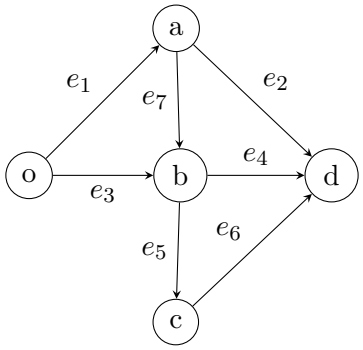
Taha Kamalisadeghian
Email: s328266@studenti.polito.it
Network Dynamics and Learning
Instructor: Prof. Fabio Fagnani

Homework - #1
ID: s328266
Term: Fall 2024 - 25
Due Date: 10^{th} November, 2024

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

$$c_1 = c_3 = c_5 = 3, \quad c_6 = c_7 = 1, \quad c_2 = c_4 = 2.$$



Problem 1-a
Compute the capacity of all the cuts and find the minimum capacity to be removed for no feasible flow from o to d to exist.

Solution:
We are given a network with vertices and directed edges, where each edge has a specified capacity c_i. Our goal is to calculate the capacity of all cuts and identify the minimum cut capacity for which no feasible flow exists from o to d.

Identify All Cuts: We examine possible cuts in the network that separate the source o from the sink d. A cut (S, T) of a flow network is a partition of the vertices into two disjoint subsets S and T, such that the source node $o \in S$ and the sink node $d \in T$.

1. Cut S_1: $S = \{o\}$, $T = \{a, b, c, d\}$ *Edges crossing:* e_1, e_3
2. Cut S_2: $S = \{o, a\}$, $T = \{b, c, d\}$ *Edges crossing:* e_2, e_3, e_7
3. Cut S_3: $S = \{o, b\}$, $T = \{a, c, d\}$ *Edges crossing:* e_1, e_4, e_5
4. Cut S_4: $S = \{o, c\}$, $T = \{a, b, d\}$ *Edges crossing:* e_1, e_3, e_5
5. Cut S_5: $S = \{o, a, b\}$, $T = \{c, d\}$ *Edges crossing:* e_2, e_4, e_5
6. Cut S_6: $S = \{o, b, c\}$, $T = \{a, d\}$ *Edges crossing:* e_1, e_4, e_6
7. Cut S_7: $S = \{o, a, b, c\}$, $T = \{d\}$ *Edges crossing:* e_6, e_4, e_2

Calculate the Capacity of Each Cut: For each cut, we calculate the capacity $C(S,T)$ by summing the capacities of the edges crossing from S to T. The minimum cut capacity is the smallest capacity among all the cuts.

1. Cut S_1: $C(S_1) = c_1 + c_3 = 3 + 3 = 6$
2. Cut S_2: $C(S_2) = c_2 + c_3 + c_7 = 2 + 3 + 1 = 6$
3. Cut S_3: $C(S_3) = c_1 + c_4 + c_5 + c_7 = 3 + 3 + 2 = 8$
4. Cut S_4: $C(S_4) = c_1 + c_3 + c_5 = 3 + 3 + 3 = 9$
5. Cut S_5: $C(S_5) = c_2 + c_4 + c_5 = 2 + 2 + 3 = 7$
6. Cut S_6: $C(S_6) = c_1 + c_4 + c_6 = 3 + 2 + 1 = 6$
7. Cut S_7: $C(S_7) = c_2 + c_4 + c_6 = 2 + 2 + 1 = 5$

Answer: For part (a), the capacity of the minimum cut is 5. This implies that if the capacity of edges e_2, e_4, and e_6 were removed it would be impossible to achieve a feasible flow from o to d.

$$\min\{C(S_1), C(S_2), C(S_3), C(S_4), C(S_5), C(S_6), C(S_7)\} = \min\{6, 6, 8, 9, 7, 6, 5\} = 5$$

Problem 1-b
You are given $x > 0$ extra units of capacity ($x \in \mathbb{Z}$). 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 $x \geq 0$.

