# Chapter 16

# Directed graphs

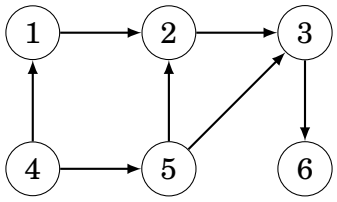
In this chapter, we focus on two classes of directed graphs:

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

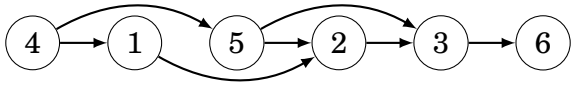
It turns out that in both cases, we can design efficient algorithms that are based on the special properties of the graphs.

## Topological sorting

A **topological sort** is an ordering of the nodes of a directed graph such that if there is a path from node $a$ to node $b$, then node $a$ appears before node $b$ in the ordering. For example, for the graph




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




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

[1] Directed acyclic graphs are sometimes called DAGs.

## Algorithm

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

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

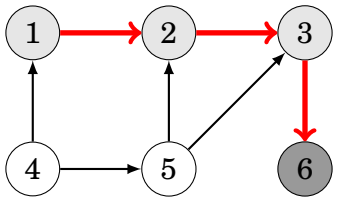
Initially, the state of each node is 0. When a search reaches a node for the first time, its state becomes 1. Finally, after all successors of the node have been processed, its state becomes 2.

If the graph contains a cycle, we will find this out during the search, because sooner or later we will arrive at a node whose state is 1. In this case, it is not possible to construct a topological sort.

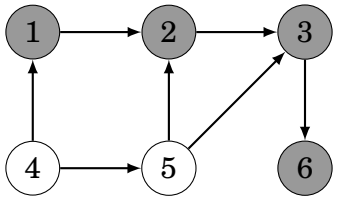
If the graph does not contain a cycle, we can construct a topological sort by adding each node to a list when the state of the node becomes 2. This list in reverse order is a topological sort.

## Example 1

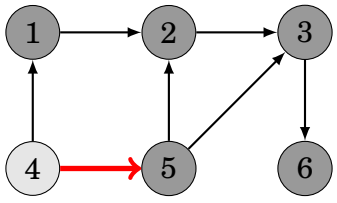
In the example graph, the search first proceeds from node 1 to node 6:



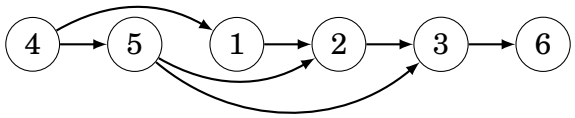

Now node 6 has been processed, so it is added to the list. After this, also nodes 3, 2 and 1 are added to the list:




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



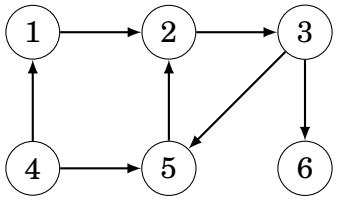
Thus, the final list is $[6,3,2,1,5,4]$. We have processed all nodes, so a topological sort has been found. The topological sort is the reverse list $[4,5,1,2,3,6]$:



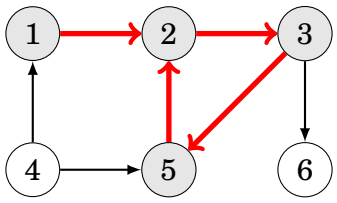

Note that a topological sort is not unique, and there can be several topological sorts for a graph.

### Example 2

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




The search proceeds as follows:




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

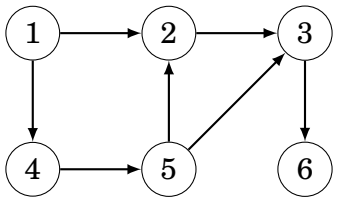
## Dynamic programming

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

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

## Counting the number of paths

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



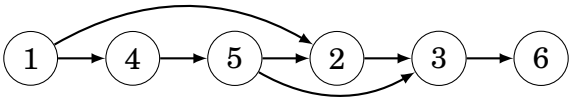

There are a total of three such paths:

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

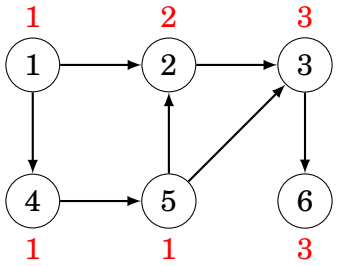
Let $\mathrm{paths}(x)$ denote the number of paths from node 1 to node $x$. As a base case, $\mathrm{paths}(1) = 1$. Then, to calculate other values of $\mathrm{paths}(x)$, we may use the recursion

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

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



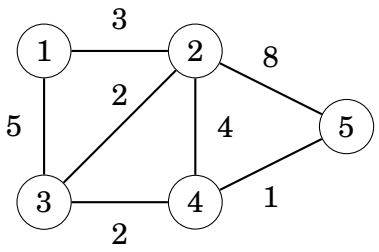

Hence, the numbers of paths are as follows:



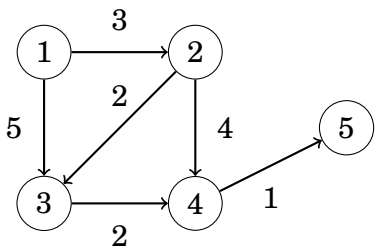

For example, to calculate the value of $\mathrm{paths}(3)$, we can use the formula $\mathrm{paths}(2) + \mathrm{paths}(5)$, because there are edges from nodes 2 and 5 to node 3. Since $\mathrm{paths}(2) = 2$ and $\mathrm{paths}(5) = 1$, we conclude that $\mathrm{paths}(3) = 3$.

