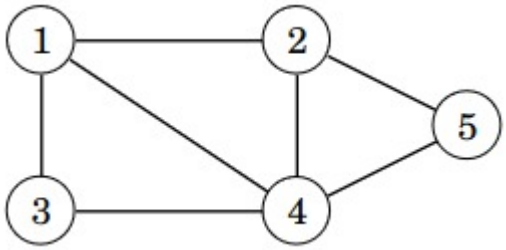
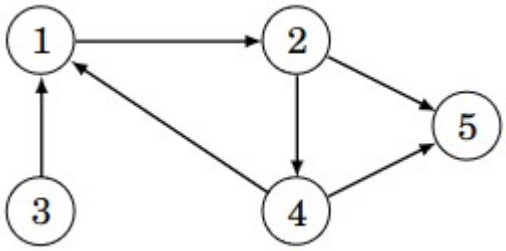
13. Directed graphs

Earlier we assumed that the edges of a graph go both directions, i.e, that the graph is *undirected*. For example, the following graphs is undirected:



In this chapter, we consider *directed* graphs, where edges go only one direction, which is indicated by arrows on edges. For example, the following graph is directed:



Dealing with directed graphs is similar to dealing with undirected graphs, but the edge directions do change some things. This chapter covers algorithms that are specifically designed for directed graphs.

Representing directed graphs

In programming, a directed graph can represented using adjacency lists in the same way as an undirected graph, but each edge is added to only one adjacency list. The following class can used for directed graphs:

```
class Graph:
    def __init__(self, nodes):
        self.nodes = nodes
        self.graph = {node: [] for node in self.nodes}

    def add_edge(self, a, b):
```

```
            self.graph[a].append(b)
```

Since the graph is directed, the method `add_edge` adds the node `b` into the adjacency list of the node `a`, but it does not add the node `a` into the adjacency list of the node `b`. Thus the adjacency list contains only those nodes that can be reached by following an edge in the specified direction.

Now the above example graph can be created as follows:

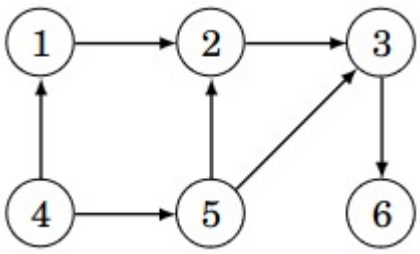
```
g = Graph([1, 2, 3, 4, 5])

g.add_edge(1, 2)
g.add_edge(2, 4)
g.add_edge(2, 5)
g.add_edge(3, 1)
g.add_edge(4, 1)
g.add_edge(4, 5)
```

Depth-first search and breadth-first search work on directed graphs too. Since they use the adjacency lists, edges are followed in one direction only during the search.

Topological ordering

A *topological ordering* of a directed graph is an ordering of the nodes that satisfies the following condition: If there is a (directed) path from a node a to a node b, then the node a is before the node b in the ordering.

Consider the following graph as an example:



One possible topological ordering for the graph is $[4, 1, 5, 2, 3, 6]$. In the following figure, the nodes are displayed in the topological order, which makes all edges to go from left to right.

A topological ordering exists only for *acyclic* graphs, which are graphs that contain no *cycles*. A cycle is a path in the graph that returns back to the node it started from. A cycle makes a topological ordering impossible, because no node of the cycle can occur in the ordering before all other nodes of the cycle.

Directed acyclic graphs are useful in many applications. The abbreviation *DAG* is sometimes used as a short hand for directed acyclic graph.

Topological sorting

A topological ordering can be computed using depth-first searches. During the computation, each node is in one of three states:

- State 0: the node has not been visited yet
- State 1: the processing of the node has started but has not finished yet
- State 2: the node has been fully processed

Initially, every node is in state 0. The algorithm iterates through all nodes, and starts a depth-first search for each node that is still in state 0. When the search reaches a node, the state of the node changes to 1. When all the edges leaving the node have been processed, the state becomes 2.

The algorithm constructs a list of all the nodes of the graph. Each node is added to the list when it reaches the state 2. At the end of the algorithm, the list is reversed to obtain a topological ordering.

