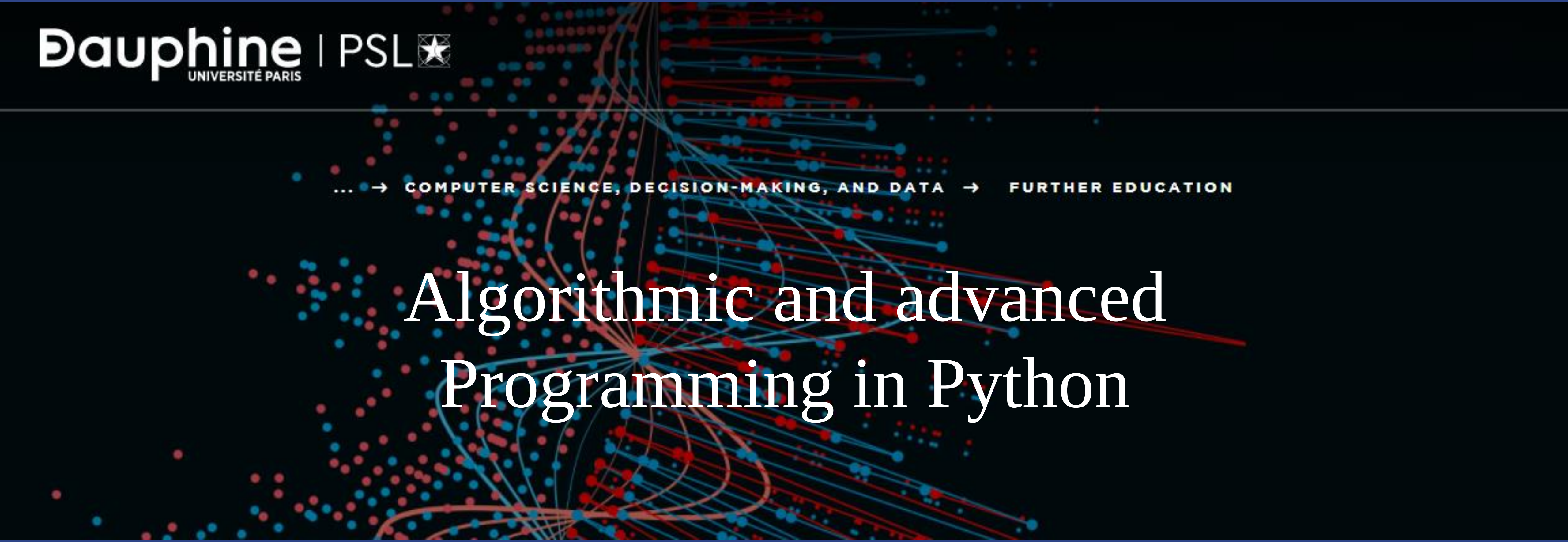

... → COMPUTER SCIENCE, DECISION-MAKING, AND DATA → FURTHER EDUCATION

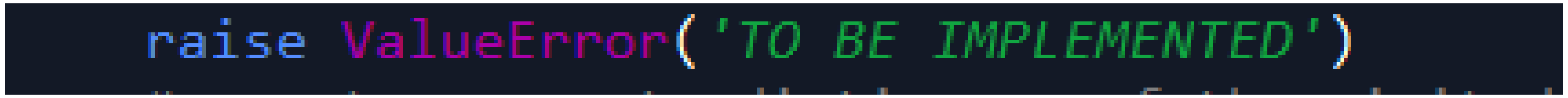
Algorithmic and advanced Programming in Python

Remy Belmonte remy.belmonte@dauphine.eu

Lab 6

Problem 1: Implement DFS

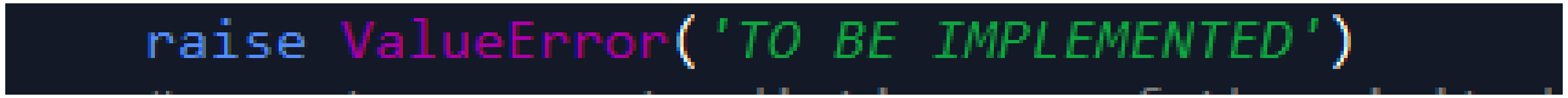
- Upload corresponding code and fill the part of the code with



```
    raise ValueError('TO BE IMPLEMENTED')
```

Problem 2: Implement BFS

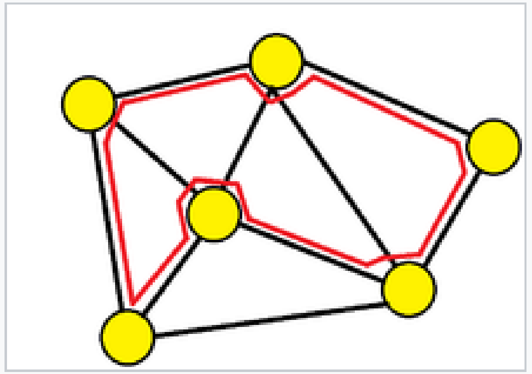
- Upload corresponding code and fill the part of the code with



```
    raise ValueError('TO BE IMPLEMENTED')
```

Problem 3: Hamiltonian path

- In the mathematical field of graph theory, a Hamiltonian path (or traceable path) is a path in an undirected or directed graph that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian path that is a cycle. Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem, which is NP-complete.