Solution:
Given $x > 0$ extra units of capacity, our goal is to distribute these additional units to maximize the throughput from o to d. We analyze two different approaches to distribute the extra capacity and calculate the maximum throughput $T(x)$ as a function of x.

1. **Bottleneck Strategy:**
   The Bottleneck Strategy focuses on increasing the capacity of known bottleneck edges. From part (a), we identified the minimum cut, which includes edges e_2, e_4 and e_6. The total capacity of these edges (the cut value) limits the maximum flow. In the bottleneck approach, extra capacity x is distributed only among the edges in the minimum cut to directly address the flow constraints.

   (a) Compute the **minimum cut** of the graph:

   $$C_{\text{min}} = \text{min\_cut}(G, o, d)$$

   where C_{min} is the total capacity of the edges in the minimum cut.

   (b) Distribute the extra capacity x proportionally among the edges in the minimum cut. The updated capacity for each edge (i,j) is given by:

   $$c'_{ij} = c_{ij} + x \cdot \frac{c_{ij}}{\sum c_{ij} \text{ in minimum cut}}$$

   where c_{ij} is the current capacity of edge (i,j).

   (c) Recompute the **maximum flow** with the updated capacities:

   $$F_{\text{new}} = \text{max\_flow}(G', o, d)$$

   where G' is the graph after updating the capacities.

2. **Critical Edge Approach:**

The critical edge approach differs from the bottleneck approach in that it dynamically identifies the edges that have the highest impact on throughput at each step. Rather than distributing x to all edges in the minimum cut, it allocates extra capacity incrementally to the single edge that provides the largest improvement in throughput. In each iteration, the edge that yields the greatest increase in flow when its capacity is increased is identified as the critical edge. The extra capacity is assigned to this edge, and the process is repeated until x is exhausted.

(a) Compute the **current maximum flow**:

$$F_{\text{current}} = \text{max\_flow}(G, o, d)$$

(b) Identify Saturated Edges:

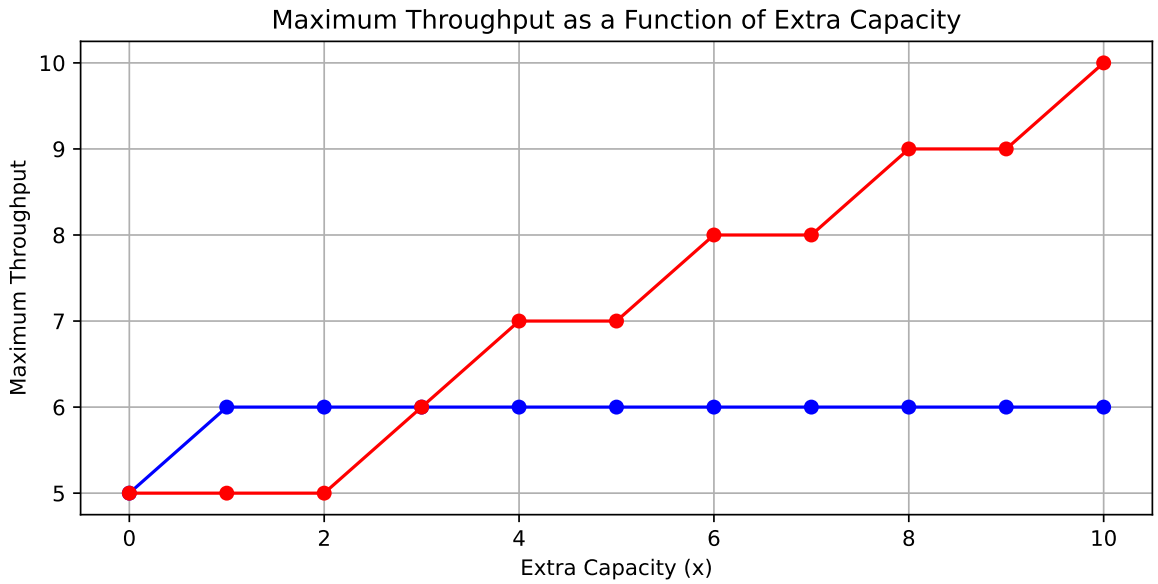
$$\text{Saturated Edges: } \{(i,j) \mid f_{ij} = c_{ij}\}$$

