Student Name: Fatemeh Ahmadvand
Student ID: s301384

Network Dynamics, and Learning

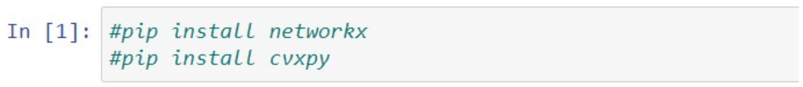
Assignment 1

1 Introduction

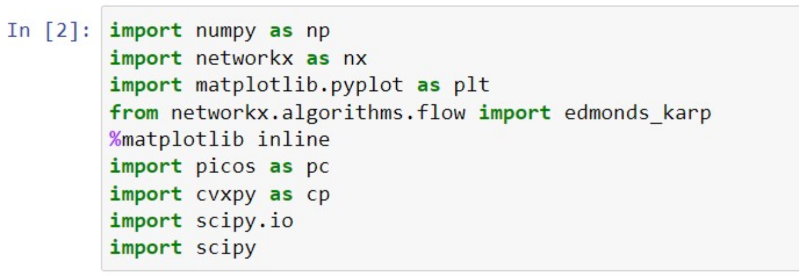
Graph theory principles are commonly utilized in diverse fields to research and model different applications. For instance, this knowledge is critical for modeling transport networks by finding the shortest route from two vertices in a diagram. The vertices mark the locations and the edges represent the movements.

2 Install and import all the libraries

graphs are represented and can be manipulated with Python using libraries, packages, and modules - NumPy, SciPy, Picos, Matplotlib, CVXPY and NetworkX. But the library which plays a significant rule to use throughout this assignment is the NetworkX package. It provides data structures and methods for storing graphs. The choice of graph class depends on the structure of the graph we want to represent.



```
In [1]: #pip install networkx
        #pip install cvxpy
```



```
In [2]: import numpy as np
        import networkx as nx
        import matplotlib.pyplot as plt
        from networkx.algorithms.flow import edmonds_karp
        %matplotlib inline
        import picos as pc
        import cvxpy as cp
        import scipy.io
        import scipy
```

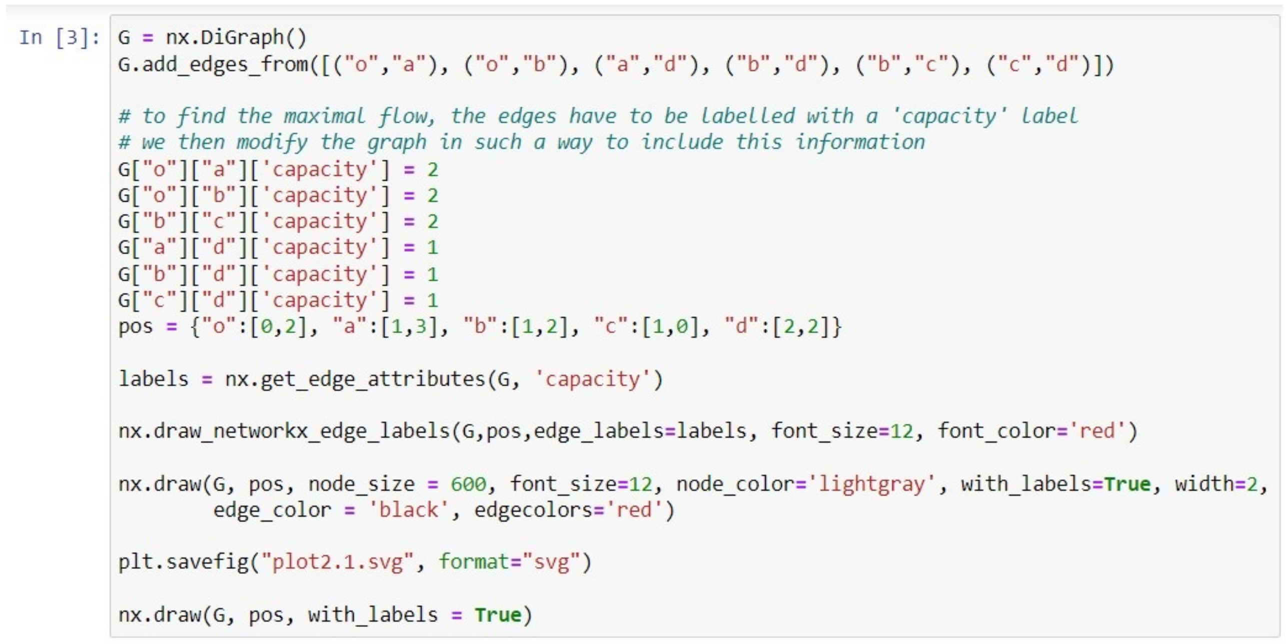
Exercise 1. Consider the network in Figure 2 with link capacities

$$c_2 = c_4 = c_6 = 1, c_1 = c_3 = c_5 = 2$$

Algorithms:NetworkX offers the implementation of a lot of network-related algorithms, which allow performing complex graph analysis in a simple way. we present the following functions contained in the modules nx.algorithms.flow.minimum_cut to compute the value and the node partition of a minimum (Source o, Sink d-cut, and nx.algorithms.flow.maximum_flow is to route as much flow as possible from the source to the sink, in other words, find the flow f_{max} with maximum value. The theorem equates two quantities: the maximum flow through a network, and the minimum capacity of a cut of the network.

Main Theorem: In the above situation, one can prove that the value of any flow through a network is less than or equal to the capacity of any o-d cut and that furthermore a flow with maximal value and a cut with minimal capacity exists. The main theorem links the maximum flow value with the minimum cut capacity of the network. ***Max-flow min-cut theorem***, the maximum value of an o-d flow is equal to the minimum capacity over all o-d cuts.

Data preparation: we create the graph and then to find the maximal flow, the edges have to be labeled with a 'capacity' label (Figure 1.).



```
In [3]: G = nx.DiGraph()
        G.add_edges_from([("o","a"), ("o","b"), ("a","d"), ("b","d"), ("b","c"), ("c","d")])

        # to find the maximal flow, the edges have to be labelled with a 'capacity' label
        # we then modify the graph in such a way to include this information
        G["o"]["a"]['capacity'] = 2
        G["o"]["b"]['capacity'] = 2
        G["b"]["c"]['capacity'] = 2
        G["a"]["d"]['capacity'] = 1
        G["b"]["d"]['capacity'] = 1
        G["c"]["d"]['capacity'] = 1
        pos = {"o":[0,2], "a":[1,3], "b":[1,2], "c":[1,0], "d":[2,2]}

        labels = nx.get_edge_attributes(G, 'capacity')

        nx.draw_networkx_edge_labels(G,pos,edge_labels=labels, font_size=12, font_color='red')

        nx.draw(G, pos, node_size = 600, font_size=12, node_color='lightgray', with_labels=True, width=2,
                edge_color = 'black', edgecolors='red')

        plt.savefig("plot2.1.svg", format="svg")

        nx.draw(G, pos, with_labels = True)
```

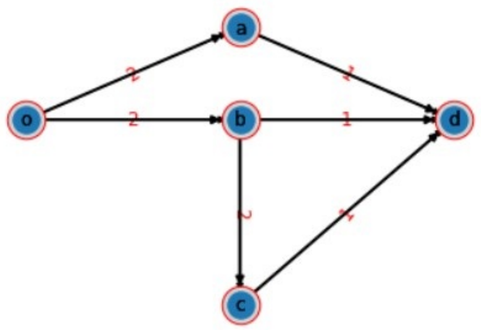


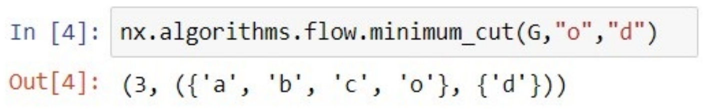
Figure.1: Network with link capacity

1. What is the minimum aggregate capacity that needs to be removed for no feasible flow from o to d to exist?

   As can be seen in the graph we have three paths from Source o to Sink d:

   - Path 1: $o \xrightarrow{2} a \xrightarrow{1} d$
   - Path 2: $o \xrightarrow{2} b \xrightarrow{1} d$
   - Path 3: $o \xrightarrow{2} b \xrightarrow{2} c \xrightarrow{1} d$

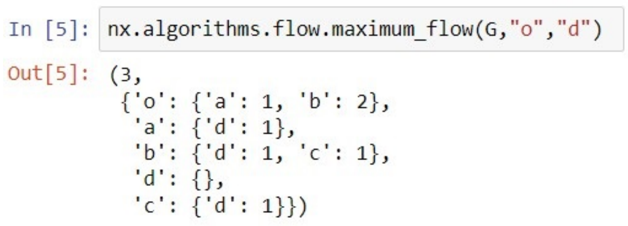
   According to choosing the edge with minimum capacity, in the first path and second path, we remove e(a,d) and e(b,d) respectively. Afterward, The new capacity of e(o, a) and e(o, b) calculates by subtracting the capacity of the removal edge, and in this case, it changes to 1. When it comes to the third path, the capacity of e(o,b) is the same as e(c,d) and it is equal to 1 but we choose the edge(c,d) which has the real lower capacity. To achieve the minimum aggregate capacity that needs to be removed for no feasible flow from o to d to exist, we use nx.algorithms.flow.minimum_cut.



```
In [4]: nx.algorithms.flow.minimum_cut(G,"o","d")
Out[4]: (3, ({'a', 'b', 'c', 'o'}, {'d'}))
```

2. What is the maximum aggregate capacity that can be removed from the links without affecting the maximum throughput from o to d?

   maximum_flow returns the maximal throughput, plus a dictionary containing the value of the flow that goes through each edge achieved by nx.algorithms.flow.maximum_flow.



```
In [5]: nx.algorithms.flow.maximum_flow(G,"o","d")
Out[5]: (3,
         {'o': {'a': 1, 'b': 2},
          'a': {'d': 1},
          'b': {'d': 1, 'c': 1},
          'd': {},
          'c': {'d': 1}})
```

   Finally, the max-flow min-cut theorem, represents the capacity of a minimum capacity cut is equal to the flow value of a maximum flow.

3. You are given x>0 extra units of capacity. How should you distribute them in order to maximize the throughput that can be sent from o to d? Plot the maximum throughput from o to d as a function of x$\geq$ 0.

$$\begin{aligned} \max_{F} \quad & F = \sum_{e \in E} f(e) \\ \text{s.t.} \quad & \forall m \in V - \{o,d\}, \quad \sum_{i,m} f(i,m) = \sum_{m,j} f(m,j) \\ & \forall e \in E, \quad 0 \leq f(e) \leq c(e) + \lambda_e \\ & \sum \lambda_e \ = \ \text{extra units of capacity} \\ & \lambda_e \geq 0 \end{aligned} \tag{1}$$

   In this part, we use Picos library to enable us to better deal with extra capacity that should be added to the edges of the graph. So, we extract capacities as a dictionary, and then by using picos, we convert capacities to a picos expression. Now we can instantiate a problem by using the problem method. We set the extra capacity value as a constraint and Set the objective. In the next step, we use the "solve()" method to find maximize distribution capacity from Source O to sink d.

```
In [6]: import picos as pc

        for x in range(0, 10):
          # Extra capacity available
          gamma = x

          # Extracting capacities as dictionary
          c = {}
          for e in sorted(G.edges(data=True)):
            capacity = e[2]['capacity']
            c[(e[0], e[1])]  = capacity

          # Convert the capacities to a PICOS expression.
          cc = pc.new_param('c',c)

          s, t = 'o', 'd'

          maxflow=pc.Problem()

          # Add the flow variables.
          f={}
          for e in G.edges():
            f[e]=maxflow.add_variable('f[{0}]'.format(e))

          # Add the extra capacity variable
          ex={}
          for e in G.edges():
            ex[e]=maxflow.add_variable('ex[{0}]'.format(e))

          # Add the objective variable for the total flow.
          F=maxflow.add_variable('F')

          # CONSTRAINTS
          # Enforce flow conservation.
          maxflow.add_list_of_constraints([
              pc.sum([f[p,i] for p in G.predecessors(i)])
              == pc.sum([f[i,j] for j in G.successors(i)])
              for i in G.nodes() if i not in (s,t)])

          # Set source flow at s.
          maxflow.add_constraint(
            pc.sum([f[p,s] for p in G.predecessors(s)]) + F
            == pc.sum([f[s,j] for j in G.successors(s)]))

          # Set sink flow at t.
          maxflow.add_constraint(
            pc.sum([f[p,t] for p in G.predecessors(t)])
            == pc.sum([f[t,j] for j in G.successors(t)]) + F)

          # Enforce flow nonnegativity.
          maxflow.add_list_of_constraints([f[e] >= 0 for e in G.edges()])

          # Enforce edge capacities.
          maxflow.add_list_of_constraints([f[e] <= cc[e] + ex[e] for e in G.edges()])

          # Enforce extra capacity nonnegativity.
          maxflow.add_list_of_constraints([ex[e] >= 0 for e in G.edges()])

          # Set extra capacity value constraint.
          maxflow.add_constraint(pc.sum([ex[e] for e in G.edges()]) <= gamma)

          # Set the objective.
          maxflow.set_objective('max', F)

          # Solve the problem.
          maxflow.solve(solver='glpk')
```

```
In [7]: import math
        import random

        print(math.floor(F))

        for val in ex.items():
          print(val[0], round(val[1]))
```

```
8
('o', 'a') 4
('o', 'b') 0
('a', 'd') 5
('b', 'd') 0
('b', 'c') 0
('c', 'd') 0
```

Exercise 2.

Main Theorem:

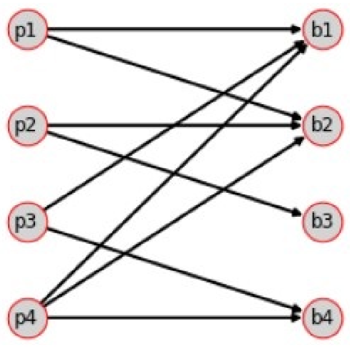
Max-flow problems and perfect matching: in the mathematical field of graph theory, a bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint and independent sets U and V, that is every edge connects a vertex in U to one in V. Vertex sets U and V are usually called the parts of the graph.

Equivalently, a bipartite graph is a graph that does not contain any odd-length cycles. A perfect matching in G = (V, E) is a matching of all vertices in V. We want to find conditions ensuring the existence of large or perfect matching in arbitrary graphs.