If the graph contains a cycle, this is detected by the algorithm when it encounters an edge that leads to a node in state 1. Such a situation occurs only with a cycle and indicates that there is no topological ordering.

An implemenation of the algorithm in Python follows:

```
class TopologicalSort:
    def __init__(self, nodes):
        self.nodes = nodes
        self.graph = {node: [] for node in self.nodes}

    def add_edge(self, a, b):
        self.graph[a].append(b)

    def visit(self, node):
```

```
        if self.state[node] == 1:
            self.cycle = True
            return
        if self.state[node] == 2:
            return

        self.state[node] = 1
        for next_node in self.graph[node]:
            self.visit(next_node)

        self.state[node] = 2
        self.order.append(node)

    def create(self):
        self.state = {}
        for node in self.nodes:
            self.state[node] = 0

        self.order = []
        self.cycle = False

        for node in self.nodes:
            if self.state[node] == 0:
                self.visit(node)

        if self.cycle:
            return None
        else:
            self.order.reverse()
            return self.order
```

The algorithm can be used as follows:

```
t = TopologicalSort([1, 2, 3, 4, 5, 6])

t.add_edge(1, 2)
t.add_edge(2, 3)
t.add_edge(3, 6)
t.add_edge(4, 1)
t.add_edge(4, 5)
t.add_edge(5, 2)
t.add_edge(5, 3)
```

```
print(t.create()) # [4, 5, 1, 2, 3, 6]
```

Notice that the topological ordering computed by the algorithm, $[4, 5, 1, 2, 3, 6]$, is different from the earlier example topological ordering $[4, 1, 5, 2, 3, 6]$. This is because a graph can have multiple topological orders and the algorithm finds one of them.

Why does the algorithm work correctly?

The correctness of the algorithm is based on the fact that, if the graph has en edge from a node a to a node b, the node a reaches state 2 later than the node b. Thus the node a is added to the list later than the node b.

Since the list is reversed at the end, the node a is before b in the computed ordering. This ensures that the edge goes from left to right in the ordering.

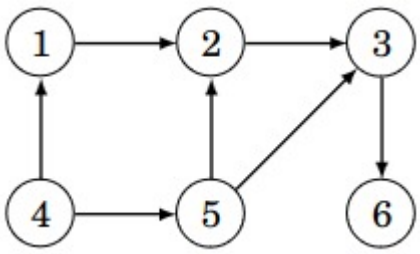
If the graph contains a cycle, at some point the algorithm reaches one of the nodes in the cycle. Following the edges out of the node will lead back to the node itself along the cycle before the node changes to state 2, which is detected by the algorithm.

Dynamic programming

Dynamic programming can be used for computing with directed acyclic graphs. For example, the following questions can be answered using dynamic programming:

- How many different paths are there from a node a to a node b?
- What is the smallest number of edges on a path from a node a to a node b?
- What is the largest number of edges on a path from a node a to a node b?

As an example, consider path counting in the following graph:



There are 3 different paths from the node 4 to the node 3 in this graph:

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

When counting the paths from the node 4 to the node 3, we can divide the problem into two subproblems:

- The first edge of the path is $4 \to 1$. In this case, we then count the paths from the node 1 to the node 3.
- The first edge of the path is $4 \to 5$. In this case, we then count the paths from the node 5 to the node 3.

There is 1 path from the node 1 to the node 3 and 2 paths from the node 5 to the node 3. Thus the number of paths from the node 4 to the node 3 is $1 + 2 = 3$.

More generally, when counting paths from a node x to a node y, we iterate through all nodes that are reacheable by an edge from the node x. Summing up all paths from such nodes to the node y gives the number of paths from the node x to the node y. When the graph has no cycles, all the subproblems can be computed efficiently using dynamic programming.

