Politecnico di Torino

Network Dynamics

Homework 1

Micol Rosini s302935

1 Exercise 1

Consider the network in Fig. 1 with link capacities:

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

Figure 1

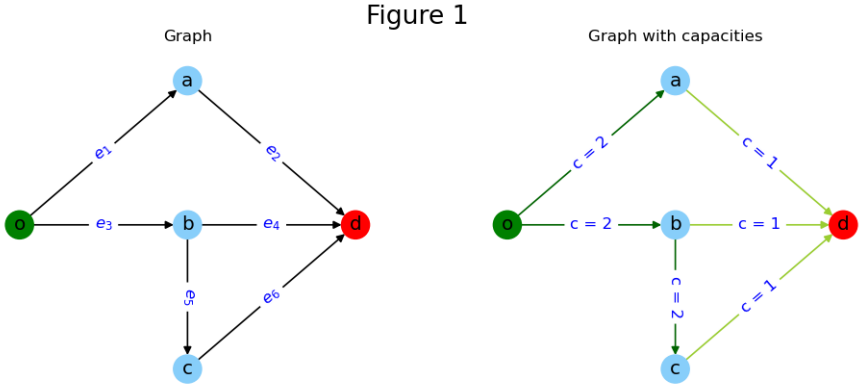

Figure 1: Graph with normal *edge* labels on the left and graph with the *capacities* as labels on the right.

1.1 Minimum capacity removed for no existence of feasible flow

The **Max-Flow Min-Cut Theorem** states that, given a multigraph $G = (\mathcal{V}, \mathcal{E}, c)$ with a capacity vector $c > 0$, and two distinct nodes $o \neq d$ in $\mathcal{V}$, then the **maximum throughput** $\tau^*_{o,d}$ from o to d is equal to the **min-cut capacity** $c^*_{o,d}$ of the network :

$$\tau^*_{o,d} = c^*_{o,d}$$

This is equivalent to state that the **minimum total capacity** that needs to be removed from the network in order to make node d *not reachable* from node o coincides with the **min-cut capacity** $c^*_{o,d}$.

Indeed, if all the links are removed in a minimum capacity $o - d$ cut (and so, removing a total capacity $c^*_{o,d}$) then, the min-cut capacity of the resulting graph would be 0 and the theorem above would imply that there exists **no feasible** flow from node o to node d in such resulting graph, except the one with all-zero, hence d would not be reachable from o.

Furthermore, if the total capacity removed is less than $c^*_{o,d}$, the min-cut capacity of the resulting graph would still be positive so that the theorem would imply that there exists a **feasible flow** with positive throughput from node o to node d.

In order to obtain the min-cut capacity we need to compute the capacities of all cuts of the network that are 2^{n-2} where n is the number of nodes:

- $U = \{o, a, b, c\},\ U^C = \{d\} \rightarrow C_U = 3$
- $U = \{o, a, c\},\ U^C = \{b, d\} \rightarrow C_U = 4$
- $U = \{o, a\},\ U^C = \{b, c, d\} \rightarrow C_U = 3$
- $U = \{o, c\},\ U^C = \{a, b, d\} \rightarrow C_U = 5$

$U = \{o, a, b\},\ U^C = \{c, d\} \rightarrow C_U = 4$

$U = \{o, b, c\},\ U^C = \{a, d\} \rightarrow C_U = 4$

$U = \{o, b\},\ U^C = \{a, c, d\} \rightarrow C_U = 5$

$U = \{o\},\ U^C = \{a, b, c, d\} \rightarrow C_U = 4$

The results are obtained with the following functions: `nodepartition()` and `calculation_cut_capacity`.

```python
def nodepartition(G, o, d):
    allnodepartition = []
    od= [o,d]
    tocombine = list(set(G.nodes()) - set(od))
    for i in range(len(tocombine)+1):
        combination = combinations(tocombine, i)
        for c in list(combination):
            partition1 = list(c)
            partition1.insert(0, o)
            partition2 = list(set(G.nodes()) - set(partition1))

            allnodepartition.append([partition1,partition2])
    return allnodepartition
```

```python
def calculation_cut_capacity(G, allpartitions):
    allcutcapacity = []
    for p in allpartitions:
        partition1 = p[0]
        partition2 = p[1]
        cutcapacity = 0
        for node1 in partition1:
            for node2 in partition2:
                if (node1, node2) in G.edges():
                    cutcapacity += G[node1][node2]['capacity']
        allcutcapacity.append(cutcapacity)
    return allcutcapacity
```

So, the $o-d$ min-cut has capacity **3**, thus the **minimum aggregate capacity** that needs to be removed for no feasible flow from o to d to exist is **3**.

The results are checked with NetworkX functions for flow applications, e.g., `networkx.algorithms.flow.maximum_flow` and `networkx.algorithms.flow.minimum_cut`. They both returns **3**, so, as the **Max-Flow Min-Cut Theorem** states, the minimum cut capacity and the maximum throughput are equal, and they also coincide with the **minimum aggregate capacity** that needs to be removed for no feasible flow from o to d to exist.

The flow distribution is the following:

$$
\begin{aligned}
&o : \{a : 1, b : 2\},\\
&a : \{d : 1\},\\
&b : \{d : 1, c : 1\},\\
&c : \{d : 1\},\\
&d : \{\}
\end{aligned}
$$

These results are shown in Fig.2:

1.2 Maximum capacity removed without affecting the maximum throughput

The maximum throughput is 3 and it is strictly related to the node partitions which gives the minimum cut, that are:

- $U = \{o, a, b, c\} \quad U^C = \{d\} \rightarrow \ C_U = 3$
- $U = \{o, a\} \quad U^C = \{b, c, d\} \rightarrow \ C_U = 3$

So, if we want to remove the maximum capacity without affecting it, the capacities of the links that contributes to that minimum capacity cannot be changed. Therefore, the capacities of e_2, e_3, e_4 and e_6 cannot be decreased, otherwise the min-cut capacity and so the maximum throughput will decrease.