An important special case of matching considers bipartite graphs G = (V, E) where $V = X \cup Y$ and $X \cap Y =$. We study this special case through the lens of the max-flow min-cut theorem. First, we transform G in a flow network $G' = (V', E')$ where $V'V \cup s, t$ and $E' = E \cup (s, x) : (x \in X \cup (y, t) : y \in Y$ is called Matchings in bipartite graphs. There are a set of people $\{p_1, p_2, p_3, p_4\}$ and a set of book $\{b_1, b_2, b_3, b_4\}$.Each person is interested in a subset of books, specifically

$$p_1 \rightarrow \{b_1, b_2\}, p_2 \rightarrow \{b_2, b_3\}, p_3 \rightarrow \{b_1, b_4\}, p_4 \rightarrow \{b_1, b_2, b_4\}$$

```
In [8]: #G bipartite graph
        G = nx.DiGraph()
        G.add_edges_from([("p1","b1"), ("p1","b2"), ("p2","b2"), ("p2","b3"), ("p3","b1"), ("p3","b4"),
                          ("p4","b1"), ("p4","b2"), ("p4","b4")])

        #Draw the Bipartite Graph
        fig, ax = plt.subplots(figsize=(4,4))

        pos = {"p1":[0,2], "p2":[0,1], "p3":[0,0], "p4":[0,-1], "b1":[1,2], "b2":[1,1], "b3":[1,0], "b4":[1,-1]}
        nx.draw(G, pos, node_size = 600, font_size=12, node_color='lightgray', with_labels=True, width=2,
                edge_color = 'black', edgecolors='red', ax=ax)

        plt.savefig("plot2.1.svg", format="svg")
```



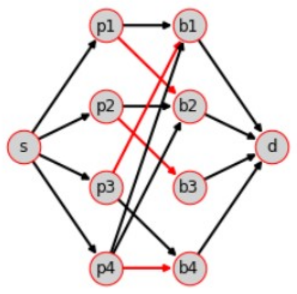
1. Exploit max-flow problems to find a perfect matching (if any).

   Based on the above-designed graph, s and d are defined as Source and Destination.For calculating the max-flow of the bipartite graph, We use nx.algorithms.flow.maximum_flow(dG,"s","d") that four red edges achieved as Max Flow

$$p_1 \rightarrow \{b_2\}, p_2 \rightarrow \{b_3\}, p_3 \rightarrow \{b_1\}, p_4 \rightarrow \{b_4\}$$

```
In [9]: # add source and destionation to bipartite graph
        dG = nx.DiGraph()
        dG.add_edges_from([("p1","b1"), ("p1","b2"), ("p2","b2"), ("p2","b3"), ("p3","b1"), ("p3","b4"),
                           ("p4","b1"), ("p4","b2"), ("p4","b4")])
        dG.add_edges_from([("s","p1"), ("s","p2"), ("s","p3"), ("s","p4"), ("b1","d"), ("b2", "d"),
                           ("b3","d"), ("b4","d")])
        # unitary capacity
        dG["p1"]["b1"]['capacity']=1
        dG["p1"]["b2"]['capacity']=1
        dG["p2"]["b2"]['capacity']=1
        dG["p2"]["b3"]['capacity']=1
        dG["p3"]["b1"]['capacity']=1
        dG["p3"]["b4"]['capacity']=1
        dG["p4"]["b1"]['capacity']=1
        dG["p4"]["b2"]['capacity']=1
        dG["p4"]["b4"]['capacity']=1
        dG["s"]["p1"]['capacity']=1
        dG["s"]["p2"]['capacity']=1
        dG["s"]["p3"]['capacity']=1
        dG["s"]["p4"]['capacity']=1
        dG["b1"]["d"]['capacity']=1
        dG["b2"]["d"]['capacity']=1
        dG["b3"]["d"]['capacity']=1
        dG["b4"]["d"]['capacity']=1
        # Max-flow of Graph
        print("Max flow: ", nx.algorithms.flow.maximum_flow(dG, "s", "d"))
        # draw graph
        fig, ax = plt.subplots(figsize=(4,4))

        edge_colors = ["black","red","black","black","black","red","black","red","black","black","black",
                       "black","red","black","black","black","black"]
        pos = {"p1":[0,2], "p2":[0,1], "p3":[0,0], "p4":[0,-1], "b1":[1,2], "b2":[1,1], "b3":[1,0],
               "b4":[1,-1], "s":[-1,0.5], "d":[2,0.5]}
        nx.draw(dG, pos, node_size = 600, font_size=12, node_color='lightgray', with_labels=True, width=2,
                edge_color = edge_colors, edgecolors='red', ax=ax)
        plt.savefig("plot2.2.svg", format="svg")
```

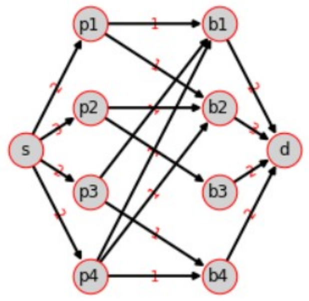
```
Max flow:  (4, {'p1': {'b1': 0, 'b2': 1}, 'b1': {'d': 1}, 'b2': {'d': 1}, 'p2': {'b2': 0, 'b3': 1}, 'b3': {'d': 1}, 'p3': {'b
1': 1, 'b4': 0}, 'b4': {'d': 1}, 'p4': {'b1': 0, 'b2': 0, 'b4': 1}, 's': {'p1': 1, 'p2': 1, 'p3': 1, 'p4': 1}, 'd': {}})
```



2. Assume now that there are multiple copies books, and the distribution of the number of copies is (2, 3, 2, 2). Each person can take an arbitrary number of different books. Exploit the analogy with max-flow problems to establish how many books of interest can be assigned in total.

```
In [10]: # weighted bipartite graph
         G_weighted = nx.DiGraph()
         G_weighted.add_edges_from([("p1","b1"), ("p1","b2"), ("p2","b2"), ("p2","b3"), ("p3","b1"), ("p3","b4"),
                                    ("p4","b1"), ("p4","b2"), ("p4","b4")])
         G_weighted.add_edges_from([("s","p1"), ("s","p2"), ("s","p3"), ("s","p4"), ("b1","d"), ("b2", "d"),("b3","d"), ("b4","d")])
         # weighted capacity
         G_weighted["p1"]["b1"]['capacity']=1
         G_weighted["p1"]["b2"]['capacity']=1
         G_weighted["p2"]["b2"]['capacity']=1
         G_weighted["p2"]["b3"]['capacity']=1
         G_weighted["p3"]["b1"]['capacity']=1
         G_weighted["p3"]["b4"]['capacity']=1
         G_weighted["p4"]["b1"]['capacity']=1
         G_weighted["p4"]["b2"]['capacity']=1
         G_weighted["p4"]["b4"]['capacity']=1
         G_weighted["s"]["p1"]['capacity']=2
         G_weighted["s"]["p2"]['capacity']=3
         G_weighted["s"]["p3"]['capacity']=2
         G_weighted["s"]["p4"]['capacity']=2
         G_weighted["b1"]["d"]['capacity']=2
         G_weighted["b2"]["d"]['capacity']=3
         G_weighted["b3"]["d"]['capacity']=2
         G_weighted["b4"]["d"]['capacity']=2
         # Max-flow of Graph
         print("Max flow: ", nx.algorithms.flow.maximum_flow(G_weighted,"s","d"))
         # draw Graph
         fig, ax = plt.subplots(figsize=(4,4))
         edge_colors = ["black","black","black","black","black","black","black","black","black","black","black",
                        "black","black","black","black","black","black"]
         pos = {"p1":[0,2], "p2":[0,1], "p3":[0,0], "p4":[0,-1], "b1":[2,2], "b2":[2,1], "b3":[2,0], "b4":[2,-1],
                "s":[-1,0.5], "d":[3,0.5]}
         labels = nx.get_edge_attributes(G_weighted, 'capacity')
         nx.draw_networkx_edge_labels(G_weighted,pos,edge_labels=labels, font_size=10, font_color='red')
         nx.draw(dG, pos, node_size = 600, font_size=12, node_color='lightgray', with_labels=True, width=2,
                 edge_color = edge_colors, edgecolors='red', ax=ax)
         plt.savefig("plot2.3.svg", format="svg")
```

```
Max flow:  (8, {'p1': {'b1': 1, 'b2': 1}, 'b1': {'d': 2}, 'b2': {'d': 3}, 'p2': {'b2': 1, 'b3': 1}, 'b3': {'d': 1}, 'p3': {'b
1': 1, 'b4': 1}, 'b4': {'d': 2}, 'p4': {'b1': 0, 'b2': 1, 'b4': 1}, 's': {'p1': 2, 'p2': 2, 'p3': 2, 'p4': 2}, 'd': {}})
```



According to the distribution of the number of copies (2, 3, 2, 2), we defined a Source s and a Sink d. As can be shown in figure 8, the input weight is equal to the output weight.We call nx.algorithms.flow.maximum_flow(G_weighted,”s”,”d”) method by weighted graph. In this way, eight books are chosen by this method.