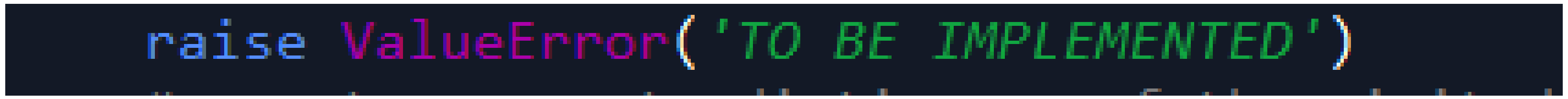


A Hamiltonian cycle around a network of six vertices

Problem 3: Hamiltonian path

- A bit of history: Hamiltonian paths and cycles are named after William Rowan Hamilton who invented the icosian game, now also known as Hamilton's puzzle, which involves finding a Hamiltonian cycle in the edge graph of the dodecahedron. Hamilton solved this problem using the icosian calculus, an algebraic structure based on roots of unity with many similarities to the quaternions (also invented by Hamilton).

- Upload corresponding code and fill the part of the code with



```
    raise ValueError('TO BE IMPLEMENTED')
```

Problem 4: Dystra

- What do GPS navigation devices and websites for booking flights have in common? As it turns out, a lot! For one, both technologies employ Dijkstra’s shortest path algorithm.

What Is Dijkstra’s Algorithm?

- A bit of history: In 1956, Dutch programmer Edsger W. Dijkstra had a practical question. He wanted to figure out the shortest way to travel from Rotterdam to Groningen. But he did not simply consult a map to calculate the distances of the roads he would need to take. Instead, Dijkstra took a computer scientist’s approach: he abstracted from the problem by filtering out the specifics such as traveling from city A to city B. This allowed him to discover the more general problem of graph search. Thus, Dijkstra’s algorithm was born.

A powerful algorithm

- Dijkstra’s algorithm is a popular search algorithm used to determine the shortest path between two nodes in a graph. In the original scenario, the graph represented the Netherlands, the graph’s nodes represented different Dutch cities, and the edges represented the roads between the cities.

