

**“AZƏRBAYCAN HAVA YOLLARI” CJSC NATIONAL AVIATION ACADEMY**

**Individual Work № : 8**

**Topic:** **Dijkstra's algorithm**

**Subject: Obyektyönümlü proqramlaşdırma**

**Teacher: Mamed Shahmaliyev**

**Group: 1459i Student: Mehrali Babayev**

**Date: 11.10.2021 Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_**

**­ Baku 2022**

# **What Is Dijkstra’s Algorithm?**

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.

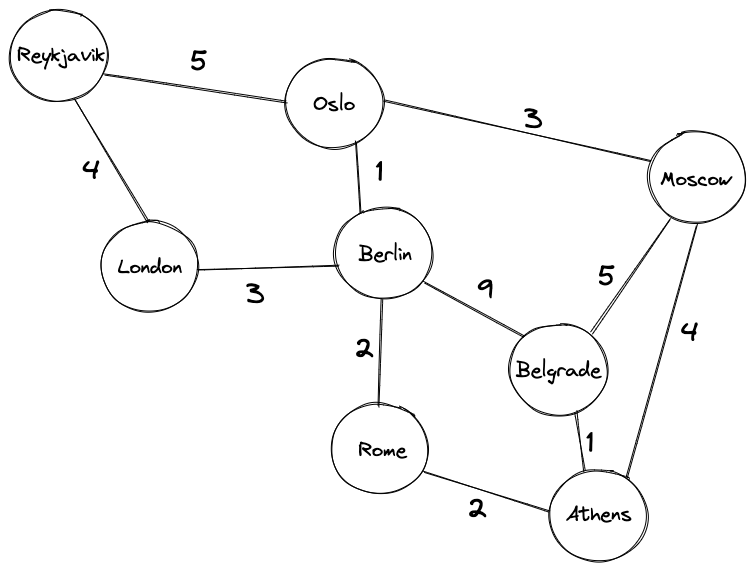
Dijkstra’s algorithm is a popular search algorithm used to determine the shortest path between two nodes in a graph. In the original scenario, the graph represented the Netherlands, the graph’s nodes represented different Dutch cities, and the edges represented the roads between the cities.

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).



## Code for Dijkstra's Algorithm

The implementation of Dijkstra's Algorithm in C++ is given below. The complexity of the [code](http://www.reviewmylife.co.uk/blog/2008/07/15/dijkstras-algorithm-code-in-c/) can be improved, but the abstractions are convenient to relate the code with the algorithm.

# Dijkstra's Algorithm in Python

import sys

# Providing the graph

vertices = [[0, 0, 1, 1, 0, 0, 0],

[0, 0, 1, 0, 0, 1, 0],

[1, 1, 0, 1, 1, 0, 0],

[1, 0, 1, 0, 0, 0, 1],

[0, 0, 1, 0, 0, 1, 0],

[0, 1, 0, 0, 1, 0, 1],

[0, 0, 0, 1, 0, 1, 0]]

edges = [[0, 0, 1, 2, 0, 0, 0],

[0, 0, 2, 0, 0, 3, 0],

[1, 2, 0, 1, 3, 0, 0],

[2, 0, 1, 0, 0, 0, 1],

[0, 0, 3, 0, 0, 2, 0],

[0, 3, 0, 0, 2, 0, 1],

[0, 0, 0, 1, 0, 1, 0]]

# Find which vertex is to be visited next

def to\_be\_visited():

global visited\_and\_distance

v = -10

for index in range(num\_of\_vertices):

if visited\_and\_distance[index][0] == 0 \

and (v < 0 or visited\_and\_distance[index][1] <=

visited\_and\_distance[v][1]):

v = index

return v

num\_of\_vertices = len(vertices[0])

visited\_and\_distance = [[0, 0]]

for i in range(num\_of\_vertices-1):

visited\_and\_distance.append([0, sys.maxsize])

for vertex in range(num\_of\_vertices):

# Find next vertex to be visited

to\_visit = to\_be\_visited()

for neighbor\_index in range(num\_of\_vertices):

# Updating new distances

if vertices[to\_visit][neighbor\_index] == 1 and \

visited\_and\_distance[neighbor\_index][0] == 0:

new\_distance = visited\_and\_distance[to\_visit][1] \

+ edges[to\_visit][neighbor\_index]

if visited\_and\_distance[neighbor\_index][1] > new\_distance:

visited\_and\_distance[neighbor\_index][1] = new\_distance

visited\_and\_distance[to\_visit][0] = 1

i = 0

# Printing the distance

for distance in visited\_and\_distance:

print("Distance of ", chr(ord('a') + i),

" from source vertex: ", distance[1])

i = i + 1

1. **The Graph Class**

First, we’ll create the Graph class. This class does not cover any of the Dijkstra algorithm’s logic, but it will make the implementation of the algorithm more succinct.

