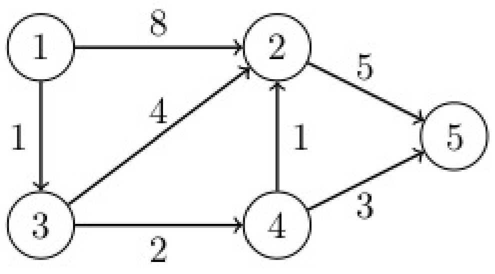
# 14. Shortest paths

The topic of this chapter is finding a shortest path in a *weighted* graph, where each edge has a weight. The length of a path in a weighted graph is the sum of the edge weights on the path.

The following figure shows an example of a weighted graph:



Each edge is labelled by its weight. In this graph, for example, the length of the path $1 \to 3 \to 2$ is $1 + 4 = 5$, because the weight of the edge $1 \to 3$ is $1$ and the weight of the edge $3 \to 2$ is $4$.

Earlier we have used breadth-first search to find shortest paths, but it does not work correctly in weighted graphs. In this chapter, we will see three algorithms for finding shortest paths in weighted graphs.

## Bellman-Ford algorithm

The Bellman-Ford algorithm computes the distance (the length of the shortest path) from a given start node to all nodes. The algorithm maintains an estimate for the distance to each node. Initially, the distance of the start node is $0$ and the distance to all other nodes is $\infty$.

The algorithm performs $n - 1$ rounds of computation, where $n$ is the number of nodes in the graph. In each round, the algorithm iterates through all edges of the graph and tries to use each edge to reduce a distance estimate. When the algorithm is processing an edge $a \to b$, it checks if the distance to $b$ through this edge is smaller than the previous distance estimate. If it is, the distance to $b$ is updated.

After $n - 1$ rounds, all distances have reached their final values and correspond to the lengths of the shortest paths.

The Bellman-Ford algorithm can be implemented as follows:

```python
class BellmanFord:
    def __init__(self, nodes):
        self.nodes = nodes
        self.edges = []

    def add_edge(self, node_a, node_b, weight):
        self.edges.append((node_a, node_b, weight))

    def find_distances(self, start_node):
        distances = {}
        for node in self.nodes:
            distances[node] = float("inf")
        distances[start_node] = 0

        num_rounds = len(self.nodes) - 1
        for _ in range(num_rounds):
            for edge in self.edges:
                node_a, node_b, weight = edge
                new_distance = distances[node_a] + weight
                if new_distance < distances[node_b]:
                    distances[node_b] = new_distance

        return distances
```

The list `nodes` contains the nodes of the graph and the list `edges` contains the edges of the graph. The dictionary `distances` contains the distance estimates from the node `start_node` to all nodes. In each round, the algorithm goes through the list `edges` and tries to use the edges to reduce the distances. If a new smaller distance is found, the algorithm updates the distance in the dictionary `distances`.

The following code uses the algorithm to compute the distances from the node $1$:

```python
b = BellmanFord([1, 2, 3, 4, 5])

b.add_edge(1, 2, 8)
b.add_edge(1, 3, 1)
b.add_edge(2, 5, 5)
```

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

distances = b.find_distances(1)
print(distances) # {1: 0, 2: 4, 3: 1, 4: 3, 5: 6}
```

In this case, the distances and the corresponding shortest paths are:

| Target node | Distance | Path |
|---|---|---|
| $1$ | $0$ | $1$ |
| $2$ | $4$ | $1 \to 3 \to 4 \to 2$ |
| $3$ | $1$ | $1 \to 3$ |
| $4$ | $3$ | $1 \to 3 \to 4$ |
| $5$ | $6$ | $1 \to 3 \to 4 \to 5$ |

## Examining the algorithm

To obtain a better understanding of the operation of the algorithm, we can make the algorithm to print out each update of a distance estimate:

```
                if new_distance < distances[node_b]:
                    print("update", node_a, node_b, new_distan
                    distances[node_b] = new_distance
```

The output of the modified algorithm for the example graph looks like this:

