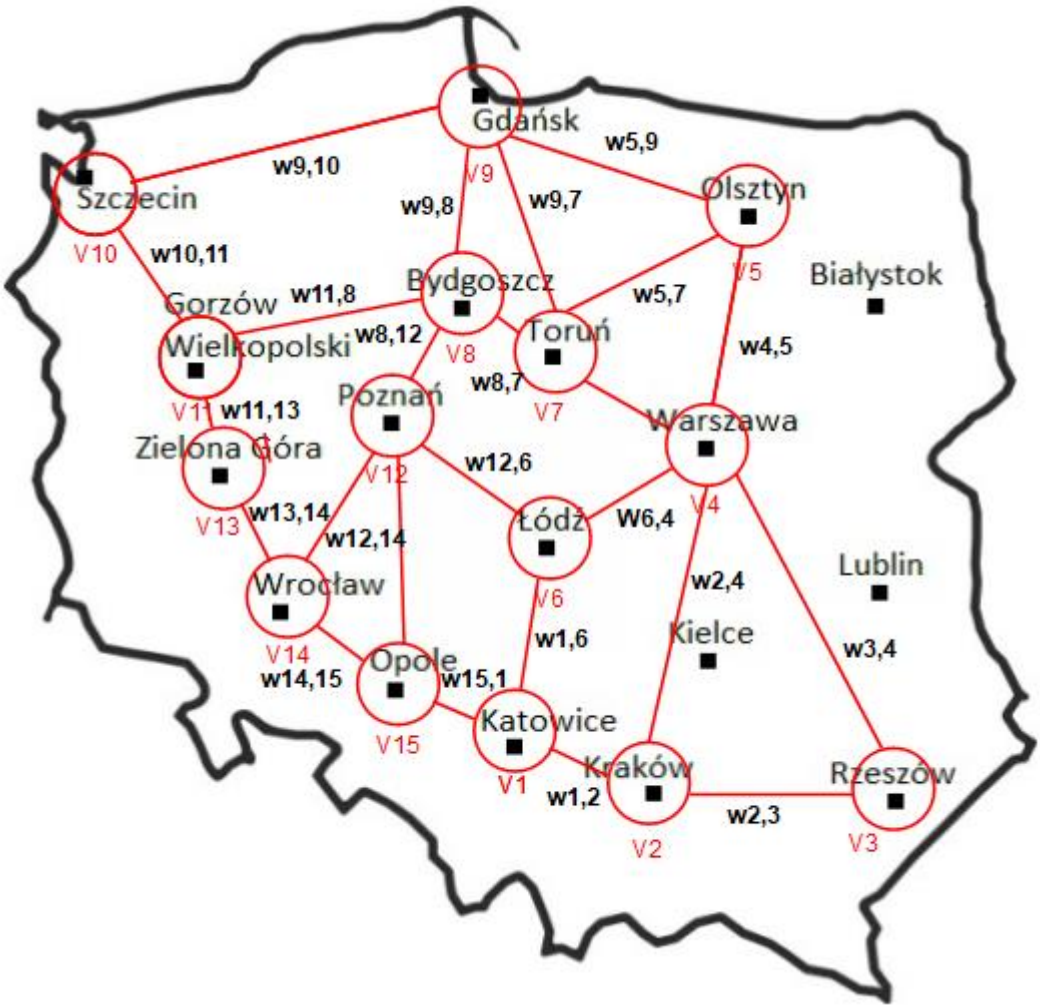
Filip Uherek
Michał Wilkosz

Graph Optimization Problems – Laboratory 20/04/20

A. Problem description

1. Graph structure



2. Weighting factors

- Katowice-Kraków(**w1,2**) - 78km
- Kraków-Rzeszów(**w2,3**) – 168km
- Rzeszów-Warszawa(**w3,4**) – 332km
- Kraków-Warszawa(**w2,4**) – 294km
- Warszawa-Olsztyn(**w4,5**) -214km
- Olsztyn-Gdańsk(**w5,9**)- 167km
- Olsztyn-Toruń(**w5,7**)-170km
- Warszawa-Toruń(**w4,7**)- 260km
- Gdańsk-Toruń(**w9,7**) -170km
- Gdańsk-Bydgoszcz(**w9,8**)-175km
- Bydgoszcz-Toruń(**w8,7**)-46km

