Homework 1

Gioele Giachino
s295380

Silva Bashllari
s299317

November 17, 2024

Disclaimer: We have worked in collaboration for all the exercises by discussing all the points together and reflecting on the possible solutions, by solving them both by hand and through Python libraries. The solutions reflected below are the final versions made with the help of Python Libraries.

1 Exercise 1

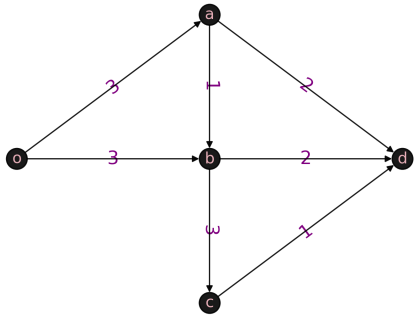

Figure 1: Structure of the Graph 1

For this first exercise, we take into account the network represented in Figure 1, that contains 5 nodes and 7 edges, that have the following capacities: $c_1 = c_3 = c_5 = 3, \quad c_6 = c_7 = 1, \quad c_2 = c_4 = 2$.

1.1

Here we are asked to compute the capacity of all the cuts and then to find the minimum capacity to be removed for no feasible flow from node 'o' to node 'd' to exist, thus creating two disjoint sets of nodes U_1 and U_2, such that their intersection is an empty set and their union is N(the total nodes in the original graph). Practically, we are interested in the **min-cut capacity**.
Firstly, we enumerate all the cuts, with their respective capacities.

$$
\begin{aligned}
&U_1 = \{o\}, \quad U_2 = \{a, b, c, d\}, \quad C = 6;\\
&U_1 = \{o, a\}, \quad U_2 = \{b, c, d\}, \quad C = 6;\\
&U_1 = \{o, b\}, \quad U_2 = \{a, c, d\}, \quad C = 8;\\
&U_1 = \{o, c\}, \quad U_2 = \{a, b, d\}, \quad C = 7;\\
&U_1 = \{o, a, b\}, \quad U_2 = \{c, d\}, \quad C = 7;\\
&U_1 = \{o, a, c\}, \quad U_2 = \{b, d\}, \quad C = 6;\\
&U_1 = \{o, b, c\}, \quad U_2 = \{a, d\}, \quad C = 6;\\
&U_1 = \{o, a, b, c\}, \quad U_2 = \{d\}, \quad C = 5.
\end{aligned}
$$

So, we can easily see that the minimum capacity to be removed corresponds to the last cut of capacity=5 and it's associated to the cut that presents on one side nodes o, a, b, c and on the other side alone the sink node d. More specifically, edges that correspond to this minimum capacity are $e2, e4$ and $e6$. We did this enlisting manually but then to verify the accuracy, we have verified that with the method available in the Networkx library, namely:

```
nx.algorithms.flow.minimum_cut(graph1, source, sink, capacity="capacity")
```

The output of the method above confirms the same results we obtained. Furthermore, as we know from the theory of the min-cut max-flow theorem, the minimum cut capacity is also the maximum flow that can be injected in this network from nodes 'o' to 'd' and this can also be further verified with the function in python:

```
nx.algorithms.flow.maximum_flow(graph1, source, sink, capacity="capacity")
```

1.2

Here we are asked to consider the addition of $x > 0$ extra units of capacity and our objective is to distribute them in order to maximize the throughput.
Since we know that min-cut capacity is a sort of bottleneck for the maximum throughput, logically we have to allocate this extra capacity to the edges that correspond to the minimum cut. In our case, the minimum cut includes the following edges: $e6, e2, e4$. Let's take a first step and suppose we want to add just 1 capacity, so $x = 1$. This would change the minimum-cut capacity from 5 to 6. However, we must observe that now the minimum-cut is no longer unique, given the fact that we have multiple cuts with a capacity of 6.
Developing this in a more systematic manner, in our code implementation we create a method, to distribute extra capacity in the network incrementally from 0 to 50. Before allocating the extra capacity, each time it re-computes the minimum cut and saves the edges corresponding to the minimum cut. By recalculating the min-cut after each step, the algorithm ensures that the next unit of capacity is added to the most restrictive edges at that moment. Obviously, there may be more than 1 same-capacity minimum-cut and the limitation of our solution is the minimum-cut provided first by the Networkx library. Then, we allocate the extra capacity to one edge in the cut. We experimented with different selection of the edges in the list, so both 0 and 1 as positions, and with the exception of some initial transient, the pattern of the graph of Extra Capacity vs. Maximum Throughput from o to d was the same. These graphs can be observed in Figure 2 and 3. The throughput_values list records the maximum flow (max_flow) after each unit of capacity is added. This provides a direct relationship between the number of extra units of capacity added (x) and the resulting throughput.