3. Suppose that the library can sell a copy of a book and buy a copy of another book. Which books should be sold and bought to maximize the number of assigned books?

   To calculate the maximum number of books that we can assign to one person, we define two variables to store the values of weighted graph nodes and edges, and two variables to store the values of input edges and output edges (inflow, outflow). Then we navigate the graph to see which node is balanced. If its value is positive, it means that the output degree is greater than the input degree, and it is the opposite if it is negative. Finally, maxPossibleOutflow shows the highest assigned value.

```
In [11]: nodes = G_weighted.nodes
         edges = G_weighted.edges

         for node_i in nodes:

             inflow = 0
             outflow = 0
             maxPossibleOutflow = 0

             # source node is not considered
             if node_i == "s": continue

             # inflow of destination node is the maximal possible flow of graph
             if node_i == "d":
                  for u, v, data in G_weighted.in_edges(node_i, data=True):
                     maxPossibleOutflow = sum(data.values()) + maxPossibleOutflow

             # calculate inflow of node_i
             for u, v, data in G_weighted.in_edges(node_i, data=True):
                 inflow = sum(data.values()) + inflow

             # calculate outflow of node_i
             for u, v, data in G_weighted.out_edges(node_i, data=True):
                 outflow = sum(data.values()) + outflow

             # netflow of node_i
             netflow = inflow - outflow

             # if some nodes has inflow greater or minor than outflow means that it has not an optimized flow
             if netflow != 0 and node_i != "s" and node_i != "d":
                 print(f"Node {node_i} not optimized, netflow is {netflow}")

         print("Max possible outflow: ", maxPossibleOutflow)
```

```
Node b1 not optimized, netflow is 1
Node p2 not optimized, netflow is 1
Node b3 not optimized, netflow is -1
Node p4 not optimized, netflow is -1
Max possible outflow:  9
```

Exercise 3.

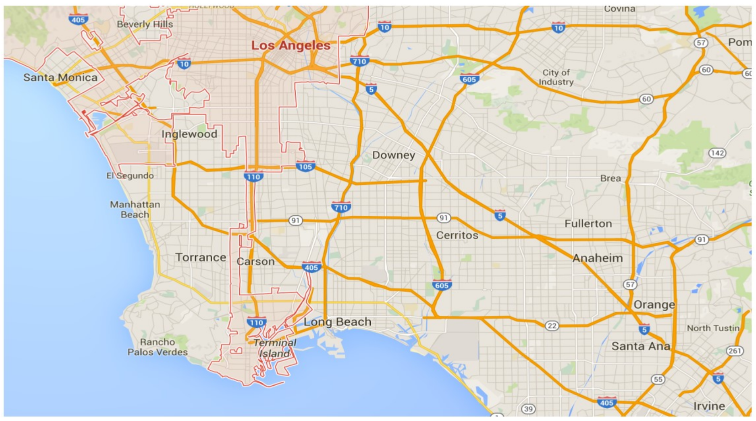


Figure.2: The highway network in Los Angeles.

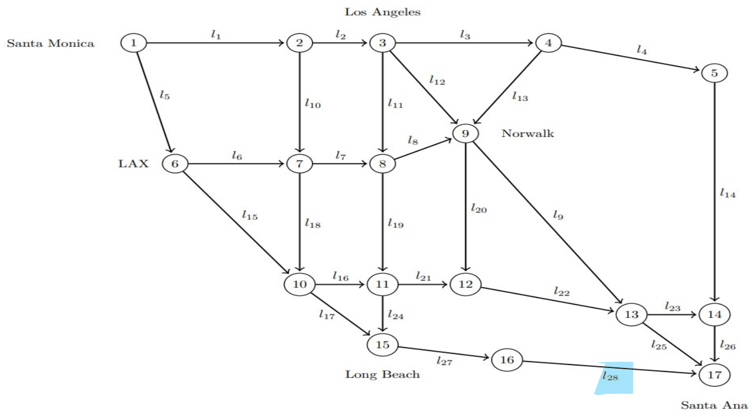
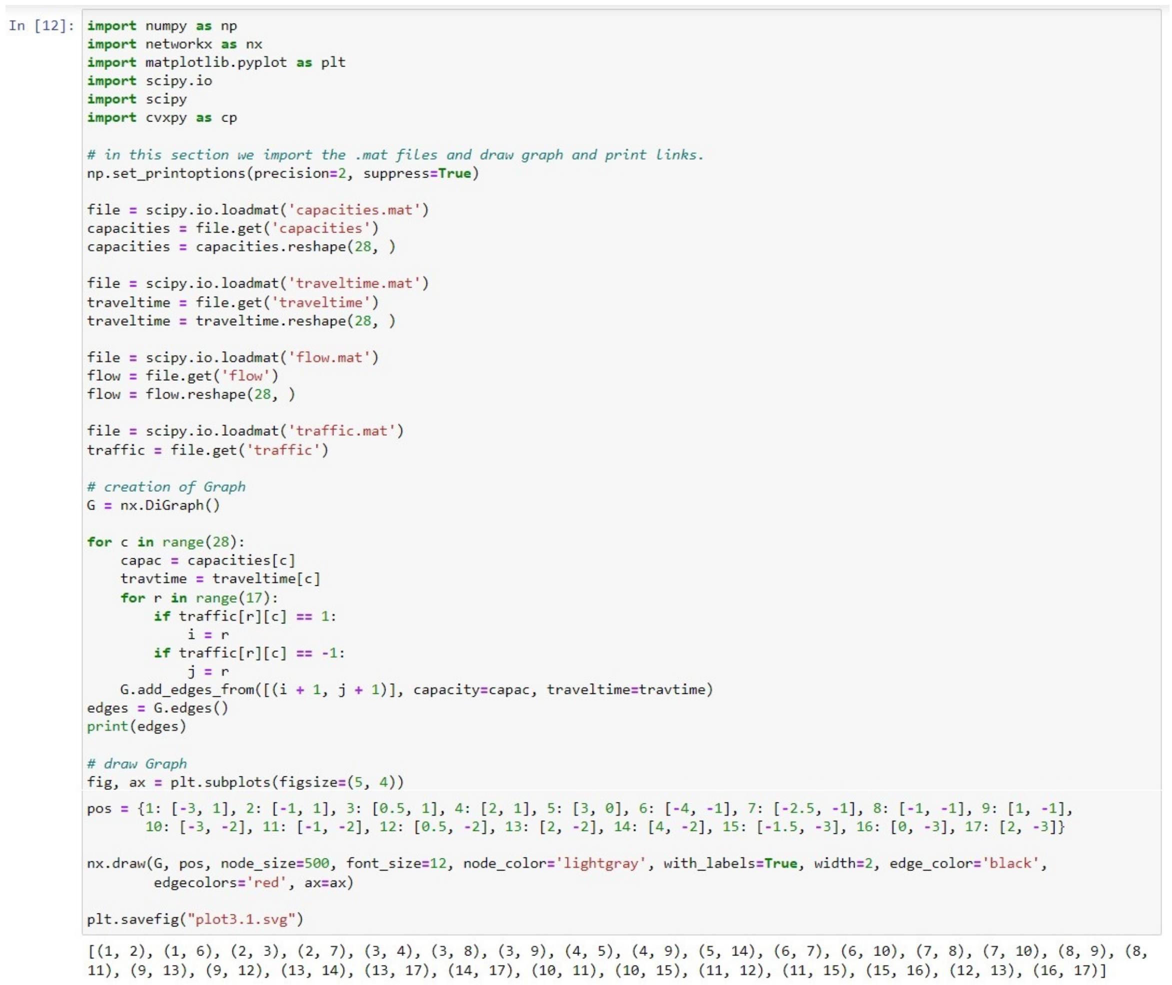

