Network Dynamics, and Learning (Homework 1)

Hesam Khanjani 310141
Shaghayegh Abedi 309894
Nastaran Ahmadi Bonakdar 310073
Mohammadreza Mashhadigholamali 308499
Ali Samimi Fard 308956

November 19, 2023

1 Introduction

Graph theory ideas are widely applied in many domains to study and simulate various applications. For example, by determining the shortest path between two vertices in a diagram, this knowledge is essential for modeling transport networks. The edges show the movements, while the vertices indicate the locations.

2 Install and Import all the libraries

Python provides libraries, packages, and modules for representing and manipulating graphs. SciPy, Matplotlib, NetworkX, Picos, NumPy, and CVXPY. However, the NetworkX package is the library extensively used in this assignment. It offers techniques and data structures for storing graphs. Graph classes are selected based on the graph's desired representation's structure.

```
[1]  !pip install networkx
     !pip install cvxpy
```

Figure 1

2.1 exercise one

Consider the network with link capacities:

$$c1 = c3 = c5 = 3,\ c6 = 1,\ c2 = c4 = 2$$

Main Theorem. The minimum-cut function is supported by the max-flow min-cut theorem. This theorem posits that the maximum flow through a network from a source to a sink is equal to the minimum cut's capacity, which is the least total capacity that must be removed to entirely disconnect the sink from the source. The minimum-cut function returns both the value of the minimum cut (the capacity) and the partition (the specific nodes on either side of the cut).

Data Preparation. The data preparation involves creating a directed graph with nodes and edges corresponding to the given network. The graph is initialized, and capacities are assigned to the edges according to the network's characteristics. This graph is then used as input for the minimum-cut function. The nodes are positioned and labeled for visualization purposes, and the graph is drawn using matplotlib to show the original network and the edges involved in the minimum cut.

```python
import networkx as nx
import matplotlib.pyplot as plt

# Create the directed graph as described in your previous code
G = nx.DiGraph()
G.add_edges_from([("o","a"), ("o","b"), ("a","d"), ("b","d"), ("b","c"), ("c","d")])

# Add edge capacities to the graph
capacities = {("o", "a"): 3, ("o", "b"): 3, ("b", "c"): 3, ("a", "d"): 2, ("b", "d"): 2, ("c", "d"): 1}
nx.set_edge_attributes(G, capacities, 'capacity')

# Specify the position for vertices
pos = {"o": [0, 2], "a": [1, 3], "b": [1, 2], "c": [1, 0], "d": [2, 2]}

# Draw the graph
labels = nx.get_edge_attributes(G, 'capacity')

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

# Show the plot
plt.show()
```

Figure 2

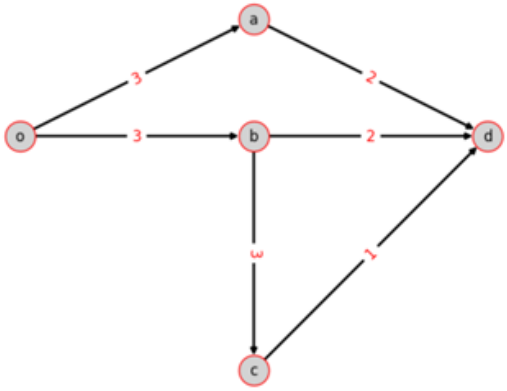

Figure 3

Question 1. What is the minimum aggregate capacity that needs to be removed for no feasible flow from o to d to exist? For the network with link capacities c1 = c3 = c5 = 3, c6 = 1, c2 = c4 = 2, the flow paths from the source node o to the sink node d are:

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

Figure 4

In evaluating the minimum aggregate capacity to be removed, we assess the paths and identify the edges that have the smallest capacities, which become the bottlenecks for flow in their respective paths. The minimum cut will involve these edges. For Path 1, the edge e(a,d) with capacity 2 is the bottleneck, since it has the lower capacity compared to the edge e(o,a) with capacity 3. Similarly, in Path 2, the edge e(b,d) also has a capacity of 2, making it the constraining edge for that path. In Path 3, while e(o,b) and e(c,d) have the same capacity of 3 and 1 respectively, we choose to remove the edge e(c,d) because it is the actual bottleneck for the entire path, having the smallest capacity of all edges in this path. The minimum-cut function identifies the edges that constitute this cut and returns the total capacity of these edges, which is the answer to the problem. In the context of this network, the minimum cut would involve the edges e(a,d) and e(b,d) with capacities of 2, and the edge e(c,d) with a capacity of 1. The aggregate of these capacities 2+2+1=5, is the minimum capacity that needs to be removed to stop all flows from o to d.

```python
# Find the minimum cut
cut_value, partition = nx.minimum_cut(G, "o", "d")

# Calculate the minimum aggregate capacity to remove
min_capacity_to_remove = sum(G[u][v]['capacity'] for u, v in G.edges if u in partition[0] and v in partition[1])
print("Minimum aggregate capacity to remove:", min_capacity_to_remove)

# Display the two sets of nodes that define the minimum cut
print("Minimum cut partition:", partition)

# Highlight the edges in the minimum cut
cut_edges = [(u, v) for u, v in G.edges if u in partition[0] and v in partition[1]]
nx.draw_networkx_edges(G, pos, edgelist=cut_edges, width=4, edge_color='blue')

# Show the plot
plt.show()
```

Figure 5

```
Minimum aggregate capacity to remove: 5
Minimum cut partition: ({'o', 'a', 'c', 'b'}, {'d'})
```