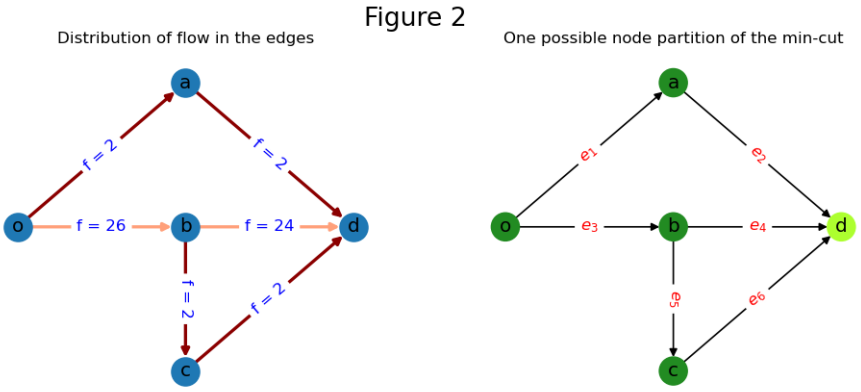

Figure 2: the distribution of the flow on the left and one possible node partition of the min-cut on the right

The only capacities that can be removed without having any change on $\tau^*_{o,d}$ are the ones of e_1 and e_5.

For example, we try to remove 1 from c_1 and 1 from c_5, let's compute the new updated cut capacities:

- $U = \{o, a, b, c\},\ U^C = \{d\} \rightarrow C_U = 3$ $\quad$ $U = \{o, a\},\ U^C = \{b, c, d\} \rightarrow C_U = 3$
- $U = \{o, a, b\},\ U^C = \{c, d\} \rightarrow C_U = 3$ $\quad$ $U = \{o, b\},\ U^C = \{a, c, d\} \rightarrow C_U = 3$
- $U = \{o, a, c\},\ U^C = \{b, d\} \rightarrow C_U = 4$ $\quad$ $U = \{o, c\},\ U^C = \{a, b, d\} \rightarrow C_U = 4$
- $U = \{o, b, c\},\ U^C = \{a, d\} \rightarrow C_U = 3$ $\quad$ $U = \{o\},\ U^C = \{a, b, c, d\} \rightarrow C_U = 4$

The min-cut capacity will remain 3.

Instead, if we remove 2 from c_1 and 0 from c_5, we will obtain the following cut capacities:

- $U = \{o, a, b, c\},\ U^C = \{d\} \rightarrow C_U = 3$ $\quad$ $U = \{o, a\},\ U^C = \{b, c, d\} \rightarrow C_U = 3$
- $U = \{o, a, b\},\ U^C = \{c, d\} \rightarrow C_U = 4$ $\quad$ $U = \{o, b\},\ U^C = \{a, c, d\} \rightarrow C_U = 3$
- $U = \{o, a, c\},\ U^C = \{b, d\} \rightarrow C_U = 4$ $\quad$ $U = \{o, c\},\ U^C = \{a, b, d\} \rightarrow C_U = 3$
- $U = \{o, b, c\},\ U^C = \{a, d\} \rightarrow C_U = 2$ $\quad$ $U = \{o\},\ U^C = \{a, b, c, d\} \rightarrow C_U = 2$

And so, the min-cut capacity and the maximum throughput will be 2.

As last try, if we remove 2 from c_5 and 0 from c_1:

- $U = \{o, a, b, c\},\ U^C = \{d\} \rightarrow C_U = 3$ $\quad$ $U = \{o, a\},\ U^C = \{b, c, d\} \rightarrow C_U = 3$
- $U = \{o, a, b\},\ U^C = \{c, d\} \rightarrow C_U = 2$ $\quad$ $U = \{o, b\},\ U^C = \{a, c, d\} \rightarrow C_U = 3$
- $U = \{o, a, c\},\ U^C = \{b, d\} \rightarrow C_U = 4$ $\quad$ $U = \{o, c\}, U^C = \{a, b, d\} \rightarrow C_U = 5$
- $U = \{o, b, c\},\ U^C = \{a, d\} \rightarrow C_U = 4$ $\quad$ $U = \{o\}, U^C = \{a, b, c, d\} \rightarrow C_U = 4$

Also in this case, the min-cut capacity and the maximum throughput will be 2.

The only feasible solution is to remove 1 from c_1 and 1 from c_5 , since this change doesn't affect the maximum throughput $\tau^*_{o,d}$. In my work this result is demonstrated and checked with an algorithm, that returns:

- The total amount of capacity that can be removed which is **2**.
- The amount of capacity that can be removed from every node: $(o, a) : 1,\ (a, d) : 0,\ (o, b) : 0,\ (b, d) : 0,\ (b, c) : 1,\ (c, d) : 0$.

Fig. 3 shows us the graph with the original capacities and the graph with the updated ones.

