Chapter 11

Basics of graphs

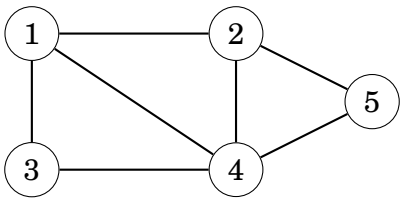
Many programming problems can be solved by modeling the problem as a graph problem and using an appropriate graph algorithm. A typical example of a graph is a network of roads and cities in a country. Sometimes, though, the graph is hidden in the problem and it may be difficult to detect it.

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

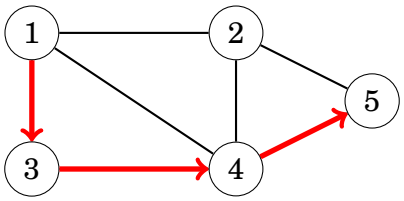
11.1 Graph terminology

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

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



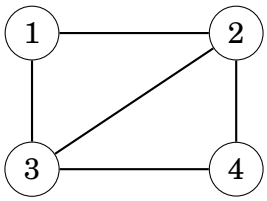
A **path** leads from node a to node b through edges of the graph. The **length** of a path is the number of edges in it. For example, the above graph contains the path $1 \to 3 \to 4 \to 5$ from node 1 to node 5:



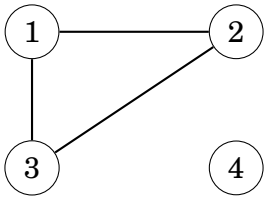
A path is a **cycle** if the first and last node is the same. For example, the above graph contains the cycle $1 \to 3 \to 4 \to 1$. A path is **simple** if each node appears at most once in the path.

Connectivity

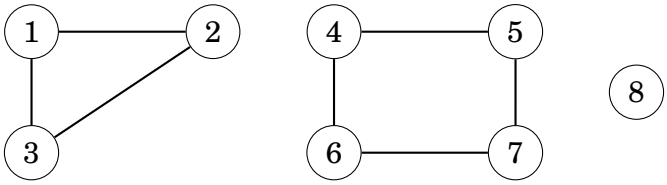
A graph is **connected** if there is path between any two nodes. For example, the following graph is connected:



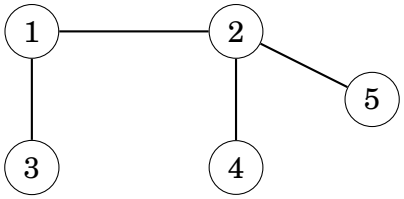
The following graph is not connected, because it is not possible to get from node 4 to any other node:



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 $\{8\}$.

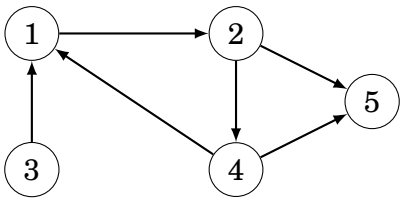


A **tree** is a connected graph that consists of n nodes and $n-1$ edges. There is a unique path between any two nodes in a tree. For example, the following graph is a tree:



Edge directions

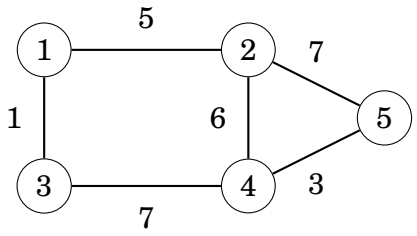
A graph is **directed** if the edges can be traversed in one direction only. For example, the following graph is directed:



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

Edge weights

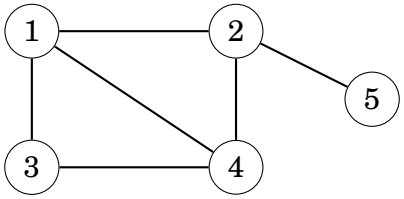
In a **weighted** graph, each edge is assigned a **weight**. Often the weights are interpreted as edge lengths. For example, the following graph is weighted:



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

Neighbors and degrees

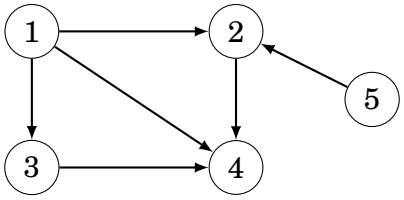
Two nodes are **neighbors** or **adjacent** if there is an edge between them. The **degree** of a node is the number of its neighbors. For example, in the following graph, the neighbors of node 2 are 1, 4 and 5, so its degree is 3.