Figure 6

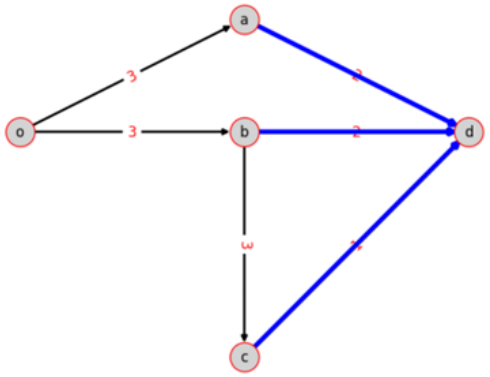

Figure 7

Question 2. What is the maximum aggregate capacity that can be removed from the links without affecting the maximum throughput from o to d? The maximal throughput is returned by maximum flow. The output of the function is a tuple, where: The first element is the value of the maximum flow, which in this case is 5. The second element is a dictionary that represents a flow network, showing the flow between each pair of nodes.

```
[11] nx.algorithms.flow.maximum_flow(G,"o","d")

(5,
 {'o': {'a': 2, 'b': 3},
  'a': {'d': 2},
  'b': {'d': 2, 'c': 1},
  'd': {},
  'c': {'d': 1}})
```

Figure 8

Question 3. You are given $x > y$ extra units of capacity ($x \in z$). 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 ($x \geq 0$).

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

In the given section of the report, the Picos library is utilized for its adept handling of the additional capacity allocation within a graph network. This additional capacity, symbolized by x, represents the supplementary potential for flow along the edges. Initially, the existing capacities are gathered into a dictionary format, creating a structured representation of the graph's limits. Utilizing Picos, these capacities are transformed into mathematical expressions that Picos can operate on, enabling the formulation of a network flow problem. Once the problem is instantiated, we introduce the extra capacity as a variable constraint within the model, ensuring that the sum of the additional capacities distributed across the edges does not exceed x. With the objective function set to maximize the total flow from the source node O to the sink node d, the Picos 'solve()' function is called into action. This function computes the optimal distribution of the extra capacity across the edges, thereby maximizing the overall throughput from O to d. The solution provided by 'solve()' gives us the maximum flow value along with the configuration of how the additional capacity x is allocated throughout the network to achieve this optimal state.

```
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
```

Figure 9

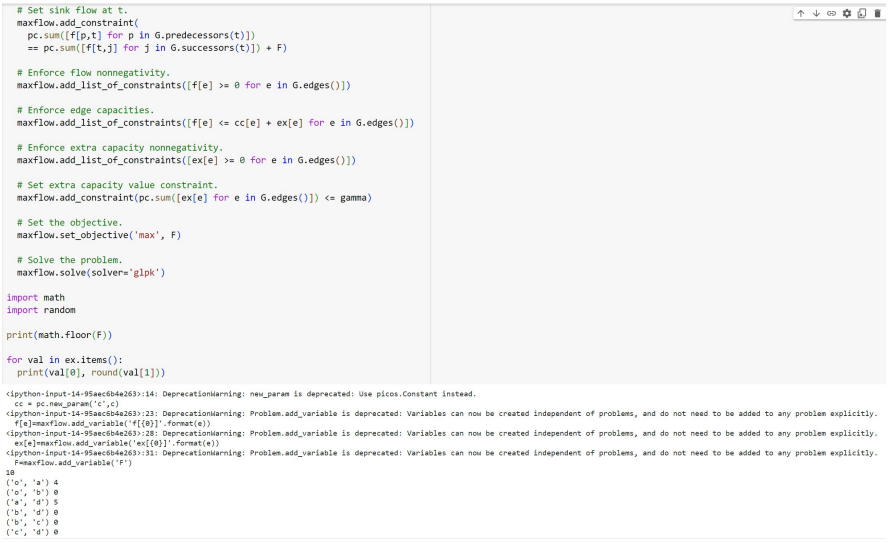

Figure 10

2.2 exercise Two

For the problem described, the main theorem applicable is the Max-Flow Min-Cut Theorem, which in the context of bipartite matching can be used to determine a perfect matching. In graph theory, the concept of matchings in bipartite graphs is closely tied to the max-flow min-cut theorem. A bipartite graph is one where the set of vertices can be divided into two distinct groups, U and V, such that every edge connects a vertex from U to one in V. This structure naturally lends itself to scenarios like matching a group of people to a set of objects, where each edge represents a person's interest in a specific object.

To explore matchings within these graphs, especially to identify a perfect matching where every vertex in one partition is matched to a unique vertex in the other, we can apply the principles of network flow. We transform the bipartite graph G=(V,E) into a flow network G'=(V',E') by adding a source node s and a sink node t, with 'V' including V along with s and t, and 'E' encompassing E as well as additional edges connecting s to every vertex in one partition and every vertex in the other partition to t. In this transformed flow network, we seek a flow from s to t that maximizes the number of matched pairs—this is where the max-flow min-cut theorem becomes instrumental. The theorem posits that the maximum flow through the network from s to t will be equal to the capacity of the minimum cut, which is the smallest number of edges that can be removed to disconnect s from t. For the set of people p1,p2,p3,p4 and books b1,b2,b3,b4, with each person interested in a specific subset of books, a perfect matching would pair each person with one book such that all books are distributed, and each person receives a book they are interested in. To determine if such a perfect matching exists, we apply the max-flow min-cut theorem to the corresponding flow network. If the maximum flow equals the number of people, then a perfect matching is indeed possible, and the flow values along the edges from people to books will indicate the pairs in the perfect matching. If the maximum flow is less than the number of people, then a perfect matching is not possible under the given preferences.