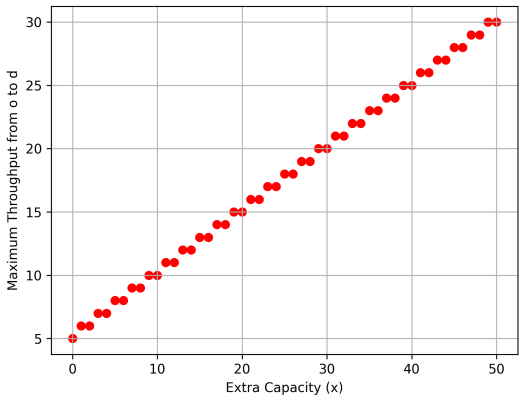

Figure 2: Maximum Throughput as a Function of Extra Capacity (edge 0)

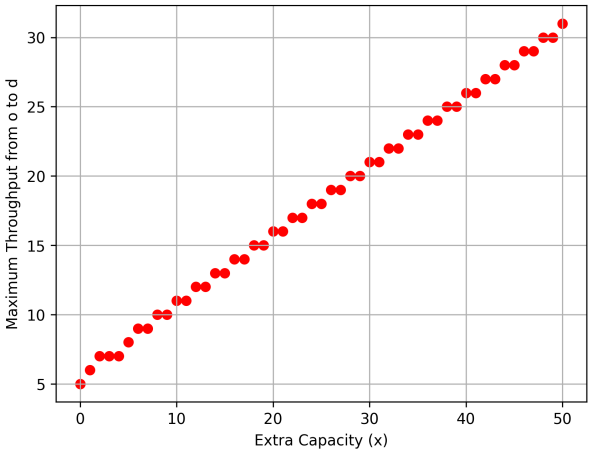

Figure 3: Maximum Throughput as a Function of Extra Capacity (edge 1)

1.3

Here, besides the extra units of capacity, there is also another possibility, the one of adding to the network a new directed link $e8$ with capacity $c_8 = 1$. Just like in the question 1.2, our objective is to strategically insert the new link and the extra units of capacity in order to maximize the throughput.

The new link can be added between different nodes. We decided arbitrarily to not consider the possibility of creating a multi-graph when adding the new link $e8$, so we explore the two following possibilities:

- Link $e8$ connects node 'o' and node 'c', which corresponds to the Graph in Figure 4.
- Link $e8$ connects node 'o' directly with node 'd' which corresponds to the Graph in Figure 5.

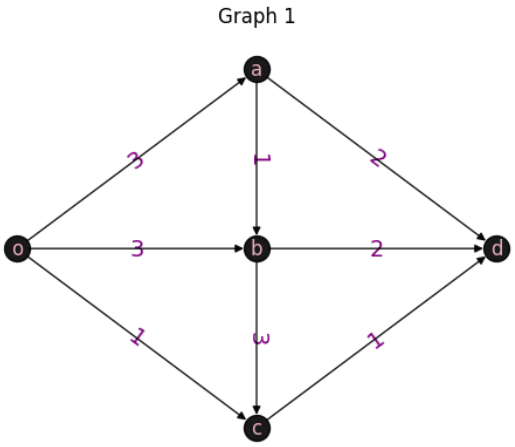

Figure 4: Graph 1 adjusted (adding link o-c)