(c) For each saturated edge (i, j), increment capacity and reduce the remaining extra capacity::

$$c'_{ij} = c_{ij} + 1$$

$$x' = x - 1$$

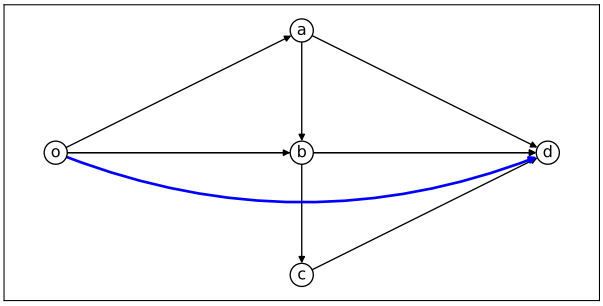
(d) Allocate capacity incrementally to the critical edge until x is exhausted.



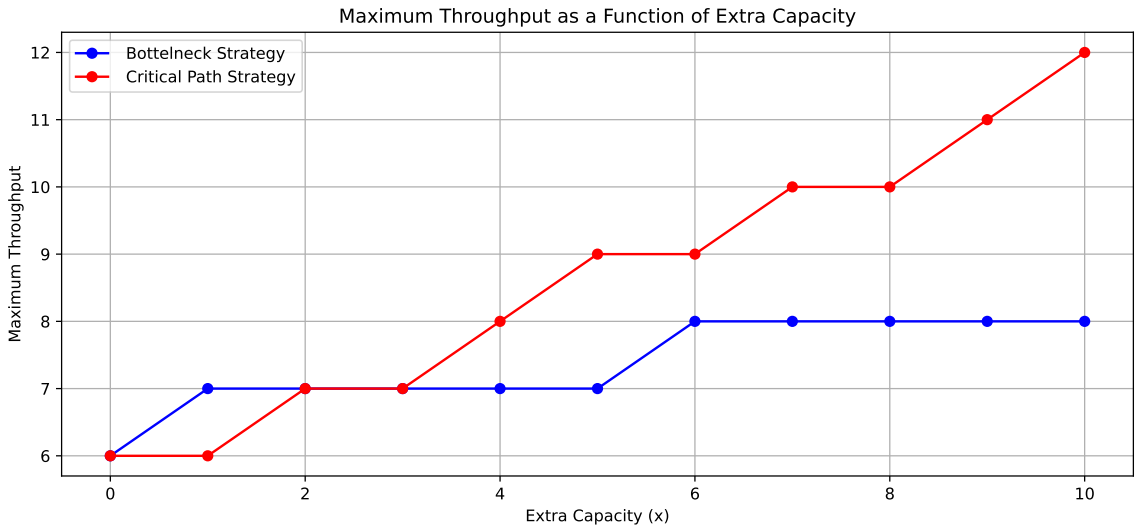
The comparison between the bottleneck and critical edge approaches, as depicted in Fig.1, highlights distinct characteristics. The bottleneck approach (blue curve) focuses exclusively on the edges in the minimum cut, resulting in an initial sharp increase in throughput, followed by a plateau as the extra capacity is fully utilized in addressing the bottleneck. In contrast, the critical edge approach (red curve) dynamically identifies and allocates capacity to the most impactful edge at each step, resulting in a smoother and more gradual increase in throughput. This method efficiently adapts to evolving constraints in the network, achieving a consistently higher throughput compared to the bottleneck approach, especially as the allocated capacity increases.

Problem 1-c
You are given the possibility of adding to the network a directed link e_8 with capacity $c_8 = 1$ and $x > 0$ extra units of capacity ($x \in \mathbb{Z}$). Where should you add the link and how should you distribute the additional capacity 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 $x \geq 0$.