The following code implements the algorithm in Python:

```
class CountPaths:
    def __init__(self, nodes):
        self.nodes = nodes
        self.graph = {node: [] for node in self.nodes}

    def add_edge(self, a, b):
        self.graph[a].append(b)

    def count_from(self, node):
        if node in self.result:
            return self.result[node]

        path_count = 0
        for next_node in self.graph[node]:
            path_count += self.count_from(next_node)

        self.result[node] = path_count
        return path_count
```

```
    def count_paths(self, x, y):
        self.result = {y: 1}
        return self.count_from(x)
```

The algorithm can be used as follows:

```
c = CountPaths([1, 2, 3, 4, 5, 6])

c.add_edge(1, 2)
c.add_edge(2, 3)
c.add_edge(3, 6)
c.add_edge(4, 1)
c.add_edge(4, 5)
c.add_edge(5, 2)
c.add_edge(5, 3)

print(c.count_paths(4, 3)) # 3
```

Here dynamic programming is realized by storing path counts from each node to the target node into a dictionary `result`. If the path count is already stored, it is not computed again, which makes the algorithm efficient.

Problems as graphs

Dynamic programming in general can be seen as operating on a directed acyclic graph. Consider the following task, which was solved using dynamic programming in Chapter 10:

Task

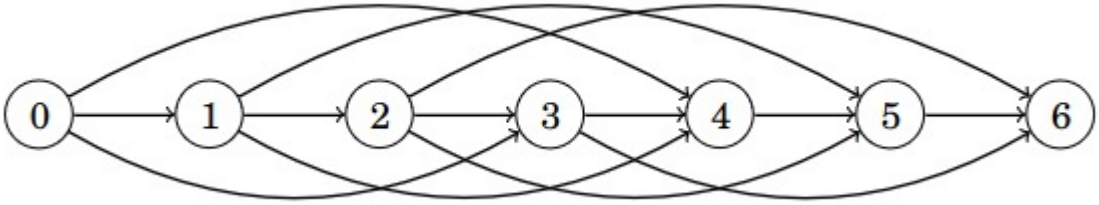
You have an unlimited number coins with values given as a list. How many ways can you choose the coins so that their sum is x?

For example, when the coins are $[1, 3, 4]$ and $x = 6$, there are 9 ways:

- $[1, 1, 1, 1, 1, 1]$
- $[1, 1, 1, 3]$
- $[1, 1, 4]$
- $[1, 1, 3, 1]$
- $[1, 3, 1, 1]$
- $[1, 4, 1]$

- $[3, 1, 1, 1]$
- $[3, 3]$
- $[4, 1, 1]$

This problem can be represented as a graph, where each node represents one of the sums $0, 1, \ldots, x$, and there is an edge from a node a to a node b if adding one coin to the sum a gives the sum b. When the coins are $[1, 3, 4]$ ja $x = 6$, the graph looks like this:

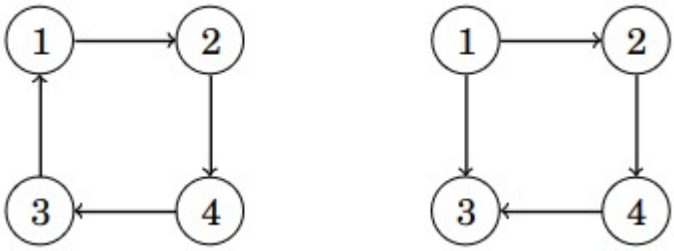


Each path from the node 0 to the node x represents one way of obtaining the sum x. Thus the number of such paths is the number of ways for choosing the coins. We could use above class `CountPaths` almost directly for solving the task.

Strong connectivity

In directed graphs, the concept of connectivity is more complicated than in undirected graphs, because two nodes are not necessarily connected by a directed path even if they are in the same component.

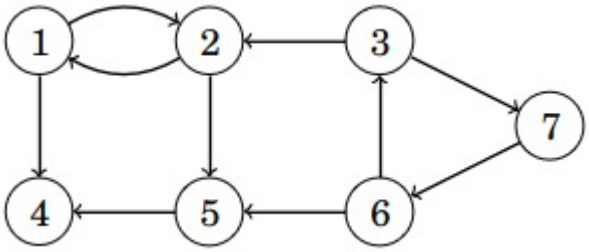
For example, consider the graphs in the following figure. In the graph on the left, there is a path from any node to any other node, but the graph on the right has no path from the node 2 to the node 1, for example.



