Network Dynamics and Learning (01TXLSM)
Homework 1

Professor: Fabio Fagnani
Student: Juan Aragón (S291466)
S291466@studenti.polito.it

Abstract

In the following report it is presented a complete solution for the first assignment of the course Network Dynamics and Learning, a.a. 2021/2022. The main goal was to put in practice the concepts already learned in the class by following an interactive procedure.

The related code was written in python using libraries such as ***networkx****, used to handle graph structures with their related algorithms, and* ***cvxpy****, for solving convex optimization requirements that will be explained with more detail in the following content. All theoretical aspects, cited on this paper, make reference to the professor's lecture notes [1].*

1. Exercise 1

For the first part of this assignment it was required to consider a graph given in the form $G = (V, E)$, with a unitary o-d network flow, assuming for each link l an integer capacity C_l as in Fig.1.

1.1. Max-Flow Min-Cut Capacity Theorem

What is infimum of the total capacity that needs to be removed for no feasible unitary flows from o to d to exist?

This question can be easily answered by looking at the Max-Flow Min-Cut Capacity Theorem (Theorem 3.3, on [1]). Which states that *"For a given multigraph $G = (V, E, c)$ with a capacity vector $c > 0$, and two distinct nodes $o \neq d$ in V. The maximum throughput from o to d coincides with the minimum capacity among all o-d cuts."*

More explicitly, *Remark 3*, on [1], says that *"the minimum total capacity that a hypothetical adversary needs to remove from the network in order to make node d not reachable from node o coincides with the min-cut capacity $c^*_{o,d}$."*

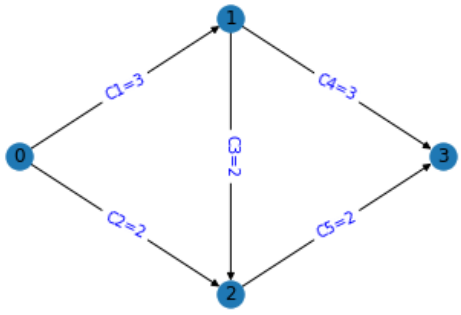

Figure 1: Graph with link capacities required for exercise 1

1.2. Maximal Throughput

With the graph described at the beginning of this section, showed on Fig.1, it was required to determine where should 1 unit of additional capacity be located to maximize the feasible throughput.

For this scope, due to the simplicity of the graph, it was decided to compute a for-loop, increasing the capacity of a single link at a time, and calculating the Min-Cut capacity with the obtained graph using the *networkx* built-in function $algorithms.flow.minimum\_cut()$.

From the obtained results, it was determined that the throughput of the mentioned graph could not be modified by the increment of just 1 unit on any of its capacities. It's maximum throughput remains equivalent to 5 units. This result can be understood by looking at the structure of the graph, where in order to augment the throughput, at least two distinct links capacities need to be incremented.

1.3. Maximal Throughput 2

On this part of the exercise, it was required to determine where 2 additional capacity units should be placed in order to maximize the network's throughput. As in the previous step, all possible cases where analyzed by implementing two nested for-loops and iterating over all links capacities, computing the Min-Cut's algorithm on each scenario.

The obtained results showed that there are three main configurations for distributing the additional capacities, for which the current graph's throughput is augmented, obtaining a maximal throughput of 6 units. Those configurations are: $(C_1, C_3), (C_1, C_5)$ and (C_2, C_5).

1.4. Maximal Throughput 3

The last part of the exercise followed the same approach, this time allocating 4 additional capacities into the same graph. In the implementation, 4 nested for-loops were considered, selecting 4 links at a time and computing the Min-Cut's algorithm for each case. This time the maximal throughput obtained was about 7 units. Which is obtained, for example, by distributing the capacities in the following way: $(C1, C1, C3, C3)$.

2. Exercise 2

For this exercise it was taken into consideration a bipartite graph, connecting a set of people (p_1, p_2, p_3, p_4) with a set of books (b_1, b_2, b_3, b_4) as shown in Fig.2, where each link represents the fact that a given book is considered as interesting for a certain person.

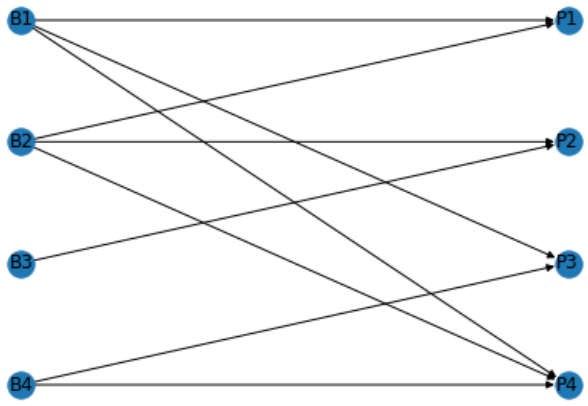

Figure 2: Bipartite graph obtained at exercise 2

2.1. Perfect Matching

According to [1], a *matching* is any subset of links $M \subseteq E$ that have no nodes in common, and it is considered as perfect or complete when all nodes of the graph are matched.

The goal of this exercise was to determine if a perfect match could be found on the bipartite graph shown previously. This exercise's solution was based on the already mentioned Max-Flow Min-Cut Theorem and the Hall's Theorem (Theorem 1.1 on [1]).

Hall's Theorem establishes that *"For a simple bipartite graph $G = (V, E)$ and $V_0 \subseteq V$, there exists a V_0-perfect matching in G if and only if: $|U| \leq |N_U|, \forall U \subseteq V_0$. Where $N_U = \cup_{i \in U} N_i$ is the neighborhood of U in G"*. And as it is mentioned on [1], *"The cardinality of a maximum matching is given by $c^*_{o,d}$"* obtained from the following equation: $c^*_{o,d} = \min_{U \subseteq V_0} \{|V_0| - |U| + |N^+_U|\}$. In this sense, there exists a perfect matching if and only if Hall's Theorem is satisfied ($c^*_{o,d} = |V_0|$).

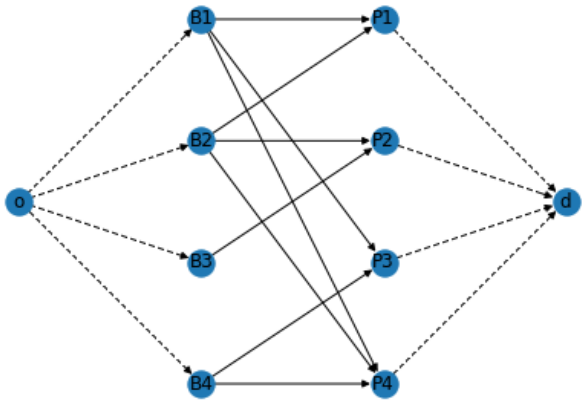

Figure 3: Max-Flow applied to bipartite graph to find Matching cardinality

In order to prove the theoretical aspects previously mentioned, the original graph was modified by adding source and sink nodes as it is shown in Fig.3. Establishing a unitary capacity to the dotted links coming from the source while the rest of the links were set with unlimited capacities. The result of applying the Min-Cut's algorithm to the existing graph was the following, $c^*_{o,d} = 4$, which indicates the presence of at least one perfect matching set.