Solution:
The best placement for e_8 was determined by systematically evaluating all possible locations where e_8 could be added between nodes that were not already directly connected. For each available placement a temporary graph was created with e_8 added with an initial capacity of 1. The maximum flow was then calculated and then recorded for each candidate. The placement that resulted in the highest throughput was chosen. In this case, $e_8 = (o, d)$ was selected because it yielded the maximum throughput of 6.



The plot shows that adding e_8 between (o, d) provides an initial throughput of 6, and the subsequent distribution of $x > 0$ extra capacity results in higher throughput for the **Critical Path Strategy** (red curve) compared to the **Bottleneck Strategy** (blue curve). The critical path approach dynamically allocates x to the most impactful edges, leading to a smooth and consistent increase in throughput. In contrast, the bottleneck strategy quickly resolves the initial bottlenecks but plateaus as it fails to adapt to shifting constraints. Compared to Problem 1.2, where $x > 0$ was distributed among pre-existing edges, the addition of e_8 creates a direct path from o to d, fundamentally altering the network and amplifying the effect of capacity allocation, resulting in more significant throughput improvements.



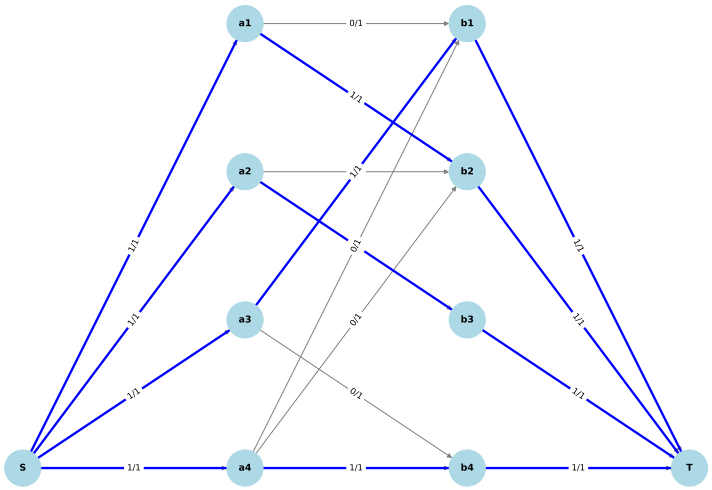
Exercise 2. There is a set of people $\{a_1, a_2, a_3, a_4\}$ and a set of foods $\{b_1, b_2, b_3, b_4\}$. Each person is interested in a subset of foods, specifically:

$$a_1 \to \{b_1, b_2\}, \quad a_2 \to \{b_2, b_3\}, \quad a_3 \to \{b_1, b_4\}, \quad a_4 \to \{b_1, b_2, b_4\}.$$

Problem 2-a
Exploit max-flow problems to find a perfect matching (if any).

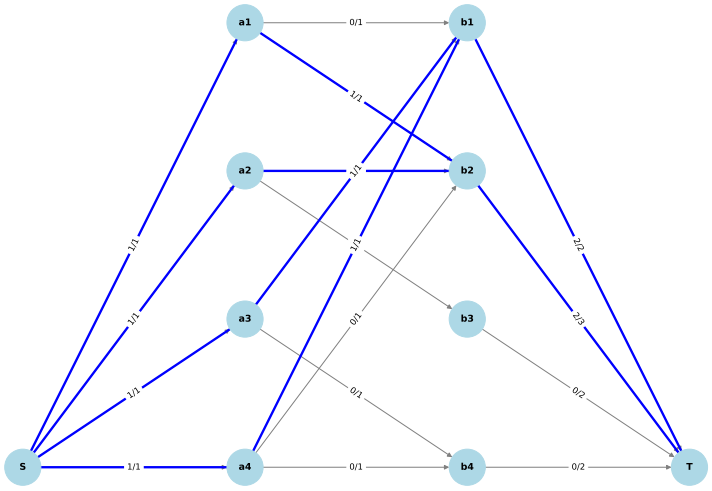
Solution:
In part (a), we model the perfect matching problem as a maximum flow problem. We construct a directed graph with a source node S and a sink node T. Each person a_i is connected to S with capacity 1, and each food item b_j is connected to T with capacity 1. Edges between people and foods are based on individual preferences, also with capacity 1.