## Extending Dijkstra's algorithm

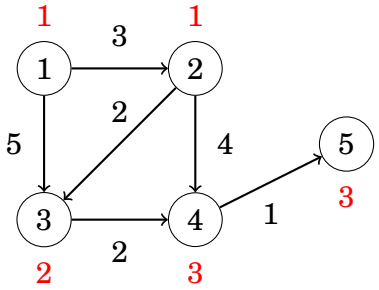
A by-product of Dijkstra's algorithm is a directed, acyclic graph that indicates for each node of the original graph the possible ways to reach the node using a shortest path from the starting node. Dynamic programming can be applied to that graph. For example, in the graph




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



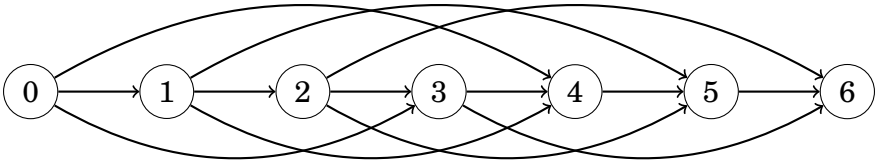

Now we can, for example, calculate the number of shortest paths from node 1 to node 5 using dynamic programming:




## Representing problems as graphs

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

As an example, consider the problem of forming a sum of money $n$ using coins $\{c_1, c_2, \ldots, c_k\}$. In this problem, we can construct a graph where each node corresponds to a sum of money, and the edges show how the coins can be chosen. For example, for coins $\{1, 3, 4\}$ and $n = 6$, the graph is as follows:




Using this representation, the shortest path from node 0 to node $n$ corresponds to a solution with the minimum number of coins, and the total number of paths from node 0 to node $n$ equals the total number of solutions.

# Successor paths

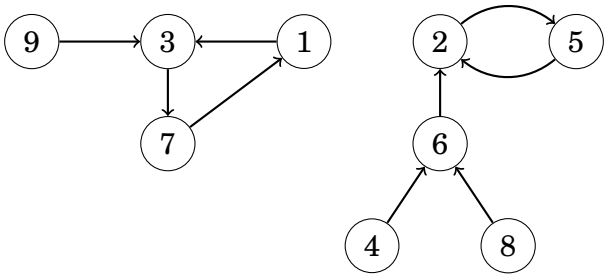
For the rest of the chapter, we will focus on **successor graphs**. In those graphs, the outdegree of each node is 1, i.e., exactly one edge starts at each node. A successor graph consists of one or more components, each of which contains one cycle and some paths that lead to it.

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

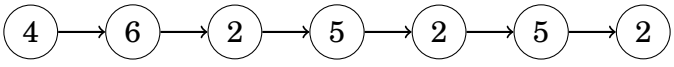
For example, the function

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

defines the following graph:




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




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

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

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

Precalculating the values takes $O(n \log u)$ time, because $O(\log u)$ values are calculated for each node. In the above graph, the first values are as follows:

| $x$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| succ$(x,1)$ | 3 | 5 | 7 | 6 | 2 | 2 | 1 | 6 | 3 |
| succ$(x,2)$ | 7 | 2 | 1 | 2 | 5 | 5 | 3 | 2 | 7 |
| succ$(x,4)$ | 3 | 2 | 7 | 2 | 5 | 5 | 1 | 2 | 3 |
| succ$(x,8)$ | 7 | 2 | 1 | 2 | 5 | 5 | 3 | 2 | 7 |
| $\cdots$ | | | | | | | | | |

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

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

For example, in the previous graph

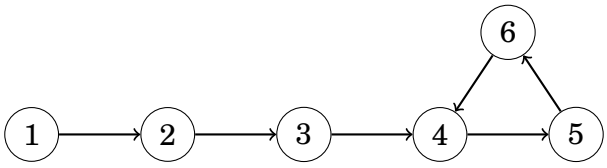
$$\text{succ}(4,11) = \text{succ}(\text{succ}(\text{succ}(4,8),2),1) = 5.$$

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

# Cycle detection

Consider a successor graph that only contains a path that ends in a cycle. We may ask the following questions: if we begin our walk at the starting node, what is the first node in the cycle and how many nodes does the cycle contain?

For example, in the graph




we begin our walk at node 1, the first node that belongs to the cycle is node 4, and the cycle consists of three nodes (4, 5 and 6).

A simple way to detect the cycle is to walk in the graph and keep track of all nodes that have been visited. Once a node is visited for the second time, we can conclude that the node is the first node in the cycle. This method works in $O(n)$ time and also uses $O(n)$ memory.

However, there are better algorithms for cycle detection. The time complexity of such algorithms is still $O(n)$, but they only use $O(1)$ memory. This is an important improvement if $n$ is large. Next we will discuss Floyd's algorithm that achieves these properties.

## Floyd's algorithm

**Floyd's algorithm**[2] walks forward in the graph using two pointers $a$ and $b$. Both pointers begin at a node $x$ that is the starting node of the graph. Then, on each turn, the pointer $a$ walks one step forward and the pointer $b$ walks two steps forward. The process continues until the pointers meet each other:

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

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

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

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

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

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