In order to prove the presence of at least one perfect matching, the *networkx* algorithm *is_perfect_matching()* was computed on every possible combination of the edges of the original graph (at Fig.2). This is how three perfect matching sets were identified, as shown on Fig.4.

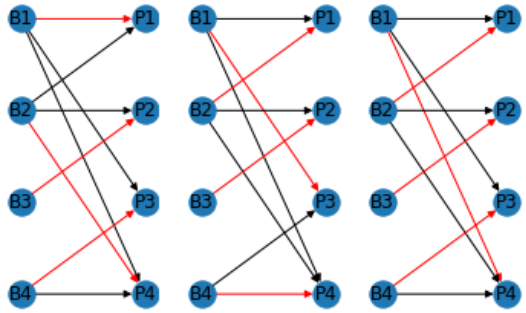

Figure 4: Identified perfect matching sets

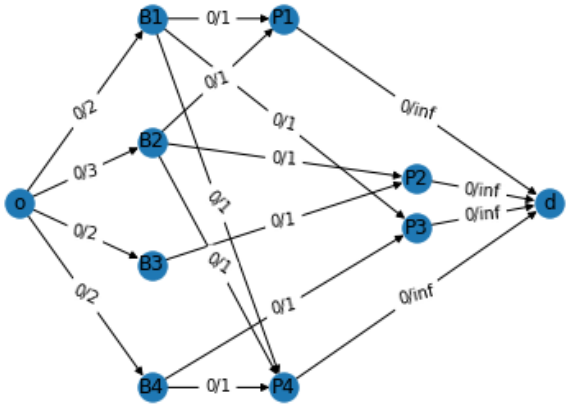

Figure 5: Max-Flow adaptation for the books-assignment problem

2.2. Book copies assignment

For the current exercise it was required to assume the availability of multiple copies of the books, distributed in the following manner $(2, 3, 2, 2)$. The goal was to determine the maximum amount of books that could be assigned to the set of people while respecting their preferences, assuming that each person can have at most one copy for each of his preferred books.

This problem was solved by using a Max-Flow approach, modifying the initial graph to the one shown on Fig.5, where the capacities of the links from the source to any book b represent the amount of copies a book can have, determined by the previously shown distribution. Similarly, the capacities for the existing links, going from any b to a person p are all equal to 1, representing the limited amount of copies a person can have for each of his preferred book. Lastly, the links from any person p to the sink, have unlimited capacities, due to the fact that any person can have unlimited number of books, assigned to him as long as the previous limitations are respected.

After applying the Min-Cut's algorithm to the current graph it was easily obtained the maximum amount of books that could be assigned with the given constraints, which in this case were 8 books. By analysing the result on the graph, it is easy to determine that all the available books were assigned, excepting for one of the two copies available for b_3, which that is indeed demanded only by one person (p_2). All the other copies were assigned successfully.

2.3. Buy & Sell

The last exercise for the second part of the homework consisted of identifying for which of the books it should be necessary to buy/sell one copy, in order to increase the number of assigned copies among the people.

By looking at Fig.5, it is evident that with the current distribution of copies, there is a copy of for b_3 that is not demanded, while there is missing a copy of b_1 to satisfy its entire demand. In order to verify this reasoning it was computed a two nested for-loops procedure, analyzing all possible options regarding selling one copy and buying another. On each iteration it was computed the Min-Cut algorithm, analyzing if the obtained result was superior to the one obtained for the previous exercise.

As determined by the reasoning explained before, the algorithm found an optimal arrangement, modifying the distribution of the copies by adding a copy for b_1 and removing a copy for b_3. In this way it was obtained a final throughput of 9 successfully assigned copies, this time satisfying the total demand.

3. Exercise 3

For the last part of the homework, it was given a highway network of the city "Los Angeles" (Fig.6), consisting of 17 nodes and 28 links, with all its information provided on the following files:

1. ***traffic.mat***. A node-link incidence matrix, where the i-th column has a $+1$ value at the row corresponding to the tail node of link e_i, and a -1 at the corresponding head node.
2. ***traveltime.mat***. A column vector with many rows as number of links, where each value describes the length of the i-th highway divided by the speed limit (60 miles/hour).
3. ***capacities.mat***. A column vector as the previous one, describing the maximum flow capacity for every link.

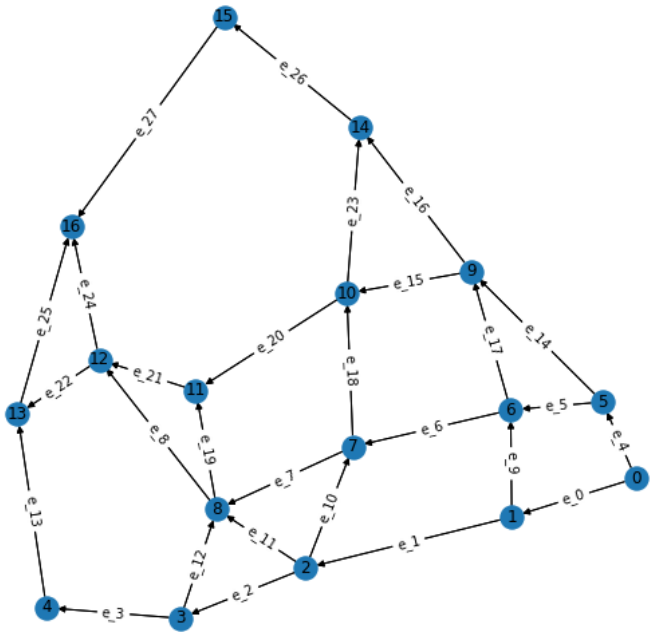

Figure 6: Los Angeles highway network

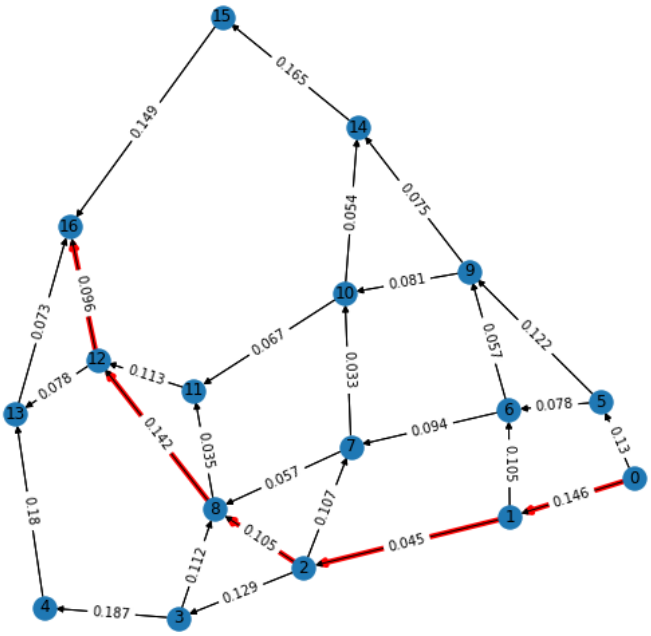

Figure 7: Shortest Path from $node_0$ to $node_{16}$, evaluated on *traveltime* values

3.1. Fastest Path

This exercise consisted of finding the fastest path connecting $node_0$ with $node_{16}$, which is the path with the shortest traveling time.