Using the maximum flow algorithm, we compute a flow of 4, matching the number of people, indicating a perfect matching exists. This confirms that each person can be assigned a unique preferred food item without conflicts. The flow results are displayed in the plot, with each edge labeled f/c showing flow f over capacity c.



Problem 2-a
Now, assume that there are multiple portions of every food, and the distribution of the portions is (2, 3, 2, 2). Each person can take an arbitrary number of different foods. Exploit the analogy with max-flow problems to establish how many portions of food can be assigned in total.

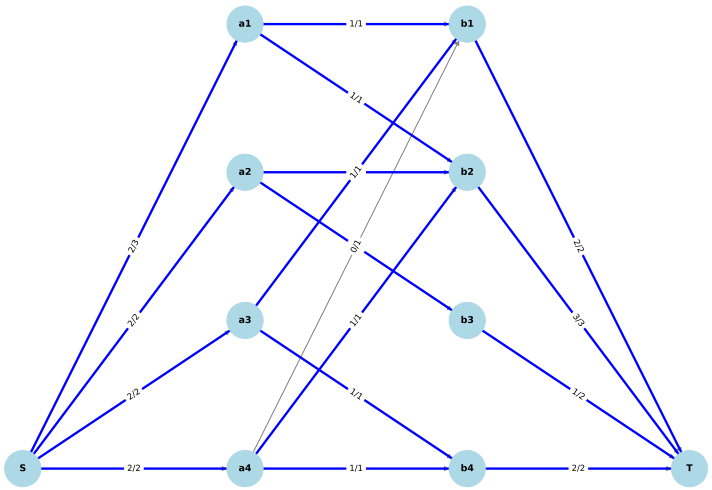
Solution:
Each food now has a specified capacity based on the number of available portions: b_1 has 2 portions, b_2 has 3 portions, b_3 has 2 portions, and b_4 has 2 portions. We model this by setting the edges from each food node to the sink T with these respective capacities, allowing each food to be assigned to multiple people as long as the total assignments do not exceed the available portions. Using the maximum flow algorithm, we compute the flow from the source S to the sink T with these adjusted capacities. The resulting flow indicates the maximum number of portions that can be distributed based on each persons preferences and the food capacities. This flow value represents the maximum feasible allocation of food portions under these conditions.