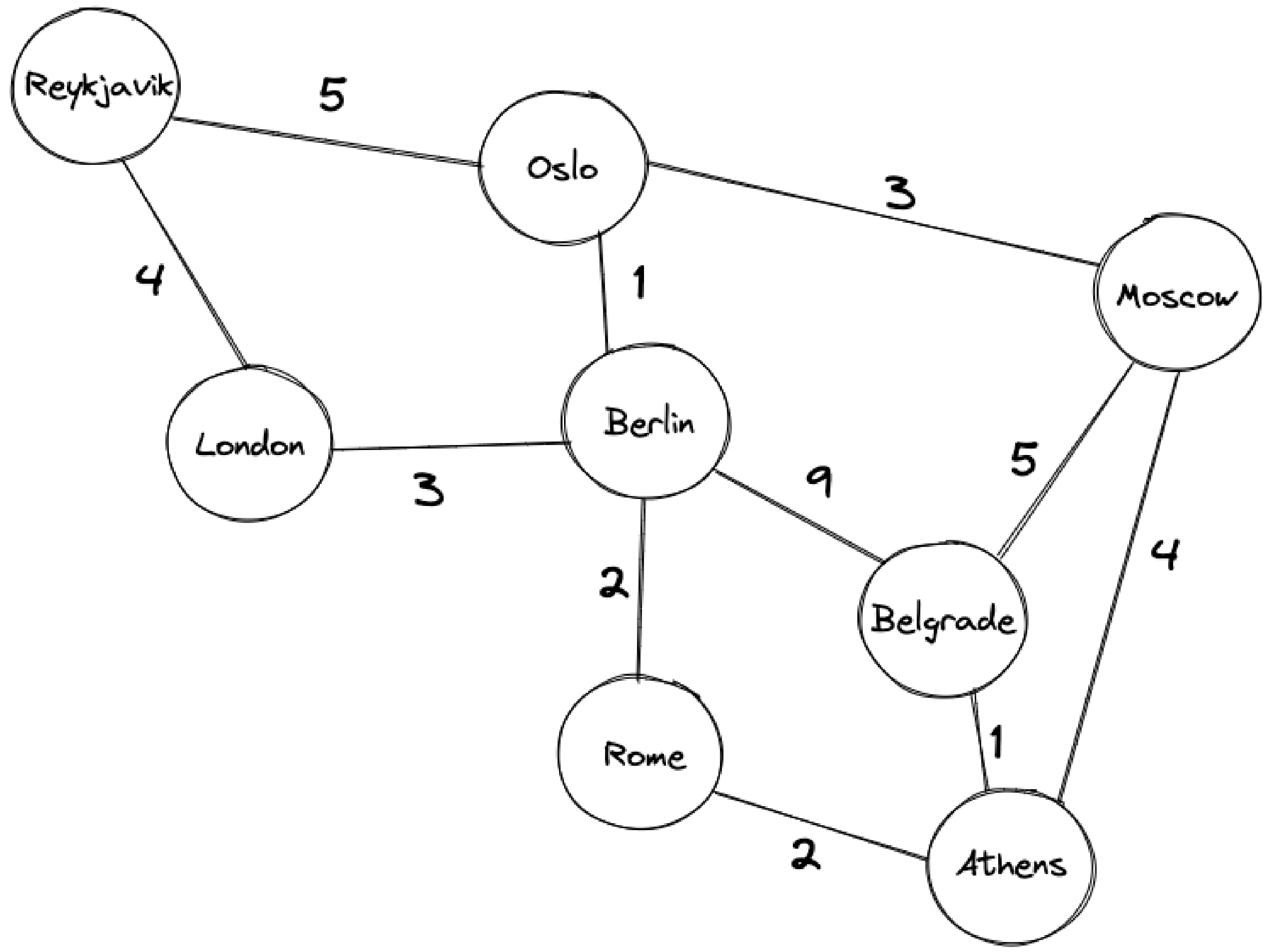
- You can apply Dijkstra’s algorithm to any problem that can be represented as a graph. Friend suggestions on social media, routing packets over the internet, or finding a way through a maze—the algorithm can do it all. But how does it actually work?

Dijkstra’s Algorithm: Problem Setting

- Recall that Dijkstra’s algorithm operates on graphs, meaning that it can address a problem only if it can be represented in a graph-like structure. The example we’ll use throughout this tutorial is perhaps the most intuitive: the shortest path between two cities.

- We’ll be working with the map below to figure out the best route between the two European cities of Reykjavik and Belgrade. For the sake of simplicity, let’s imagine that all cities are connected by roads (a real-life route would involve at least one ferry).

To keep it simple!



Some color!

- Note the following:
  - Each city is represented as a node.
  - Each road is represented as an edge.
  - Each road has an associated value. A value could be the distance between cities, a highway toll, or the amount of traffic. Generally, we'll favor edges with lower values. In our specific case, the associated value is defined by the distance between two cities.
- You also may have noticed that we cannot reach Belgrade from Reykjavik directly; that would render our exercise pointless. But there are several paths from Reykjavik to Belgrade that go through other cities:

  - Reykjavik –> Oslo –> Berlin –> Belgrade
  - Reykjavik –> London –> Berlin –> Rome –> Athens –> Belgrade
  - Reykjavik –> London –> Berlin –> Rome –> Athens –> Moscow –> Belgrade

- Each of these paths end in Belgrade, but they all have different values. We can use Dijkstra's algorithm to find the path with the lowest total value.

Dijkstra’s Algorithm: Step-by-Step

- Before diving into the code, let’s start with a high-level illustration of Dijkstra’s algorithm. First, we initialize the algorithm as follows:

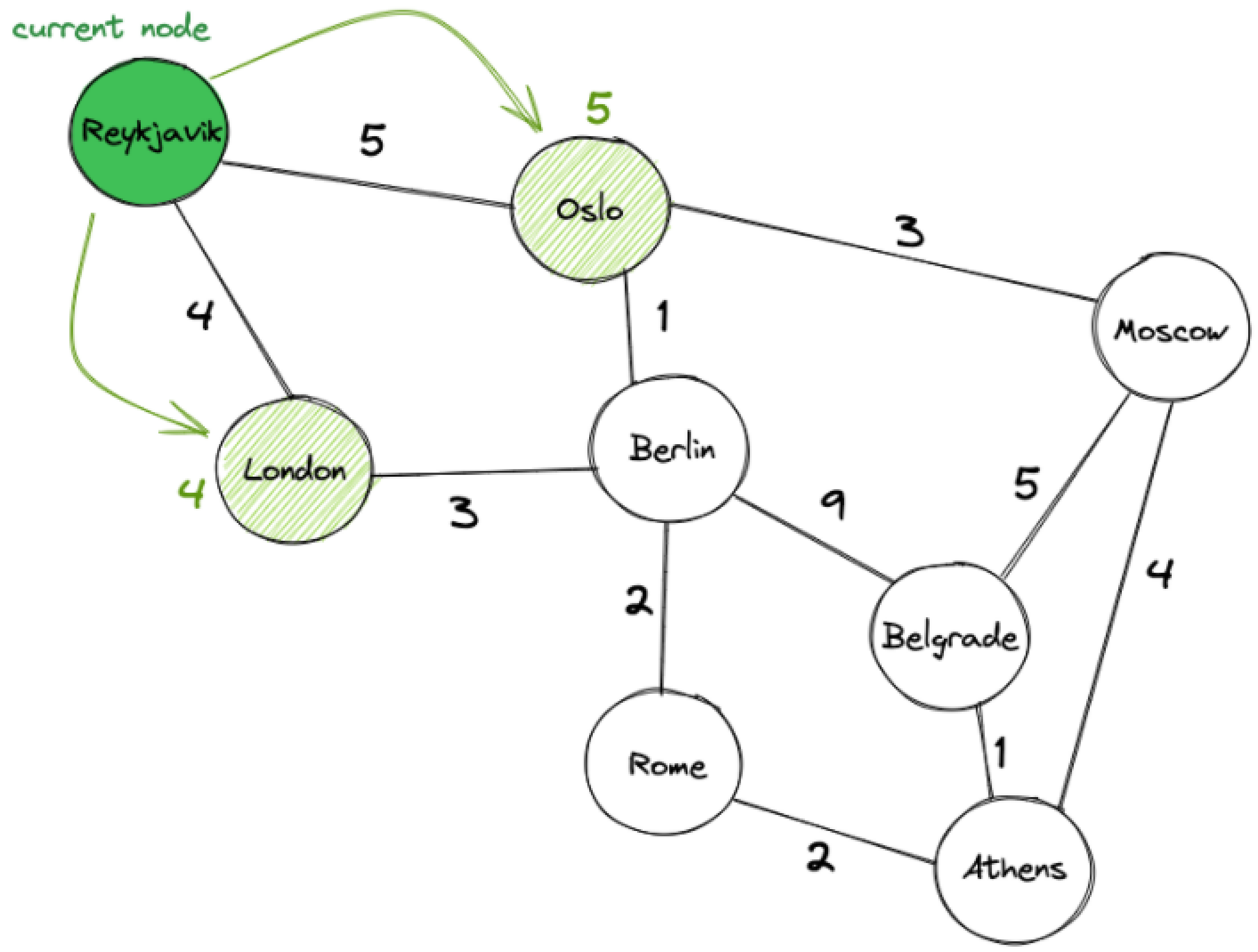
1. We set Reykjavik as the starting node.
2. We set the distances between Reykjavik and all other cities to infinity, except for the distance between Reykjavik and itself, which we set to 0.

- After that, we iteratively execute the following steps:

1. We choose the node with the smallest value as the “current node” and visit all of its neighboring nodes. As we visit each neighbor, we update their tentative distance from the starting node.
2. Once we visit all of the current node’s neighbors and update their distances, we mark the current node as “visited.” Marking a node as “visited” means that we’ve arrived at its final cost.
3. We go back to step one. The algorithm loops until it visits all the nodes in the graph.