For this scope, a *Deep First Search* approach was used. That is, all possible paths were analyzed comparing the lengths of those who started on $node_0$ and arrived into $node_{16}$. This approach was implemented on a recursive way, adding the length of a given link into a global variable when the link was considered part of the path, and removing its length after the recursive exploration was finished.

The optimal solution was found on the path made by the links $[e_0, e_1, e_{11}, e_8, e_{24}]$. As it can be seen from Fig.7, where each link shows its corresponding *traveltime* value. This optimal path has the lowest time sum, which is of about 0.533 hours.

3.2. Maximum Flow

The following exercise consisted of finding the maximum flow between $node_0$ and $node_{16}$. For this purpose two different procedures were computed. One based on an implementation of the *Edmonds Karp*'s algorithm and the other based on the previously mentioned *networkx* built-in function, *min_cut()*.

For the implementation of the *Edmonds Karp*'s algorithm, using the data from ***traffic.mat***, it was built the adjacency matrix, a square matrix $|V| \times |V|$, where $c_{i,j} \in |V| \times |V|$ represents the capacity of the link going from $node_i$ to $node_j$. The next step consisted of implementing the *Breadth First Search (bfs)* algorithm, that finds an existing path from any two nodes and returns the "parent" array, consisting of a row vector of the same size as $|V|$, storing on each node's index the index of its corresponding parent. In this way, the *Edmonds Karp*'s algorithm, calls the *bfs()* and while there still exists a feasible path from the origin to the destination, it computes the minimum capacity along the path found, adding its value into the max-flow variable. After that, it modifies the adjacency matrix by subtracting, to any link of the path the minimum of the capacities, and adding it to the links going in the opposite direction. Finaly, the algorithm stops when there are no more feasible paths between the specified nodes and it returns its max-flow value.

After executing both procedures, the obtained results were identical, both giving a maximum flow of 22448 units.

3.3. External Inflow

For this exercise, an additional file was taken into consideration, ***flow.mat***, consisting of an hypothetical flow present on the network, respecting the capacity constraints for every link.

The goal of the exercise was to compute the External Inflow, ν, from the equation $Bf = \nu$, where B is the

incidence matrix, and f the considered flow. The obtained results are shown on Table.1.

As it can be seen from the results, the 17 non-zero values, represent the external inflow for each node on the network, where the negative sign indicates that the given node acts as sink while the negative sign represent additional sources on the network, all nodes introducing or taking out part of the flow.

$node_i$	ν_i	$node_i$	ν_i
$node_0$	16806	$node_9$	1169
$node_1$	8570	$node_{10}$	-5
$node_2$	19448	$node_{11}$	-7131
$node_3$	4957	$node_{12}$	-380
$node_4$	-746	$node_{13}$	-7412
$node_5$	4768	$node_{14}$	-7810
$node_6$	413	$node_{15}$	-3430
$node_7$	-2	$node_{16}$	-23544
$node_8$	-5671		

Table 1: ν, external inflow values obtained from ***flow.mat***

3.4. Social Optimum

The *Social Optimum* is a traffic assignment procedure which is considered non realistic behaviourally speaking. Given a network throughput, it assigns the distribution of the flow in the optimal way, respecting all constraints along the network, and taking into consideration a cost function to minimize, which is usually expressed in terms of delay or time required to arrive into a certain destination.

For this exercise it was assumed a zero exogenous inflow for all the nodes excepting for $node_0$ and $node_{16}$, whose values were ν_0 and $-\nu_0$ respectively, both values taken from the previous exercise (Table 1). In this case, the goal was to find f^*, the social optimum, considering the delays, $d_e(f_e)$, as described on (1).

As mentioned before, it was required to minimize a cost function subject to the flow distribution, representing the total delay experienced on the network. Obtaining the most efficient way for distributing the flow. The cost function for this exercise is shown on (2).

$$d_e(f_e) = \frac{l_e}{1 - f_e/C_e}, \quad 0 \leq f_e < C_e \tag{1}$$

$$\sum_{e \in E} f_e d_e(f_e) = \sum_{e \in E} \left(\frac{l_e C_e}{1 - f_e/C_e} - l_e C_e \right) \tag{2}$$

The minimization problem was solved by using the python library *cvxpy*, which by defining a convex objective and the constraints, as shown in (3), it was able to find the values for the independent variable, f, that minimized the expression (2). The obtained results are shown on Fig.8.

$$Bf = \nu, \quad f \geq 0, \quad f < c \tag{3}$$

3.5. Wardrop Equilibrium

Similar to the previous exercise, the *Wardrop Equilibrium* procedure gives a more realistic intuition of a driver's behavior on the network. It is based on the principle that a given driver will always take the most convenient path for reaching his destination, also by minimizing a cost function.

The purpose of this exercise was to find the *Wardrop Equilibrium*, $f^{(0)}$ which was the flow obtained after minimizing the expression shown in (4). Similarly to the previous exercise, it was solved by using the python's library *cvxpy*.

$$\sum_{e \in E} \int_0^{f_e} d_e(s) ds = -\sum_{e \in E} l_e C_e \ln (C_e - f_e) \tag{4}$$

The first graph on Fig.8 shows the distribution differences between the *Social Optimum flow*, $f^{(0)}$, and the *Wardrop Equilibrium flow*, $f^{(w)}$. From this graph it can be deduced that by following selfish approach, looking at a single's driver optimal solution at a time, as in $f^{(w)}$ results can vary from the system's optimal version.