There are a set of people p1,p2,p3,p4 and a set of book b1,b2,b3,b4. Each person is interested in a subset of books, specifically:

$$p1 \rightarrow \{b1, b2\},\ p2 \rightarrow \{b2, b3\}\ ,\ p3 \rightarrow \{b1, b4\},\ p4 \rightarrow \{b1, b2, b4\ \}$$

as you can see in Figure 11 and 12 we visualize our bipartite graph

```python
import networkx as nx
import matplotlib.pyplot as plt

def create_bipartite_graph():
    G = nx.DiGraph()
    edges = [("p1", "b1"), ("p1", "b2"), ("p2", "b2"), ("p2", "b3"),
             ("p3", "b1"), ("p3", "b4"), ("p4", "b1"), ("p4", "b2"), ("p4", "b4")]
    G.add_edges_from(edges)
    return G

def visualize_bipartite_graph(G):
    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='lightblue', with_labels=True, width=2,
            edge_color='black', edgecolors='red', ax=ax)

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

# Create and visualize the bipartite graph
G_bipartite = create_bipartite_graph()
visualize_bipartite_graph(G_bipartite)
```

Figure 11

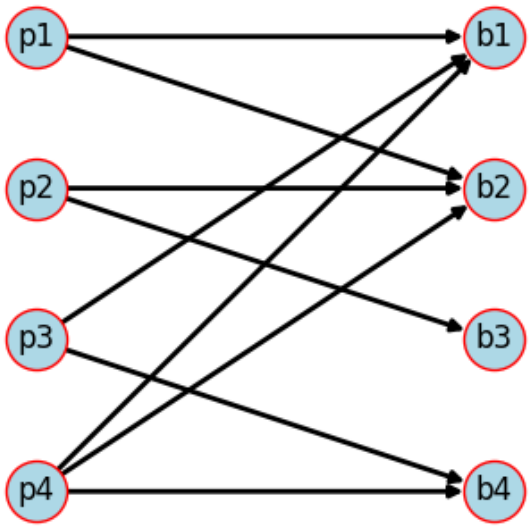

Figure 12

Question 1. Exploit max-flow problems to find a perfect matching (if any).
The code constructs a bipartite graph to match people with books based on their interests using a max-flow algorithm. It creates a flow network, calculates the maximum flow from a source to a sink node, and checks for a perfect matching where each person is paired with a distinct book they are interested in. If a perfect matching is possible, it prints the matched pairs. you can reach the codes and result in Figure 13 and visualized graph in Figure 14.

```python
# Create a directed graph
G = nx.DiGraph()
# Add nodes for the source and sink
G.add_node('s')
G.add_node('t')

# Add edges from source to people
people = ['p1', 'p2', 'p3', 'p4']
for person in people:
    G.add_edge('s', person, capacity=1)

# Define the interests of each person
interests = {
    'p1': ['b1', 'b2'],
    'p2': ['b2', 'b3'],
    'p3': ['b1', 'b4'],
    'p4': ['b1', 'b2', 'b4']
}

# Add edges from people to books they are interested in
for person, books in interests.items():
    for book in books:
        G.add_edge(person, book, capacity=1)

# Add edges from books to sink
books = ['b1', 'b2', 'b3', 'b4']
for book in books:
    G.add_edge(book, 't', capacity=1)

# Calculate the maximum flow
flow_value, flow_dict = nx.maximum_flow(G, 's', 't')

# Check if there's a perfect matching
perfect_matching_exists = flow_value == len(people)
matching = {}

# If there's a perfect matching, extract it
if perfect_matching_exists:
    for person in people:
        for book, flow in flow_dict[person].items():
            if flow > 0:  # If the flow is greater than 0, then the book is matched to the person
                matching[person] = book

# Output the matching
print("Perfect matching exists:", perfect_matching_exists)
print("Matching pairs:")
for person, book in matching.items():
    print(f"{person} -> {book}")



# Define edge colors and node positions
edge_colors = ["black", "black", "black", "black", "black", "red", "black", "red", "red", "black", "black", "black", "red", "black", "black", "black", "black"]
position = {"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], "t": [2, 0.5]}

# Visualize the graph
labels = nx.get_edge_attributes(G, 'capacity')
nx.draw(G, pos=position, with_labels=True, node_size=700, node_color='skyblue', font_size=8, edge_color=edge_colors)
nx.draw_networkx_edge_labels(G, position, edge_labels=labels)

plt.show()
```

```
Perfect matching exists: True
Matching pairs:
p1 -> b2
p2 -> b3
p3 -> b1
p4 -> b4
```

Figure 13

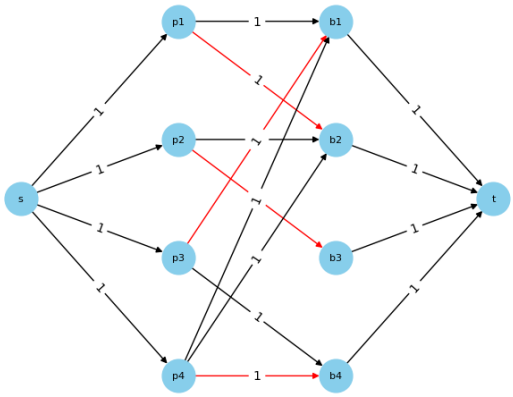

Figure 14

Question 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.

```python
import networkx as nx
import matplotlib.pyplot as plt

# Define the new capacities for the edges from books to sink (number of copies for each book)
book_copies = {'b1': 2, 'b2': 3, 'b3': 2, 'b4': 2}

# Create a new directed graph
G = nx.DiGraph()

# Add nodes for the source and sink
G.add_node('s')
G.add_node('t')

# Add edges from source to people with specified capacities
people_capacities = {'p1': 2, 'p2': 3, 'p3': 2, 'p4': 2}
for person, capacity in people_capacities.items():
    G.add_edge('s', person, capacity=capacity)

# Define the interests of each person
interests = {
    'p1': ['b1', 'b2'],
    'p2': ['b2', 'b3'],
    'p3': ['b1', 'b4'],
    'p4': ['b1', 'b2', 'b4']
}

# Add edges from people to books they are interested in with capacity of 1
for person, books in interests.items():
    for book in books:
        G.add_edge(person, book, capacity=1)

# Add edges from books to sink with the updated capacities
books = ['b1', 'b2', 'b3', 'b4']
for book, copies in book_copies.items():
    G.add_edge(book, 't', capacity=copies)


# Output the total number of books assigned
flow_value, _ = nx.maximum_flow(G, 's', 't')
print("Total number of books that can be assigned:", flow_value)

# Define node positions for better visualization
position = {'s': (-1, 0.5), 't': (2, 0.5), 'p1': (0, 2), 'p2': (0, 1), 'p3': (0, 0), 'p4': (0, -1),
            'b1': (1, 2), 'b2': (1, 1), 'b3': (1, 0), 'b4': (1, -1)}

# Visualize the graph
labels = nx.get_edge_attributes(G, 'capacity')
nx.draw(G, pos=position, with_labels=True, node_size=700, node_color='skyblue', font_size=8, edge_color='black')
nx.draw_networkx_edge_labels(G, position, edge_labels=labels)

plt.show()
```

```
Total number of books that can be assigned: 8
```