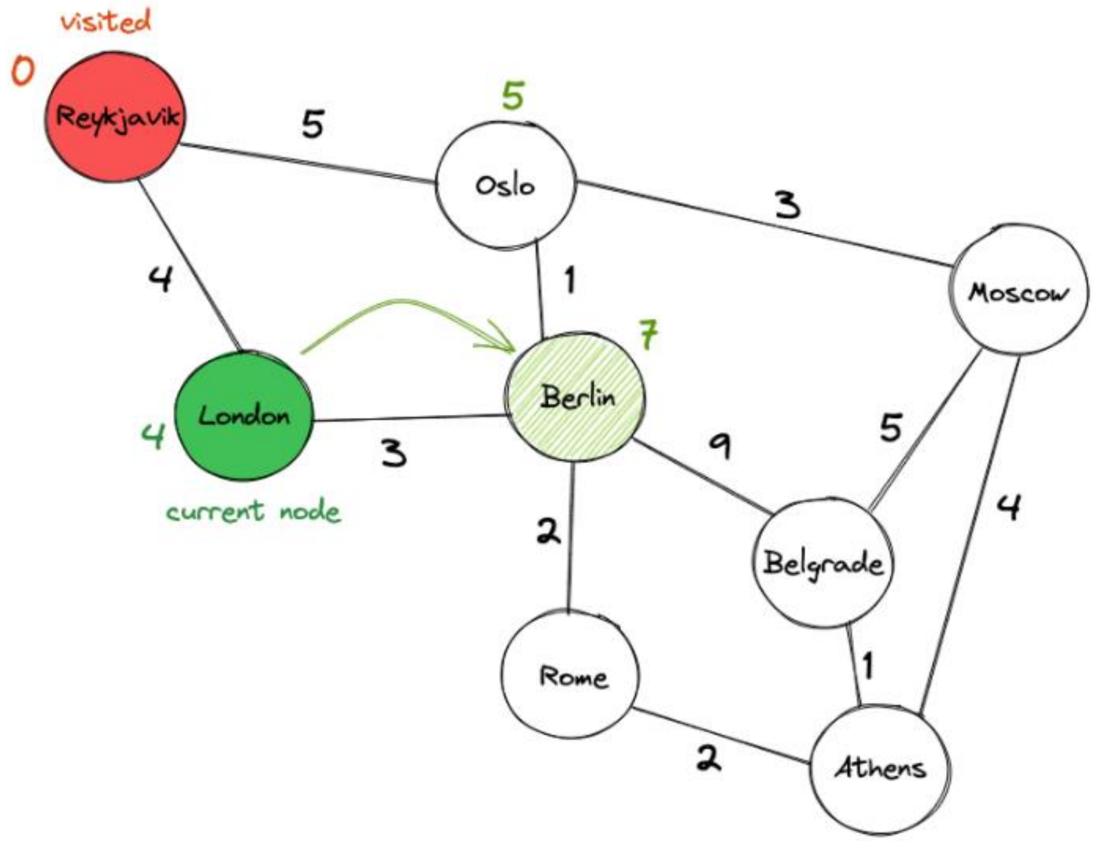
- In our example, we start by marking Reykjavik as the “current node” since its value is 0. We proceed by visiting Reykjavik’s two neighboring nodes: London and Oslo. At the beginning of the algorithm, their values are set to infinity, but as we visit the nodes, we update the value for London to 4, and Oslo to 5.

On our example:



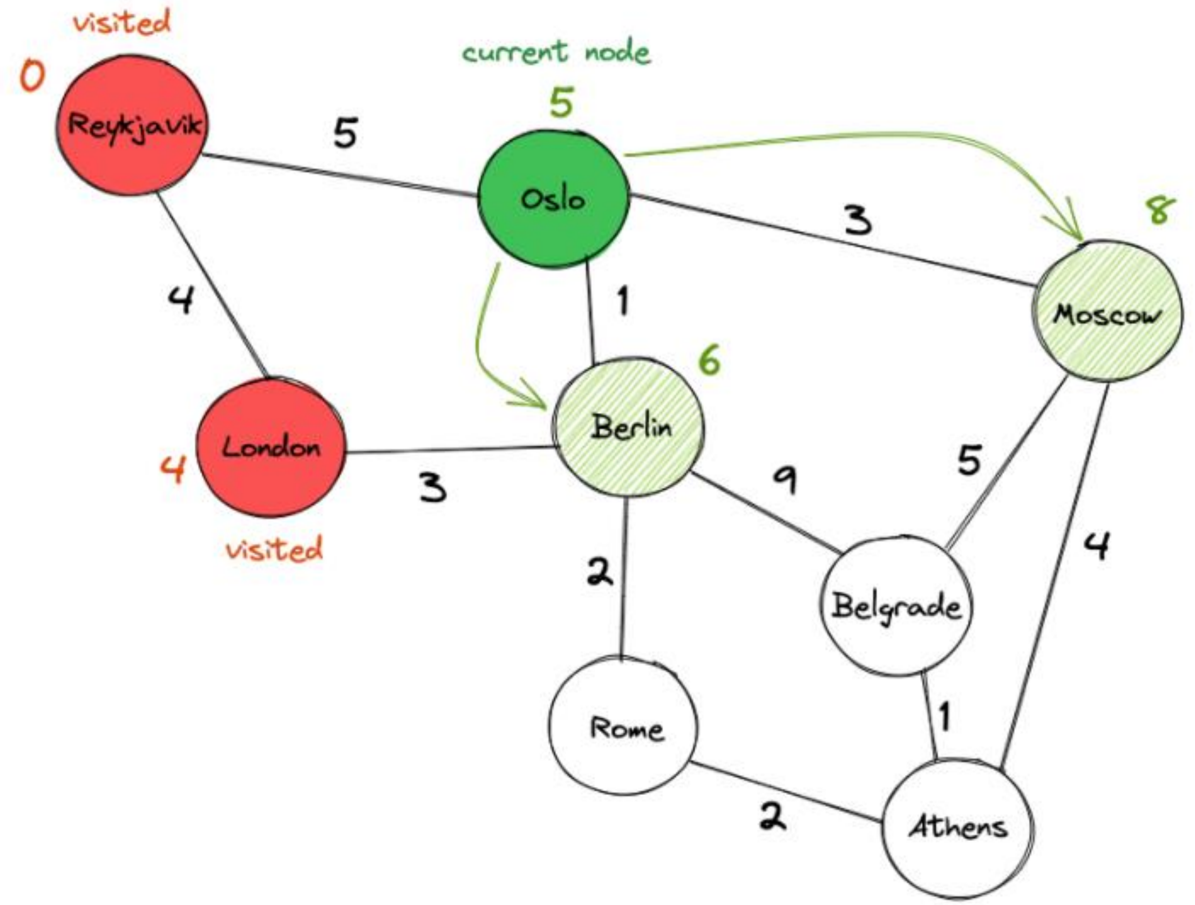
Dystra follows

- We then mark Reykjavik as "visited." We know that its final cost is zero, and we don't need to visit it again. We continue with the next node with the lowest value, which is London.
- We visit all of London's neighboring nodes which we haven't marked as "visited." London's neighbors are Reykjavik and Berlin, but we ignore Reykjavik because we've already visited it. Instead, we update Berlin's value by adding the value of the edge connecting London and Berlin (3) to the value of London (4), which gives us a value of 7.



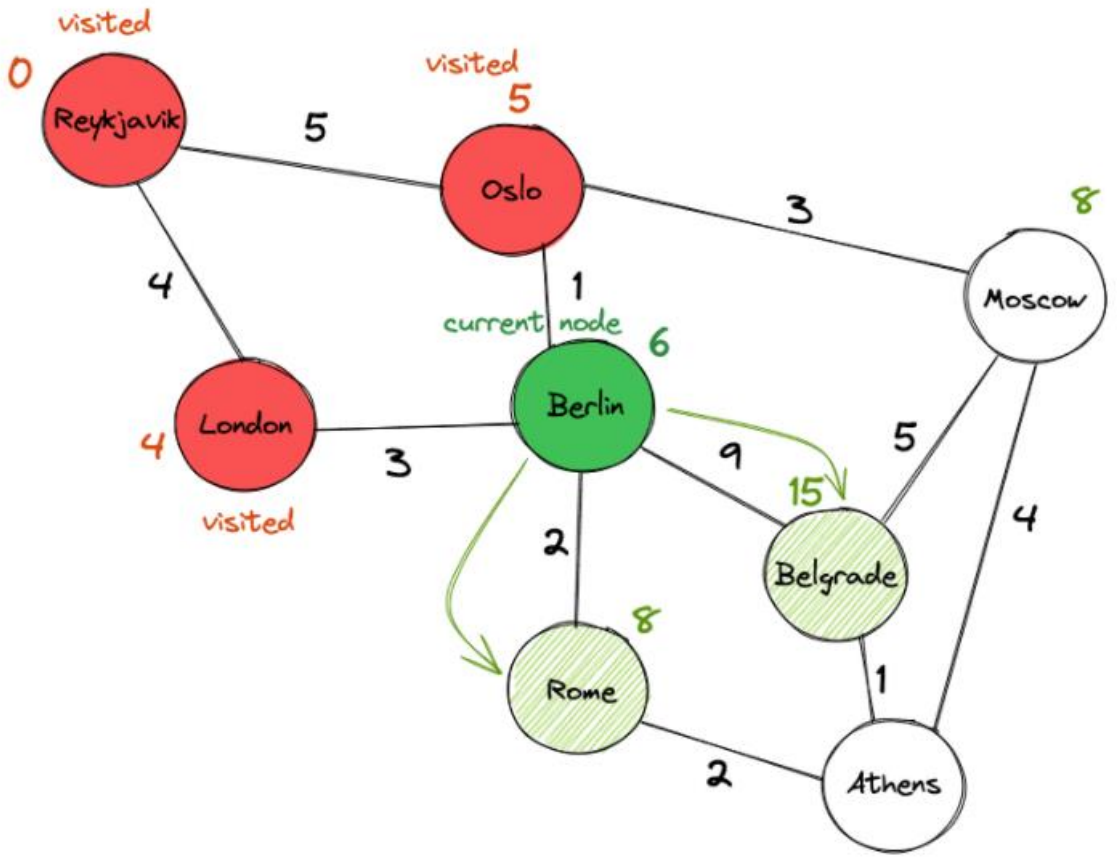
To continue

- We mark London as visited and choose the next node: Oslo. We visit Oslo’s neighbors and update their values. It turns out that we can better reach Berlin through Oslo (with a value of 6) than through London, so we update its value accordingly. We also update the current value of Moscow from infinity to 8.