3.6. Wardrop Equilibrium with tolls

This was a variant of the previous exercise where the concept of tolls was introduced into the network. Usually, a toll is associated to an additional cost for each link, that modifies the behavior of the flow distribution. For a given link e its respective toll was defined as $w_e = f_e^* d_e'(f_e^*)$, where f_e^* corresponds to the link's flow on the *Social Optimum*. For this reason, the new delay function taken into consideration is shown on (5). On this exercise it was required to compute the *Wardrop Equilibrium* considering the new cost function, that is minimizing expression shown in 6.

$$d_e(f_e) = \frac{l_e}{1 - f_e/C_e} + f_e^* d_e'(f_e^*), \quad 0 \leq f_e < C_e \tag{5}$$

$$\begin{aligned} &\sum_{e \in E} \int_0^{f_e} d_e(s) ds = \\ &= \sum_{e \in E} -l_e C_e \ln (C_e - f_e) + f_e \frac{f_e^* l_e C_e}{(C_e - f_e^*)^2} \end{aligned} \tag{6}$$

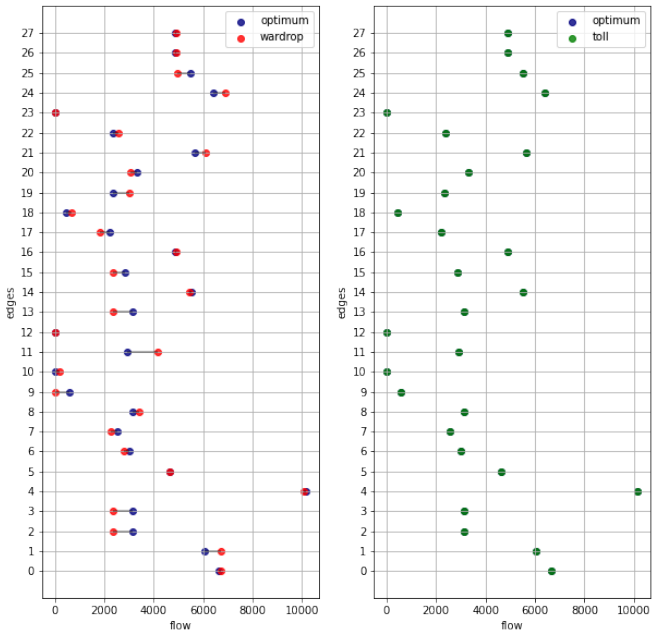

Figure 8: Comparison between the computed Wardrop flows $f^{(0)}$ and $f^{(w)}$ with respect the social optimum flow f^*.

In Fig.8, the graph on the right, shows a comparison between the *Social Optimum* and the currently obtained result. It can be noticed, that paradoxically by increasing the delays at each link the obtained solution is more close to the *Social Optimum* than the *Wardrop flow* obtained at the previous exercise. With a mean relative error of 7.586%, due basically to numerical errors obtained in the computatinal process.

In order to give a better explanation of this phenomena, lets take a look at the Theorem 4.2 on [1], which establishes that *"for a given Wardrop Equilibrium flow, $f^{(w)}$, if f^* is a solution of a network optimization problem, with convex non decreasing costs $\psi_e(f_e)$, and*

$$w_e^* = \psi_e'(f_e^*) - \tau_e(f_e^*) \quad e \in E,$$

then

$$f^* = f^{(w^*)}$$

More explicitly, Corollary 4.3 on [1] says that by choosing $w_e^* = f_e^* \tau_e'(f_e^*)$ for $e \in E$. The Wardrop equilibrium flow $f^{(w^*)}$ coincides with the system optimum flow f^*.

3.7. New Cost Function

For the last exercise a new delay function was considered, being the total additional delay compared to the total delay in free flow as it is shown in (7).

$$\tau_e(f_e) = d_e(f_e) - l_e \tag{7}$$

The exercise required to compute the new *Social Optimum*, f^*, considering the new delay function and satisfying the constraints mentioned on (3). The new cost function to minimize is shown in (8).

$$\begin{aligned} &\sum_{e \in E} f_e(d_e(f_e) - l_e) = \\ &= \sum_{e \in E} \left(\frac{l_e C_e}{1 - f_e/C_e} - l_e C_e - l_e f_e \right) \end{aligned} \tag{8}$$

For the next part of the exercise, it was required to compute w_e^* such that the *Wardrop Equilibrium flow*, $f^{(w_e^*)}$ concides with the *Social Optimum*, f^*. It was done taking into consideration the previously mentioned Corollary 4.3, obtaining w_e^* as shown in (9).

$$w_e^* = \frac{f_e^* l_e C_e}{(C_e - f_e^*)^2} \tag{9}$$

By solving the *Wardrop Equilibrium* problem, minimizing the expression shown in (10) it was possible to verify that indeed $f^{(w^*)}$ was a very good approximation for f^*. Obtaining a mean relative error of 5.006%

$$\begin{aligned} &\sum_{e \in E} \int_0^{f_e} (d_e(s) - l_e) ds = \\ &= \sum_{e \in E} -l_e C_e \ln(C_e - f_e) + (w_e^* - l_e) f_e^* \end{aligned} \tag{10}$$

References

[1] Fabio Fagnani Giacomo Como. Lecture notes on network dynamics. page 1–187, December, 2020.

Network Dynamics and Learning (01TXLSM)
Homework 2

Professor: Fabio Fagnani
Student: Juan Aragón (S291466)
S291466@studenti.polito.it

Abstract

In the following report it is presented a complete solution for the second assignment of the course Network Dynamics and Learning, a.y. 2021/2022. The main goal was to put in practice the concepts already learned in the class, focusing on the continuous time Markov chains, and the French-DeGroot dynamics model.

All simulations and experiments carried on, whose procedures are described bellow, were written in python, using the professor's lecture notes ([1]) as the theoretical background for all cited concepts.

1. Exercise 1

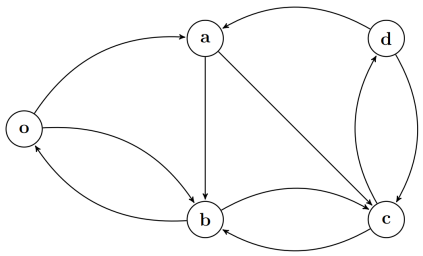

Figure 1: $G_\Lambda = (V, E, \Lambda)$

In this exercise it was required to simulate the behavior of a single particle that moves around the nodes of a given network, $G_\Lambda = (V, E, \Lambda)$, in continuous time according to the transition rate matrix:

$$\Lambda = \begin{array}{c} \begin{matrix} o & a & b & c & d \end{matrix} \\ \begin{pmatrix} 0 & 2/5 & 1/5 & 0 & 0 \\ 0 & 0 & 3/4 & 1/4 & 0 \\ 1/2 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 1/3 & 0 & 2/3 \\ 0 & 1/3 & 0 & 1/3 & 0 \end{pmatrix} \end{array} \begin{matrix} o \\ a \\ b \\ c \\ d \end{matrix}$$

For this exercise it was necessary to define a Poisson clock with rate r. The clock allows us to define a Poisson process N_t, that specifies the number of ticks of the clock occurred by time t. In that way, it was possible to consider an independent discrete-time Markov chain $U(k)$, also called the *jump chain*, to describe the sequence of moves of its associated continuous-time Markov chain $X(t)$.

$$X(t) = U(N_t) \tag{1}$$

The *jump chain*, $U(k)$, was built considering the transition probability matrix $\overline{P}_{ij}$ as it is shown in Eq.2 and Eq.3:

$$\omega_i = \sum_{j \neq i} \Lambda_{ij}, \quad \omega^* = \max_{i \in \chi} \{\omega_i\} \tag{2}$$

$$\overline{P}_{ij} = \frac{\Lambda_{i,j}}{\omega^*}, \quad i \neq j \qquad \overline{P}_{ii} = 1 - \sum_{i \neq j} \overline{P}_{ij} \tag{3}$$

1.1. Return Time of a single particle

What is, according to the simulations, the average time it takes a particle that starts in node a to leave the node and then return to it?