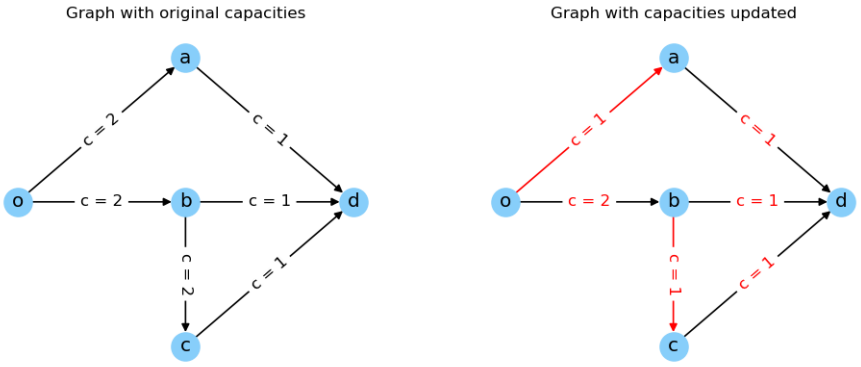

Figure 3: The graph with the original capacities on the left and the and the graph with the capacities updated on the right

1.3 Allocate units $x > 0$ of capacity

If $x > 0$ extra units of capacity are given, they should be distributed in order to maximize the throughput that can be sent from o to d, i.e. the capacity of the min-cut. So, they should be allocated on the links between the set of the minimal cut:

- $U = \{o, a, b, c\},\ U^C = \{d\} \rightarrow C_U = 3$
- $U = \{o, a\},\ U^C = \{b, c, d\} \rightarrow C_U = 3$

i.e. on e_2, e_3, e_4, e_6.

To solve this exercise, I have implemented an algorithm that automatically distribute the units x of capacity in such a way that the **throughput is maximized**, by calculating each time that I allocate a new capacity, all the partitions that gives the minimum cut and find between all the set of edges that contribute to the minimum cut, the one which gives the maximum throughput with the addition of a 1 unit of capacity.

In Fig.4 it is shown how the maximum throughput $\tau^*_{o,d}$ of the graph varies by the units of capacity c added:

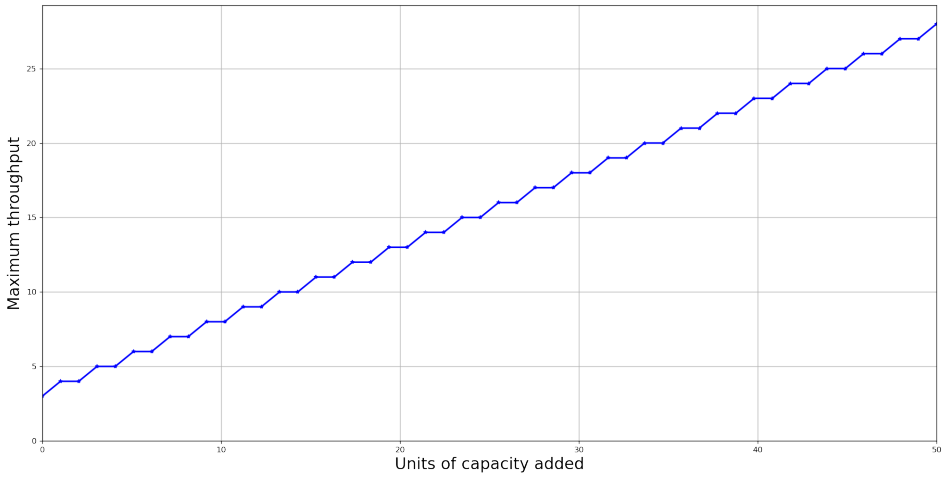

Figure 4: How the maximum throughput $\tau^*_{o,d}$ of the graph varies by the units of capacity c added.

2 Exercise 2

There are a set of *people* $\{p_1, p_2, p_3, p_4\}$ and a set of *books* $\{b_1, b_2, b_3, b_4\}$. Each person is interested in a subset of books, specifically:

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

This problem can be represented with a *bipartite* graph where the nodes represent the people and the books, and the edges represent the interest of a specific person to a specific book. Fig. 5 shows the bipartite graph of the exercise:

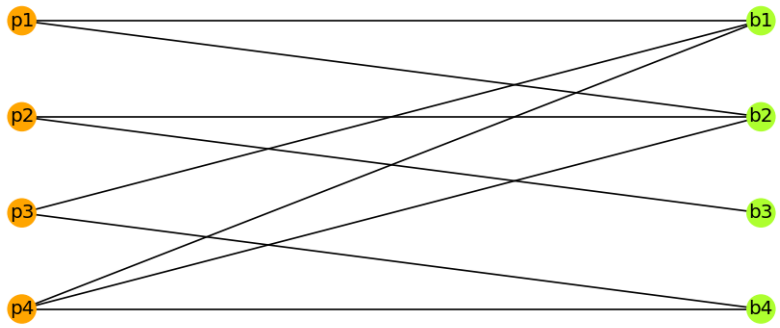

Figure 5: Bipartite graph

2.1 Perfect matching

A matching M in G is perfect for a specific partition of node V if every node in V is matched in M. In this case we have a bipartite graph with:

$$V_0 = \{p_1, p_2, p_3, p_4\} \qquad V_1 = \{b_1, b_2, b_3, b_4\}$$

The theory states that for a simple bipartite graph $G = (V, E)$, in order to have a perfect matching for a partition $V_0 \subseteq V$, the cardinality of every subset U of V_0 must be:

$$|\mathcal{U}| \leq |\mathcal{N}_\mathcal{U}| \quad \forall \mathcal{U} \subseteq V_0,$$

where

$$\mathcal{N}_\mathcal{U} = \sum_{i \in \mathcal{U}} \mathcal{N}_i$$

is the neighborhood of $\mathcal{U}$ in G.
This is the **Hall's marriage theorem**. Note that a perfect matching (both for V_0 and V_1) can exist only if $|V_0| = |V_1|$. To verify that a perfect matching exists, I have implemented the following function that returns `True` if a perfect matching exists given a bipartite graph:

```python
def exist_perfect_matching(G, V_0, V_1):
    for k in range(len(V_0)):
        for w in range(k+1, len(V_0)+1):
            people = V_0[k:w] # Take all the possible combination of node of
                V_0
            neighbour = set() # List of all the neighbour of people, i.e., the
                books of interest
            for i in people:
                for j in V_1:
                    if (i,j) in list(G.edges()) or (j,i)  in list(G.edges()): #
                         if the edge (i,j) exists
                        neighbour.add(j)# Add the node j to the list of
                            neighbour
            if len(neighbour) < len(people): # if |N_s| < |U| where U = any
                partition of V_0
                        return False
    return True
```

