Network Dynamics and Learning

Seyed Mohammad Sheikh Ahmadi - s327914
Ehsan Dashti - s316511

November 12, 2024

1 Exercise 1

This exercise focuses on analyzing a flow network with given link capacities to determine the minimum cut that stops flow from o to d, optimizing throughput by distributing additional capacity $x > 0$, and finding the best placement and distribution for an added directed link to maximize throughput. It involves calculating cuts, distributing extra capacity, and plotting the maximum throughput as functions of x for network optimization.

1.1 Preparation

1.1.1 Importing Dependencies

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

1.1.2 Defining the Graph

```
G = nx.DiGraph()
edges = [
    ('o', 'a', 3,'e1'),  # e1
    ('a', 'd', 2,'e2'),  # e2
    ('o', 'b', 3,'e3'),  # e3
    ('b', 'd', 2,'e4'),  # e4
    ('b', 'c', 3,'e5'),  # e5
    ('c', 'd', 1,'e6'),  # e6
    ('a', 'b', 1,'e7')   # e7
]
for u, v, weight, label in edges:
    G.add_edge(u, v, weight=weight, label=label)
```

1.1.3 Plotting the Graph

```
## defining the positions of each nodes
pos = {
    'o': (-1, 0),
    'a': (0, 1),
    'b': (0, 0),
    'c': (0, -1),
    'd': (1, 0)
}
# Draw the nodes and edges
plt.figure(figsize=(4, 3))

nx.draw(G, pos, with_labels=True, node_size=700,
        node_color="lightblue",
        font_size=10,
        font_weight="bold")

# Draw edge labels for capacities
edge_labels = {(u, v): f"{d['label']} ({d['weight']})"
                for u, v, d in G.edges(data=True)}

nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
                             font_size=10)
## Plotting the Graph
plt.title("Network Flow Graph with Capacities")
plt.savefig('ex1')
plt.show()
```

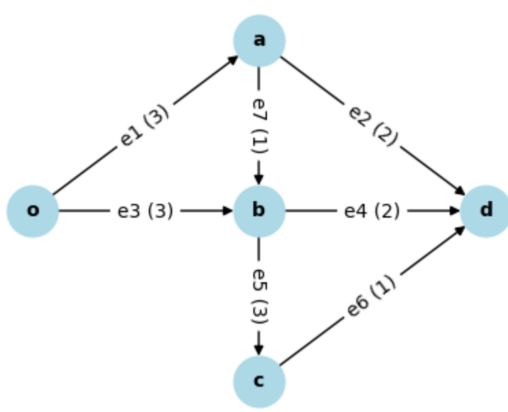

Figure 1: Directed Graph Representing For Exercise 1

1.2 Part A

1.2.1 Computation of the Minimum Cut

```
cut_value, partition = nx.minimum_cut(G, 'o', 'd',
                                      capacity='weight')

print(f"The minimum cut capacity is: {cut_value}")
print(f"partitions: {partition}")
```

1.2.2 Results

1. **Cut 1:** $\{o\}$ and $\{a, b, c, d\}$
   Crossing edges: $e1$ and $e3$
   Capacity: $3 + 3 = 6$

2. **Cut 2:** $\{o, a\}$ and $\{b, c, d\}$
   Crossing edges: $e2, e3$ and $e7$
   Capacity: $2 + 3 + 1 = 6$

3. **Cut 3:** $\{o, b\}$ and $\{a, c, d\}$
   Crossing edges: $e1, e4$, and $e5$
   Capacity: $3 + 2 + 3 = 8$

4. **Cut 4:** $\{o, a, b\}$ and $\{c, d\}$
   Crossing edges: $e2, e4$, and $e5$
   Capacity: $2 + 2 + 3 = 7$

5. **Cut 5:** $\{o, b, c\}$ and $\{a, d\}$
   Crossing edges: $e1, e4$, and $e6$
   Capacity: $3 + 2 + 1 = 6$

6. **Cut 6:** $\{o, a, b, c\}$ and $\{d\}$
   Crossing edges: $e2, e4$, and $e6$
   Capacity: $2 + 2 + 1 = 5$

1.2.3 Minimum Cut

The minimum capacity cut among these is **Cut 6**, with a capacity of **5**.

1.2.4 Conclusion

The minimum capacity that needs to be removed to prevent any feasible flow from o to d is **5** units.