Los Angeles
Santa Monica
LAX
Norwalk
Long Beach
Santa Ana
l_1 l_2 l_3 l_4 l_5 l_6 l_7 l_8 l_9 l_{10} l_{11} l_{12} l_{13} l_{14} l_{15} l_{16} l_{17} l_{18} l_{19} l_{20} l_{21} l_{22} l_{23} l_{24} l_{25} l_{26} l_{27} l_{28}

Figure.3: Some possible paths from Santa Monica (node 1) to Santa Ana (node 17)

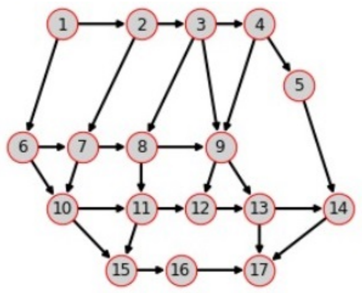
We are given the highway network in Los Angeles, see Figure 2. To simplify the problem, an approximate highway map is given in Figure 3, covering part of the real highway network. The node-link incidence matrix B, for this traffic network is given in the file traffic.mat. The rows of B are associated with the nodes of the network and the columns of B with the links.The i-th column of B has 1 in the row corresponding to the tail node of link e_i and (1) in the row corresponding to the head node of link e_i. Each node represents an intersection between highways (and some of the area around). Each link $e_i \in \{e_1, ..., e_{28}\}$, has a maximum flow capacity c_{e_1}. The capacities are given as a vector c_e in the file capacities.mat. Furthermore, each link has a minimum travelling time l_{e_i}, which the drivers experience when the road is empty. In the same manner as for the capacities, the minimum travelling times are given as a vector l_e in the file traveltime.mat. These values are simply retrieved by dividing the length of the highway segment with the assumed speed limit 60 miles/hour. For each link, we introduce the delay function

$$\tau_e(f_e) = \frac{l_e}{1-f_e/c_e} \text{ , } 0 \leq \mathrm{f_e} < c_e$$

For $f_e \geq c_e$, the value of $\tau_e(f_e)$ is considered as $+\infty$.
As we use pythone, it is essential to load following codes for this exercise:
f = scipy.io.loadmat('flow.mat')["flow"].reshape(28,)
C = scipy.io.loadmat('capacities.mat')["capacities"].reshape(28,)
B = scipy.io.loadmat('traffic.mat')["traffic"]
l = scipy.io.loadmat('traveltime.mat')["traveltime"].reshape(28,)
We make the graph according to the given information.



```
In [12]: import numpy as np
         import networkx as nx
         import matplotlib.pyplot as plt
         import scipy.io
         import scipy
         import cvxpy as cp

         # in this section we import the .mat files and draw graph and print links.
         np.set_printoptions(precision=2, suppress=True)

         file = scipy.io.loadmat('capacities.mat')
         capacities = file.get('capacities')
         capacities = capacities.reshape(28, )

         file = scipy.io.loadmat('traveltime.mat')
         traveltime = file.get('traveltime')
         traveltime = traveltime.reshape(28, )

         file = scipy.io.loadmat('flow.mat')
         flow = file.get('flow')
         flow = flow.reshape(28, )

         file = scipy.io.loadmat('traffic.mat')
         traffic = file.get('traffic')

         # creation of Graph
         G = nx.DiGraph()

         for c in range(28):
             capac = capacities[c]
             travtime = traveltime[c]
             for r in range(17):
                 if traffic[r][c] == 1:
                     i = r
                 if traffic[r][c] == -1:
                     j = r
             G.add_edges_from([(i + 1, j + 1)], capacity=capac, traveltime=travtime)
         edges = G.edges()
         print(edges)

         # draw Graph
         fig, ax = plt.subplots(figsize=(5, 4))
         pos = {1: [-3, 1], 2: [-1, 1], 3: [0.5, 1], 4: [2, 1], 5: [3, 0], 6: [-4, -1], 7: [-2.5, -1], 8: [-1, -1], 9: [1, -1],
                10: [-3, -2], 11: [-1, -2], 12: [0.5, -2], 13: [2, -2], 14: [4, -2], 15: [-1.5, -3], 16: [0, -3], 17: [2, -3]}

         nx.draw(G, pos, node_size=500, font_size=12, node_color='lightgray', with_labels=True, width=2, edge_color='black',
                 edgecolors='red', ax=ax)

         plt.savefig("plot3.1.svg")

         [(1, 2), (1, 6), (2, 3), (2, 7), (3, 4), (3, 8), (3, 9), (4, 5), (4, 9), (5, 14), (6, 7), (6, 10), (7, 8), (7, 10), (8, 9), (8,
         11), (9, 13), (9, 12), (13, 14), (13, 17), (14, 17), (10, 11), (10, 15), (11, 12), (11, 15), (15, 16), (12, 13), (16, 17)]
```

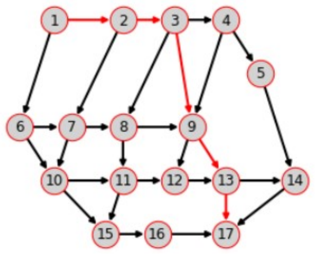


1. Find the shortest path between node 1 and 17. This is equivalent to the fastest path (path with shortest traveling time) in an empty network.

   One of the methods of the NetworkX library is the use of the Shortest path algorithm, which calculates the shortest path in the graph. The parameters of this algorithm are determining ***Source***:the Source starting node for path, ***Target***: Ending node of the path, ***Weight***:the weight of an edge is the value returned by the traveltime function and ***Method***: is the algorithm to use to compute the path, Supported options as the default is 'dijkstra'.

```
In [13]: print("The shortest path from node 1 to node 17: ", nx.shortest_path(G, source=1, target=17, weight='traveltime'))
         fig, ax = plt.subplots(figsize=(5, 4))

         edge_colors = ["red", "black", "red", "black", "black", "black", "red", "black", "black", "black",
                        "black", "black", "black", "black", "black", "black", "red", "black", "black", "red",
                        "black", "black", "black", "black", "black", "black", "black", "black"]
         nx.draw(G, pos, node_size=500, font_size=12, node_color='lightgray', with_labels=True, width=2, edge_color=edge_colors,
                edgecolors='red', ax=ax)

         plt.savefig("plot3.2.svg")
```

```
The shortest path from node 1 to node 17:  [1, 2, 3, 9, 13, 17]
```



Output path with red directed edges include both the source and target in the path and a single list of nodes in a shortest path from the source to the target.