But there is also an analogy between maximal flows and perfect matching. In order to use this analogy we need to modify our graph. We start by adding a source node s and a tail node t to our graph. Then, we add directed edges from the source s to all the node-partition V_0, and from all the node-partition V_1 to the tail t.

Then, for every undirected edge (i, j) in G, we transform it in a directed one (i, j) where $i \in V_0$ and $j \in V_1$.

The graph now is directed, and we set all the edges with capacity $c = 1$.

The theory states that: **a V_0-perfect matching on G exists if and only if it there exists a flow with throughput $|V_0|$ on the network G.**

Moreover, given a node $i \in |V_0|$, if a $|V_0|$-perfect matching exists, then there exists $k \in V_1$ such that

$$f_{(i,j)} = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{otherwise} \end{cases}$$

Thus, given a maximum flow with throughput $|V_0|$, the associated V_0-perfect matching can be found by selecting all the edges (i, j) $(i \in V_0, j \in V_1)$ such that $f_{(i,j)} = 1$. Fig.6 represents the new modified graph: Since the

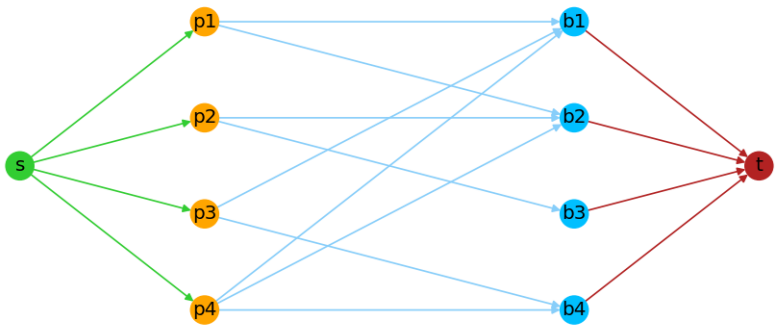

Figure 6: Graph with source s and sinkt

maximum flow that can be send on the new graph is 4 and it is equal to the cardinality of both set V_0 and V_1 **a perfect matching for both sets can exist**.

The perfect matching is obtained with the following function:

```python
def perfect_matching(G, o , d):
    flow, flow_distribution = nx.algorithms.flow.maximum_flow(G, o, d)

    dict1 = flow_distribution.keys()
    dict2 = flow_distribution.values()
    perfect_matching = []
    for node1 in dict1:
        dict2 = flow_distribution[node1]
        for node2 in dict2:
            if dict2[node2] == 1 and node1 != 's' and node2 != 't':
                perfect_matching.append((node1,node2))
    return perfect_matching
```

And in this case is:

$$(p1, b2) \quad (p2, b3) \quad (p3, b1) \quad (p4, b4)$$

Fig. 7 will show the edges that create this perfect matching:

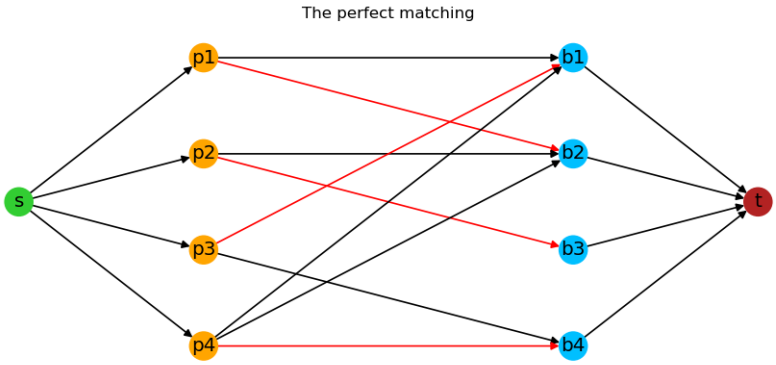

Figure 7: The perfect matching with the edges which make it underlined in *red*

2.2 Matching with multiple copies of books

Now, there are multiple copies of books: the distribution is $(2, 3, 2, 2)$. Each person can take an arbitrary number of different books.

In this case the perfect matching for both people and books can exist only if the copies of each book available are equal to the number of people that have a preference for that specific book

$$| \textit{ available copies of book } i | = | \textit{ people interested in book } i | \quad \forall i = 1, 2, 3, 4$$

It will be perfect if all the people satisfies all their preference and all the available copies are taken.

A matching will be **perfect** for the **set of people** if each person satisfy all his preferences, i.e., if he can take all the books he is interested in.

For the **set of books** a matching will be **perfect** if all the available copies are taken.

We can see that the number of people interested in book 1 is 3, but the available copies for that specific book are just 2. In this case a perfect matching for the set of people cannot exist since there aren't enough copies of book 1. On the other hand, we have 2 copies of book 3 and just 1 person interested in it, so it is impossible that all the copies of all the possible books will be taken. Also in this case, a perfect matching for the set of books cannot exist.