1.3 Part B

1.3.1 Defining a Function to Compute Max-Flow by Increasing X

We should add extra capacity to the edges with min cut value, but here we can add all edges x unit and there wouldn't be any differences.

```
def max_flow_with_extra_capacity(G, x):
    G_temp = G.copy()
    for u, v in G.edges():
        G_temp[u][v]['weight'] += x
    flow_value,_ = nx.maximum_flow(G_temp, 'o', 'd',
                                   capacity='weight')
    print("max flow with x =  ",x," is ",flow_value)
    return flow_value
```

1.3.2 Generate Data for Different Values of X

```
x_values = list(range(0, 11))  # for x from 0 to 10
flow_values = [max_flow_with_extra_capacity(G, x)
               for x in x_values]
```

1.3.3 Plotting The Result

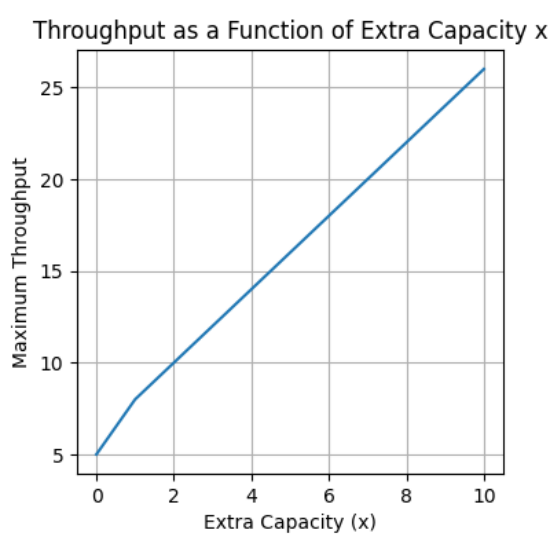

Figure 2: Max Throughput from o to d as a Function of Extra Capacity x

1.4 Part C

1.4.1 Defining New Edges for Considering as Possible Edge

```
new_edges = [('o', 'd'), ('o', 'c'), ('a', 'c')]
```

1.4.2 Defining a Function for Calculating All the Possible Ways

```
def max_flow_with_new_edge(G, edge, x):
    G_temp = G.copy()
    G_temp.add_edge(*edge, weight=1 + x)
    flow_value, _ = nx.maximum_flow(G_temp, 'o', 'd',
                                    capacity="weight")
    return flow_value
```

1.4.3 Generate Data for Different Values of X

```
x_values = list(range(0, 11))  # for x from 0 to 10
results = {edge: [max_flow_with_new_edge(G, edge, x)
           for x in x_values] for edge in new_edges}
```

1.4.4 Plotting The Result

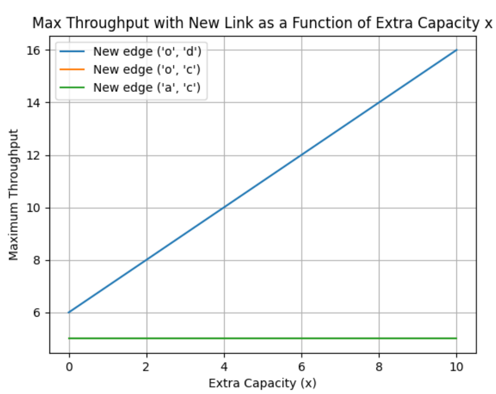

Figure 3: Max Throughput with New Link as a Function of Extra Capacity X

2 Exercise 2

This exercise involves a bipartite matching problem modeled as a max-flow network. Given a set of people $\{a_1, a_2, a_3, a_4\}$ and foods $\{b_1, b_2, b_3, b_4\}$, each person has specific food preferences. Part (a) asks for a perfect matching using max-flow. Part (b) introduces multiple food portions and explores how many can be assigned in total. Part (c) changes the portion demands for individuals and uses max-flow to find the maximum total allocation. This exercise illustrates the application of max-flow in solving matching and distribution problems.