For this purpose it was implemented a function *returnTime*$(position, n\_steps)$, whose parameters represent the node from which the particle leaves and returns to, and the maximum number of steps allowed to complete this task. Inside this function a Poisson clock was implemented with rate ω^*, obtained according to (2).

For each step inside the function, the output of the clock is added into a global variable containing the cumulative time since the beginning of the execution. After that, according to the probabilities specified on $\overline{P}$ the particle will jump into a new node or remain in its current state, if it is allowed. When the particle decides to leave the initial node, a flag variable, *isOut*, is activated so it is possible to

identify the moment at which it turns back. At the end of the execution the cumulative time is returned.

This procedure was simulated 100,000 times, studying the return time related to node a. By taking the average of the obtained output it was possible to determine that $\mathrm{E}(T_a^+) \approx 6.769$.

1.2. Theoretical $\mathrm{E}(T_a^+)$

How does the result in 1.1 compare to the theoretical return-time $\mathrm{E}(T_a^+)$?

From the *Theorem 7.2 (iv)* on [1], it is shown that, for this case, the expected return time for a given node *i* can be computed as:

$$\mathrm{E}(T_i^+) = \frac{1}{\omega_i \overline{\pi}_i}, \quad i \in \chi \tag{4}$$

Considering the information related to node a, it was obtained the theoretical version of the expected return time, $\mathrm{E}(T_a^+) = 6.750$. Finally, by using hypothesis testing techniques, it was possible to validate the previous experiment with a level of significance $\alpha = 5\%$.

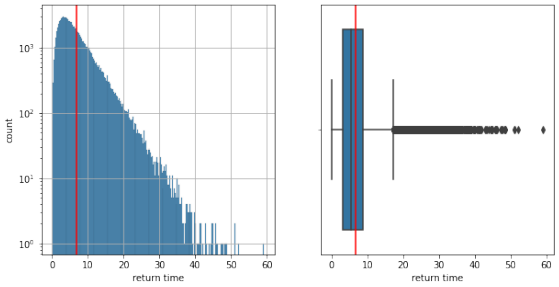

Figure 2: Distribution of T_a^+ after 100,000 simulations. In red, $\mathrm{E}(T_a^+) = 6.750$.

1.3. Expected Hitting Time

What is, according to the simulations, the average time it takes to move from node o to node d?

This implementation followed a similar approach of the one introduced at 1.1. In this case, the declared function was *hitTime(src, dest, num_steps)*, whose parameters determine the source node, the destination node and the number of steps a generic particle is able to move before reaching *dest*.

Once again, a Poisson clock was implemented with rate ω^*, whose value is stored in a cumulative time variable at every step, and at the end of the execution the total time is returned as output of the function.

This procedure was simulated 100,000 times using as parameters the required nodes for this exercise. At the end, it was computed the average of all the returned times, obtaining $\mathrm{E}_o(T_d) \approx 8.781$.

1.4. Theoretical $\mathrm{E}_o(T_d)$

How does the result in 1.3 compare to the theoretical hitting-time $\mathrm{E}_o(T_d)$?

From the *Theorem 7.2 (v)* on [1], it is shown that the expected hitting time $\mathrm{E}_o(T_d)$ is the solution of the system of equations obtained from:

$$\overline{\tau}_d = 0; \qquad \overline{\tau}_i = \frac{1}{\omega_i} + \sum_{j \in \chi} P_{ij} \overline{\tau}_j, \quad i \neq d \tag{5}$$

For this purpose, it was necessary to compute, by first, the probability distribution matrix, $P = D^{-1}\Lambda$, whose values were then used to elaborate the system of equations, cited on (6), and whose unique solution establishes that $\overline{\tau}_o = 123/14 \approx 8.786$.

$$\begin{aligned} \overline{\tau}_o &= \frac{5}{3} + \frac{2}{3}\overline{\tau}_a + \frac{1}{3}\overline{\tau}_b \\ \overline{\tau}_a &= 1 + \frac{3}{4}\overline{\tau}_b + \frac{1}{4}\overline{\tau}_c \\ \overline{\tau}_b &= 1 + \frac{1}{2}\overline{\tau}_c + \frac{1}{2}\overline{\tau}_o \\ \overline{\tau}_c &= 1 + \frac{1}{3}\overline{\tau}_b \end{aligned} \tag{6}$$

Once more, by assuming the theoretical result as our hypothesis, it was possible to verify the level of accuracy of the experimental result, obtaining a satisfactory acceptance with a level of significance $\alpha = 5\%$. A graphical representation of this experiment is shown in Fig.3

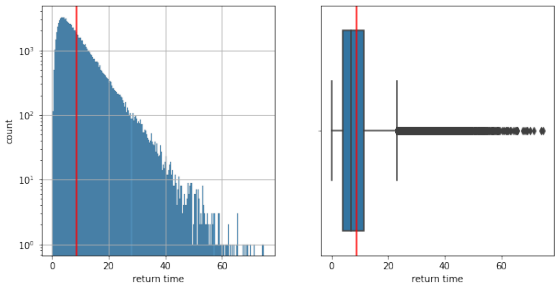

Figure 3: Distribution of $\mathrm{E}_o(T_d)$ after 100,000 simulations. In red $\overline{\tau}_o = 8.786$.

1.5. French-DeGroot

Interpret the matrix Λ as the weight matrix of a graph $G = (V, E, \Lambda)$, and simulate the French-DeGroot dynamics on G with an arbitrary initial condition $x(0)$. Does the dynamics converge to a consensus state?

According to [1], the French-DeGroot dynamics model is defined with the following expression:

$$x(t+1) = Px(t), \quad t = 0, 1, 2, ... \tag{7}$$

Where P is the Probability Distribution Matrix we cited before, and x is a vector of states, each of which evolves in time as a result of the influence of its adjacent states in a way determined by P.

From Fig.1 it can be observed that the considered graph is strongly connected, since for every two node i and j we have that i is reachable from j. Similarly, we can observe that the graph is also aperiodic, since there exists at least a cycle of length 2 and a cycle of length 3. These two conditions are important in terms of the French-DeGroot model, since according to [1], are enough to guarantee that for $t \to \infty$, the vector of states x is able to reach a consensus.