For this exercise, the capacities must be updated:

1. All the capacities of the edges from s to the set of people will be equal to the number of preferences of each different person since all the people can take an arbitrary number of different book that prefer;
2. All the edges from the set of people to the set of books have capacity $c = 1$ since a person can take just one copy of the same book;
3. The edges from the set of books to t will have capacity c equals to the number of copies that are available for that specific book.

Fig.8 shows the graph with the updated capacities: Using the same function `perfect_matching()` with the

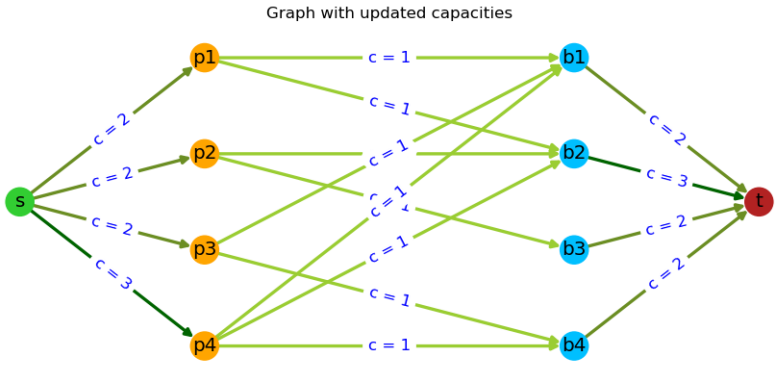

Figure 8: graph with updated *capacities*

capacities updated we can obtain one of the possible matching:

$$[(p1, b2),\ (p2, b2),\ (p2, b3),\ (p3, b1),\ (p3, b4),\ (p4, b1),\ (p4, b2),\ (p4, b4)]$$

And by calculating the length of the array of this possible matching we have the number of book of interest that can be assigned in total, that is: 8.

Fig. 9 represents the matching returned by the function:

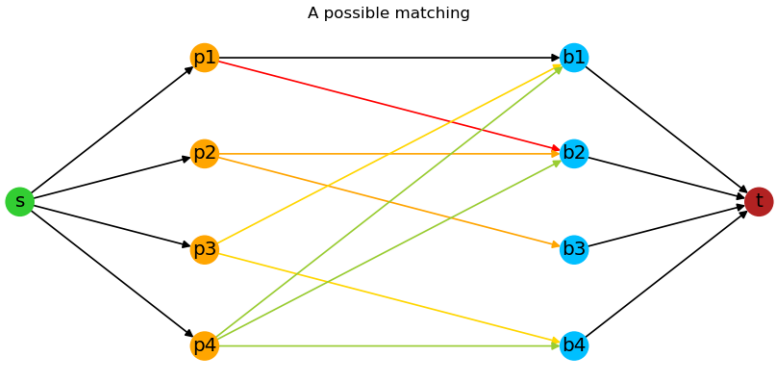

Figure 9: One of the possible matching

2.3 Exchange books to maximize the number of assignments

If the library can choose to sell a copy of a book and buy a copy of another book,it should choose the book in order to maximize the number of assignments. From the figure9 we can see that p_1 wanted b_1 copy, but he couldn't obtain it since there aren't enough copies. Instead book b_3, which had 2 copies, still has one copy left that is not taken by any people. So the library should sell one copy of b_3 and buy one of b_1 With this function the result can be checked :

```python
def find_book_tochange(G, books):
    i = 1 # No books need some changes
    for book in V_1:
        if G.in_degree(book) != G[book]['t']['capacity']: # If the interest of
            that book is different from the number of the copies
            i += 1
            n = G.in_degree(book) - G[book]['t']['capacity']
            if n > 1:
                print('The library should buy',n,'more copies of book', book)
            elif n == 1:
                print('The library should buy',n,'more copy of book', book)
            elif n < -1:
                print('The library should sell',-n,'copies of book', book)
            else:
                print('The library should sell',-n,'copy of book', book)
            G[book]['t']['capacity'] +=  n #Update capacities
    if i == 1:
        print('No more books must be bought or sold')
```

Fig. 10 represents the graph with the new capacities after having sold a copy of book 3 and bought a copy o book 1

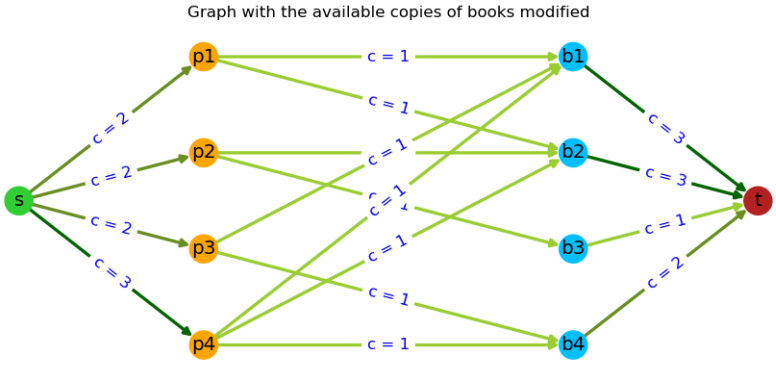

Figure 10: graph with the new capacities after having sold a copy of book *3* and bought a copy o book *1*.

3 Exercise 3

An approximate highway map of the highway network in Los Angeles is given in Fig. 11, covering part of the real highway network.

