Homework 1 - Network Dynamics and Learning

Fabrizio Pisani s305391, Matteo Zulian s310384

November 19, 2023

The hand-written solutions, the code and the report has been done in collaboration without a strict division of work.

Exercise 1

Ex 1.a

Introduction

To solve the following exercise, we applied two solving methods: hand-written method and using Python (Networkx Library) for result verification.

Hand-written solution

A $o - d$ cut divides the network in two parts: on one side (the left) there is the origin, on the other side (the right) there is the destination. The aggregate capacity of the links on the out-boundary U is

$$C_U = \sum_{e \in \Delta U} c_e$$

The goal of this point was to find the minimum aggregate capacity, i.e. we solved this problem:

$$C^*_{od} = \min C_U$$

To do this, we found the totality of the paths and all the cuts, as shown in figure(1).

We calculated the aggregate capacity of the cuts:

$$C_r = c_1 + c_3 = 6$$

$$C_y = c_2 + c_3 = 5$$

$$C_b = c_2 + c_4 + c_5 = 7$$

$$C_g = c_2 + c_3 + c_6 = 5$$

We found that the minimum aggregate capacity that needs to be removed to disconnect o to d is

$$C^*_{od} = C_g = C_y = 5$$

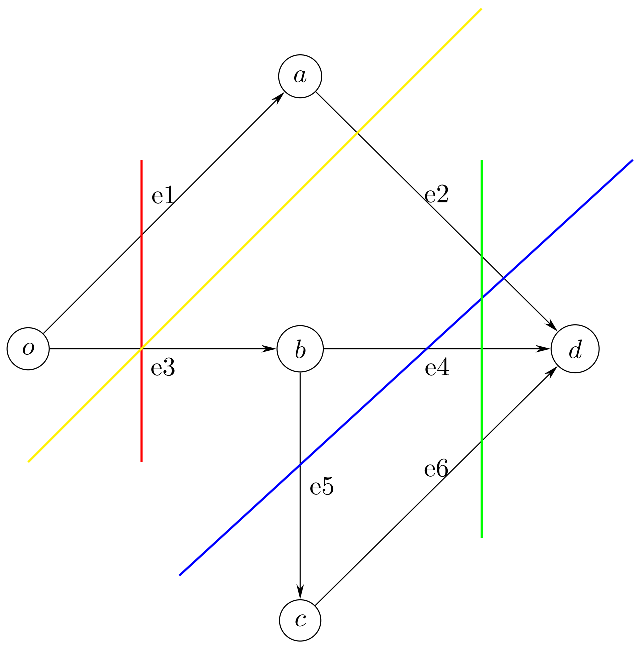


Figure 1: Graph with cuts

Python solution

To solve this point we used the function `minimum_cut_value(G, source_node, sink_node, capacity)` to find the minimum aggregate capacity to disconnect o to d. This function returns the result *cut_value*. The python solution verified our hand-written one.

Ex 1.b

Introduction

To solve the following exercise, we used the same approach of the point a.

Hand-written solution

We exploited the max flow-min cut theorem to obtain the throughput τ, which is:

$$\tau = C^*_{od} = 5.$$

We operated as follow: since the throughput $\tau = 5$ must not be affected, we then removed the excess capacities from the links ensuring that minimum aggregate capacity of the graph remains $C^*_{od} = 5$. To do this, we removed capacity units only from non-critical links, i.e. the links that do not belong to the minimum cut found in the point a.

$$c_{e1} = c_{e2} = c_{e4}$$

$$c_{e3} = 3$$

$$c_{e5} = c_{e6} = 1$$

We obtained that the maximum aggregate capacity that can be removed from the links without affecting the maximum throughput from o to d is 3, and the links with excess capacity are e_1 (capacity removed = 1) and e_5 (capacity removed = 2).

Python solution

To solve this problem, we calculated the maximum flow in the graph with the function `maximum_flow(G, 'o', 'd', capacity)`. Subsequently, we determine the excess capacity on each edge, i.e., the difference between the maximum capacity of the edge and the actual maximum flow through it. We did it using a loop that iterates through each edge in the graph and calculates the difference between the maximum capacity of the edge and the actual maximum flow. Finally, we calculate the maximum aggregate capacity that can be removed without affecting the maximum flow from $'o'$ to $'d'$. The results are the same of the hand-written solution.