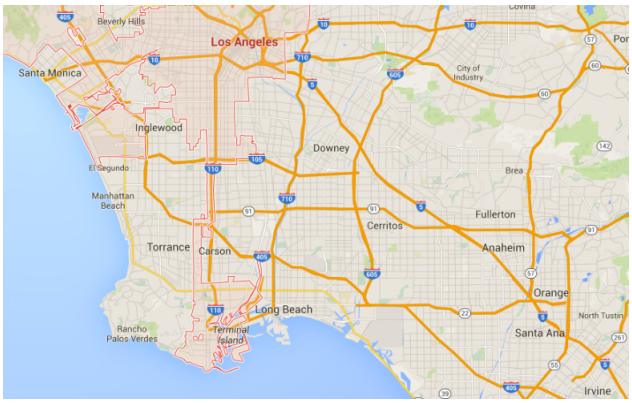


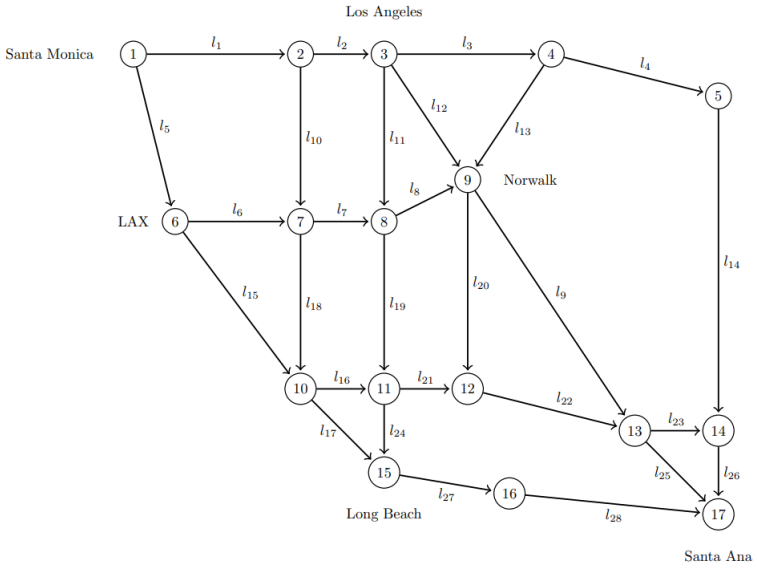
Problem 2-c

Now, assume that a_1 wants 3 portions of food, and a_i (for every $i \neq 1$) wants 2 portions of food. Every person can take multiple portions of the same food, and the distribution of the portions is $(2, 3, 2, 2)$. Exploit the analogy with max-flow problems to establish how many portions of food can be assigned in total.

Solution:
For answering part (c), we consider a scenario where each person requires a specified number of portions, and each food item has a limited number of portions available. Specifically, a_1 requires 3 portions, while each of a_2, a_3, and a_4 requires 2 portions. The food capacities remain as in part (b), with $b_1 = 2$, $b_2 = 3$, $b_3 = 2$, and $b_4 = 2$. We represent these requirements by adding edges from S to each person with capacities reflecting their portion needs, and from each food to T with edges reflecting the available portions. Using the maximum flow algorithm, we calculate the total flow from S to T, which represents the maximum number of food portions that can be assigned while satisfying both the individual portion requirements and the food item capacities. This total flow provides the answer to the maximum feasible distribution under these conditions.







Exercise 3. 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 $e_i \in \{e_1, \ldots, e_{28}\}$, has a maximum flow capacity c_{e_i}. 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 of 60 miles/hour. For each link, we introduce the delay function

$$\tau_e(f_e) = \frac{l_e}{1 - \frac{f_e}{c_e}}, \quad 0 \leq f_e < c_e.$$

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

If you use Python to solve the Exercise, you can load the .mat files by the following code:

```
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,)
```

Problem 3-a
Find the shortest path between nodes 1 and 17. This is equivalent to the fastest path (path with the shortest traveling time) in an empty network.

Solution:
To determine the shortest path between nodes 1 (Santa Monica) and 17 (Santa Ana) in an empty network, we aim to find the route that minimizes total travel time.

Let $G = (N, E)$ represent a directed graph, Each link $e \in E$ has an associated minimum travel time l_e, which is the time experienced by drivers when the link is uncongested. The shortest path P from source node 1 to target node 17 minimizes the cumulative travel time, expressed as:

$$P = \arg\min_{P \subset E} \sum_{e \in P} l_e$$

where l_e is the minimum travel time on each link e in the path.