2.1 Preparation

2.1.1 Defining the Graph

```
G = nx.DiGraph()
# Add nodes for people and foods
people = ['a1', 'a2', 'a3', 'a4']
foods = ['b1', 'b2', 'b3', 'b4']
# Define person to food interests
interests = {
    'a1': ['b1', 'b2'],
    'a2': ['b2', 'b3'],
    'a3': ['b1', 'b4'],
    'a4': ['b1', 'b2', 'b4']
}
# Add edges from people to foods with capacity 1
for person, food_list in interests.items():
    for food in food_list:
        G.add_edge(person, food, capacity=1)
```

2.1.2 Plotting the Original Graph

```
plt.figure(figsize=(5, 4))
# Position the nodes using a bipartite layout
pos = {}
# Left side for people
pos.update((node, (0, i)) for i, node in enumerate(people))
# Right side for foods
pos.update((node, (1, i)) for i, node in enumerate(foods))

nx.draw_networkx(G, pos, with_labels=True, node_size=1000,
                 node_color='lightblue',font_size=10,
                 edge_color='gray')
```

```python
# Draw edge labels showing the capacity
edge_labels = {(u, v): f"{d['capacity']}"
               for u, v, d in G.edges(data=True)}

nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels,
                            font_size=8)

plt.title("Directed Graph Representing People and Food Interests")
plt.savefig('ex2_0')
plt.show()
```

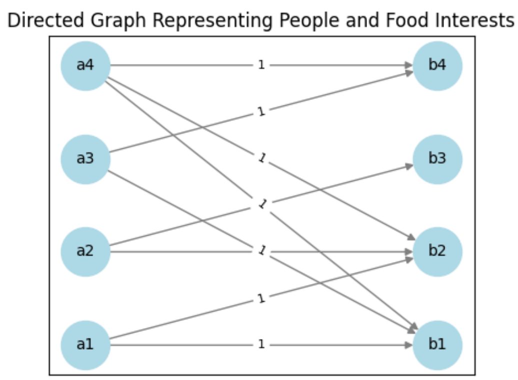

Figure 4: Directed Graph Representing People and Food Interests

2.1.3 Source and Sink

for using max flow we should add a source as S then connect it to all people with capacity 1, Also we need a sink as T to connect to all foods with capacity 1

```python
source = 'S'
sink = 'T'


# Add edges from source to people with capacity 1
for person in people:
    G.add_edge(source, person, capacity=1)

# Add edges from foods to sink with capacity 1
for food in foods:
    G.add_edge(food, sink, capacity=1)
```

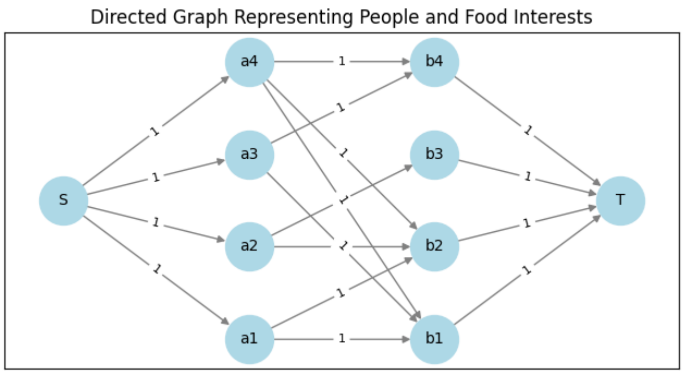

Figure 5: Directed Graph Representing People and Food Interests With Source and Sink

2.2 Part A

2.2.1 Finding a Perfect Matching

```
flow_value, flow_dict = nx.maximum_flow(G, source, sink)

print("Maximum Flow Value:", flow_value)
print("Flow Details:")
for key, value in flow_dict.items():
    if(key.startswith('a')):
        print(key, "->", value)
```

2.2.2 Result

Maximum Flow Value: 4
Flow Details:

$$\begin{aligned} a_1 &\to \{b_1 : 0,\ b_2 : 1\} \\ a_2 &\to \{b_2 : 0,\ b_3 : 1\} \\ a_3 &\to \{b_1 : 1,\ b_4 : 0\} \\ a_4 &\to \{b_1 : 0,\ b_2 : 0,\ b_4 : 1\} \end{aligned}$$

2.3 Part B

For this part, we need to change the capacity of edges:

2.3.1 Updating the Capacities

- Update the food-sink edges based on the number of existing portions.
- The capacity of the person-to-food edges should be set to 1 because people cannot take more than 1 portion of each food.
- The capacity of source-to-person edges should be infinite, as each person can take an arbitrary number of different foods.

```
G['b1']['T']['capacity'] = 2
G['b2']['T']['capacity'] = 3
G['b3']['T']['capacity'] = 2
G['b4']['T']['capacity'] = 2
G['S']['a1']['capacity'] = float('inf')
G['S']['a2']['capacity'] = float('inf')
G['S']['a3']['capacity'] = float('inf')
G['S']['a4']['capacity'] = float('inf')
```

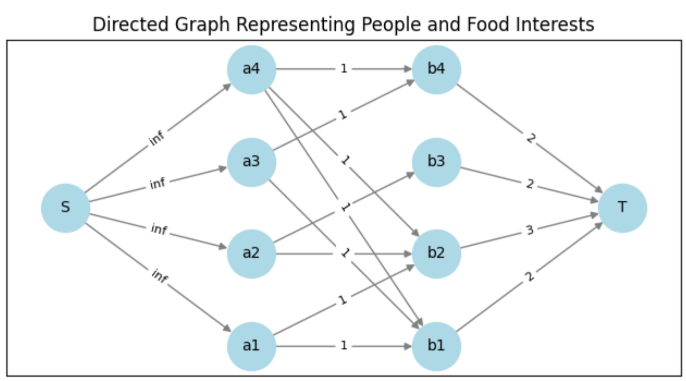

Figure 6: Updated Capacities Based on Exercise.

2.3.2 Result

Maximum Flow Value: 8

Flow Details:

$$
\begin{aligned}
a_1 &\to \{b_1 : 0,\ b_2 : 1\} \\
a_2 &\to \{b_2 : 1,\ b_3 : 1\} \\
a_3 &\to \{b_1 : 1,\ b_4 : 1\} \\
a_4 &\to \{b_1 : 1,\ b_2 : 1,\ b_4 : 1\}
\end{aligned}
$$

2.4 Part C

For this part, we need to change the capacity of edges:

2.4.1 Updating the Capacities

- This part we update the capacity of People-food to infinite because people can have more than 1 portion from each food.
- Also the capacity of source-people should be based on demand number of each person.

```
for person, food_list in interests.items():
    for food in food_list:
        G.add_edge(person, food, capacity=float('inf'))
G['S']['a1']['capacity'] = 3
G['S']['a2']['capacity'] = 2
G['S']['a3']['capacity'] = 2
G['S']['a4']['capacity'] = 2
```

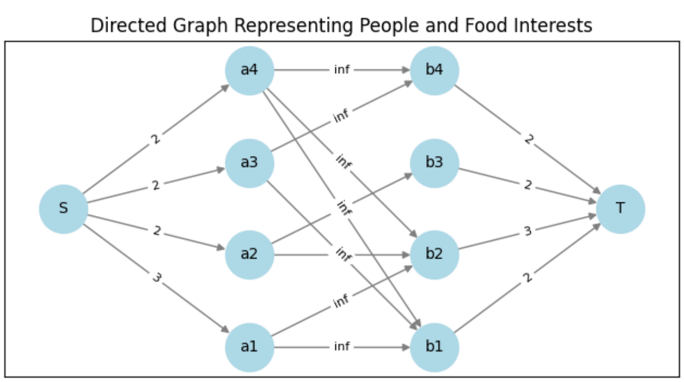

Figure 7: Updated Capacities Based on Exercise.

2.4.2 Result

Maximum Flow Value: 9
Flow Details:

$$a_1 \to \{b_1 : 0,\ b_2 : 3\}$$
$$a_2 \to \{b_2 : 0,\ b_3 : 2\}$$
$$a_3 \to \{b_1 : 2,\ b_4 : 0\}$$
$$a_4 \to \{b_1 : 0,\ b_2 : 0,\ b_4 : 2\}$$

3 Exercise 3

3.1 Part A

3.1.1 Travel Time Graph

Here we should create a graph that weight of edges are travel time.

```
G = nx.DiGraph()

num_nodes = B.shape[0]   # Number of nodes in the network
num_links = B.shape[1]   # Number of links in the network


for i in range(num_links):
    tail_node = np.where(B[:, i] == 1)[0][0] + 1
    head_node = np.where(B[:, i] == -1)[0][0] + 1
    travel_time = l[i]

    G.add_edge(tail_node, head_node, weight=travel_time)
```