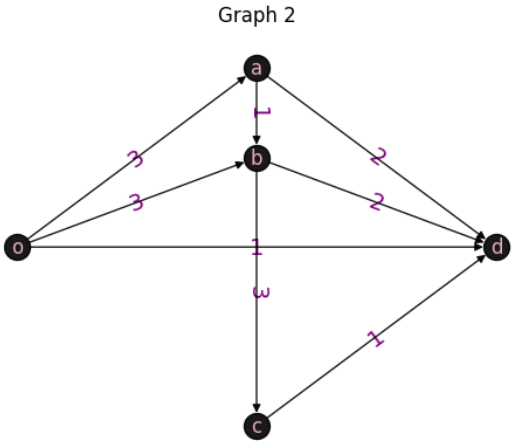

Figure 5: Graph 1 adjusted (adding link o-d)

Just like in Exercise 1.B we are also provided with $x > 0$ extra units of capacity to add and we have to distribute this additional capacity in order of course to maximize the throughput and then plot it. Using the same methods we used in Exercise 1.B we obtained the following plots:

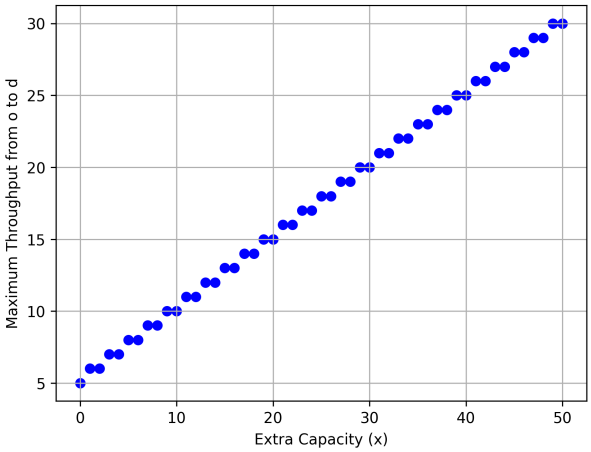

Figure 6: Maximum Throughput as a Function of Extra Capacity(adding link o-c)

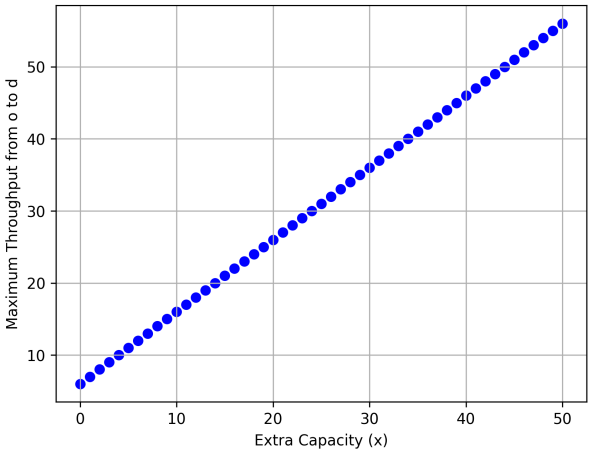

Figure 7: Maximum Throughput as a Function of Extra Capacity (adding link o-d)

We can observe that adding the new link between nodes 'o' and 'c' makes little difference in terms of altering the behavior of our function with respect to this link not being present. However, when this extra link is instead added in the second case, so between nodes 'o' and 'd' directly, we can observe that the function is steeper than in the first case and with the same number of iterations of extra capacity up to 50 it reaches an higher throughput of 56. Hence, we can conclude that the optimal positioning for this extra link $e8$ is between nodes 'o' and 'd'.

2 Exercise 2

We deal with a network perfectly represented by the bipartite graph in Figure 4, because our system presents on one side a set of people a_1, a_2, a_3, a_4 and on the other side a set of foods b_1, b_2, b_3, b_4. Each person is interested in a set of foods, as indicated by edges in the graph.