Figure 15

A Source (s) and a Sink (t) were defined based on the distribution of the number of copies (2, 3,

2, 2). Our algorithm is nx.maximum_flow(G, 's', 't') which calculates the maximum flow (G, 's', 't'). Thus, this methodology results in the selection of eight books. this graph visualized in Figure 16.

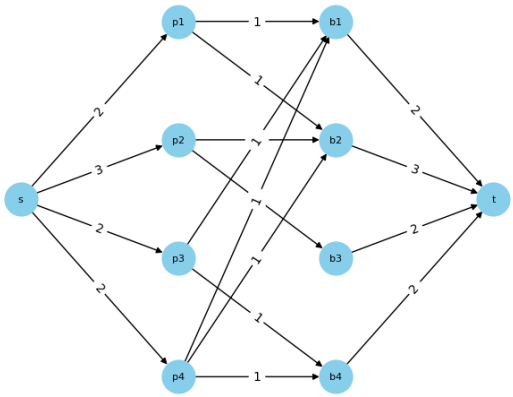

Figure 16

Question 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 ascertain the highest number of books that can be assigned to each individual, we initiate variables to maintain the values for the nodes and edges within the weighted graph, alongside additional variables to capture the inflow and outflow edges. Traversing the network allows us to determine the balance of each node; a positive value indicates a node whose outflow exceeds its inflow, while a negative value signifies the reverse. The variable maxPossibleOutflow is then employed to reveal the maximum value that can be assigned, reflecting the greatest number of books that can be distributed to a person based on the available copies and their preferences.

```python
nodes = G.nodes  # store nodes
edges = G.edges  # store edges

for node_i in nodes:

    inflow = 0  # no of input edge
    outflow = 0  # no of output edge

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

    # person nodes are not considered
    if node_i in ["p1", "p2", "p3", "p4"]:
        continue

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

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

    # print inflow and outflow of node_i (Book nodes)
    print(f"{node_i} inflow: {inflow}")
    print(f"{node_i} outflow: {outflow}")

    # netflow of node_i (Book nodes)
    netflow = inflow - outflow

    # if some nodes have inflow greater or minor than outflow, means that it has not an optimized flow
    if netflow > 0 and node_i != "s" and node_i != "t":
        print(f"1 copy of the book {node_i} can be sold ")
    elif netflow < 0 and node_i != "s" and node_i != "t":
        print(f"{-netflow} copy of the book {node_i} can be bought ")

# print inflow and outflow of the destination node 't'
inflow_t = sum(data['capacity'] for u, v, data in G.in_edges('t', data=True))
outflow_t = sum(data['capacity'] for u, v, data in G.out_edges('t', data=True))
print("t inflow:", inflow_t)
print("t outflow:", outflow_t)

# print the maximum possible outflow
maxPossibleOutflow = max(inflow_t, 0)
print("Max possible outflow:", maxPossibleOutflow)
```

```
t inflow: 9
t outflow: 0
b1 inflow: 3
b1 outflow: 2
1 copy of the book b1 can be sold
b2 inflow: 3
b2 outflow: 3
b3 inflow: 1
b3 outflow: 2
1 copy of the book b3 can be bought
b4 inflow: 2
b4 outflow: 2
t inflow: 9
t outflow: 0
Max possible outflow: 9
```

Figure 17

2.3 exercise Three

We have information about the highway network in Los Angeles, as discussed in the text. An approximate highway map is provided to simplify the problem. The node-link incidence matrix B for this traffic network is available in the file traffic.mat. The rows of B correspond to the nodes of the network, and the columns represent the links. Each node signifies an intersection between highways, and links are denoted as $ei \in e1, ..., e28$. The maximum flow capacity of each link is given by the vector ce in the file capacities.mat. Additionally, the minimum traveling times for each link, denoted as lei, are available in the file traveltime.mat. These values are determined by dividing the length of the highway segment by the assumed speed limit of 60 miles/hour. The delay function for each link is introduced accordingly.

$\Gamma_e(f_e) = \frac{le}{1-\frac{f_e}{c_e}}, \quad 0 \leq f_e < c_e$

For $fe \geq c_e$, the value of Γe(fe) is considered as $+\infty$.

At First, we install dependencies in order to be able to solve the problem. Then we iterate over traffic array, loaded using the provided code, and try to create the graph of the highways. In order to create the previously mentioned graph, we loop over the second and first dimensions of the traffic array respectively. In the case we found the cell value as 1, we consider it as the source node and in we consider the destination node once we find -1 as a cell value during the iteration.