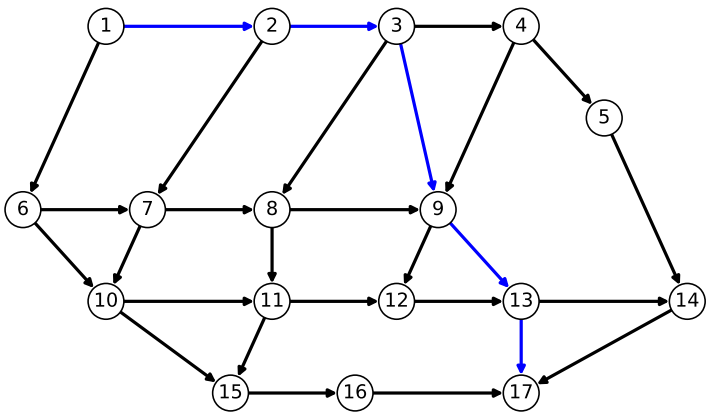
In Python, we employ the NetworkX library, which provides an implementation of this algorithm. The `nx.shortest_path_length()` function calculates the shortest path length based on travel time. the result is:

```
shortest_path = nx.shortest_path(traffic_Graph,
                                 source=1,
                                 target=17,
                                 weight='min_capacity')

shortest_path_len = nx.shortest_path_length(traffic_Graph,
                                            source=1,
                                            target=17,
                                            weight='min_capacity')

print(f"Shortest Path : {shortest_path} \nShortest Path Length : {shortest_path_len}")
```

```
Shortest Path : [1, 2, 3, 9, 13, 17]
Shortest Path Length : 0.559833
```



Problem 3-b
Find the maximum flow between node 1 and 17.

Solution:

Each link $e \in E$ has a maximum capacity c_e, which represents the maximum allowable flow on that link. We define the flow vector f such that f_e represents the flow on link e. The objective is to maximize the total flow f_{total} from source node 1 (Santa Monica) to target node 17 (Santa Ana), constrained by link capacities. This can be expressed as:

$$\max f_{total} = \sum_{e \in \text{outgoing}(1)} f_e$$

In Python, we use NetworkXs `maximum_flow()` function, which efficiently computes the maximum feasible flow from node 1 to node 17 while respecting all capacity constraints. The function returns both the maximum flow value (`flow_a`) and a dictionary (`flow_dict_a`) showing the flow across each link.

```
flow_a, flow_dict_a = nx.maximum_flow(traffic_Graph, _s=1, _t=17,  capacity='capacity')
f"flow_a =  {flow_a} "
```

```
flow_a = 22448
```

Problem 3-c
Given the flow vector in `flow.mat`, compute the vector ν satisfying $Bf = \nu$.

Solution:

To compute ν, multiply the incidence matrix B by the flow vector f. This matrix-vector multiplication gives the net inflow or outflow at each node.

In Python, this can be achieved using the following code:

```
# external inflow vector
nu = B_matrix @ flow
```

```
[ 16282   9094  19448   4957   -746   4768    413     -2  -5671
  1169   -5  -7131   -380  -7412  -7810  -3430 -23544]
```

Problem 3-d
Find the social optimum f^* with respect to the delays on the different links $\tau_e(f_e)$. For this, minimize the cost function

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

Solution:
To find the **social optimum flow** f^* with respect to the delays on the different links $\tau_e(f_e)$, we minimize the given cost function. This problem is formulated and solved using convex optimization. The decision variable is f, which represents the flow on each edge of the network. The objective is to minimize the total cost by solving a constrained optimization problem. The constraints include flow conservation ($B \cdot f = \nu$, ensuring inflow equals outflow plus external inflows), non-negativity of flow ($f \geq 0$), and capacity limits ($f \leq c_e$). Using Python's `cvxpy` library, we define the cost function and constraints, set up the problem with `cp.Problem`, and solve it to find f^* and the corresponding optimal cost. This approach guarantees that the social cost is minimized while adhering to network constraints. The computed results can be accessed using `f.value` for f^* and `cost_opt` for the optimal cost.

```
nu = [ 16282,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
                                                                                          -16282]

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)
```

The final result for f^* was:

```
Optimal cost: 23997.160893545046

Social optimal flow: [6.37458648e+03 5.66544280e+03 2.90469700e+03 2.90469515e+03
 9.90741352e+03 4.52798777e+03 2.95050425e+03 2.48738468e+03
 3.01825442e+03 7.09143680e+02 8.94419224e-03 2.76073686e+03
 1.84265590e-03 2.90469515e+03 5.37942575e+03 2.76619021e+03
 4.89986274e+03 2.28662720e+03 4.63128517e+02 2.22986896e+03
 3.22931627e+03 5.45918523e+03 2.30731755e+03 2.45326148e-03
 6.17012210e+03 5.21201270e+03 4.89986520e+03 4.89986520e+03]
```