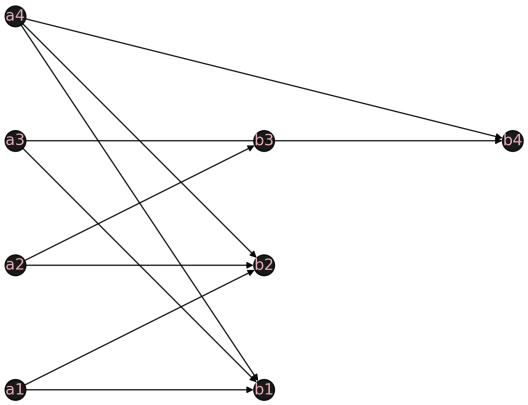

Figure 8: Bipartite Graph of people and foods

2.1

Here our goal is to search for perfect matching given that one exists. Every edge is considered with a unitary capacity. A perfect matching corresponds to the precise association between one single person and one single food. In order to find a perfect matching, assuming one exists, the max-flow algorithm can be exploited, given the fact that as has been discussed in the lectures can also be linked to Hall's Theorem (for perfect matching). In order to have a perfect matching of 4 edges, we expect to have a maximum flow with a value of 4.
To search for maximum flow, we have to apply the Ford-Fulkerson algorithm, as implemented by the Networkx library. Thus, firstly we must adapt the structure of our graph adding a dummy source node and a dummy sink node. The source node will be connected to all people nodes a_1, a_2, a_3, a_4 with unitary capacity, while all the food nodes b_1, b_2, b_3, b_4 will be connected to the sink node with unitary capacity.
In Figure 5 we can observe the graph with the two new added nodes.

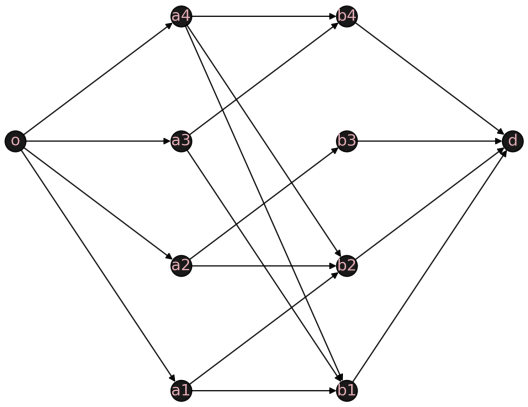

Figure 9: Set of people and set of foods plus source and sink node

The function that we use to find the perfect matching is that of maximum flow, as illustrated in the Question 1.a. The output of that method provides not only the maximum flow but also the perfect matching. We can observe, as illustrated in Figure 10, that there exists a perfect matching.

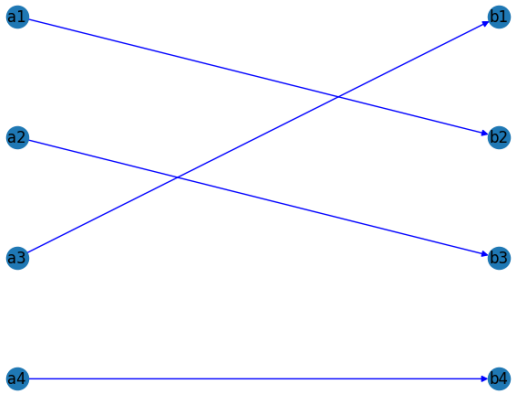

Figure 10: Perfect matching

The results illustrate that the throughput has a value of 4 and the following matching is the perfect one:

- Person 1 (a1) - Food 2 (b2)
- Person 2 (a2) - Food 3 (b3)
- Person 3 (a3) - Food 1 (b1)
- Person 4 (a4) - Food 4 (b4)

To ensure the accuracy of our perfect matching, we also verify the matching using a function of Networkx. First we save our matching in a dictionary and then we pass that dictionary and the graph itself input to the following function:

```
nx.is_perfect_matching(test_graph,perf_matching)
```

The function outputs a boolean value that in this case is **"true"**.

2.2

Now, we consider an instance of the same problem but with multiple portions of foods, distributed with this multiplicity: (2,3,2,2). Clearly, every person can now take an arbitrary number of different foods (but always 1 portion from each).
Always from the perspective of max-flow, we have to establish how many portions of food can be assigned globally.
Again, as before, we apply F-F algorithm, but now we have to introduce some slight modifications. Firstly, we assign to the edges going from the source node to the people nodes a sufficiently large capacity value, in order to be sure to enable enough flow in the network. Secondly, we assign to the edges connecting the foods nodes with the sink node a capacity corresponding to the available portions of each food as illustrated in the distribution above.
After the necessary iterations of F-F algorithm, we can observe that up to a maximum of 8 (value of the max-flow) portions of food can be assigned in total.
The assignment of portions is the following:

- Person a_1 gets 1 portion of food (b_2)
- Person a_2 gets 1 portion of food (b_2) and 1 portion of food (b_3)
- Person a_3 gets 1 portion of food (b_1) and 1 portion of food (b_4)
- Person a_4 gets 1 portion of food (b_1), 1 portion of food (b_2) and 1 portion of food (b_4)

As we can observe the only not allocated portion of food is 1 from b_3.