```
update 1 2 8
update 1 3 1
update 2 5 13
update 3 2 5
update 3 4 3
update 4 2 4
update 4 5 6
```

Each line shows one update to a distance. On the first line, the distance of the node $2$ is updated to $8$. The update uses the edge $1 \to 2$ of weight $8$. Since the distance to the node $1$ is $0$, the new distance to the node $2$ is $8$, which is smaller than the old distance $\infty$.

The output demonstrates how the distance to a node can change multiple times during the algorithm. For example, the distance to the node $2$ is first updated to $8$ through the edge $1 \to 2$, then to $5$ through the edge $3 \to 2$, and finally to $4$ through the edge $4 \to 2$.

## Algorithm analysis

After the first round, the Bellman-Ford algorithm has found all shortest paths consisting of a single edge. After the second round, it has found all shortest paths consisting of at most two edges. And in general, after $k$ rounds, the algorithm has found all shortest paths consisting of at most $k$ edges.

When the graph has $n$ nodes, every shortest path has at most $n-1$ edges. This is because, if a path has $n$ or more edges, it visits some node multiple times. Such a path cannot be a shortest path, because the section of the path between the first visit to a node and the last visit to a node is unnecessary. Thus the Bellman-Ford algorithm needs at most $n-1$ rounds to find all shortest paths.

Since the algorithm performs $n-1$ rounds and each round processes $m$ edges, the time complexity of the algorithm is $O(nm)$.

## Negative cycles

The Bellman-Ford algorithm does not produce a sensible result if the graph has a negative cycle. A negative cycle is a cycle, where the sum of the edge weights is negative. By going around the negative cycle repeatedly, the length of a path can be reduced without bound, and thus the concept of shortest path is not well defined.

The following example shows a situation with a negative cycle:

```
b = BellmanFord([1, 2, 3, 4])

b.add_edge(1, 2, 1)
b.add_edge(2, 3, 1)
b.add_edge(3, 2, -2)
b.add_edge(2, 4, 1)
```

```
distances = b.find_distances(1)
print(distances) # {1: 0, 2: -2, 3: 0, 4: -1}
```

The graph has an edge $2 \to 3$ of weight $1$ and an edge $3 \to 2$ of weight $-2$. Each round through the cycle reduces the length of a path by one. Thus the path length to the nodes $2$ and $3$ as well as to the node $4$ can be made arbitrarily small, and the distances produced by the algorithm are meaningless.

However, the Bellman-Ford algorithm can be used for detecting a negative cycle by performing one more round after the $n-1$ rounds. If the graph has a negative cycle that is reachable from the start node, some distances still change in the extra round, which can be detected by the algorithm.

## Dijkstra’s algorithm

Like the Bellman-Ford algorithm, Dijkstra’s algorithm finds the shortest paths from a given start node to all nodes. Dijkstra’s algorithm is significantly more efficient than the Bellman-Ford algorithm for large graphs. Dijkstra’s algorithm cannot be used if the graph has negative edge weights (even if there are no negative cycles).

The starting point in Dijkstra’s algorithm is the same as in the Bellman-Ford algorithm: Each node has a distance from the start node, initially $0$ for the start node and $\infty$ for all other nodes. Then the algorithm updates the distances until they reach their final value.

In each step, the algorithm selects the node with the smallest distance among nodes that have not been selected before. At this point, the distance to the selected node has reached its final value and does not change anymore. The algorithm goes through the edges leaving the node and uses them to reduce the distances of other nodes. Then the node is marked visited and will not be processed again.

When the algorithm has processed all nodes, the distance of each node is the length of the shortest path from the start node to that node.

Dijkstra’s algorithm can be implemented as follows:

```
import heapq

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

    def add_edge(self, node_a, node_b, weight):
        self.graph[node_a].append((node_b, weight))

    def find_distances(self, start_node):
        distances = {}
        for node in self.nodes:
            distances[node] = float("inf")
        distances[start_node] = 0

        queue = []
        heapq.heappush(queue, (0, start_node))

        visited = set()
        while queue:
            node_a = heapq.heappop(queue)[1]
            if node_a in visited:
                continue
            visited.add(node_a)

            for node_b, weight in self.graph[node_a]:
                new_distance = distances[node_a] + weight
                if new_distance < distances[node_b]:
                    distances[node_b] = new_distance
                    new_pair = (new_distance, node_b)
                    heapq.heappush(queue, new_pair)

        return distances
```

The graph is implemented using adjacency lists so that each element of an adjacency list is a pair containing the target node and the weight. The dictionary `distances` is initialized in the same way as in the Bellman-Ford algorithm.

The algorithm needs to repeatedly find the unvisited node with the smallest distance. To do this efficiently, the algorithm uses a heap `queue`, where each element is a pair containing the distance and the identifier of a node. The element with the smallest distance is extracted from the heap, and if the corresponding node has not been marked visited, the algorithm goes through the edges on the adjacency list of the node. If an edge causes an update of the distance to a node, that node is added to the heap.

Notice that a node can occur multiple times on the heap with different distances. This is not a problem, since each node is processed only once when it is selected from the heap for the first time.

The algorithm can be used as follows:

```
d = Dijkstra([1, 2, 3, 4, 5])

d.add_edge(1, 2, 8)
d.add_edge(1, 3, 1)
d.add_edge(2, 5, 5)
d.add_edge(3, 2, 4)
d.add_edge(3, 4, 2)
d.add_edge(4, 2, 1)
d.add_edge(4, 5, 3)

distances = d.find_distances(1)
print(distances) # {1: 0, 2: 4, 3: 1, 4: 3, 5: 6}
```

## Algorithm analysis

In each step, Dijkstra's algorithm selects the node with the smallest distance. The distance to the selected node should not change later, since the node will not be processed again.

The algorithm relies on the assumption that there are no negative edge weights. When a node with the smallest distance is selected, all the remaining distances on the heap are greater or equal. Thus no later distance update can produce a smaller distance, and in particular, the distance to the selected node will not change any more.

The time complexity of the algorithm is $O(n + m \log m)$. The time complexity $O(n)$ comes from iterating through the nodes. The time complexity $O(m \log m)$ comes from the fact that at most one element is added to the heap for each edge. Thus there are $O(m)$ heap operations each needing $O(\log m)$ time.

## Negative edges

The following code demonstrates how Dijkstra's algorithm can produce an incorrect result if the graph has a negative edge:

```
d = Dijkstra([1, 2, 3, 4])

d.add_edge(1, 2, 3)
d.add_edge(2, 3, -4)
d.add_edge(1, 3, 1)
d.add_edge(3, 4, 1)

distances = d.find_distances(1)
print(distances) # [0, 3, -1, 2]
```

Here the shortest path from the node $1$ to the node $4$ is $1 \to 2 \to 3 \to 4$ and has length $3 - 4 + 1 = 0$. However, Dijkstra's algorithm chooses the path $1 \to 3 \to 4$ of length $2$. When Dijkstra's algorithm selects the node $3$, its distance is $1$, which causes the distance to the node $4$ to be updated to $2$. Later, the distance to the node $3$ is updated to $-1$, thanks to the negative edge, but because the node has already been marked visited, it is not processed again and the distance to the node $4$ does not get updated.

## Constructing shortest paths

So far we have used the Bellman-Ford algorithm and Dijsktra's algorithm to compute distances, but they can also be modified to construct a shortest path from a start node to an end node. This is achieved by augmenting each distance with the edge that caused the update to its current value. Then the shortest path to each node can be computed by walking the path backwards.

As an example, let us modify the implementation of the Bellman-Ford algorithm so that it returns a shortest path instead of the distances:

```
    def shortest_path(self, start_node, end_node):
        distances = {}
        for node in self.nodes:
            distances[node] = float("inf")
        distances[start_node] = 0
        previous = {}
        previous[start_node] = None

        for _ in range(len(self.nodes) - 1):
            for edge in self.edges:
                node_a, node_b, weight = edge
                new_distance = distances[node_a] + weight
                if new_distance < distances[node_b]:
                    distances[node_b] = new_distance
                    previous[node_b] = node_a

        if distances[end_node] == float("inf"):
            return None

        path = []
        node = end_node
        while node:
            path.append(node)
            node = previous[node]

        path.reverse()
        return path
```

The method uses a dictionary `previous` that stores for each node the preceding node on the shortest path. At the end, the method uses this dictionary to walk the shortest path backwards. The nodes on the path are collected to a list `path`, which is then reversed before returning it.

The following code demonstrates the use of the method:

```
b = BellmanFord([1, 2, 3, 4, 5])

b.add_edge(1, 2, 8)
b.add_edge(1, 3, 1)
b.add_edge(2, 5, 5)
b.add_edge(3, 2, 4)
b.add_edge(3, 4, 2)
b.add_edge(4, 2, 1)
b.add_edge(4, 5, 3)