Problem 3-e
Find the Wardrop equilibrium $f^{(0)}$. For this, use the cost function

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

Solution:

To find the Wardrop equilibrium flow $f^{(0)}$, we minimize the cost function where the travel delay on each edge is given by $\tau_e(s) = \dfrac{l_e}{1 - s/c_e}$. The integral evaluates to:

$$\int_0^{f_e} \frac{l_e}{1 - s/c_e}\, ds = -l_e c_e \ln\left(1 - \frac{f_e}{c_e}\right).$$

Therefore, the cost function becomes:

$$\text{func} = -\sum_{e\in\mathcal{E}} l_e c_e \ln\left(1 - \frac{f_e}{c_e}\right).$$

The optimization problem is solved using Pythons `cvxpy` library along with `scipy.integrate`. The constraints are the same as in problem 3.d. The Wardrop equilibrium flow $f^{(0)}$ is obtained by solving this optimization problem. The computed flow minimizes the total travel cost while satisfying all constraints. The solution flow vector is stored in `f.value`, and the corresponding equilibrium cost is stored in `cost_w`.

```
from scipy import integrate
f = cp.Variable(n_edges)
nu = [ 16282, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -16282]
#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: [6.34959872e+03 6.17825285e+03 2.03772189e+03 2.03772188e+03
 9.93240128e+03 4.56732491e+03 2.73809135e+03 2.14408534e+03
 3.27080302e+03 1.71345873e+02 6.92016417e+01 4.07132931e+03
 8.50623010e-06 2.03772188e+03 5.36507637e+03 2.20294354e+03
 5.16271226e+03 2.00057943e+03 6.63207643e+02 2.94461164e+03
 2.86615117e+03 5.81076281e+03 2.43670616e+03 1.34964045e-05
 6.64485968e+03 4.47442804e+03 5.16271227e+03 5.16271227e+03]
```

To evaluate the inefficiency in the network The *Price of Anarchy (PoA)* metric is used. It compares decentralized decision-making in networks to a centralized socially optimal solution. It quantifies the ratio of the total cost incurred at the Wardrop equilibrium, where individual users minimize their own travel times, to the total cost incurred in the socially optimal state, where overall system efficiency is maximized. In this study, the need for PoA arises to assess how much worse the system performs due to selfish routing behaviors.

The total delay cost $\sum_e f_e \tau_e(f_e)$ was determined using the flow vector obtained from the CVXPY optimization to compute the Wardrop equilibrium cost. While the optimization problem minimized the integral of the delay function to ensure compliance with Wardrop's conditions, the actual cost experienced by users (denoted as `cost_w`) was calculated by summing the product of the flow and delay for each link. This approach provides a realistic measure of the system's performance under equilibrium and serves as the numerator in the PoA computation, enabling a meaningful comparison to the socially optimal cost.

```
# cost, defined as \sum_e f_e tao_e(f_e)
war_vect = f.value

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)
```

```
Wardrop cost: 24341.2689271447
```

```
PoA = cost_w/cost_opt

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

```
The price of anarchy: 1.0143395310439502
```

The Price of Anarchy (PoA) of 1.0143 indicates that the system's performance under selfish routing (Wardrop equilibrium) is only about 1.43% worse than the socially optimal performance. This small inefficiency suggests that selfish behavior has minimal impact on the network's overall performance. As a result, the network operates near optimally even without centralized control, and interventions like tolls or traffic regulations may offer limited additional benefits. The plots below demonstrate traffic flows with a starting flow of 5000 units, showing the Wardrop Equilibrium where selfish routing balances individual travel costs (left) against the socially optimal flow distribution minimizing total travel cost (right), using a red gradient to indicate flow intensity.