2. Find the maximum flow between node 1 and 17.

   We calculate the maximum flow between node 1 to node 17 by nx.algorithms.flow.maximum_flow.

```
In [14]: #maximum flow between 1 to 17:
         print("Maximum flow from node 1 to 17: ", nx.algorithms.flow.maximum_flow(G, 1, 17))
```

```
Maximum flow from node 1 to 17:  (22448, {1: {2: 8741, 6: 13707}, 2: {3: 8741, 7: 0}, 3: {4: 0, 8: 0, 9: 8741}, 4: {5: 0, 9:
0}, 5: {14: 0}, 6: {7: 4624, 10: 9083}, 7: {8: 4624, 10: 0}, 8: {9: 4624, 11: 0}, 9: {13: 6297, 12: 7068}, 13: {14: 3835, 17: 1
0355}, 14: {17: 3835}, 10: {11: 825, 15: 8258}, 11: {12: 825, 15: 0}, 15: {16: 8258}, 12: {13: 7893}, 17: {}, 16: {17: 8258}})
```

3. Given the flow vector in flow.mat, compute the external inflow ν satisfying Bf $= \nu$. In the following, we assume that the exogenous inflow is zero in all the nodes except for node 1, for which ν_1 has the same value computed in the point (c), and node 17, for which ν_{17} =-ν_1.

   The value of the traffic matrix mentioned above is read from the traffic file and stored in a variable. Then we define the unitary inflow (exogenous flow vector) from node 1 to node 17 and the value of the traveltime function as an array in the new variable. we determine a problem that encloses an objective and a set of constraints. Then, with 'solve()' method, the problem is encoded by the instance, returns the optimal value, and set variables values to optimal points. By multiplying two matrices of flow and traffic, we compute the external inflow ν satisfying Bf $= \nu$.

```
In [15]: # C :external inflow 'v' satisfying Bf = v
         B_matrix = traffic
         n_edges = B_matrix.shape[1]
         # unitary inflow from node 1 to node 17
         nu = np.array([1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1])  # exogenous flow vector
         l = np.array(traveltime)

         # Construct the problem.
         f = cp.Variable(n_edges)

         objective = cp.Minimize(l.T @ f)
         constraints = [B_matrix @ f == nu, f >= 0]
         prob = cp.Problem(objective, constraints)

         # The optimal objective value is returned by `prob.solve()`.
         cost_opt = prob.solve()

         # The optimal value for f is stored in `f.value`.
         print("Optimal f:", f.value)
```

```
Optimal f: [1. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 1. 0. 0. 0.]
```

```
In [16]: external_inflow = B_matrix @ flow
         print("external_inflow:", external_inflow)

         external_inflow: [ 16806   8570  19448   4957   -746   4768    413     -2  -5671
          1169 -5   -7131   -380  -7412  -7810  -3430 -23544]
```

4. Find the social optimum f^* with respect to the delays on the different links $\tau_e(f_e)$. For this,minimize the cost function

$$\sum_{e\in E} f_e(\tau_e(f_e)) = \sum_{e\in E} \frac{l_e}{1-f_e/c_e} = \sum_{e\in E} \frac{l_e}{1-f_e/c_e} \text{ - } l_e c_e$$

For obtaining social optimum f^* concerning the delays on the different links $\tau_e(f_e)$, we implement the formula and store it in the Func variable. After minimizing the Func, considering it as the objective, and defining these constraints $0 \leq f_e < c_e$ to solve this problem. Then we call the Solve() method to achieve the optimal objective value.

```
In [17]: # D: Find the social optimum f* and optimum cost:
         # exogenous inflow vector
         nu = np.array([16806, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -16806])
         f = cp.Variable(n_edges)

         # cost function
         func = cp.multiply(traveltime * capacities, cp.inv_pos(1 - cp.multiply(f, 1 / capacities))) - traveltime * capacities
         func = cp.sum(func)

         # Construct the problem.
         # Minimize cost function
         objective = cp.Minimize(func)
         constraints = [B_matrix @ f == nu, f >= 0, f <= capacities]
         prob = cp.Problem(objective, constraints)

         # The optimal objective value is returned by `prob.solve()`.
         cost_opt = prob.solve()

         # The optimal value for f is stored in `f.value`.
         opt_flow = f.value
         print("Social optimal flow:", opt_flow)
         print("Optimal cost:", cost_opt)

         Social optimal flow: [ 6642.2    6058.94  3132.33  3132.33 10163.8    4638.32  3006.34  2542.63
           3131.54   583.26     0.01  2926.6      0.     3132.33  5525.48  2854.27
           4886.45  2215.24   463.72  2337.69  3317.99  5655.68  2373.11     0.
           6414.12  5505.43  4886.45  4886.45]
         Optimal cost: 25943.62261121288
```

5. (e1)Find the Wardrop equilibrium $f^{(0)}$. For this, use the cost function

$$\sum_{e\in E} \int_0^{f_e} \tau_e(\text{s}) \text{ ds}$$

for calculating the Wardrop equilibrium f(0) with respect to the above formula, first of all, we calculate Integral as an input parameter of the problem and then apply the constraints mentioned in the previous part.

```
In [18]: # E1: Find the Wardrop equilibrium
         f = cp.Variable(n_edges)

         # cost function
         integral = - cp.multiply(traveltime * capacities, cp.log(1 - (cp.multiply(f, 1 / capacities))))
         func2 = cp.sum(integral)

         # minimize cost function
         objective = cp.Minimize(func2)
         constraints = [B_matrix @ f == nu, f >= 0, f <= capacities]
         prob = cp.Problem(objective, constraints)

         cost_w = prob.solve()

         print("Wardrop equilibrium flow:", f.value)
```

```
Wardrop equilibrium flow: [ 6715.65  6715.65  2367.41  2367.41 10090.35  4645.39  2803.84  2283.56
  3418.48     0.     176.83  4171.41     0.    2367.41  5444.96  2353.17
  4933.34  1841.55   697.11  3036.49  3050.28  6086.77  2586.51     0.
  6918.74  4953.92  4933.34  4933.34]
```

(e2) Introduce tolls, such that the toll on link e is $\omega_e = {f_e}^* \tau_e'({f_e}^*)$ where ${f_e}^*$ is the flow at the system optimum. Now the delay on link e is given by $\tau_e(f_e)+\ \omega_e$. compute the new Wardrop equilibrium $f^{(\omega)}$. What do you observe?