Since index values during the iteration started from 0 and we wanted the first node to have the value of 1, we summed up i and j with 1.

```python
# Import necessary libraries
import scipy.io
import cvxpy as cp
import numpy as np
import networkx as nx

# Load data from mat files
flow = scipy.io.loadmat("flow.mat")["flow"].reshape(28,)
capacities = scipy.io.loadmat("capacities.mat")["capacities"].reshape(28,)
traffic = scipy.io.loadmat("traffic.mat")["traffic"]
traveltime = scipy.io.loadmat("traveltime.mat")["traveltime"].reshape(28,)

# Create a directed graph
G2 = nx.DiGraph()

# Populate the graph with edges and capacities based on traffic data
for i in range(traffic.shape[1]):
    for j in range(traffic.shape[0]):
        if traffic[j][i] == 1:  # 1 -> source
            src = j + 1
        elif traffic[j][i] == -1:  # -1 -> destination
            dst = j + 1
    G2.add_edges_from([(src, dst)], capacity=capacities[i])

# Define positions for nodes in the graph
pos = {1: [-2, 1], 2: [-1, 1], 3: [0.5, 1], 4: [2, 1], 5: [4, 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]}

# Draw the graph using networkx and matplotlib
nx.draw(G2,pos, with_labels=True, node_size=700, node_color='lightgreen', font_size=8, font_color='black', edge_color='darkorange')
```

Figure 18

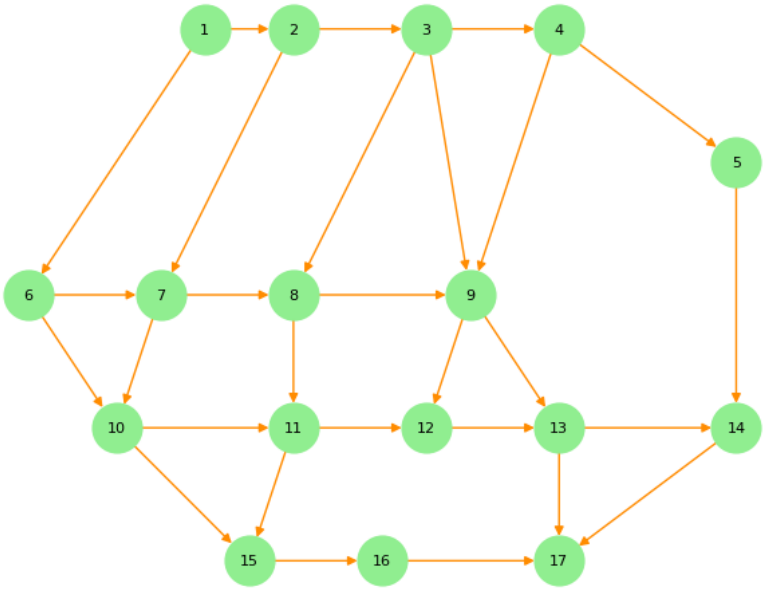

Figure 19

Question 1. In order to find the shortest path between nodes 1 and 17, we used shortest-path method introduced by NetworkX library. This function is designed to compute shortest paths in the graph. It has several parameters namely:

G: NetworkX graph

source: node, optional Starting node for path. If not specified, compute shortest paths for each possible starting node.

target: node, optional Ending node for path. If not specified, compute shortest paths to all possible nodes.

weight: None, string or function, optional (default = None) If None, every edge has weight/distance/cost 1. If a string, use this edge attribute as the edge weight. Any edge attribute not present defaults to 1. If this is a function, the weight of an edge is the value returned by the function. The function must accept exactly three positional arguments: the two endpoints of an edge and the dictionary of edge attributes for that edge. The function must return a number.

method: string, optional (default = 'dijkstra') The algorithm to use to compute the path. Supported options: 'dijkstra', 'bellman-ford'. Other inputs produce a ValueError. If weight is None, unweighted graph methods are used, and this suggestion is ignored. In result part, you can see the array of the shortest path.

```
# Find the shortest path in the graph G2 from source node 1 to target node 17 based on travel time as the weight
shortestPath = nx.shortest_path(G2, source=1, target=17, weight='traveltime')

# Print the result
print("Shortest path from node 1 to node 17: ", shortestPath)
```

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

Figure 20

Question 2. To find the maximum flow between nodes 1 and 17, we used maximum_flow method introduced by NetworkX library. This function is designed to Find a maximum single-commodity flow. It has several parameters which are described in the following:

flowG: NetworkX graph Edges of the graph are expected to have an attribute called 'capacity'. If this attribute is not present, the edge is considered to have infinite capacity.

_s: node Source node for the flow.

_t: node Sink node for the flow.

capacity: string Edges of the graph G are expected to have an attribute capacity that indicates how much flow the edge can support. If this attribute is not present, the edge is considered to have infinite capacity. Default value: 'capacity'.

flow_func: function A function for computing the maximum flow among a pair of nodes in a capacitated graph. The function has to accept at least three parameters: a Graph or Digraph, a source node, and a target node. And return a residual network that follows NetworkX conventions (see Notes). If flow_func is None, the default maximum flow function (preflow_push()) is used. See below for alternative algorithms. The choice of the default function may change from version to version and should not be relied on. Default value: None.

kwargs Any other keyword parameter is passed to the function that computes the maximum flow.

In result part, we have a number and a disctionary printed. As NetworkX explains in its document, They stand for:

flow_value: integer, float Value of the maximum flow, i.e., net outflow from the source.

flow_dict: dict A dictionary containing the value of the flow that went through each edge.

```
# Calculate the maximum flow in the graph G2 from source node 1 to target node 17
maxFlow = nx.algorithms.flow.maximum_flow(G2, 1, 17)

# Print the result
print("Maximum flow from node 1 to node 17: ", maxFlow)
```

Figure 21

the result is Maximum flow from node 1 to node 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: 10355, 14: 17: 3835, 10: 11: 825, 15: 8258, 11: 12: 825, 15: 0, 15: 16: 8258, 12: 13: 7893, 17: , 16: 17: 8258)