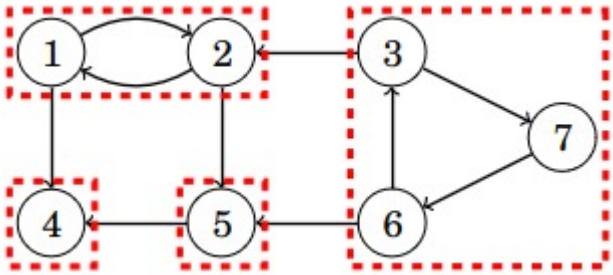
A directed graph is *strongly connected*, if there is a directed path from any node to any other node. In the above figure, the graph on the left is strongly connected but the graph on the right is not.

The *strongly connected components* of a directed graph are maximal sets on nodes such that there is a path from any node in the set to any other node in the set.

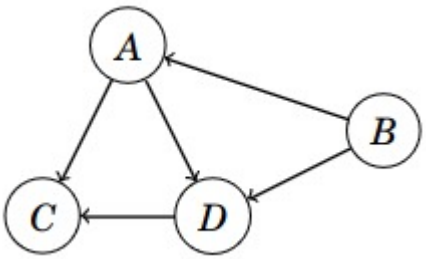
Any directed graph can be partitioned into strongly connected components. Consider the following graph as an example:



The strongly connected components of this graph are:



The following figure shows the strongly connected components as a graph, where each node corresponds to one component:



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

Such a graph of components is always acyclic and summarizes the connectivity structure of the graph, i.e., which nodes have a path between them.

Kosaraju's algorithm

The strongly connected components of a graph can be computed using Kosaraju's algorithm. The algorithm has two phases, each of which visits all nodes using depth-first search.

The first phase constructs a list of all nodes in the same way as in topological sorting. The difference is that the algorithm does not keep track of the states of the nodes and does not check for cycles.

The second phase is performed in the *transpose* graph, where the direction of

every edge has been reversed. The algorithm iterates through the node list constructed in the first phase in reverse order, and starts a depth-first search for each node that has not been visited yet. For each such depth-first search, the nodes visited during the search form a strongly connected component.

The algorithm can be implemented as follows in Python:

```
class Kosaraju:
    def __init__(self, nodes):
        self.nodes = nodes
        self.graph = {node: [] for node in self.nodes}
        self.reverse = {node: [] for node in self.nodes}

    def add_edge(self, a, b):
        self.graph[a].append(b)
        self.reverse[b].append(a)

    def visit(self, node, phase):
        if node in self.visited:
            return
        self.visited.add(node)

        if phase == 1:
            graph = self.graph
        if phase == 2:
            graph = self.reverse

        for next_node in graph[node]:
            self.visit(next_node, phase)

        if phase == 1:
            self.order.append(node)

    def count_components(self):
        self.visited = set()
        self.order = []

        for node in self.nodes:
            self.visit(node, 1)

        self.order.reverse()
        self.visited.clear()
```

```
        count = 0
        for node in self.order:
            if node not in self.visited:
                count += 1
                self.visit(node, 2)

        return count
```

The algorithm can be used as follows:

```
k = Kosaraju([1, 2, 3, 4, 5, 6, 7])

k.add_edge(1, 2)
k.add_edge(1, 4)
k.add_edge(2, 1)
k.add_edge(2, 5)
k.add_edge(3, 2)
k.add_edge(3, 7)
k.add_edge(5, 4)
k.add_edge(6, 3)
k.add_edge(6, 5)
k.add_edge(7, 6)

print(k.count_components()) # 4
```

Here the method `count_components` counts the number of strongly connected components. In the example graph, the algorithm finds 4 strongly connected components.

Why does the algorithm work correctly?

During the second phase of Kosaraju's algorithm, each depth-first search visits all nodes that are reachable from the start node and have not been visited before. How do we know that the set of visited nodes forms exactly one strongly connected component and does not contain nodes from other components?

Consider an edge $a \to b$ in the original graph that goes from one component to another component. In the first phase, the nodes in the component of the node b are added to the list before the nodes in the component of the node a. In the reversed list the order is the opposite, and thus the second phase constructs the component of the node a before the component of the node b. Since the edges

have been reversed, the edge $a \to b$ has become the edge $b \to a$, which prevents the search for the component of the node a from spilling into the component of the node b.

Since the above deduction works for any edge that goes from one component to another, no depth-first search in the second phase visits nodes in other components.