We’ll implement the graph as a [Python dictionary](https://www.udacity.com/blog/2020/12/how-to-work-with-python-dictionaries.html). The dictionary’s keys will correspond to the cities and its values will correspond to dictionaries that record the distances to other cities in the graph.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | import sys    class Graph(object):      def \_\_init\_\_(self, nodes, init\_graph):          self.nodes = nodes          self.graph = self.construct\_graph(nodes, init\_graph)        def construct\_graph(self, nodes, init\_graph):          '''          This method makes sure that the graph is symmetrical. In other words, if there's a path from node A to B with a value V, there needs to be a path from node B to node A with a value V.          '''          graph = {}          for node in nodes:              graph[node] = {}            graph.update(init\_graph)            for node, edges in graph.items():              for adjacent\_node, value in edges.items():                  if graph[adjacent\_node].get(node, False) == False:                      graph[adjacent\_node][node] = value            return graph        def get\_nodes(self):          "Returns the nodes of the graph."          return self.nodes        def get\_outgoing\_edges(self, node):          "Returns the neighbors of a node."          connections = []          for out\_node in self.nodes:              if self.graph[node].get(out\_node, False) != False:                  connections.append(out\_node)          return connections        def value(self, node1, node2):          "Returns the value of an edge between two nodes."          return self.graph[node1][node2] |

1. **The Dijkstra Algorithm**

Next, we’ll implement the Dijkstra algorithm. We’ll start by defining the function.

|  |  |
| --- | --- |
| 1 | def dijkstra\_algorithm(graph, start\_node): |

The function takes two arguments: graph and start\_node. graph is an instance of the Graph class that we created in the previous step, whereas start\_node is the node from which we’ll start the calculations. We’ll call the get\_nodes() method to initialize the list of unvisited nodes:

|  |  |
| --- | --- |
| 1 | unvisited\_nodes = list(graph.get\_nodes()) |

Next, we’ll create two dicts, shortest\_path and previous\_nodes:

* shortest\_path will store the best-known cost of visiting each city in the graph starting from the start\_node. In the beginning, the cost starts at infinity, but we’ll update the values as we move along the graph.
* previous\_nodes will store the trajectory of the current best known path for each node. For example, if we know the best way to Berlin to be via Oslo, previous\_nodes["Berlin"] will return “Oslo”, and previous\_nodes["Oslo"] will return “Reykjavik.” We’ll use this dictionary to backtrace the shortest path.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | shortest\_path = {}  previous\_nodes = {}  # We'll use max\_value to initialize the "infinity" value of the unvisited nodes  max\_value = sys.maxsize  for node in unvisited\_nodes:      shortest\_path[node] = max\_value  # However, we initialize the starting node's value with 0  shortest\_path[start\_node] = 0 |

Now we can start the algorithm. Remember that Dijkstra’s algorithm executes until it visits all the nodes in a graph, so we’ll represent this as a condition for exiting the while-loop.

|  |  |
| --- | --- |
| 1 | while unvisited\_nodes: |

Now, the algorithm can start visiting the nodes. The code block below first instructs the algorithm to find the node with the lowest value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | current\_min\_node = None  for node in unvisited\_nodes: # Iterate over the nodes      if current\_min\_node == None:          current\_min\_node = node      elif shortest\_path[node] < shortest\_path[current\_min\_node]:          current\_min\_node = node |

Once that’s done, the algorithm visits all node’s neighbors that are still unvisited. If the new path to the neighbor is better than the current best path, the algorithm makes adjustments in the shortest\_path and previous\_nodes dictionaries.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | # The code block below retrieves the current node's neighbors and updates their distances  neighbors = graph.get\_outgoing\_edges(current\_min\_node)  for neighbor in neighbors:      tentative\_value = shortest\_path[current\_min\_node] + graph.value(current\_min\_node, neighbor)      if tentative\_value < shortest\_path[neighbor]:          shortest\_path[neighbor] = tentative\_value          # We also update the best path to the current node          previous\_nodes[neighbor] = current\_min\_node |

After visiting all of its neighbors, we can mark the current node as “visited”:

|  |  |
| --- | --- |
| 1 | unvisited\_nodes.remove(current\_min\_node |

At last, we can return the two dictionaries:

|  |  |
| --- | --- |
| 1 | return previous\_nodes, shortest\_path |

1. **A Helper Function**

Lastly, we need to create a function that prints out the results. This function will take the two dictionaries returned by the dijskstra\_algorithm function, as well as the names of the beginning and target nodes. It’ll use the two dictionaries to find the best path and calculate the path’s score.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | def print\_result(previous\_nodes, shortest\_path, start\_node, target\_node):      path = []      node = target\_node        while node != start\_node:          path.append(node)          node = previous\_nodes[node]        # Add the start node manually      path.append(start\_node)        print("We found the following best path with a value of {}.".format(shortest\_path[target\_node]))      print(" -> ".join(reversed(path))) |