3.1.2 Compute the Fastest Path

```
shortest_path = nx.dijkstra_path(G, source=1,
                                 target=17,
                                 weight='weight')

shortest_path_time = nx.dijkstra_path_length(G, source=1,
                                             target=17,
                                             weight='weight')
```

3.1.3 Results

Shortest Path: $\{1, 2, 3, 9, 13, 17\}$
Travel Time: 0.56

3.2 Part B

3.2.1 Capacity Graph

Here we should create a graph that weight of edges are Capacity.

```
G_flow = nx.DiGraph()

for link_index in range(num_links):
```

```
    tail_node = np.where(B[:, link_index] == 1)[0][0] + 1
    head_node = np.where(B[:, link_index] == -1)[0][0] + 1
    capacity = C[link_index]

    G_flow.add_edge(tail_node, head_node, capacity=capacity)
```

3.2.2 Compute the Max-Flow

```
max_flow_value, max_flow_dict = nx.maximum_flow(G_flow, 1, 17)
```

3.2.3 Results

Maximum Flow Value: 22448

3.3 Part C

Here we should compute dot product of B and f.

```
nu = np.dot(B, f)
## or
nu = B @ f
```

3.3.1 Results

Resulting vector v: $\{16282, 9094, 19448, 4957, -746, 4768, 413, -2, -5671, 1169, -5, -7131, -380, -7412, -7810, -3430, -23544\}$

3.4 Updating nu

```
nu_modified = np.zeros_like(nu)
nu_modified[0] = nu[0]
nu_modified[-1] = -nu[0]

nu = nu_modified
```

3.5 Part D

3.5.1 Definition of Variables and Cost Function and Constrains

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

```
f__Variable = cp.Variable(28)


lc = cp.multiply(l, C)   ## l*c
f_c= f__Variable/C       ## f/c
cost_function = cp.sum(cp.multiply(lc, cp.inv_pos(1 - f_c)) - lc)

constraints = [
    B @ f__Variable == nu,
    f__Variable >= 0,
    f__Variable <= c
]
```

3.5.2 Solve the Problem

```
problem = cp.Problem(cp.Minimize(cost_function), constraints)
result = problem.solve()

f_optimal = f__Variable.value

delay = np.sum(l * C / (1 - f_optimal / C) - l * C)
```

3.5.3 Results

Optimal Flow Values:

[6374.58648, 5665.44280, 2904.69700, 2904.69515, 9907.41352, 4527.98777, 2950.50425, 2487.38468, 3018.25442, 709.14368, 0.00894, 2760.73686, 0.00184, 2904.69515, 5379.42575, 2766.19021, 4899.86274, 2286.62720, 463.12852, 2229.86896, 3229.31627, 5459.18523, 2307.31755, 0.00245, 6170.12210, 5212.01270, 4899.86520, 4899.86520]

Total Delay:

23997.160893545046

3.6 Part E

3.6.1 Definition of Variables and Cost Function and Constrains

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

```
f__Variable = cp.Variable(f.shape)

cost_function = cp.sum(cp.multiply(-l * C,
                        cp.log(1 - cp.multiply(f__Variable,
                                   cp.inv_pos(C)))))



constraints = [
    B @ f__Variable == nu,
    f__Variable >= 0,
    f__Variable <= C
]
```

3.6.2 Solve the Problem

```
problem_cp_we = cp.Problem(cp.Minimize(cost_function), constraints)
result = problem_cp_we.solve()

# Extract the Wardrop equilibrium flow
f_optimal = f__Variable.value
delay = np.sum(l * C / (1 - f_optimal / C) - l * C)
```

3.6.3 Results

Optimal f (Wardrop Equilibrium):

[6349.59545, 6178.22408, 2037.75222, 2037.75221, 9932.40455, 4567.32063, 2738.11937, 2144.13313, 3270.77894, 171.37137, 69.21041, 4071.26145, 0.00001, 2037.75221, 5365.08392, 2202.95556, 5162.70099, 2000.57263, 663.19664, 2944.61565, 2866.15219, 5810.76784, 2436.69203, 0.00001, 6644.85475, 4474.44424, 5162.70100, 5162.70100]

Total Delay (Wardrop Equilibrium):

24341.24307789489

3.7 Part F

Unfortunately, I couldn't solve next parts.