For this purpose, using the same P matrix mentioned in section 1.4, it was computed P^{1000}. This was done by implementing a for-loop with 1000 iterations, computing $P^{t+1} = P^t P$ at each step. At the end, it was observed a matrix in the form $P^t = 1\pi'$, that according to the *Proposition 5.3*, the vector π is also called "invariant probability distribution" of the initial graph.

Finally, the existence of a consensus was proved by computing $x(t) = P^t x(0)$, where $x(0)$ was randomly generated. The obtained result, $x(t)$, was in line with Corollary 5.1, meaning that the expressions in (8) were satisfied.

$$\lim_{t \to \infty} x(t) = \alpha 1, \quad \alpha = \pi' x(0) \tag{8}$$

1.6. Consensus 1

Assume that the initial state of the dynamics for each node i V is given by $x_i(0) = \xi i$, where $\{\xi i\}\, i \in V$ are i.i.d random variables with variance σ^2. Compute the variance of the consensus value, and compare your results with numerical simulations.

For this exercise it was implemented an initial vector of states $x(0) = \mu + \xi$ where ξ was normally distributed with $\mathrm{E}(\xi) = 0$ and $\sigma^2(\xi) = 1$. In the consensus, as it was seen in Eq.8, we expect all the values in $x(t)$ to be identical, so the effect of ξ vanishes, i.e. the $\sigma^2(x(t)) = 0$.

In the simulations, mainly due to computational errors, the obtained output was just very near the expected output, $\sigma^2(x(t)) \approx 10^{-27}$, however, due to the magnitude of the values, the result is considered in line with the expectations.

1.7. Consensus 2

Remove the edges (d, a) and (d, c). Describe and motivate the asymptotic behaviour of the dynamics.

From Fig.1 we can observe that by removing the required links, the resulting graph is no more strongly connected, since we cannot reach any node from node d. However, we can still find consensus, since the number of sinks in the current graph, S_G, is equal to 1, and since the only available sink is aperiodic.

Similar to the previous point, at the beginning it was computed the matrix P^{1000} by iterating over a for-loop, multiplying the modified version of P with itself. It was still evident that the output was in the form $P^t = 1\pi'$. The obtained vector $\pi \approx [0, 0, 0, 0, 1]$.

Also in this case, the existence of a consensus was proved by computing $x(t) = P^t x(0)$, where $x(0)$ was randomly generated. As it was expected from the obtained π, in the consensus all the states are equal to $x_4(0)$. Meaning that also for this scenario the variance of the consensus goes to zero.

The fact that all values in the consensus are equal to $x_4(0)$ makes sense if we associate a given ij-edge to the meaning "node i is influenced by node j". In that way, given the current graph, the node d influences the rest of the graph while no one influences the initial state of node d.

1.8. Consensus 3

Consider the graph, $G = (V, E, \Lambda)$, and remove the edges (c, b) and (d, a). Analyse the French-DeGroot dynamics on the new graph.

For this exercise, similar to the previous point, by removing the required edges it was obtained a graph with a single sink, $S_G = 1$, however, this time the obtained sink was not aperiodic.

As in the previous point, it was computed also the matrix P^{1000}, but as it was expected, in this case it was not possible to identify the vector π, since the rows were not all identical. Moreover, an interesting behavior was noticed, the matrix P^{1001} differed considerably with respect to

P^{1000}. It was determined that the behavior was due to the fact that the nodes c and d swap values at each iteration, and that is why P^t swaps the last two columns on every further iteration.

For the same reason, it was not possible to find a consensus from a randomly generated $x(0)$ since its values keep changing even for t growing large.

2. Exercise 2

For the second part of the assignment, it was required to work once more with the graph shown in Fig 1, using the same transition rate matrix as in the previous part. However, the goal for this case was to simulate the behavior of multiple particles that move around the network in a continuous time.

2.1. Particles Perspective

If 100 particles all start in node a, what is the average time for a particle to return to node a?

This exercise was similar to the first exercise of the previous part. The only difference is that now it was required to consider 100 particles all together. For this purpose it was decided to compute a function, *returnTime_particles()*, with a global Poisson clock having rate equal to the amount of particles on the network, and two auxiliary vectors of length 100, *isOut* and *Finished*. The former vector determines whether a particle has left node a, while the latter indicates whether a particle has returned to a after leaving the node previously.

On every iteration of the implemented function a new tick of the clock was generated and stored on a global variable containing the time required for all particles to arrive into the "Finished" status. On every tick, a particle was selected randomly from a uniform distribution and it was moved according to previously computed $\overline{P}$. Finally, the returned output was the global time variable divided into the total amount of particles.

At the end of this experiment it was obtained an experimental $\mathrm{E}(T_a^+) \approx 6.86$, representing a relative error $\epsilon \approx 1.64\%$ with the theoretical value shown in section 1.2.

This result is similar to the output obtained from the experiment carried on section 1.1. In this scenario, from the particles perspective, nothing changed at all, since the movement of each particle was considered independent from the rest of particles. Regarding the implementation, one of the main changes was the rate of the clock, however, from the way each particle is selected, it can be still considered that each particle was moving around independently at a rate = $w^* = 1$.

2.2. Node perspective

If 100 particles start in node o, and the system is simulated for 60 time units, what is the average number of particles in the different nodes at the end of the simulation?

For this exercise, similar to the previous point, it was implemented a global Poisson clock with rate equal to the total amount of particles in the network. This time, all particles started from node o and the experiment consisted of letting the particles to move freely over the network according to the transition probability matrix $\overline{P}$.

On this implementation, it was no more necessary to implement the auxiliary vectors described before, since the algorithm stops when the cumulative time variable, storing the sum of all the outputs of the clock, surpasses the 60 time units.

The output of the function was the relative distribution of the particles over the nodes of the network, as it is shown in Fig 4. The experiment was carried on with 100 and 10,000 particles, and it is clear that the more the particles on the experiment the more the output becomes similar to the distribution of the "invariant probability vector for the jump chain", $\overline{\pi}$, which is also associated to the probability that a given particle will be found on a specific node over the time, a meaning that has coherence with these types of experiments involving large numbers.