Ex 1.c

Introduction

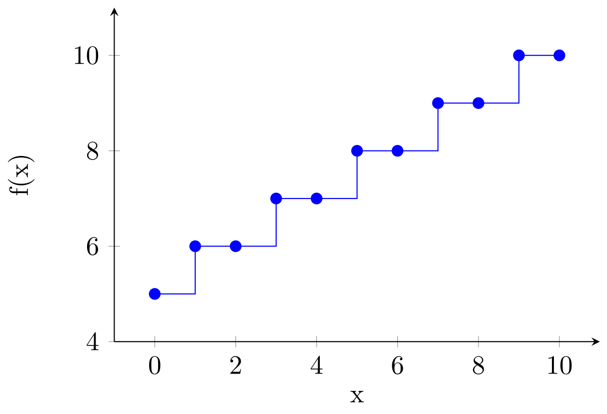
Given the computational complexity, to solve the following exercise we wrote a Python program(NetworkX library).

Python solution

The algorithm we designed follow this intuitively idea: if we want to increase the max flow we have to increase the so called *bottleneck* in the graph; that's why we look for the edges that belong to a min-cut partition and we try to increase each one of theme to see if there is an improvement. To be precise the optimal solution is always found only if we look at all the partition with min-cut value, but the function in Networkx library only gives us one minimum partition, this means that sometimes it seems like there is no edge that can improve the value, even if there is one in another minimum partition, giving us sub-optimal solutions. Most of the times we can avoid that by forcing the algorithm to improve edges with bigger capacity first, that's because we want the algorithm to improve the same edges over and over, so in the end we will see a set of n edges that, one at a time, are increased, where n is the length of the shortest path from o to d. This is why after a while we see a function plot that is $stairs - like$ where the width of each step is n, (in this case n is 2, in fact we see that every 2 iteration f increases).

Algorithm 1: f(x)

```
initialize empty vector $f$;
for $i < x$ do
    find a min-cut partition;
    extract the edges of that partition;
    sort in decreasing order;
    for each edge do
        increase edge capacity;
        compute new max flow;
        if new max flow > old max flow then
            save result in $f$ and go to next iteration of i;
        else
            undo modification;
    if no edge can improve max flow then
        increase the biggest edge;
return $f$;
```



Exercise 2

Ex 2.a

Introduction

To solve the following exercise, we applied two solving methods: hand-written method and using Python (NetworkX library) for result verification.

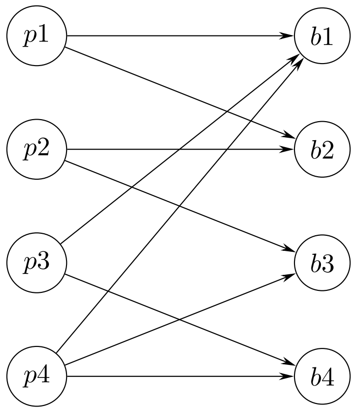


Figure 2: Bipartite graph representation of people and books.

Hand-Written Solution

Firstly, we verify the existence of the perfect matching, and to do that we use Hall's theorem. Therefore, it is possible to proceed with the identification of the perfect matching. To solve the problem manually applying the max flow theory, we added to the original graph a *source* (node o) and a *sink* (node d). We connected the source to all nodes in set p, and we connected all nodes in set b to the sink destination, as shown in figure 3

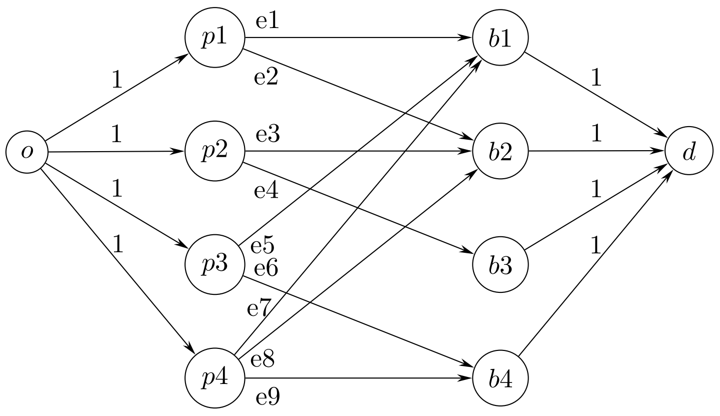


Figure 3: Graph representation with source and sink.

To solve the problem, we imposed the constraint of $in - flow = out - flow$ such that the outgoing flow from p_i and the flow coming into b_i must be equal to one. We can then write the following system of linear equations:

$$\begin{cases} e1 + e2 = 1 & \text{(out-flow constraint from } p1\text{)} \\ e3 + e4 = 1 & \text{(out-flow constraint from } p2\text{)} \\ e5 + e6 = 1 & \text{(out-flow constraint from } p3\text{)} \\ e7 + e8 + e9 = 1 & \text{(out-flow constraint from } p4\text{)} \\ e1 + e5 + e7 = 1 & \text{(in-flow constraint in } b1\text{)} \\ e2 + e3 + e8 = 1 & \text{(in-flow constraint in } b2\text{)} \\ e4 = 1 & \text{(in-flow constraint in } b3\text{)} \\ e6 + e9 = 1 & \text{(in-flow constraint in } b4\text{).} \end{cases}$$

where each e_i is a boolean variable set to 0 if that link is not part of the perfect matching, and 1 if that link is part of the perfect matching. A possible solution to the system is the perfect matching shown in figure 4:

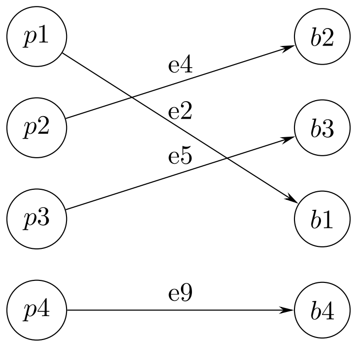


Figure 4: Graph representation of the specified connections.

Python Solution

To solve this point we used the function `maximum_flow(G, source_node, sink_node, capacity)` to find the maximum flow in G from the source node to the sink node, where G is the graph in figure (3).
This function also returns a `dictionary` of all the edges with $non - zero$ flow, that we used to determine the partition of the graph. The result obtained with Python is analogous to what was calculated manually, confirming the validity of the solution.

Ex 2.b

Introduction

Here we assume having multiple copies of the same book, the goal is to calculate the maximum number of books that can be assigned using analogies with max-flow problems. We assumed that each person can take as many different books as they want, but only one copy of the same book.

Hand-Written Solution

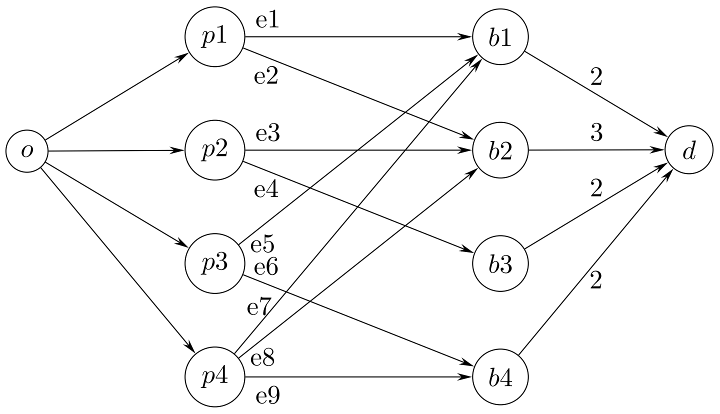


Figure 5: Graph representation with source and sink.

Starting from the solution of *Ex* 2.*a*, we changed the flow capacity of the links $b_i \longrightarrow d$ accordingly to the numbers of copies of b_i, as shown in figure (5), and we wrote the following system of inequalities where e_i is set to 0 or 1. For example, if we consider b_1, the sum of the flows of $e1$, $e5$, and $e7$ can be at most 2 since the incoming flow to b_1 must equal the outgoing flow. This reasoning holds in general. Therefore,

$$\begin{cases} e1 + e5 + e7 \leq 2 \\ e2 + e3 + e8 \leq 3 \\ e4 \leq 2 \\ e6 + e9 \leq 2 \end{cases}$$

The result obtained is that the maximum number of books that can be assigned is 8: the two books of b_i can be assigned to two among p_1, p_3, and p_4; all three books of b_2 are assigned; of the two books of b_3, only one is assigned to p_2; both books of b_4 are assigned. In total, 8 books out of 9 can be assigned (the ninth unassigned book is a copy of b_3).

Python solution

The Python solution for point *b* is analogous to the solution for *Ex* 2.*a*, we used the function `maximum_flow_value(G, source_node, sink_node, capacity)`. Where G is the graph in figure 5 with $o \longrightarrow p_i$ link-capacities equal to infinity because there are no constraints on how many books a person can buy other than the number of copies, which is represented by the $b_i \longrightarrow d$ link-capacities, and that a person can only by one copy of a book $p_i \longrightarrow b_j = 1$

Ex 2.c

Hand-written solution

In this case, to reach the result, we undertaken the following reasoning: since the bookstore can buy one copy of a book and sell the copy of another book, the optimal solution to maximize the number of assigned books (reaching 9) would be to buy the book b_3 (as it was requested by only one person) and sell another copy of book b_1 (since there are two copies, but it is requested by three people).

Python solution

The Python solution is structured in the following steps:

1. Graph Creation (analogous to the previous points).
2. Calculation of the in-degree of books, i.e., how many people are interested in a particular book.
3. Implementation of the buy-sell logic: a loop compares the number of available copies with the number of people interested in that book; if there are more copies available than interested people, the excess copies are sold; if there are more interested people than copies available, then the books in deficit are marked for purchase.
4. Consistency check of the number of books sold with the number of books bought (I can't buy more books than I sold and It doesn't make sense to sell more than I need to).