Question 3. In this part, we aim to minimize travel time subject to flow conservation constraints. In order to solve this part, we used CVXPY library. The goal is to find the external inflow vector V that satisfies the equation Bf=V, given a flow vector f and a matrix B. At first define variables and matrices. B is the matrix representing traffic flow, f is the flow vector to be optimized, exogenousFlow is a predefined vector representing exogenous inflow. In this case, there is unitary inflow from node 1 to node 17 and -1 inflow from node 17 to node 1. Then we define the optimization problem. We have objective that minimize the travel time, which is represented by the dot product of the transposed traveltime vector and the flow variable f. then we have constraints that are 2: f ¿= 0: Ensures that each flow value is non-negative. B @ f == exogenousFlow: Ensures that the flow on each edge, represented by the vector f, satisfies a linear equation determined by the matrix B. This is followed by cp.Problem(objective, constraints).solve() that solves the convex optimization problem using CVXPY. In the result section, the optimal flow vector is then printed, along with the corresponding external inflow vector.

```python
# Define the basic variables and matrices for the optimization problem
B = traffic  # B matrix representing the network structure
f = cp.Variable(B.shape[1])  # Variable representing the flow on each edge

# Define exogenous flow vector - unitary inflow from node 1 to node 17
exogenousFlow = np.array([1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1])

# Define the travel time vector
traveltime = np.array(traveltime)

# Define the optimization objective - minimize the total travel time
objective = cp.Minimize(traveltime.T @ f)  # Linear combination of the flow values

# Define the constraints for the optimization problem
constraints = [f >= 0, B @ f == exogenousFlow]  # Non-negativity constraint and flow balance constraint

# Solve the optimization problem
cp.Problem(objective, constraints).solve()

# Print the results
print("Optimal value of f:", f.value)  # Optimal flow values
print("External inflow:", B @ flow)  # Total flow on each edge (external inflow)
```

```
Optimal value of f: [1.00000000e+00 1.00000000e+00 1.19595586e-11 5.07129566e-12
 1.49988426e-10 1.33658919e-10 8.54100951e-11 3.78266693e-11
 9.99999998e-01 8.90389208e-12 3.91753215e-11 1.00000000e+00
 6.88981434e-12 5.07752202e-12 1.62878397e-11 5.42208640e-11
 1.91831469e-11 5.71274798e-11 8.67469798e-11 1.99611946e-09
 1.29358670e-10 2.12547935e-09 4.14939343e-11 1.16133834e-11
 1.00000000e+00 4.65871714e-11 3.08330688e-11 3.08494486e-11]
External inflow: [ 16282   9094  19448   4957   -746   4768    413     -2  -5671   1169
     -5  -7131   -380  -7412  -7810  -3430 -23544]
```

Figure 22

Question 4. In this section, we try to find the optimal flow on each edge in a network to minimize the given cost function while satisfying constraints on flow direction, capacity, and non-negativity. At first we define exogenous inflow vector as exogenousInflow. This vector represents the external inflow at each node in the network. In this case, there is an inflow of 16282 units at node 1 and an outflow of the same magnitude at node 17. The rest of the nodes have zero external inflow. Then we define decision variable f = cp.Variable(B.shape[1]). It represents the flow on each edge in the network. The number of elements in f is determined by the number of columns in matrix B. In the following we have cost function defined as costFunction. It is the objective function that the optimization is trying to minimize. It is a combination of travel time, capacities, and the flow variable f, which is given in the question. Then we define constraints. Three constraints are defined: f ¿= 0: Ensures that the flow on each edge is non-negative. B f == exogenousInflow: Ensures that the flow on each edge, when multiplied by matrix B, equals the specified external inflow vector. f ¡= capacities: Restricts the flow on each edge to be less than or equal to its capacity. Afterward, we solve the problem using CVXPY. The objective is to minimize the costFunction subject to the defined constraints. In the result section, we print the optimal flow on each edge in the network, as well as the optimal cost value.

```python
# Define the exogenous inflow vector
exogenousInflow = np.array([16282, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -16282])

# Define the flow variable representing the flow on each edge in the network
f = cp.Variable(B.shape[1])

# Define the cost function to be minimized
costFunction = cp.sum(
    cp.multiply(traveltime * capacities, cp.inv_pos(1 - cp.multiply(f, 1 / capacities))) - traveltime * capacities
)
# The cost function is a combination of travel time, capacities, and flow on each edge

# Define constraints for the optimization problem
# B @ f == exogenousInflow ensures that the flow on each edge, when multiplied by matrix B, equals the specified external inflow vector.
# f <= capacities restricts the flow on each edge to be less than or equal to its capacity.
# f >= 0 enforces that the flow on each edge is non-negative.
constraints = [f >= 0, B @ f == exogenousInflow, f <= capacities]

# Solve the optimization problem to find the optimal flow and cost
cost_optimal = cp.Problem(cp.Minimize(costFunction), constraints).solve()

# Print the results
print("Optimal flow:", f.value)
print("Optimal cost:", cost_optimal)
```

```
Optimal flow: [6.45410299e+03 5.91939993e+03 2.99529861e+03 2.99529847e+03
 9.82789701e+03 4.49707774e+03 2.89742668e+03 2.43599052e+03
 3.04732920e+03 5.34703061e+02 2.00406444e-03 2.92409932e+03
 1.43557705e-04 2.99529847e+03 5.33081927e+03 2.73087296e+03
 4.73430043e+03 2.13435412e+03 4.61438170e+02 2.31276078e+03
 3.19231098e+03 5.50507176e+03 2.31052058e+03 1.46285622e-04
 6.24188038e+03 5.30581904e+03 4.73430057e+03 4.73430057e+03]
Optimal cost: 23835.484474547633
```

Figure 23

Question 5. Wardrop Equilibrium Optimization In this section, we aim to determine the optimal flow distribution in the transportation network based on Wardrop equilibrium principles. The Wardrop equilibrium represents a state in which no driver can unilaterally decrease their travel time by choosing an alternate route. The optimization problem is formulated using the CVXPY library.

Cost Function for Wardrop Equilibrium The cost function for Wardrop equilibrium is defined as follows:

$$costFunction2 = \sum -(traveltime \cdot \text{capacities} \cdot \log(1 - \tfrac{\text{flow}}{\text{capacities}}))$$

This function captures the essence of Wardrop equilibrium in transportation networks, where the logarithmic term penalizes deviations from free-flow travel conditions.