The price of anarchy (POA) associated to a Wardrop equilibrium $f^{(\omega)}$ is

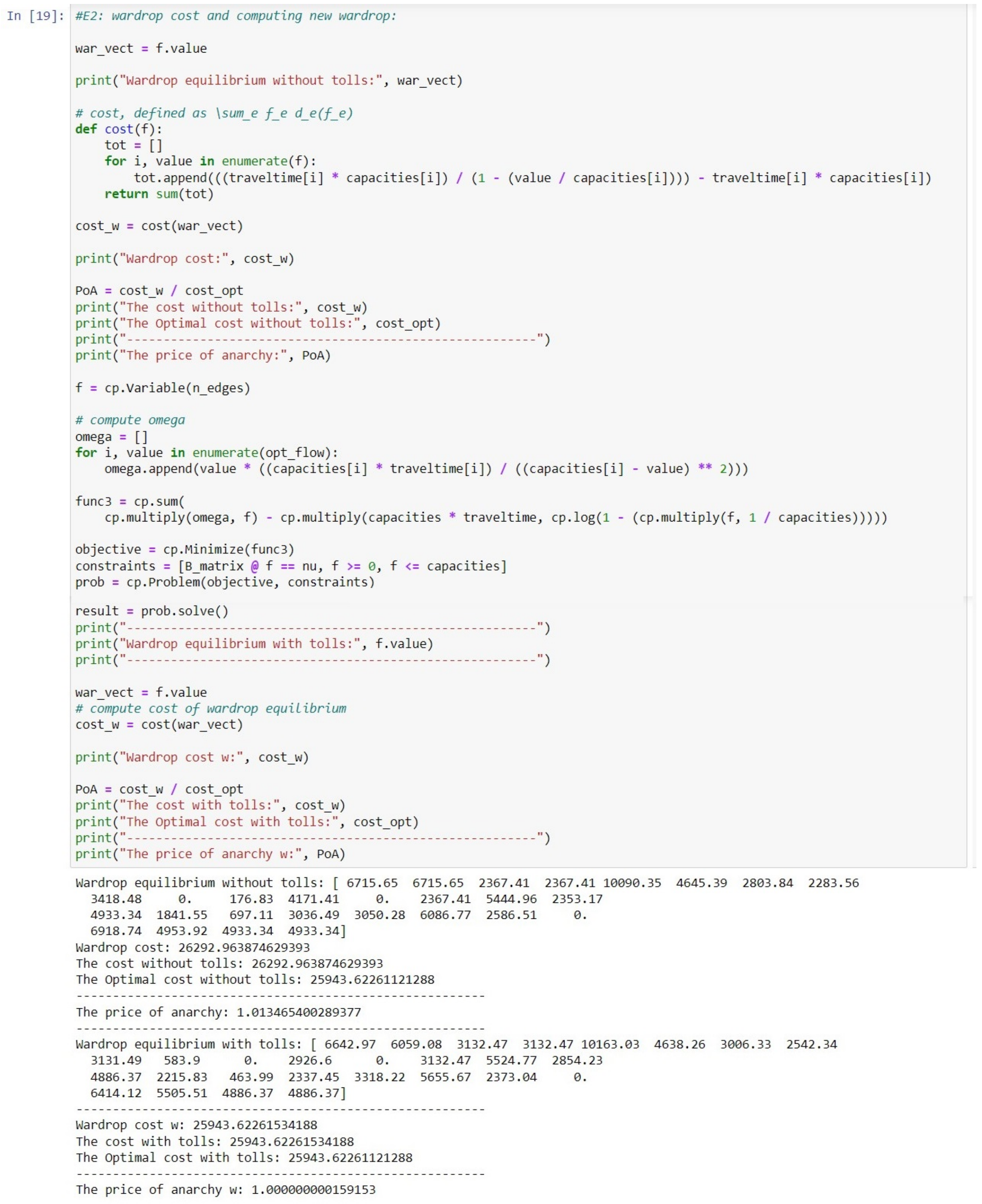
$$\text{PoA}(\omega) = \frac{\begin{array}{c}\text{total delay at}\\ \text{user optimum}\end{array}}{\begin{array}{c}\text{total delay at}\\ \text{social optimum}\end{array}}$$

While the Wardrop equilibrium (both with or without tolls) captures a user-perspective notion of optimization, it turns out that we can always interpret it as an optimal network flow, provided that we consider suitable costs on the links.

The System-Optimum Traffic Assignment Problem (SO-TAP) is the network flow optimization problem, the idea is now to model traffic flows in the network resulting not from a centralized optimization, but rather as the outcome of selfish behaviors of users. Such behavior is modeled by assuming that users choose their route so as to minimize the delay they experience along it. This is formalized by the price of anarchy by the notion of Wardrop equilibrium $f^{(\omega)}$ that we introduce below.

$$\text{PoA}(\omega) = \frac{\sum\limits_{e\in\mathcal{E}} f_e^{(\omega)} \tau_e(f_e^{(\omega)})}{\min\limits_{\substack{f\geq 0 \\ Bf=v(\delta^{(o)}-\delta^{(d)})}} \sum\limits_{e\in\mathcal{E}} f_e \tau_e(f_e)}\,.$$

it is the ratio between the total delay at the Wardrop equilibrium and the minimum possible total delay. with considering tolls on the flows and calculating POA for the flow with and without tolls, we concluded that POA decreases after tolls are applied.



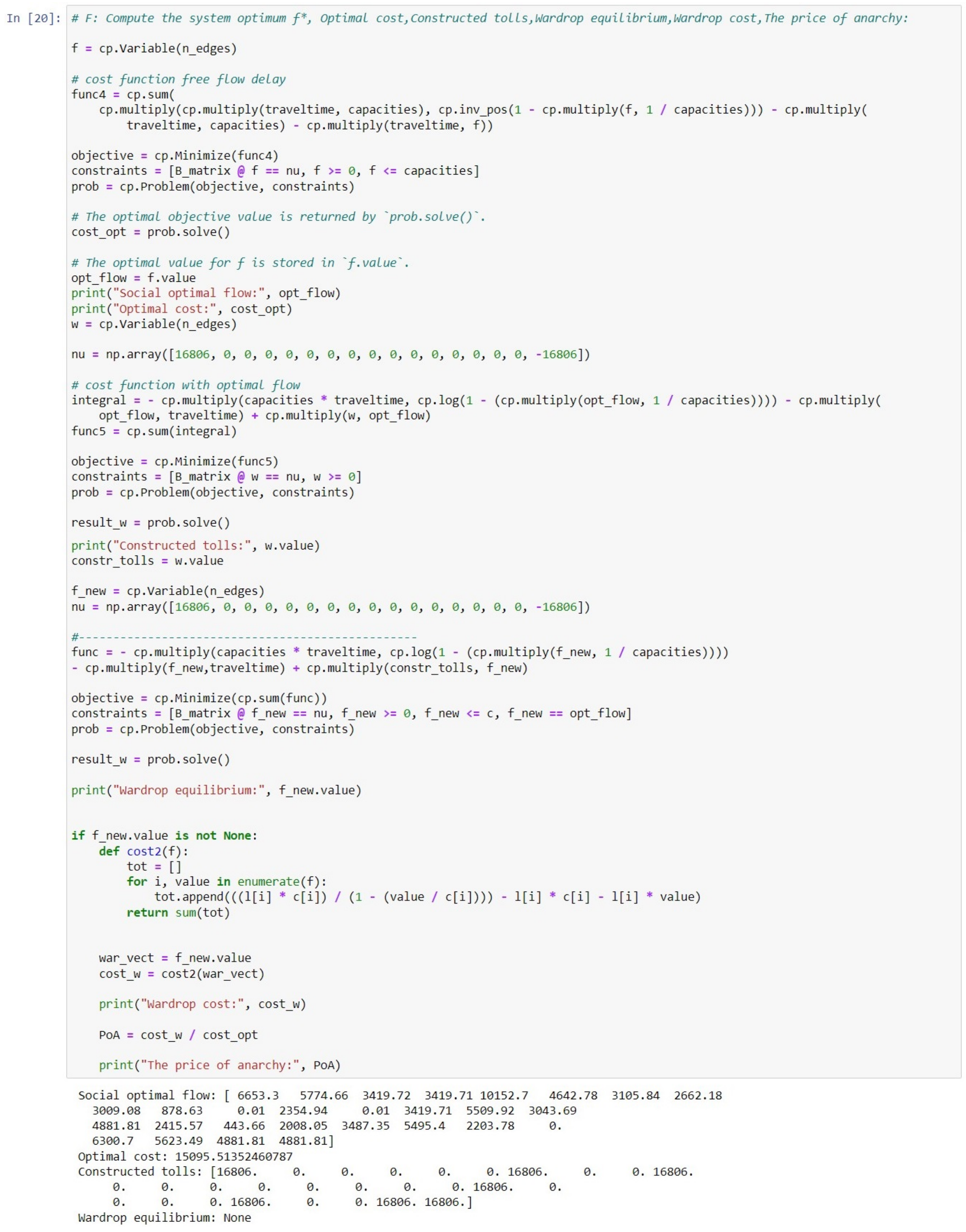
```
In [19]: #E2: wardrop cost and computing new wardrop:

         war_vect = f.value

         print("Wardrop equilibrium without tolls:", war_vect)

         # cost, defined as \sum_e f_e d_e(f_e)
         def cost(f):
             tot = []
             for i, value in enumerate(f):
                 tot.append(((traveltime[i] * capacities[i]) / (1 - (value / capacities[i]))) - traveltime[i] * capacities[i])
             return sum(tot)

         cost_w = cost(war_vect)

         print("Wardrop cost:", cost_w)

         PoA = cost_w / cost_opt
         print("The cost without tolls:", cost_w)
         print("The Optimal cost without tolls:", cost_opt)
         print("--------------------------------------------------------")
         print("The price of anarchy:", PoA)

         f = cp.Variable(n_edges)

         # compute omega
         omega = []
         for i, value in enumerate(opt_flow):
             omega.append(value * ((capacities[i] * traveltime[i]) / ((capacities[i] - value) ** 2)))

         func3 = cp.sum(
             cp.multiply(omega, f) - cp.multiply(capacities * traveltime, cp.log(1 - (cp.multiply(f, 1 / capacities)))))

         objective = cp.Minimize(func3)
         constraints = [B_matrix @ f == nu, f >= 0, f <= capacities]
         prob = cp.Problem(objective, constraints)

         result = prob.solve()
         print("--------------------------------------------------------")
         print("Wardrop equilibrium with tolls:", f.value)
         print("--------------------------------------------------------")

         war_vect = f.value
         # compute cost of wardrop equilibrium
         cost_w = cost(war_vect)

         print("Wardrop cost w:", cost_w)

         PoA = cost_w / cost_opt
         print("The cost with tolls:", cost_w)
         print("The Optimal cost with tolls:", cost_opt)
         print("--------------------------------------------------------")
         print("The price of anarchy w:", PoA)
```

```
Wardrop equilibrium without tolls: [ 6715.65  6715.65  2367.41  2367.41 10090.35  4645.39  2803.84  2283.56
  3418.48     0.     176.83  4171.41     0.     2367.41  5444.96  2353.17
  4933.34  1841.55   697.11  3036.49  3050.28  6086.77  2586.51     0.
  6918.74  4953.92  4933.34  4933.34]
Wardrop cost: 26292.963874629393
The cost without tolls: 26292.963874629393
The Optimal cost without tolls: 25943.62261121288
--------------------------------------------------------
The price of anarchy: 1.013465400289377
--------------------------------------------------------
Wardrop equilibrium with tolls: [ 6642.97  6059.08  3132.47  3132.47 10163.03  4638.26  3006.33  2542.34
  3131.49   583.9      0.    2926.6      0.    3132.47  5524.77  2854.23
  4886.37  2215.83   463.99  2337.45  3318.22  5655.67  2373.04     0.
  6414.12  5505.51  4886.37  4886.37]
--------------------------------------------------------
Wardrop cost w: 25943.62261534188
The cost with tolls: 25943.62261534188
The Optimal cost with tolls: 25943.62261121288
--------------------------------------------------------
The price of anarchy w: 1.000000000159153
```

