8, 1939.
- [32] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
- [33] S. Halim and F. Halim. *Competitive Programming 3: The New Lower Bound of Programming Contests*, 2013.
- [34] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.
- [35] C. Hierholzer and C. Wiener. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6(1), 30–32, 1873.
- [36] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.
- [37] C. A. R. Hoare. Algorithm 65: Find. *Communications of the ACM*, 4(7):321–322, 1961.
- [38] J. E. Hopcroft and J. D. Ullman. A linear list merging algorithm. Technical report, Cornell University, 1971.
- [39] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, 1974.
- [40] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [41] The International Olympiad in Informatics Syllabus, <https://people.ksp.sk/~misof/ioi-syllabus/>
- [42] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
- [43] P. W. Kasteleyn. The statistics of dimers on a lattice: I. The number of dimer arrangements on a quadratic lattice. *Physica*, 27(12):1209–1225, 1961.
- [44] C. Kent, G. M. Landau and M. Ziv-Ukelson. On the complexity of sparse exon assembly. *Journal of Computational Biology*, 13(5):1013–1027, 2006.
- [45] J. Kleinberg and É. Tardos. *Algorithm Design*, Pearson, 2005.
- [46] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*, Addison–Wesley, 1998 (3rd edition).

- [47] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Addison–Wesley, 1998 (2nd edition).
- [48] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.
- [49] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.
- [50] M. G. Main and R. J. Lorentz. An  $O(n \log n)$  algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.
- [51] J. Pachocki and J. Radoszewski. Where to use and how not to use polynomial string hashing. *Olympiads in Informatics*, 7(1):90–100, 2013.
- [52] I. Parberry. An efficient algorithm for the Knight’s tour problem. *Discrete Applied Mathematics*, 73(3):251–260, 1997.
- [53] D. Pearson. A polynomial-time algorithm for the change-making problem. *Operations Research Letters*, 33(3):231–234, 2005.
- [54] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957.
- [55] 27-Queens Puzzle: Massively Parallel Enumeration and Solution Counting. <https://github.com/preusser/q27>
- [56] M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, 151–162, 1975.
- [57] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67–72, 1981.
- [58] S. S. Skiena. *The Algorithm Design Manual*, Springer, 2008 (2nd edition).
- [59] S. S. Skiena and M. A. Revilla. *Programming Challenges: The Programming Contest Training Manual*, Springer, 2003.
- [60] SZKOpuł, <https://szkopul.edu.pl/>
- [61] R. Sprague. Über mathematische Kampfspiele. *Tohoku Mathematical Journal*, 41:438–444, 1935.
- [62] P. Stańczyk. *Algorytmika praktyczna w konkursach Informatycznych*, MSc thesis, University of Warsaw, 2006.
- [63] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.
- [64] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

- [65] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.
- [66] R. E. Tarjan and U. Vishkin. Finding biconnected components and computing tree functions in logarithmic parallel time. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, 12–20, 1984.
- [67] H. N. V. Temperley and M. E. Fisher. Dimer problem in statistical mechanics – an exact result. *Philosophical Magazine*, 6(68):1061–1063, 1961.
- [68] USA Computing Olympiad, <http://www.usaco.org/>
- [69] H. C. von Warnsdorf. *Des Rösselsprunges einfachste und allgemeinste Lösung*. Schmalkalden, 1823.
- [70] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