- Gdańsk-Szczecin(**w9,10**)-356km
- Szczecin-Gorzów Wielkopolski(**w10,11**) – 105km
- Gorzów Wielkopolski-Bydgoszcz(**w11,8**) – 215km
- Bydgoszcz- Poznań(**w8,12**) – 139km
- Poznań-Łódź(**w12,6**) – 218 km
- Gorzów Wielkopolski-Zielona Góra (**w11,13**) – 112km
- Zielona Góra-Wrocław(**w13,14**) - 187km
- Poznań-Wrocław(**w12,14**) – 182km
- Wrocław-Opole(**w14,15**) – 98km
- Opole-Katowice(**w15,1**) – 113km
- Poznań-Opole (**w12,15**) – 285km
- Katowice-Łódź(**w1,6**) – 203km
- Łódź-Warszawa(**w6,4**) – 130km

3. Adjacency matrix

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15
V1	0	78	0	0	0	203	0	0	0	0	0	0	0	0	113
V2	78	0	168	294	0	0	0	0	0	0	0	0	0	0	0
V3	0	168	0	332	0	0	0	0	0	0	0	0	0	0	0
V4	0	294	332	0	214	130	260	0	0	0	0	0	0	0	0
V5	0	0	0	214	0	0	170	0	167	0	0	0	0	0	0
V6	203	0	0	130	0	0	0	0	0	0	0	218	0	0	0
V7	0	0	0	260	170	0	0	46	170	0	0	0	0	0	0
V8	0	0	0	0	0	0	46	0	175	0	215	139	0	0	0
V9	0	0	0	0	167	0	170	175	0	356	0	0	0	0	0
V10	0	0	0	0	0	0	0	0	356	0	105	0	0	0	0
V11	0	0	0	0	0	0	0	215	0	105	0	0	112	0	0
V12	0	0	0	0	0	218	0	139	0	0	0	0	0	182	285
V13	0	0	0	0	0	0	0	0	0	0	112	0	0	187	0
V14	0	0	0	0	0	0	0	0	0	0	0	182	187	0	98
V15	113	0	0	0	0	0	0	0	0	0	0	285	0	98	0

B. Program code and executable file

1. Program code

```
import numpy as np

class DijkstraAlgorithm:
    #Constructor function for prerequisites: Adjacency matrix G, S-
initial node, T-target node
    def __init__(self, G, initial_node, target_node):
        self.S = initial_node-1
        self.T = target_node-1
        self.run(G)

    #Updates minimum distance between towns-
vertices and stores them in list - dist[].
    #In first iteration of algorithm all 15 distances are assigned as inf.
    #First minimum distance is also inf, for given starting town distance equa
ls 0 and replaces inf.
    #In next iterations distances are updated by smaller possible values.
    #At the end of every function call, returns index of added distance to lis
t.
    def min_distance(self, dist, vertices):
        minimum = float("Inf")
        index = None
        for i in range(len(dist)):
            if dist[i] < minimum and i in vertices:
                minimum = dist[i]
                index = i
        return index

    #Prints S-initial node, T-target node and length of path as a Distance.
    def print_results(self, dist, previous):
        print("Initial node {}\nTarget node: {}\nDistance: {}\nPath:".format(
            self.S+1, self.T+1, dist[self.T]))
        self.print_path(previous, self.T)

    #Prints obtained path.
    def print_path(self, previous, j):
        if previous[j] == None:
            print (j+1,)
            return
        self.print_path(previous, previous[j])
        print (j+1,)

    #Main algorithm function
    def run(self, G):
        #Copy adjacency matrix G.
        new_G = G
```

```
        #Creates dist list of adjacency matrix row length and fills with inf v
alues.
        dist = np.full(len(new_G), np.inf)
        #List of previously visited vertices
        previous = np.full(len(new_G), None)
        #Assign distance to initial node.
        dist[self.S] = 0
        #Create towns - graph vertices.
        vertices = np.arange(0,len(new_G))
        #Executing until size of processed vertices list is greater than 0.
        while vertices.size > 0:
            #Takes index of minimal distance between towns in distance list.
            u = self.min_distance(dist,vertices)
            #Delete last downloaded vertex index from vertices.
            vertices = np.delete(vertices, np.argwhere(vertices == u))
            #Iteration over town - vertices.
            for i in range(len(new_G)):
                #If there exist an edge between element of adjacency matrix wh
ich is also in list of vertices-towns.
                if new_G[u][i] and i in vertices:
                    #If distance of index u + disance between neighbour and co
nsidered node is smaller then distance in dist list.
                    if dist[u] + new_G[u][i]< dist[i]:
                        #Element i of dist list is now dist[u] with distance b
etween considered node and neighbour.
                        dist[i] = dist[u] + new_G[u][i]
                        #List previous[] is updated with added index.
                        previous[i] = u

        self.print_results(dist, previous)

if __name__ == "__main__":

    #Adjacency matrix
    G = np.array(
        [[  0.,  78.,   0.,   0.,   0., 203.,   0.,   0.,   0.,   0.,   0., 0.
,   0.,   0., 113.],
        [ 78.,   0., 168., 294.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 0.,
   0.,   0.,   0.],
        [  0., 168.,   0., 332.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 0.,
   0.,   0.,   0.],
        [  0., 294., 332.,   0., 214., 130., 260.,   0.,   0.,   0.,   0., 0.,
   0.,   0.,   0.],
        [  0.,   0.,   0., 214.,   0.,   0., 170.,   0., 167.,   0.,   0., 0.,
   0.,   0.,   0.],
        [203.,   0.,   0., 130.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 218
.,   0.,   0.,   0.],
        [  0.,   0.,   0., 260., 170.,   0.,   0.,  46., 170.,   0.,   0., 0.,
   0.,   0.,   0.],
        [  0.,   0.,   0.,   0.,   0.,   0.,  46.,   0., 175.,   0., 215., 139
.,   0.,   0.,   0.],
        [  0.,   0.,   0.,   0., 167.,   0., 170., 175.,   0., 356.,   0., 0.,
   0.,   0.,   0.],
```

```
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 356.,   0., 105., 0.,
  0.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0., 215.,   0., 105.,   0., 0.,
112.,   0.,   0.],
       [  0.,   0.,   0.,   0.,   0., 218.,   0., 139.,   0.,   0.,   0., 0.,
  0., 182., 285.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 112., 0.,
  0., 187.,   0.],
       [  0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 182., 187
.,   0.,  98.,   0.],
       [113.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 285
.,   0.,  98.,   0.]])

    dij = DijkstraAlgorithm(G, initial_node=2, target_node=9)

    input("Press Enter to continue...")

    pass
```