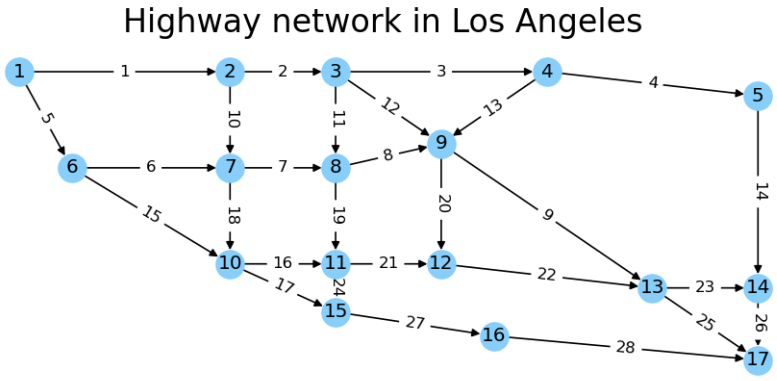

Figure 11: Highway map of network in LA

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_{ei}. Furthermore, each link has a minimum travelling time l_{ei} , which the drivers experience when the road is empty. 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 $+\infty$.

3.1 The shortest path

The shortest (fastest) path can be computed using the following function `nx.shortest_path` which implements *Dijkstra's algorithm.*

The shortest path from node 1 to node 17 is given by the following set of nodes

$$\{1, 2, 3, 9, 13, 17\}$$

which are connected by the following edges $\{(1,2),(2,3),(3,9),(9,13),(13,17)\}$, i.e, $\{e_1, e_2, e_{12}, e_9, e_{25}\}$. Fig.12 shows the shortest path:

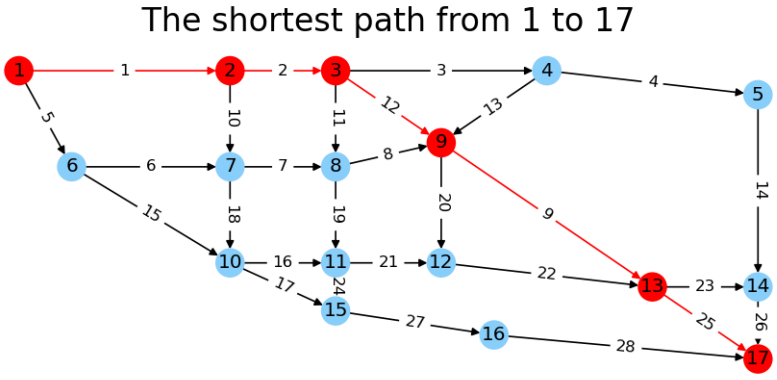

Figure 12: Shortest path.

3.2 Maximum flow

The maximum flow can be computed using the following function : `nx.algorithms.flow.maximum_flow` which represents Ford-Fulkerson's algorithm. The maximum flow from node 1 to node 17 is 22448. It can be reached with the following flow distribution:

$$1 : \{2 : 8741, 6 : 13707\}, \quad 2 : \{3 : 8741, 7 : 0\}, \quad 3 : \{4 : 0, 8 : 0, 9 : 8741\},$$
$$4 : \{5 : 0, 9 : 0\}, \quad 5 : \{14 : 0\}, \quad 6 : \{7 : 4624, 10 : 9083\},$$
$$7 : \{8 : 4624, 10 : 0\}, \quad 8 : \{9 : 4624, 11 : 0\}, \quad 9 : \{13 : 6297, 12 : 7068\},$$
$$10 : \{11 : 825, 15 : 8258\}, \quad 11 : \{12 : 825, 15 : 0\}, \quad 12 : \{13 : 7893\},$$
$$13 : \{14 : 3835, 17 : 10355\}, \quad 14 : \{17 : 3835\}, \quad 15 : \{16 : 8258\},$$
$$16 : \{17 : 8258\}, \quad 17 : \{\}$$

Fig.13 shows the flow distribution on the edges:

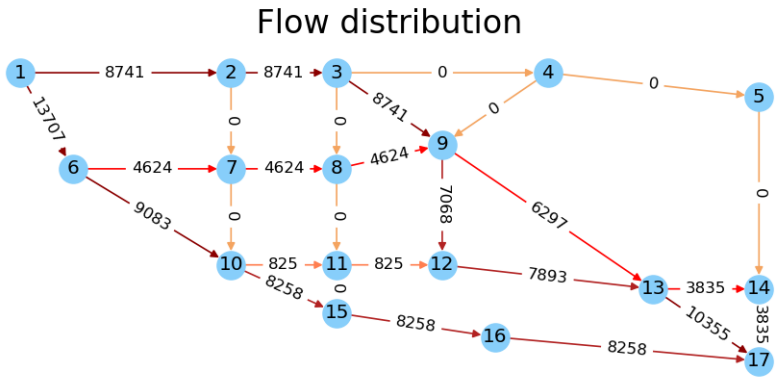

Figure 13: Flow distribution on the edges, the intensity of red is proportional to the flow

3.3 External inflow ν

Given the flow vector f given in *flow.mat*, the exogenous net flow vector is

$\nu = [16806, 8570, 19448, 4957, -746, 4768, 413, -2, -5671, 1169, -5, -7131, -380, -7412, -7810, -3430, -23544]$

Fig.14 shows the external inflow ν for each node:

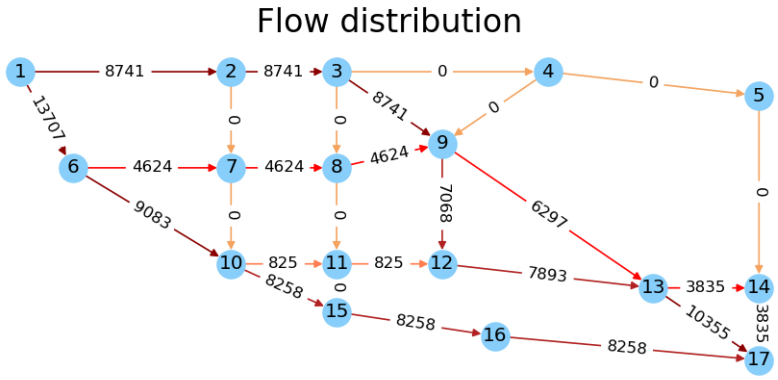

Figure 14: External inflow: red for positive flows, red for negative ones

The exogenous net flow vector that we will consider from now on is

$$\nu = [16806, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -16806]$$

3.4 Social optimum f^*

The social optimum flow distribution is defined as the one minimizing the total delay $\sum_{e\in\mathcal{E}} \psi_e(f_e)$, where on every link $e \in \mathcal{E}$, the cost is $\psi_e(f_e) = f_e\tau_e(f_e)$ with $\tau_e(f_e)$ being the delay function. In this exercise the social optimum f^* with respect to the delays on the different links $\tau_e(f_e)$ is obtained minimizing this cost function:

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

subject to the flow constraint : $Bf = \nu, \quad 0 \le f_e \le C_e \quad \forall\, e \in \mathcal{E}$.
The vector of social optimum flow is reported in `Homework1.ipybn`.
Fig.15 shows the social optimum flow for each edge:

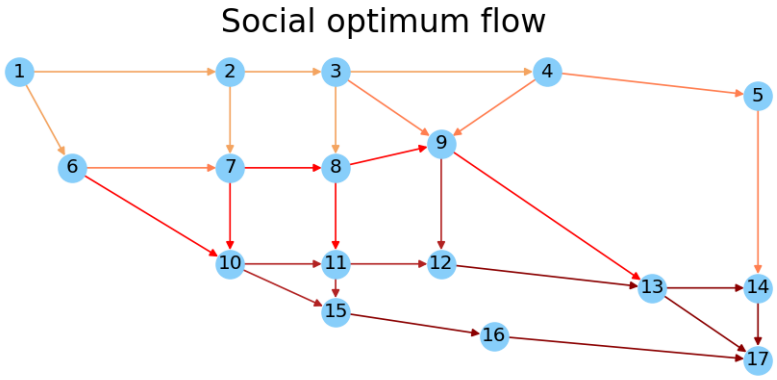

Figure 15: social optimum flow, the intensity of red is proportional to the flow

3.5 Wardrop Equilibrium

The theory states that $f^{(0)}$ is a Wardrop equilibrium if and only if it is solution of a network flow optimization given that the cost functions $\psi_e(f_e)$ are chosen as

$$\psi_e(f_e) = \int_0^{f_e} \tau_e(s)\ \mathrm{d}s.$$

For our problem, the cost function is

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

Subject to the flow constraint : $Bf = \nu, \quad 0 \le f_e \le C_e \quad \forall e \in \mathcal{E}$.
The Wardrop equilibrium flow vector is reported in `Homework1.ipybn`.
Fig.16 shows the Wardrop equilibrium for each edge.
If we introduce tolls, such that the toll on link e is $\omega_e = f_e^*\tau_e'(f_e^*)$, where f_e^* is the flow at the system optimum.
Now the delay on link e is given by $\tau_e(f_e) + \omega_e$.
The new Wardrop equilibrium with tools is reported in `Homework1.ipybn`
Fig.17 shows the the new Wardrop equilibrium with tools for each edge.

By the results, the system optimum flow and the flow in Wardrop equilibrium with tolls are very close to each other.The graph in figure 1818 will plot the system optimum flow vector, the Wardrop equilibrium vector, and the new Wardrop equilibrium with tools, and it will show the similarity.

As we can see, the vector of social optimum and the one of Wardrop equilibrium with tools are almost overlapped.

3.6 Traffic application: system optimum vs Wardrop equilibria

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

$$\psi_e(f_e) = f_e(\tau_e(f_e) - l_e)$$

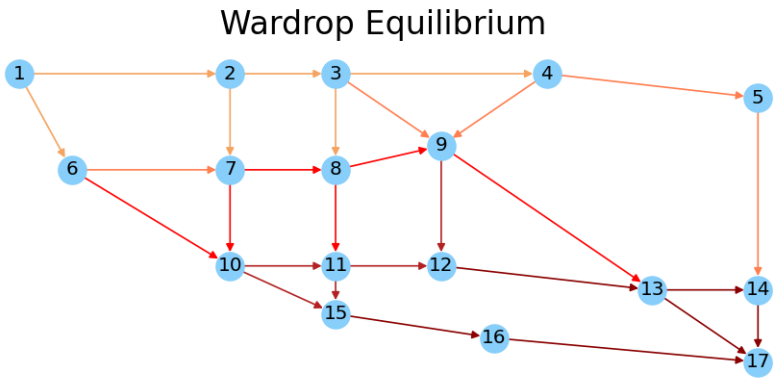

Figure 16: Wardrop equilibrium, the intensity of red is proportional to the flow

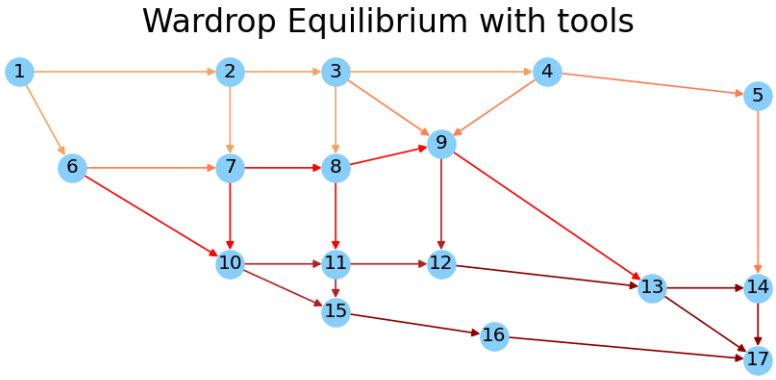

Figure 17: The new Wardrop equilibrium with tools, the intensity of red is proportional to the flow