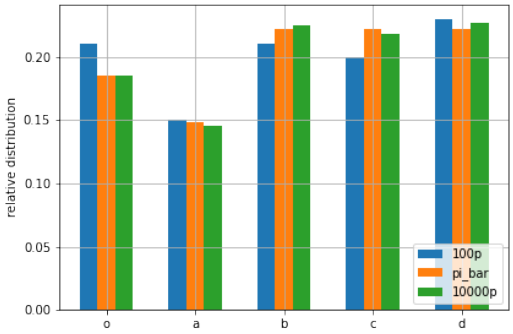

Figure 4: Distribution of 100 and 10000 particles after 60 time units

3. Exercise 3

For the last exercise of this assignment, it was required to consider the open network described in Fig 5, with tran-

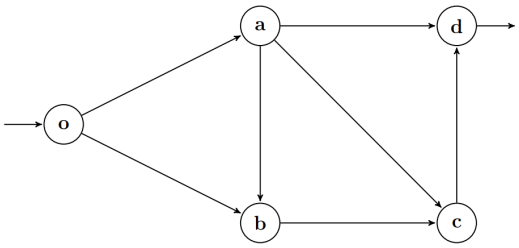

Figure 5: Open network

sition rate matrix Λ_{open} as follows:

$$\Lambda_{open} = \begin{array}{c} \begin{array}{ccccc} o & a & b & c & d \end{array} \\ \begin{pmatrix} 0 & 2/3 & 1/3 & 0 & 0 \\ 0 & 0 & 1/4 & 1/4 & 2/4 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{c} o \\ a \\ b \\ c \\ d \end{array}$$

The goal of this part was to simulate an MM1 queue with rate-λ Poisson arrivals and rate-μ Poisson departures. As it is shown in Fig 5, particles enter into the network at node o. When they are inside they can move freely according to Λ_{open} and the corresponding Poisson clock, before exiting from node d.

Before explaining the two modalities of the exercise, some common modifications were introduced into the given network. Specifically, two auxiliary nodes, o' and d', were introduced before node o and after node d respectively. The purpose of this modification was to serve as the source and the collector of the particles that are not inside the network. For this reason, on this exercise it was considered Λ'_{open} instead.

$$\Lambda'_{open} = \begin{array}{c} \begin{array}{ccccccc} o' & o & a & b & c & d & d' \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{2}{3} & \frac{1}{3} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{2}{4} & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \begin{array}{c} o' \\ o \\ a \\ b \\ c \\ d \\ d' \end{array}$$

3.1. Proportional Rate

The rate of the Poisson clock of each node is equal to the number of particles in it.

For this case, it was implemented a function *nodes_proportional(duration, input_rate)*, whose parameters specify the duration, in time units, of the simulation, and the rate at which particles enter into the network, passing from o' to o.

It was decided to implement a single Poisson clock, to determine whether a new particle was introduced into the network or an already existing particle was allowed to be moved. For this reason the rate of the introduced clock was the sum of the input_rate and the amount of particles inside the network (not in o' neither d'). Then, after storing the output of the clock into a cumulative variable, it was extracted a random number from a continuous uniform probability in the range [0, input_rate + num_particles]. If the extracted number was in the range [0, input_range] a particle from node o' was introduced into the system, otherwise a second random number determined the node from which the particle should be moved, considering the most populated nodes as the most likely to be selected. Finally, the destination node was selected taking into consideration the matrix $\overline{P}$, in order for the transaction to take place. This procedure was repeated as long as the cumulative time variable was bellow the duration of the simulation.

The final output of the function consisted of three parts. The first one was the $\overline{\mu}$-rate, consisting of the ratio between the amount of particles in d' and the duration of the simulation, a wise approximation of the output rate. The second value returned was the list of all intermediate states of the network, summarizing the distribution of the particles along the simulation. And finally, the last value contained the list of all intermediate states of the cumulative clock.

After computing several simulations at different input-rates and 600 time units, some of which are shown in Fig. 6 and Fig.7, it was observed that for any input rate, λ, the system was able to deliver the particles at a similar rate. In other words, the system was capable of modify the output-rate according to the number of particles present on the network.

According to [1], for such kind of behaviors the *Erlang's Formula*, described on (9), is able to demonstrate that systems with large amounts of particles inside are less likely to occur, since for $n \to \infty$, $\overline{\pi}_n \to 0$, even if $\lambda > \mu$.

$$\overline{\pi}_n = \frac{\rho^n/n!}{\sum_{j=0}^{n} \rho^i/j!}; \qquad \rho = \frac{\lambda}{\mu} \tag{9}$$

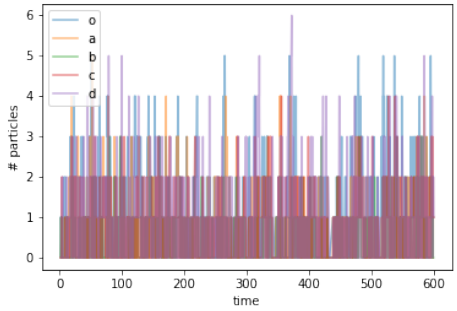

Figure 6: proportional rate, $\lambda = 0.9$, $\overline{\mu} = 0.895$

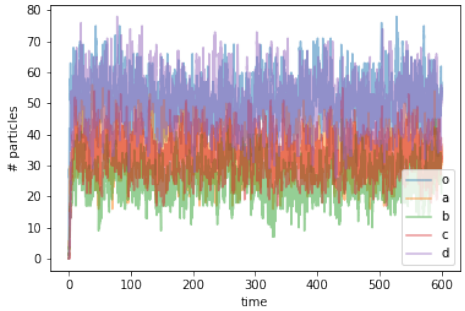

Figure 7: proportional rate, $\lambda = 50$, $\overline{\mu} = 49.88$

3.2. Fixed Rate

The rate of the Poisson clock of each node is fixed, and equal to one.

This exercise was very similar to the previous one, the main difference was that the clock for selecting the nodes was no more proportional to the amount of particles inside the network, but it was constant during the whole simulation, with rate equal to the sum of the input_rate and the number of internal nodes, since it was required to select each node at a rate equal to 1.

After computing several simulations of the system at different input rates, it was observed that for input rates close and greater than 1, the system was not able to handle the particles in a successful manner. From the observations, for input rates greater than 0.95 the systems starts behaving poorly. In other words, the system reached its maximum capacity for delivering the inner particles at the same rate as they were entering, meaning that the nodes tend to accumulate more and more particles over time. Some of this experiments are shown in Fig. 8 and Fig. 9.

According to [1], for systems that have constant input and output rates, the probability of finding large amounts of particles inside the network is given by the expression (10). So that for $\rho < 1$, π_n is exponentially decreasing to 0 as the order of n grows large. In contrast, for $\lambda > \mu$ so that $\rho > 1$, such probability π_n approaches 1.