The sum of degrees in a graph is always $2m$, where m is the number of edges, because each edge increases the degree of exactly two nodes by one. For this reason, the sum of degrees is always even.

A graph is **regular** if the degree of every node is a constant d. A graph is **complete** if the degree of every node is $n-1$, i.e., the graph contains all possible edges between the nodes.

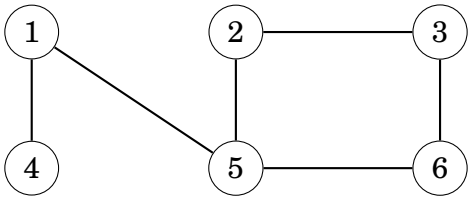
In a directed graph, the **indegree** of a node is the number of edges that end at the node, and the **outdegree** of a node is the number of edges that start at the node. For example, in the following graph, the indegree of node 2 is 2 and the outdegree of node 2 is 1.



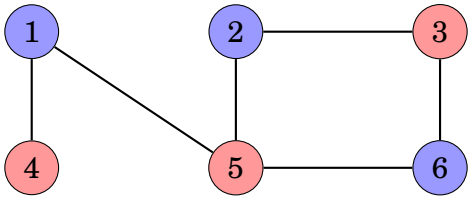
Colorings

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

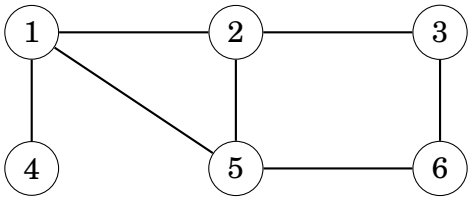
A graph is **bipartite** if it is possible to color it using two colors. It turns out that a graph is bipartite exactly when it does not contain a cycle with an odd number of edges. For example, the graph



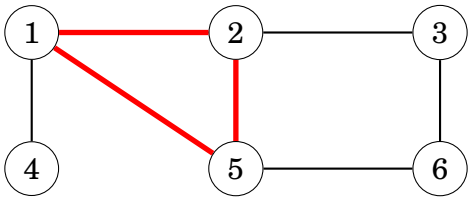
is bipartite, because it can be colored as follows:



However, the graph

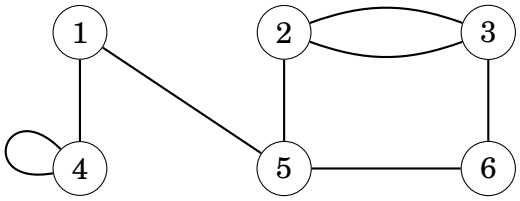


is not bipartite, because it is not possible to color the following cycle of three nodes using two colors:



Simplicity

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



11.2 Graph representation

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

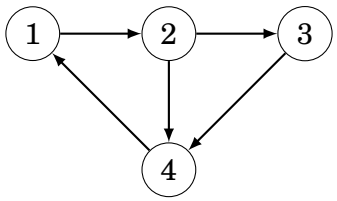
Adjacency list representation

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

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

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

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



can be stored as follows:

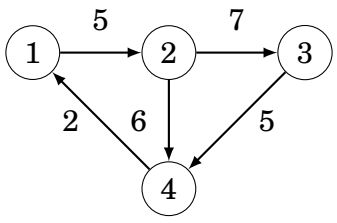
```
v[1].push_back(2);
v[2].push_back(3);
v[2].push_back(4);
v[3].push_back(4);
v[4].push_back(1);
```

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

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

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

If there is an edge from node a to node b with weight w, the adjacency list of node a contains the pair (b,w). For example, the graph



can be stored as follows:

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

The benefit in 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:

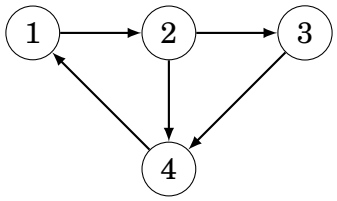
```
for (auto u : v[s]) {
    // process node u
}
```

Adjacency matrix representation

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

```
int v[N][N];
```

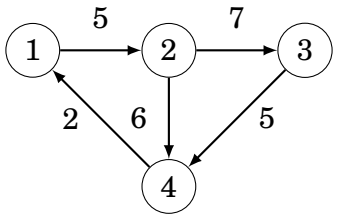
where each value v[a][b] indicates whether the graph contains an edge from node a to node b. If the edge is included in the graph, then v[a][b] = 1, and otherwise v[a][b] = 0. For example, the graph



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

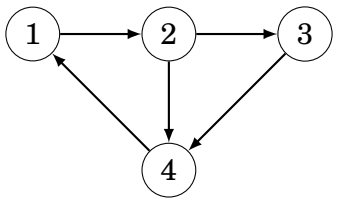
Edge list representation

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>> v;
```

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



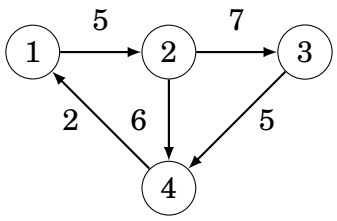
can be represented as follows:

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

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

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

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:

```
v.push_back(make_tuple(1,2,5));
v.push_back(make_tuple(2,3,7));
v.push_back(make_tuple(2,4,6));
v.push_back(make_tuple(3,4,5));
v.push_back(make_tuple(4,1,2));
```

Chapter 12

Graph traversal

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

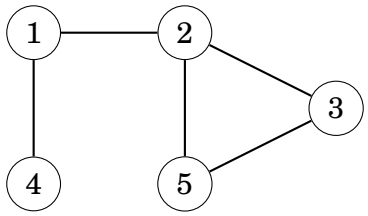
12.1 Depth-first search

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

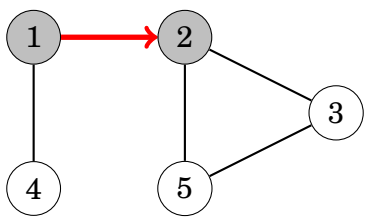
Example

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

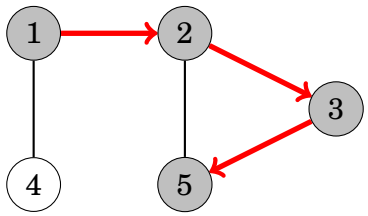


We may begin the search at any node in the graph, but we will now begin the search at node 1.

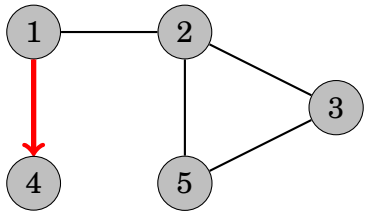
The search first proceeds to node 2:



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 previous nodes. Also the neighbors of nodes 3 and 2 have been visited, so we next move from node 1 to node 4:



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

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

Implementation

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

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

and also maintains an array

```
int z[N];
```

that keeps track of the visited nodes. Initially, each array value is 0, and when the search arrives at node s, the value of z[s] becomes 1. The function can be implemented as follows:

```
void dfs(int s) {
    if (z[s]) return;
    z[s] = 1;
    // process node s
    for (auto u: v[s]) {
        dfs(u);
    }
}
```

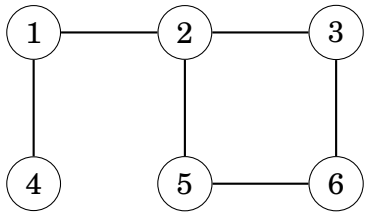
12.2 Breadth-first search

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

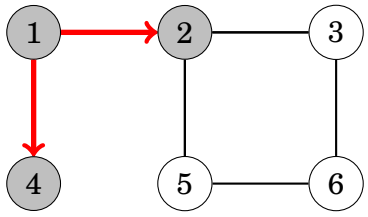
Breadth-first search goes through the nodes one level after another. First the search explores the nodes whose distance from the starting node is 1, then the nodes whose distance is 2, and so on. This process continues until all nodes have been visited.

Example

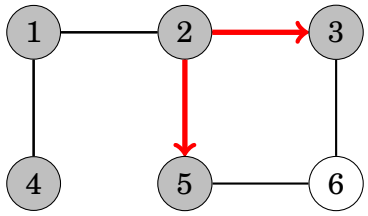
Let us consider how the algorithm processes the following graph:



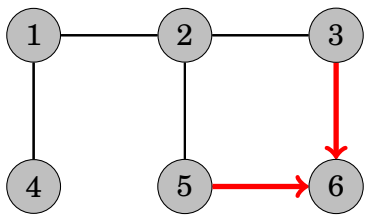
Suppose again that the search begins at node 1. First, we process all nodes that can be reached from node 1 using a single edge:



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 in 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 begins a breadth-first search at node x. The code assumes that the graph is stored as adjacency lists and maintains a queue

```
queue<int> q;
```

that contains the nodes 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.

In addition, the code uses arrays

```
int z[N], e[N];
```

so that the array z indicates which nodes the search has already visited and the array e will contain the distances to all nodes in the graph. The search can be implemented as follows:

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

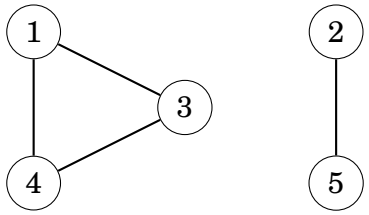
12.3 Applications

Using the graph traversal algorithms, we can check many properties of the graph. Usually, either depth-first search or bredth-first search can 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.

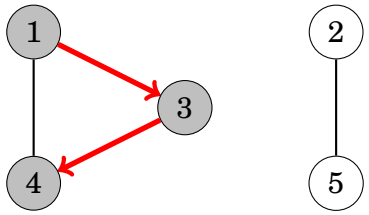
Connectivity check

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

For example, in the graph



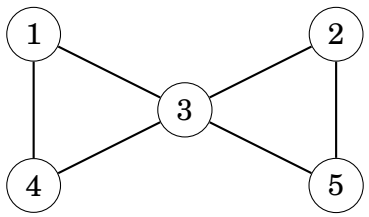
a depth-first search from node 1 visits the following nodes:



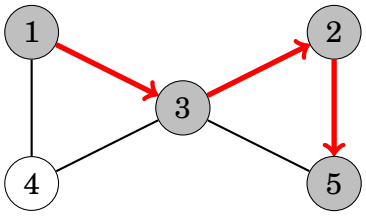
Since the search did not visit all the nodes, we can conclude that the graph is not connected. In a similar way, we can also find all connected components of a graph by iterating through the nodes and always starting a new depth-first search if the current node does not belong to any component yet.

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:



When we move from node 2 to node 5 it turns out that the neighbor 3 has already been visited. Thus, the graph contains a cycle that goes through node 3, for example, $3 \rightarrow 2 \rightarrow 5 \rightarrow 3$.

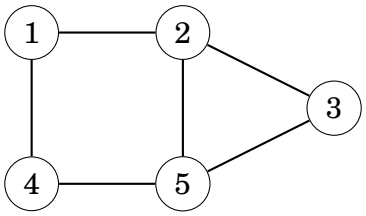
Another way to find out whether a graph contains a cycle is to simply calculate the number of nodes and edges in every component. If a component contains c nodes and no cycle, it must contain exactly $c - 1$ edges (so it has to be a tree). If there are c or more edges, the component surely contains a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

Bipartiteness check

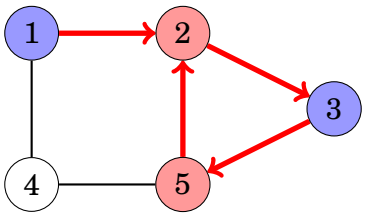
A graph is bipartite if its nodes can be colored using two colors so that there are no adjacent nodes with the same color. It is surprisingly easy to check if a graph is bipartite using graph traversal algorithms.

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 or both nodes 2 and 5 is red, while they are adjacent nodes in the graph. Thus, the graph is not bipartite.

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

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