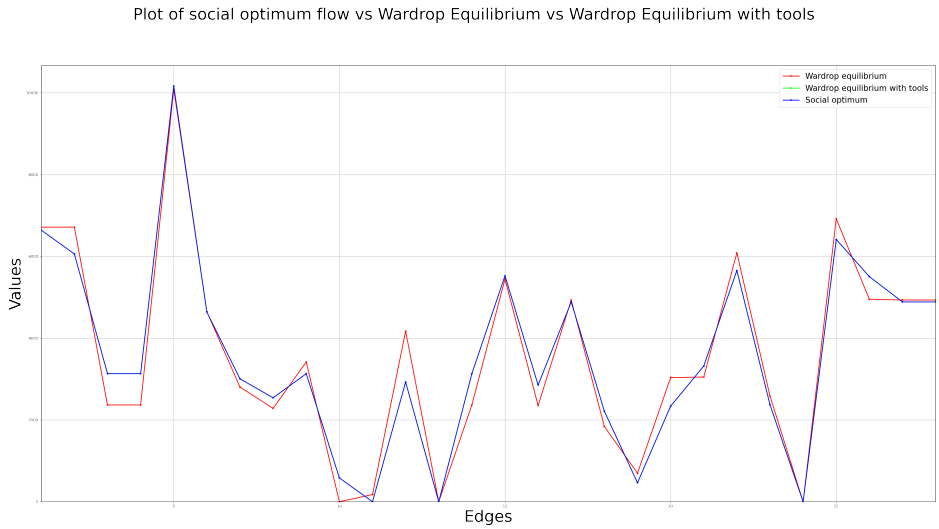

Figure 18: Plot of the 3 flow vectors: *blue* represent the social optimum, *red* the Wardrop equilibrium, *green* the Wardrop equilibrium with tools

subject to the flow constraints.

The system optimum f^* for the costs above is reported in `Homework1.ipybn`. Then, we have constructed tolls ω^* such that the Wardrop equilibrium $f^{(\omega^*)}$ coincides with f^*.

The theory states that **an optimal toll configuration is** $\omega_e^* = \psi_e'(f_e^*) - \tau_e(f_e^*)$ **for all links** e. The tools and the new Wardrop equilibrium with the constructed tolls $f^{(\omega^*)}$ are reported in `Homework1.ipybn`.

Now, the social optimum flow vector and the Wardrop equilibrium with tools differs a bit more but they are still very close to each others.

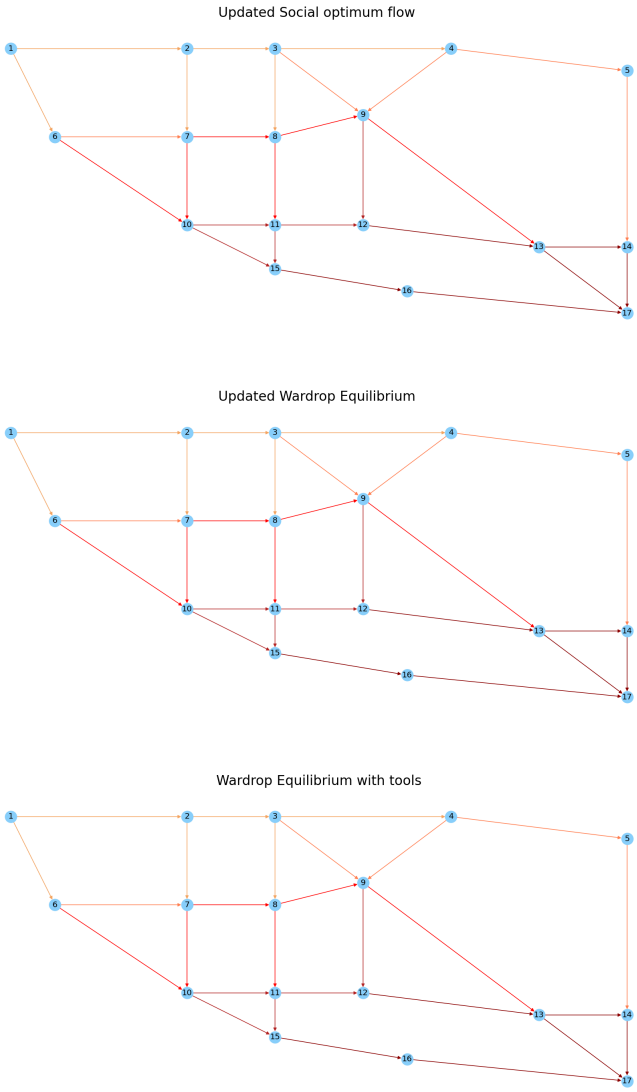

Figure 19: The social optimum on top, the Wardrop equilibrium in the middle, the Wardrop equilibrium with tools in the bottom, the intensity of red is proportional to the flow

Figure 20: Plot of the 3 flow vectors modified: *blue* represent the social optimum, *red* the Wardrop equilibrium, *green* the Wardrop equilibrium with tools.