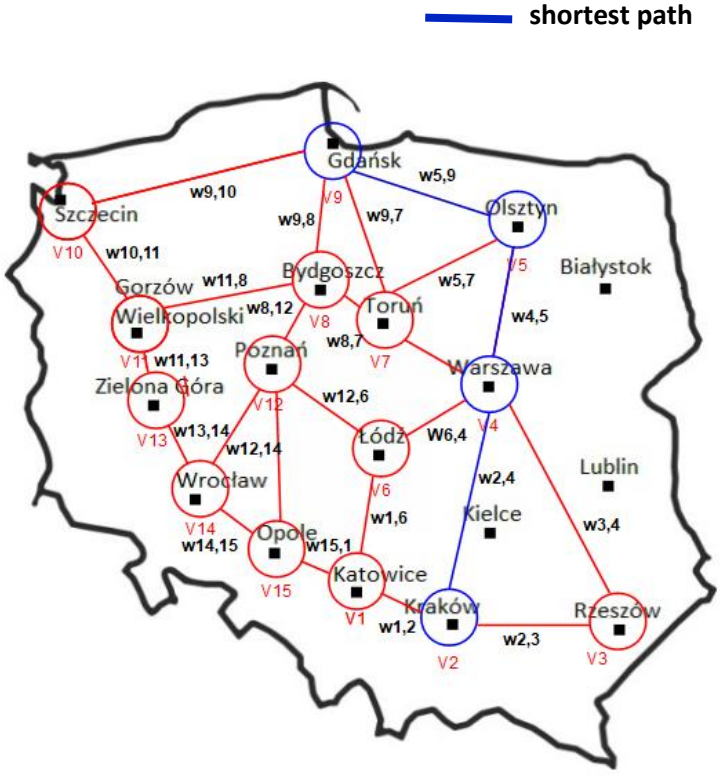
2. **Executable file**

Presented solution was developed in Python language with external library - Numpy(because of easier matrix operations). In connection with the necessity to contain used libraries. exe file exceed the allowable capacity which is possible to upload on PZE. A popular python interpreter like Anaconda allows to run the program without any problem. Therefore, here is a link which allows to download the whole program in .exe.

Link: https://drive.google.com/file/d/1tMrNSRFPontnAhPviX1AQXhq3JeFP0lr/view?usp=sharing

3. **Exemplary execution of program**

Assuming the initial node is 2 and the target node is 9. The shortest path should looks like this.



The length of the path between Kraków and Gdańsk is 675 km, and includes travel through the following cities:

Kraków(V2)--(w2,4=294km)--Warszawa(V4)--(w4,5=214)--Olsztyn(V5)--(w5,9=167km)--Gdańsk(V9).

Program output:

Initial node 2
Target node: 9
Distance: 675.0
Path:
2
4
5
9
Press Enter to continue...

Conclusion:

- For each of the cities visited, there were several options for the further route, but the final route was the shortest possible, which indicates the correctness of the algorithm.