path = b.shortest_path(1, 5)
print(path) # [1, 3, 4, 5]
```

Here the method finds a shortest path from the node $1$ to the node $5$. The method produces the path $[1, 3, 4, 5]$ of length $6$.

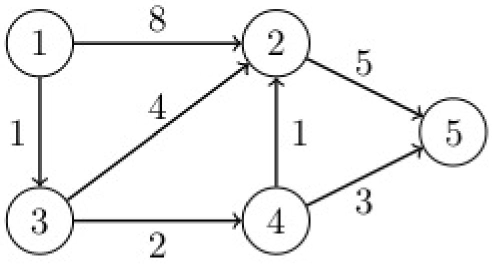
Dijkstra’s algorithm too can be modified similarly to compute a shortest path from a start node to an end node.

# Floyd-Warshall algorithm

The Floyd-Warshall algorithm finds the distances between all pairs of nodes at the same time rather than just the distances from a single start node. The algorithm works correctly for any graph as long as there are no negative cycles.

The algorithm represents the graph using an *adjacency matrix*, where the element on row $a$ and column $b$ stores the weight of the edge from the node $a$ to the node $b$. If $a = b$, the weight is $0$, and if there is no edge from the node $a$ to the node $b$, the weight is $\infty$.

Consider the following graph as an example:



The graph can be reprented as the following adjacency matrix:

| | $1$ | $2$ | $3$ | $4$ | $5$ |
|---|---|---|---|---|---|
| $1$ | $0$ | $8$ | $1$ | $\infty$ | $\infty$ |
| $2$ | $\infty$ | $0$ | $\infty$ | $\infty$ | $5$ |
| $3$ | $\infty$ | $4$ | $0$ | $2$ | $\infty$ |
| $4$ | $\infty$ | $1$ | $\infty$ | $0$ | $3$ |
| $5$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $0$ |

For example, the entry on row $3$ and column $2$ is $4$, since in the graph the edge from the node $3$ to the node $2$ has weight $4$.

The Floyd-Warshall algorithm constructs a distance matrix, where the element on row $a$ and column $b$ stores the length of the shortest path from the node $a$ to the node $b$. Initially, the distance matrix is equal to the adjacency matrix, which corresponds to the lengths of the shortest paths with at most one edge.

The main part of the algorithm consists of three nested loops. The first loop chooses a middle node $k$, and the other two loops choose a start node and an end node. Then the algorithm checks if the distance from the start node to the end node can be shortened using a path that goes through the middle node $k$.

The algorithm can be implemented as follows:

```python
class FloydWarshall:
    def __init__(self, nodes):
        self.nodes = nodes
        self.graph = {}
        for a in self.nodes:
            for b in self.nodes:
                distance = 0 if a == b else float("inf")
                self.graph[(a, b)] = distance

    def add_edge(self, a, b, w):
        self.graph[(a, b)] = min(self.graph[(a, b)], w)

    def find_distances(self):
        distances = self.graph.copy()

        for k in self.nodes:
            for a in self.nodes:
                for b in self.nodes:
                    distance = min(distances[(a, b)],
                                   distances[(a, k)] +
                                   distances[(k, b)])
                    distances[(a, b)] = distance

        return distances
```

The dictionary `graph` stores the adjacency matrix. Initially, the distance from a node $a$ to a node $b$ is $0$, if $a = b$, and $\infty$ otherwise. When an edge is added to the graph, the weight of the edge is added to the dictionary. If the same edge is added multiple times, the smallest edge weight remains as the distance.

The method `find_distances` constructs the distance matrix using the adjacency matrix. The three nested loops iterate through all combinations of three nodes. For each three nodes $k$, $a$ and $b$, the method checks if going from $a$ to $k$ to $b$ is shorter than going directly from $a$ to $b$ using the current distances. If it is, the distance from the node $a$ to the node $b$ is updated.

The following code test the algorithm:

```
f = FloydWarshall([1, 2, 3, 4, 5])

f.add_edge(1, 2, 8)
f.add_edge(1, 3, 1)
f.add_edge(2, 5, 5)
f.add_edge(3, 2, 4)
f.add_edge(3, 4, 2)
f.add_edge(4, 2, 1)
f.add_edge(4, 5, 3)

distances = f.find_distances()

print(distances[(1, 4)]) # 3
print(distances[(2, 1)]) # inf
print(distances[(3, 5)]) # 5
```

This means that the distance from the node $1$ to the node $4$ is $3$, there is no path from the node $2$ to the node $1$, and the distance from the node $3$ to the node $5$ is $5$.

The full distance matrix for the example looks like this:

| | $1$ | $2$ | $3$ | $4$ | $5$ |
|---|---|---|---|---|---|
| $1$ | $0$ | $4$ | $1$ | $3$ | $6$ |
| $2$ | $\infty$ | $0$ | $\infty$ | $\infty$ | $5$ |
| $3$ | $\infty$ | $3$ | $0$ | $2$ | $5$ |
| $4$ | $\infty$ | $1$ | $\infty$ | $0$ | $3$ |
| $5$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $0$ |

## Algorithm analysis

When the algorithm processes the middle node $k$, it finds the shortest paths, where the node $k$ occurs as an intermediate node and where all the other intermediate nodes are in the range $1 \dots k-1$. Since $k$ goes through all values $1, 2, \dots, n$, the distances at the end are the lengths of the shortest of all paths.

The time complexity of the algorithm is $O(n^3)$ because of the three nested loops, each of which goes through all nodes of the graph.

## Choice of algorithm

Dijkstra’s algorithm is usually a good choice for finding shortest paths. It is efficient, and one can usually assume that the graph has no negative edges. For example, if the edge weights represent road distances or connection prices, the values typically cannot be negative.

The Bellman-Ford algorithm works even if the graph has negative edges, as long as there are no negative cycles. The drawback is that the algorithm is slow for large graphs. The Floyd-Warshall algorithm is useful when the distances are needed for all pairs of nodes.