2.3

Finally, we still consider the presence of multiple portions of foods, again distributed (2,3,2,2), but with the possibility for every person to take multiple portions of the *same* food.
In this specific instance, we must also take into account that person a_1 wants exactly 3 portions of food, while every person a_i with i different than 1 wants exactly 2 portions of food.
Again, as before, we apply F-F algorithm, but with some new slight modifications. Now, we have to connect every person node to a *dedicated* dummy source node with an edge of capacity corresponding to the number of food portions desired. Then, we allocate to the edges going from the people nodes to the foods nodes a sufficiently large capacity value, in order to be sure to enable enough flow in the network. In the end, we allocate to the edges connecting the foods nodes with the sink node a capacity corresponding to the available portions of each food, as already done in part b.
In the result, we can observe that up to a maximum of 9 (value of the max-flow) portions of food can be assigned in total.
The distribution of the portions of foods in accordance with the constrains is be the following:

- Person a_1 takes 3 portions of food b_2.
- Person a_2 takes 2 portions of food b_3.
- Person a_3 takes 2 portions of food b_1.
- Person a_4 takes 2 portions of food b_4.

Hence, we observe an optimal allocation of all the 9 portions available.

3 Exercise 3

In this last exercise, we deal with a graph representing the highway network of the city of Los Angeles, displayed in Figure 11. The Figure represents the simplified highway as a directed graph in which the nodes are the intersections and the links are the different roads in the highway. We are also provided with 4 flies storing: the node-link incidence matrix, the capacities, the minimum travelling times and flow.

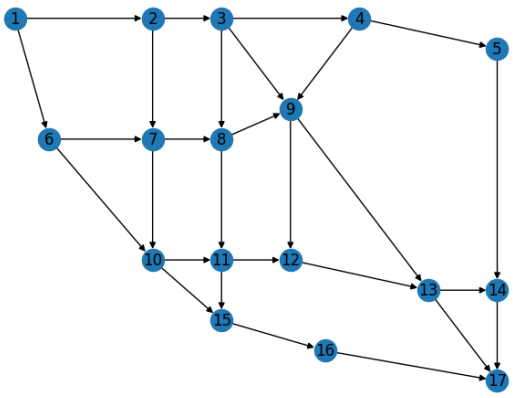

Figure 11: LA Highway network

3.1

We are asked to search for the shortest path between origin node **1** and destination node **17**. In Python we find this using the function available from the Networkx library, namely the shortest path one.

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

The nx.shortest_path function in NetworkX uses Dijkstra's algorithm by default, which is a greedy algorithm, to find the shortest path between two nodes in a graph. We can observe that the shortest path is the one containing the following nodes:

$$1- > 2- > 3- > 9- > 13- > 17.$$

3.2

Then, we are asked to calculate the maximum flow between origin node 1 and destination node 17, that corresponds to a numerical value of **22448**, which we find using the same function we have used as in the two exercises above.

3.3

Furthermore, we are asked to compute the vector v that satisfies the following equation $Bf = v$. This vector v, representing the **external flow**, corresponds to the matrix multiplication between the flow vector(flow.mat) and the node-link incidence matrix (B).
We used the following python method to perform this operation:

```
v = Bf
```

After the computation, we obtain the following result:

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

From now on, as indicated from the exercise, we will consider the exogenous inflow vector v with the following configuration:

$$v = [16282, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -16282]$$

3.4

Here, we have to find the social optimum $f*$ with respect to the delays on the different links, $\tau_e(f_e)$. The cost function we have to minimize is the following:

$$\sum_{e\in\varepsilon} f_e\tau_e(f_e) = \sum_{e\in\varepsilon} \frac{f_e l_e}{1-\frac{f_e}{C_e}} = \sum_{e\in\varepsilon}\left(\frac{l_e C_e}{1-\frac{f_e}{C_e}} - l_e C_e\right)$$