Optimization Constraints Similar to the previous sections, the optimization problem is subject to the following constraints:

1. $f \geq 0$:Ensures non-negativity of the flow on each edge.
2. B.f = exogenousInflow: Imposes conservation of flow based on
3. $f \leq capacities$: Restricts the flow on each edge to be within its specified capacity. The printed result represents the optimal flow distribution at Wardrop equilibrium in the transportation network

```python
# Define the cost function for Wardrop equilibrium
costFunction2 = cp.sum(-cp.multiply(traveltime * capacities, cp.log(1 - (cp.multiply(f, 1 / capacities)))))
# The cost function represents a form of Wardrop equilibrium in transportation networks

# Define constraints for the optimization problem
# Similar to the previous code, ensuring non-negativity, external inflow, and capacity constraints
constraints2 = [f >= 0, B @ f == exogenousInflow, f <= capacities]

# Solve the optimization problem to find the optimal flow and cost for Wardrop equilibrium
cost_optimal2 = cp.Problem(cp.Minimize(costFunction2), constraints2).solve()

# Print the result
print("Wardrop equilibrium flow:", f.value)
```

```
Wardrop equilibrium flow: [6.53248293e+03 6.53248212e+03 2.20678784e+03 2.20678769e+03
 9.74951700e+03 4.49336733e+03 2.71426833e+03 2.20386170e+03
 3.34278606e+03 8.04549658e-04 1.88858723e+02 4.13683556e+03
 1.44638393e-04 2.20678769e+03 5.25614967e+03 2.24738306e+03
 4.78786641e+03 1.77909980e+03 6.99265353e+02 2.99791135e+03
 2.94664821e+03 5.94455956e+03 2.50794044e+03 2.03543194e-04
 6.77940517e+03 4.71472814e+03 4.78786662e+03 4.78786662e+03]
```

Figure 24

Question 6.

Wardrop Equilibrium Optimization with Tolls. In this section, we extend our analysis to consider the impact of tolls on the transportation network. A toll function is introduced, and the optimization problem is adjusted to incorporate tolls into the cost function. The objective is to determine the optimal flow distribution at Wardrop equilibrium, accounting for the additional cost imposed by tolls.

Toll Function Definition. A toll function is defined, which is currently set to return the input unchanged. The toll function can be customized to model various tolling strategies or policies.

Calculate Tolls Based on Wardrop Equilibrium Flow Tolls are calculated based on the optimal flow values obtained from the Wardrop equilibrium.

Cost Function Adjustment with Tolls. The cost function is modified to include the tolls on each link. The new cost function is defined as follows:

$$costFunction_{\text{with\_tolls}} = \sum -(traveltime \cdot \text{capacities} \cdot \log(1 - \frac{flow}{\text{capacities}})) + \text{Tolls}$$

Optimization Constraints with Tolls. Similar to the previous sections, the optimization problem is subject to constraints ensuring non-negativity of flow on each edge, conservation of flow based on the external inflow vector, and capacity constraints.

Solve for Optimal Wardrop Equilibrium Flow with Tolls. The optimization problem, now incorporating tolls, is solved using the CVXPY library. The resulting optimal flow distribution at Wardrop equilibrium, along with tolls on each link and the new delay on each link, is printed for further analysis. The printed results provide insights into the optimal flow distribution considering the introduction of tolls in the transportation network.

```python
# Define a toll function, which is currently set to return the input unchanged
def toll_function(fe):
    return fe

# Calculate tolls based on the optimal flow values from the Wardrop equilibrium
tolls = toll_function(f.value)

# Define the cost function with tolls
costFunction_with_tolls = cp.sum(
    -cp.multiply(traveltime * capacities, cp.log(1 - (cp.multiply(f, 1 / capacities)))) + tolls
)
# The cost function now includes the tolls on each link

# Define constraints for the optimization problem with tolls
# Similar to previous constraints, ensuring non-negativity, external inflow, and capacity constraints
constraints_with_tolls = [f >= 0, B @ f == exogenousInflow, f <= capacities]

# Solve the optimization problem with tolls to find the new Wardrop equilibrium flow
wardrop_equilibrium_flow_with_tolls = cp.Problem(cp.Minimize(costFunction_with_tolls), constraints_with_tolls).solve()

# Print the results
print("Wardrop equilibrium flow with tolls:", f.value)
print("Tolls on each link:", tolls)
print("New delay on each link:", traveltime + tolls)
```

```
Wardrop equilibrium flow with tolls: [6.53248293e+03 6.53248212e+03 2.20678784e+03 2.20678769e+03
 9.74951700e+03 4.49336733e+03 2.71426833e+03 2.20386170e+03
 3.34278606e+03 8.04549658e-04 1.88858723e+02 4.13683556e+03
 1.44638393e-04 2.20678769e+03 5.25614967e+03 2.24738306e+03
 4.78786641e+03 1.77909980e+03 6.99265353e+02 2.99791135e+03
 2.94664821e+03 5.94455956e+03 2.50794044e+03 2.03543194e-04
 6.77940517e+03 4.71472814e+03 4.78786662e+03 4.78786662e+03]
Tolls on each link: [6.53248293e+03 6.53248212e+03 2.20678784e+03 2.20678769e+03
 9.74951700e+03 4.49336733e+03 2.71426833e+03 2.20386170e+03
 3.34278606e+03 8.04549658e-04 1.88858723e+02 4.13683556e+03
 1.44638393e-04 2.20678769e+03 5.25614967e+03 2.24738306e+03
 4.78786641e+03 1.77909980e+03 6.99265353e+02 2.99791135e+03
 2.94664821e+03 5.94455956e+03 2.50794044e+03 2.03543194e-04
 6.77940517e+03 4.71472814e+03 4.78786662e+03 4.78786662e+03]
New delay on each link: [6.53262876e+03 6.53252695e+03 2.20691684e+03 2.20697436e+03
 9.74964683e+03 4.49344566e+03 2.71436233e+03 2.20391903e+03
 3.34292773e+03 1.05974550e-01 1.88965393e+02 4.13694039e+03
 1.12474638e-01 2.20696819e+03 5.25627117e+03 2.24746373e+03
 4.78794175e+03 1.77915647e+03 6.99298020e+02 2.99794635e+03
 2.94671488e+03 5.94467239e+03 2.50801878e+03 5.43705432e-02
 6.77950101e+03 4.71480114e+03 4.78803162e+03 4.78801529e+03]
```

Figure 25

Question 7.