6. Instead of the total travel time, let the cost for the system be the total additional delay compared to the total delay in free flow, given by

   $\psi_e(\mathrm{f}_e) = f_e(\tau_e(f_e)$ - $l_e)$
   subject to the flow constraints. Compute the system optimum f^* for the costs above. Construct tolls (ω^*)such that the Wardrop equilibrium $f^{(\omega^*)}$ coincides with f^*. Compute the new Wardrop equilibrium with the constructed tolls $f^{(\omega^*)}$ to verify your result.

we calculate the optimal flow by using the cost function in the free flow delay. The optimal flow obtained from the previous function (Func4) is used as the input of this function (Func5) to calculate the Constructed tolls, which the new Wardrop Equilibrium is calculated based on optimal flow and constructed tolls. due to the input files of this question, the obtained result of Wardrop Equilibrium is None. we develop the code to support other input files to achieve Wardrop cost and the price of anarchy(PoA)



```
In [20]: # F: Compute the system optimum f*, Optimal cost,Constructed tolls,Wardrop equilibrium,Wardrop cost,The price of anarchy:

         f = cp.Variable(n_edges)

         # cost function free flow delay
         func4 = cp.sum(
             cp.multiply(cp.multiply(traveltime, capacities), cp.inv_pos(1 - cp.multiply(f, 1 / capacities))) - cp.multiply(
                 traveltime, capacities) - cp.multiply(traveltime, f))

         objective = cp.Minimize(func4)
         constraints = [B_matrix @ f == nu, f >= 0, f <= capacities]
         prob = cp.Problem(objective, constraints)

         # The optimal objective value is returned by `prob.solve()`.
         cost_opt = prob.solve()

         # The optimal value for f is stored in `f.value`.
         opt_flow = f.value
         print("Social optimal flow:", opt_flow)
         print("Optimal cost:", cost_opt)
         w = cp.Variable(n_edges)

         nu = np.array([16806, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -16806])

         # cost function with optimal flow
         integral = - cp.multiply(capacities * traveltime, cp.log(1 - (cp.multiply(opt_flow, 1 / capacities)))) - cp.multiply(
             opt_flow, traveltime) + cp.multiply(w, opt_flow)
         func5 = cp.sum(integral)

         objective = cp.Minimize(func5)
         constraints = [B_matrix @ w == nu, w >= 0]
         prob = cp.Problem(objective, constraints)

         result_w = prob.solve()

         print("Constructed tolls:", w.value)
         constr_tolls = w.value

         f_new = cp.Variable(n_edges)
         nu = np.array([16806, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -16806])

         #-------------------------------------------------
         func = - cp.multiply(capacities * traveltime, cp.log(1 - (cp.multiply(f_new, 1 / capacities))))
         - cp.multiply(f_new,traveltime) + cp.multiply(constr_tolls, f_new)

         objective = cp.Minimize(cp.sum(func))
         constraints = [B_matrix @ f_new == nu, f_new >= 0, f_new <= c, f_new == opt_flow]
         prob = cp.Problem(objective, constraints)

         result_w = prob.solve()

         print("Wardrop equilibrium:", f_new.value)


         if f_new.value is not None:
             def cost2(f):
                 tot = []
                 for i, value in enumerate(f):
                     tot.append(((l[i] * c[i]) / (1 - (value / c[i]))) - l[i] * c[i] - l[i] * value)
                 return sum(tot)


             war_vect = f_new.value
             cost_w = cost2(war_vect)

             print("Wardrop cost:", cost_w)

             PoA = cost_w / cost_opt

             print("The price of anarchy:", PoA)
```

```
Social optimal flow: [ 6653.3    5774.66   3419.72   3419.71 10152.7    4642.78   3105.84   2662.18
  3009.08    878.63     0.01   2354.94     0.01   3419.71   5509.92   3043.69
  4881.81   2415.57    443.66   2008.05   3487.35   5495.4    2203.78      0.
  6300.7    5623.49   4881.81   4881.81]
Optimal cost: 15095.51352460787
Constructed tolls: [16806.      0.      0.      0.      0.      0. 16806.      0.      0. 16806.
     0.      0.      0.      0.      0.      0.      0.      0. 16806.      0.
     0.      0.      0. 16806.      0.      0. 16806. 16806.]
Wardrop equilibrium: None
```