To begin with, we rename the vector v as "exogenous_inflow". Then, we build up the problem using the Python library "cvxpy", defining the objective function as a sum of costs of every link. First, we define the function to be minimized using the following Python method:

```
objective = cp.Minimize(sum(cost_func))
```

Then, after defining also the constraints, we use the following methods to define the convex optimization problem and to solve it:

```
prob = cp.Problem(objective, constraints)
```

```
cost_opt = prob.solve()
```

After the computations, we obtain a social optimum cost of **23997.160214062365**.

3.5

Now, we search for the Wardrop equilibrium $f^(0)$. To find it, we implement the following cost function:

$$\sum_{e\in\varepsilon}\int_0^{f(e)} \tau_e(s)\,ds$$

Similarly to the point above, we exploit "cvxpy" Python library in order to derive for our network a Wardrop equilibrium cost corresponding to the numerical value of **24341.24549203723**.
In our code the decision variable **f** represents the flow on the network, and the objective function **obj** minimizes a cost function based on a logarithmic delay model that depends on the flow **f**, the capacities of the links, and their lengths. The flow constraints ensure that the flow **f** is non-negative, does not exceed the link capacities, and satisfies an exogenous inflow balance using matrix multiplication (B @ f = exogenous_inflow). The optimization problem is then solved, and the equilibrium flow wardrop_flow is computed. Finally, the total Wardrop cost is calculated by multiplying the flow with its corresponding delay and summing it up to find the network's overall cost.
We can clearly observe as it could be anticipated, that the Wardrop equilibrium cost value is slightly larger than the social optimum cost value. This is to be expected given that by definition the Wardrop equilibrium corresponds to entities minimizing their own cost, for instance the travel time, by continuously evaluating their choices for the route. This selfish behavior leads to an equilibrium as a state where none can improve their situation by changing their routes, however selfishness guarantees only equilibrium and stability, but not the most efficient use of the network(of resources in general). Some unused capacities may still exist.
In fact, we can also calculate the Price of Anarchy, which is a measure that quantifies the inefficiency of selfish behavior compared to the social optimum situation.

$$Price\ of\ Anarchy\ (PoA) = Total\ Cost\ at\ Wardrop\ Equilibrium / Total\ Cost\ at\ SocialOptimum$$

In our case, the Price of Anarchy is equal to:

$$PoA = 24341.24549203723/23997.160214062365 = 1.01433858319$$

As expected, the value of the Price of Anarchy is bigger than 1.
We might have to consider other techniques in order to reduce the PoA to 1, e.g. adding tolls.

3.6

Then, we slightly modify the problem adding tolls. For each link e, the corresponding toll is as follows:
$\omega_e = \psi'_e(f^*_e) - \tau_e(f^*_e)$.
Consequently, the delay on link e has the following form:
$\tau_e(f_e) + \omega_e$.
With this modified configuration, calculated the new Wardrop equilibrium $f^(\omega)$.
Considering the fact that **WITHOUT TOLLS** the Wardrop equilibrium cost for our network was of **24341.24549203723**, we can now say that **WITH TOLLS** this Wardrop equilibrium cost slightly drops to **23997.16033886616**.
We can denote that the tolls incentivize entities to distribute themselves more efficiently in comparison to the Wardrop equilibrium **without** tolls across the network, achieving a performance closer to the social optimum cost. In fact, when we compare the difference between the total cost of the social optimum and the Wardrop equilibrium with tolls we observe they are really close to each other with a minor difference occurring only after the fourth digit after the decimal comma.

3.7

Finally, we introduce a different cost function, of this form: $\psi_e(f_e) = f_e(\tau_e(f_e) - l_e)$. In this new situation, the cost for the system is not considered on the total travel time, but instead on the total additional travel time compared to the total travel time in free flow.
This gives a measure of the inefficiency caused by congestion in the network.
With this new configuration, we calculate a new social optimum cost of **13550.21558388883**.
Thus, calculating the Wardrop equilibrium cost with this final cost function and also considering tolls, we reach the value of **13550.218470998525**, really close to the social optimum cost.