To continue

- We mark Oslo as “visited” and update its final value to 5. Between Berlin and Moscow, we choose Berlin as the next node because its value (6) is lower than Moscow’s (8). We proceed as before: We visit Rome and Belgrade and update their tentative values, before marking Berlin as “visited” and moving on to the next city.

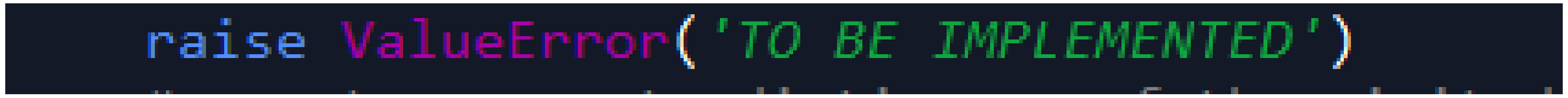


Finally

- Note that we've already found a path from Reykjavik to Belgrade with a value of 15! But is it the best one?

- Ultimately, it's not. We'll skip the rest of the steps, but you get the drill. The best path turns out to be Reykjavik –> Oslo –> Berlin –> Rome –> Athens –> Belgrade, with a value of 11.

- Upload corresponding code and fill the part of the code with



```
    raise ValueError('TO BE IMPLEMENTED')
```

Problem 6- Kruskal’s algorithm

- Kruskal's algorithm is a minimum spanning tree algorithm that uses the *greedy* approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties. Kruskal’s Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree.
- The algorithm starts with V different trees (V is the vertices in the graph). While constructing the minimum spanning tree, every time Kruskal’s algorithm selects an edge that has minimum weight and then adds that edge if it doesn’t create a cycle. So, initially, there are |V| single-node trees in the forest. Adding an edge merges two trees into one. When the algorithm is completed, there will be only one tree, and that is the minimum spanning tree.

Algorithm

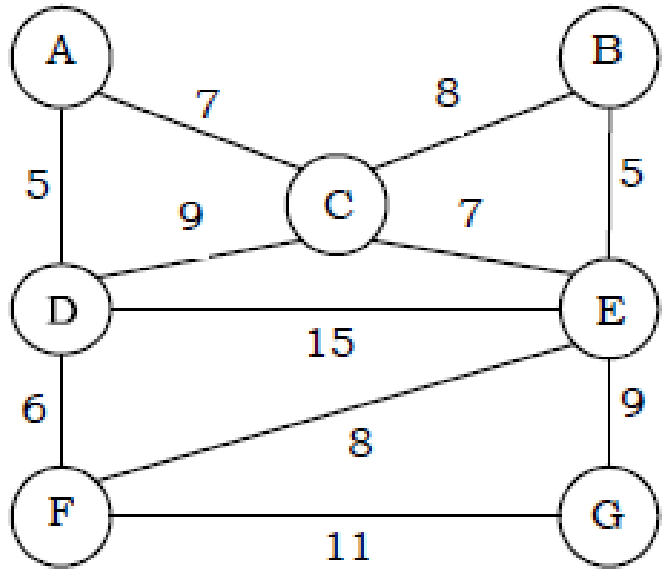
- Instructions:
  1. Sort the graph edges with respect to their weights.
  2. Start adding edges to the minimum snapping tree from the edge with the smallest weight until the edge of the largest weight.
  3. Only add edges which doesn't form a cycle, edges which connect only disconnected components.
- The greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far. There are two ways of implementing Kruskal’s algorithm:
  1. By using disjoint sets: Using UNION and FIND operations
  2. By using priority queues: Maintains weights in priority queue

Algorithm follows

- So now the question is how to check if vertices are connected or not?
- This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(E+V) where V is the number of vertices, E is the number of edges. Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.
- The appropriate data structure is the UNION/FIND algorithm [for implementing forests]. Two vertices belong to the same set if and only if they are connected in the current spanning forest. Each vertex is initially in its own set. If u and v are in the same set, the edge is rejected because it forms a cycle. Otherwise, the edge is accepted, and a UNION is performed on the two sets containing u and v.