System Optimum and Wardrop Equilibrium with Constructed Tolls. In this section, we explore the concept of system optimum and its relationship with Wardrop equilibrium. The system optimum aims to minimize the total travel time across the transportation network. Additionally, we introduce a toll vector w* such that the Wardrop equilibrium flow coincides with the system optimum flow.

Cost Function for System Optimum. The cost function for the system optimum is defined as:

$$costFunction_{\text{system\_optimum}} = \sum(flow \cdot (traveltime - traveltime[0]))$$

This cost function represents the system optimum, aiming to minimize the total travel time across the network.

Optimization Constraints for System Optimum. Similar to previous sections, the optimization problem for the system optimum is subject to constraints ensuring non-negativity of flow on each edge, conservation of flow based on the external inflow vector, and capacity constraints.

Solve for System Optimum Flow. The optimization problem for the system optimum is solved using the CVXPY library, resulting in the optimal flow distribution.

Construct Tolls for Wardrop Equilibrium. A toll vector w* is constructed such that the Wardrop equilibrium flow coincides with the system optimum flow. The cost function is updated to include tolls for the new Wardrop equilibrium. The optimization problem for the new Wardrop equilibrium, incorporating constructed tolls, is solved. The optimal flow for the system optimum, the constructed tolls, and the Wardrop equilibrium flow with constructed tolls are printed for further analysis. The printed results provide insights into the relationship between system optimum and Wardrop equilibrium, considering the introduction of tolls.

```
# Define the cost function for the system optimum
costFunction_system_optimum = cp.sum(cp.multiply(f, traveltime - traveltime[0]))
# The cost function represents the system optimum, aiming to minimize the total travel time

# Define constraints for the optimization problem for the system optimum
constraints_system_optimum = [f >= 0, B @ f == exogenousInflow, f <= capacities]

# Solve the optimization problem to find the system optimum flow
system_optimum_flow = cp.Problem(cp.Minimize(costFunction_system_optimum), constraints_system_optimum).solve()

# Construct a toll vector w* such that the Wardrop equilibrium f(w) coincides with f*
tolls_system_optimum = traveltime - traveltime[0]

# Update the cost function with tolls for the new Wardrop equilibrium
costFunction_with_tolls_system_optimum = cp.sum(
    -cp.multiply(traveltime * capacities, cp.log(1 - (cp.multiply(f, 1 / capacities)))) + tolls_system_optimum
)

# Solve the optimization problem for the new Wardrop equilibrium with tolls
wardrop_equilibrium_flow_with_tolls_system_optimum = cp.Problem(
    cp.Minimize(costFunction_with_tolls_system_optimum), constraints_with_tolls
).solve()

# Display results
print("System optimum flow (f*):", f.value)
print("Tolls for the system optimum (w*):", tolls_system_optimum)
print("Wardrop equilibrium flow with constructed tolls (f(w*)): ", f.value)
```

```
System optimum flow (f*): [6.53248293e+03 6.53248212e+03 2.20678784e+03 2.20678769e+03
 9.74951700e+03 4.49336733e+03 2.71426833e+03 2.20386170e+03
 3.34278606e+03 8.04549658e-04 1.88858723e+02 4.13683556e+03
 1.44638393e-04 2.20678769e+03 5.25614967e+03 2.24738306e+03
 4.78786641e+03 1.77909980e+03 6.99265353e+02 2.99791135e+03
 2.94664821e+03 5.94455956e+03 2.50794044e+03 2.03543194e-04
 6.77940517e+03 4.71472814e+03 4.78786662e+03 4.78786662e+03]
Tolls for the system optimum (w*): [ 0.       -0.100997 -0.01683   0.04084  -0.016    -0.067497 -0.05183
 -0.088497 -0.00416  -0.04066  -0.03916  -0.041    -0.0335    0.03467
 -0.02433  -0.065163 -0.070497 -0.089163 -0.113163 -0.11083  -0.079163
 -0.033    -0.067497 -0.091663 -0.049997 -0.07283   0.01917   0.00284 ]
Wardrop equilibrium flow with constructed tolls (f(w*)):  [6.53248293e+03 6.53248212e+03 2.20678784e+03 2.20678769e+03
 9.74951700e+03 4.49336733e+03 2.71426833e+03 2.20386170e+03
 3.34278606e+03 8.04549658e-04 1.88858723e+02 4.13683556e+03
 1.44638393e-04 2.20678769e+03 5.25614967e+03 2.24738306e+03
 4.78786641e+03 1.77909980e+03 6.99265353e+02 2.99791135e+03
 2.94664821e+03 5.94455956e+03 2.50794044e+03 2.03543194e-04
 6.77940517e+03 4.71472814e+03 4.78786662e+03 4.78786662e+03]
```

Figure 26

Shaghayegh Abedi and Nastaran Ahmadi Bonakdar solved exercise 1 with help of Hesam Khanjani

Shaghayegh Abedi and Hesam Khanjani solved exercise 2 with help of Ali Samimi Fard

Nastaran Ahmadi Bonakdar and Mohammadreza Mashhadigholamali and Ali Samimi Fard solved exercise 3

Ali Samimi Fard and Hesam Khanjani wrote the report in LaTeX with help of Mohammadreza Mashhadigholamali