As expected, the solution found is as follows: one book bought ($b3$) and one book sold ($b1$). Finally, we checked that this maximizes the number of assigned books (i.e., in our example, it increases from 8 to 9 assigned books). This check was successful: therefore, by selling one copy of $b1$ and buying one copy of $b3$ (resulting in `book_copies = [3, 3, 1, 2]`), the number of assigned books is maximized.

Exercise 3

Introduction

In this exercise we were given the highway network in Los Angeles, described by The node-link incidence matrix B, the maximum flow capacity vector c and the minimum travelling time vector l.

Ex 3.a

The shortest path is calculated by a designated function of the python library Networkx `shortest_path(G, source=1, target=17, weight=travel_time)` which gave us: $1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 9 \longrightarrow 13 \longrightarrow 17$

Ex 3.b

Similarly as before we used the function `maximum_flow_value(G, 1, 17)` and we obtained: $\nu = 22448$

Ex 3.c

Given the flow vector in *flow.mat*, we computed the external inflow ν satisfying $Bf = \nu$ and recovered the ν vector of external inflow:

$$\nu^T = \begin{pmatrix} 16282, 9094, 19448, 4957, -746, 4768, 413, -2, -5671, 1169, -5, -7131, -380, \\ -7412, -781, -3430, -23544 \end{pmatrix}$$

New exogenous flow

We assume that the exogenous flow now is :

$$\nu^T = \big(\nu_1, 0, 0, ..., 0, 0, -\nu_1\big) = \big(16282, 0, 0, ..., 0, 0, -16282\big)$$

Ex 3.d

In order to find the social optimum f^*, given a delay function $\tau(f_e)$, a cost function $\psi(f_e) = \tau(f_e)f_e$ and a capacities vector c, we had to solve the minimization problem:

$$f^* = \arg\min_f \sum_{e\in\xi} \psi(f_e) \ , \quad s.t.\ f \geq 0,\ f \leq c,\ Bf = \nu \tag{1}$$

We had to use the cvxpy library so we defined the variable f to minimize, then we created the minimization problem coping the objective function and imposing the linear constraints above (1).
Obtaining as optimal cost $\sum \psi(f_e^*) = 23835$

Ex 3.e

Then we were asked to find the Wardrop equilibrium that is found solving the minimization problem:

$$f^{(0)} = \arg\min_f \sum_{e\in\xi} \int_0^{f_e} \tau(s)ds \ , \quad s.t.\ f \geq 0,\ f \leq c,\ Bf = \nu \tag{2}$$

Solving the integral in (2) gives the new objective function : $-lc\log|1 - \frac{f}{c}|$.
So we replaced the objective function we used in (1) while the constraints remained the same. Again we solved the problem and we obtained a new flow vector $f^{(0)}$ containing the flow of the Wardrop equilibrium.
We computed the wardrop cost $\sum \psi(f_e^{(0)})$, obtaining 24162.
As we expected from the theory the wardrop cost is greater than the optimal cost.

Ex 3.f

In order to improve the wardrop cost we added tolls w to the network to induce the drivers to take paths that leads to the optimal flow of the network. Our goal was to find a $f^{(w)} = f^*$ and we accomplished that by solving:

$$f^{(w)} = \arg\min_f \sum_{e \in \xi} \int_0^{f_e} \tau(s)ds + w_e^* f_e \ , \quad s.t.\ f \geq 0,\ f \leq c,\ Bf = \nu \tag{3}$$

where $\quad w_e^* = \psi'(f_e^*) - \tau(f_e^*) = \tau'(f_e^*)f_e^* = \frac{l_e c_e}{(c_e - f_e^*)^2} f_e^*$

Obtaining as expected $\sum \psi(f_e^{(w)}) = 23835$, the same as the optimal flow.

Travel costs and PoA with $\psi(f_e)$

Optimal equilibrium cost	23835		
Wardrop equilibrium cost	24162	PoA	1.0137
Wardrop eq. cost with Tolls	23835	PoA	1.0

Ex 3.g

With the new delay function $\tau_2(f_e) = \tau_1(f_e) - l_e$ we had to compute again the objective functions of our problems :

$$\int_0^{f_e} \tau_2(s)ds = \int_0^{f_e} \tau_1(s)ds - l_e f_e \tag{4}$$

$$w_{2,e}^* = \tau_2'(f_e^*)f_e^* = \tau_1'(f_e^*)f_e^* = w_{1,e}^* \tag{5}$$

Where we found that the computation of the tolls w^* remain the same as before, while we just had to replace the integral in (2) and (3) with (4).

Travel costs and PoA with $\psi_2(f_e)$

Optimal equilibrium cost	13334		
Wardrop equilibrium cost	13378	PoA	1.0033
Wardrop eq. cost with Tolls	13334	PoA	1.0