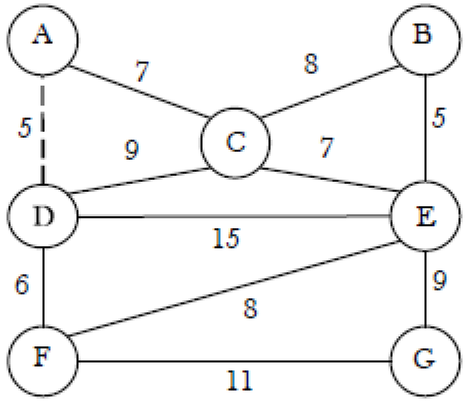
Example

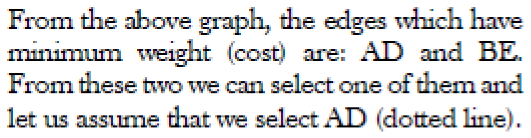
- As an example, consider the following graph (the edges show the weights). The issue is to relate with fiber optics the towns A, B, C, D, D, F, G. Cable cost are given in the graph



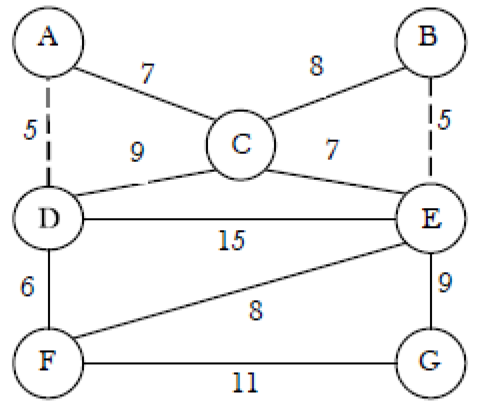
Example follows

- Now let us perform Kruskal’s algorithm on this graph. We always select the edge which has minimum weight.

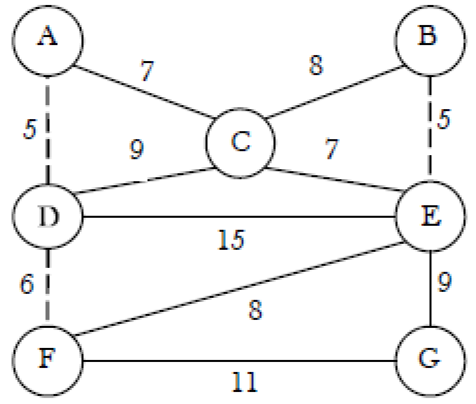



From the above graph, the edges which have minimum weight (cost) are: AD and BE. From these two we can select one of them and let us assume that we select AD (dotted line).

Example follows

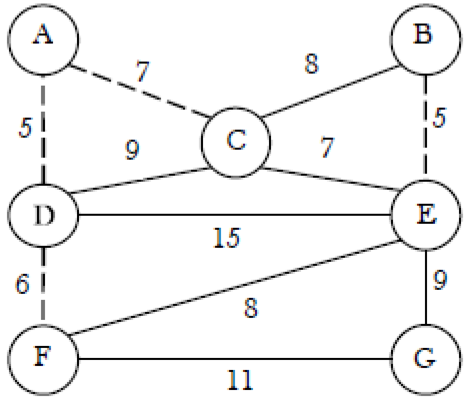


BE now has the lowest cost and we select it (dotted lines indicate selected edges).

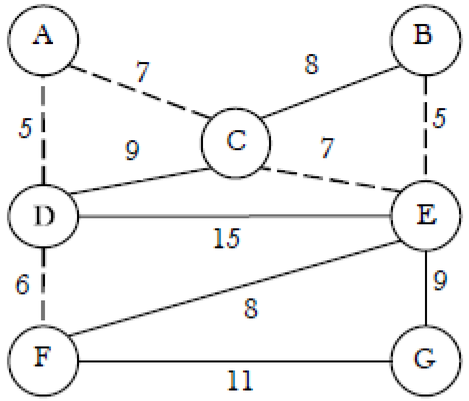


DF is the next edge that has the lowest cost (6).

Example follows

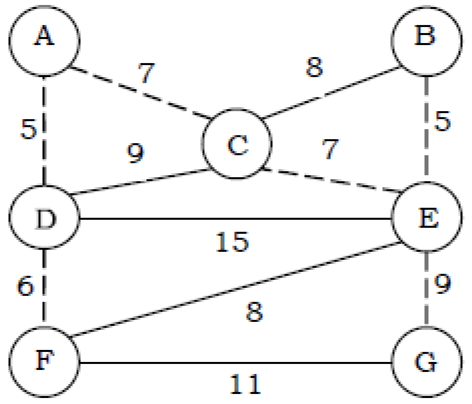


Next, AC and CE have the low cost of 7 and we select AC.



Then we select CE as its cost is 7 and it does not form a cycle.

Example follows



The next lowest cost edges are CB and EF. But if we select CB, then it forms a cycle. So, we discard it. This is also the case with EF. So, we should not select those two. And the next low cost is 9 (DC and EG). Selecting DC forms a cycle so we discard it. Adding EG will not form a cycle and therefore with this edge we complete all vertices of the graph.

- Upload corresponding code and fill the part of the code with

```
    raise ValueError('TO BE IMPLEMENTED')
```