The best explanation of why the current system blows up for input rates greater than 1 regards the individual activation rate of the given nodes, specifically the rate at which node d sends a particle outside the network. For this case all nodes are activated with rate 1, node d included, it means that the best performance of the system occurs when there is always a particle in node d to be sent out when the d-clock is on. It also means that the system cannot outperform that rate, but it is actually expected to obtain only lower performances by taking into consideration the cases for which the d-clock is on but there are not available particles in d to be sent outside.

$$\overline{\pi}_n = \frac{\rho^i}{1 + \rho + ... + \rho^n}; \qquad \rho = \frac{\lambda}{\mu} \tag{10}$$

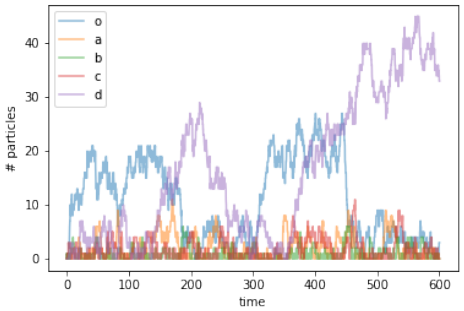

Figure 8: fixed rate, $\lambda = 0.9$, $\overline{\mu} = 0.928$

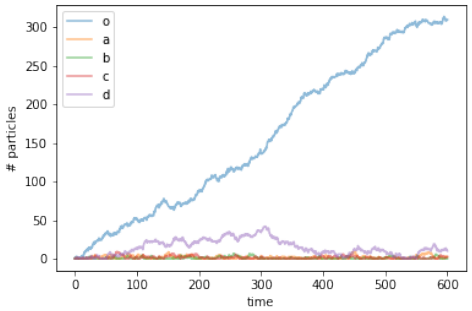

Figure 9: fixed rate, $\lambda = 1.5$, $\overline{\mu} = 0.951$

References

[1] Fabio Fagnani Giacomo Como. Lecture notes on network dynamics. page 1–187, December, 2020.

Network Dynamics and Learning (01TXLSM)
Homework 3

Professor: Fabio Fagnani
Student: Juan Aragón (S291466)
S291466@studenti.polito.it

Abstract

In the following report it is presented a complete solution for the third assignment of the course Network Dynamics and Learning, a.y. 2021/2022. The main goal was to put in practice the concepts already learned in the class, focusing on the Preferential Attachment algorithm for building random graphs, and SIR models for simulating the spreading of an epidemic over a given network.

All simulations and experiments carried on, whose procedures are described bellow, were written in python, using the professor's lecture notes ([1]) as the theoretical background for all cited concepts.

1. Exercise 1.1

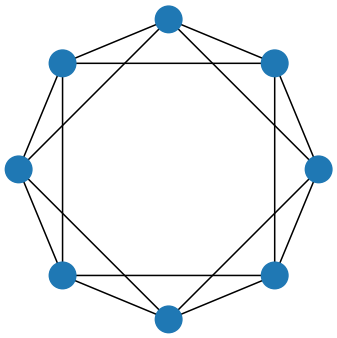

Figure 1: $G_1 = (V, E)$

In this exercise it was required to simulate an epidemic on a symmetric k-regular undirected graph, $G_1 = (V, E)$, as the one shown in Fig.1, where each of the nodes is directly connected to its $k = 4$ nearest neighbors. The real experiment was carried on considering $|V| = 500$ nodes.

For the disease propagation model, it was implemented a discrete-time version of the SIR epidemic model. It consisted of a function called *Epidemic1()*, receiving as parameters the adjacency matrix of the network described before, the number of nodes, the amount of initially infected nodes, the number of weeks at which the disease propagation model was going to be analyzed, the number of simulations and the probability values of β and ρ, related respectively to the probability of a susceptible node for acquiring the disease from an infected node, and the probability of an infected node for spontaneously recovering from the disease.

At the beginning of each simulation the algorithm randomly selects which nodes to be initially infected, while the remaining nodes are initialized as susceptible. Then, at each weekly iteration and for every node on the network, the algorithm evaluates its recovery process in case the current node is infected, by simply choosing a change of its state with probability ρ (Eq.2). However, in case the node is susceptible, then it evaluates the infection process, by analyzing the amount of infected nodes on its neighborhood, m, it decides that the node can become infected with probability $1 - (1 - \beta)^m$ (Eq.1). It can also be said that the probability of a node for not being infected by any of its m sick neighbors is given by $(1 - \beta)^m$.

$$\mathbb{P}\left(X_i(t+1) = I | X_i(t) = S, \sum_{j \in V} W_{ij} \delta^I_{X_j(t)} = m\right) = 1 - (1 - \beta)^m \tag{1}$$

$$\mathbb{P}\left(X_i(t+1) = R | X_i(t) = I\right) = \rho \tag{2}$$

Once the new states of all the nodes are computed for a given week, a summary of the network is generated, containing information regarding the amount of nodes belonging to each category on the SIR model. It is also registered the amount of nodes that became infected during each week.

After running the algorithm over the previously described network, by using $\beta = 0.3$, $\rho = 0.7$, a total of 100 simulations over 15 weeks and 10 initially infected nodes, it was obtained the results shown in Fig.2. It can be seen that the number of infections remained limited, and only a small fraction of the population contracted the disease. The reason for this behavior is mainly due to the fact that people was capable of getting cured easily (high ρ), the disease was difficult to transmit (small β), and because there was a reduced number of interactions between the nodes (small k).

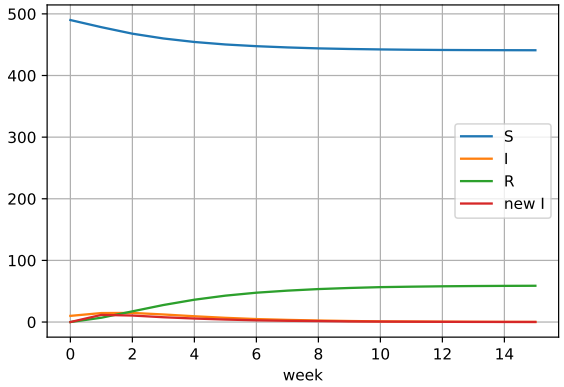

Figure 2: Average evolution of the nodes' states, for Small-world models

2. Exercise 1.2

In the previous exercise we dealt with graphs built using the so called *"Small-world"* models, that combines links due to the geographical proximity, where each node interacts only with its geographically closest neighbors. This time, we focused on the generation of random graphs following a *Preferential Attachment* procedure, where each node that is added into the network is linked with a higher probability to the already existing nodes that have the highest degree.

The goal of this exercise was to generate a random graph, with average degree k, according to the *Preferential Attachment model*. For this scope a new function, *PreferentialAttachment()*, was implemented. Initially, a complete graph with k+1 nodes was built, using the *networkx* python library. Then, one node at a time was added into the network, until the desired number of nodes was reached. For each of the nodes added into the network, a total of $c = k/2$ undirected links were used to connect the given node with the previously available network. The nodes to which every individual was linked at this stage was done considering the probability shown in (Eq. 3). In case of k being odd, the function alternates the number of links added to a given node from $\lfloor k/2 \rfloor$ and $\lceil k/2 \rceil$.

$$\begin{aligned} &\mathbb{P}\left(W_{nt,i}(t) = W_{i,nt}(t) = 1 | G_{t-1} = (V_{t-1}, E_{t-1})\right) \\ &\quad = \frac{w_i(t-1)}{\sum_{j \in V_{t-1}} w_j(t-1)}, \quad i \in V_{t-1} \end{aligned} \tag{3}$$

After running the function over a network with 1000 nodes with $k = 6$, it was obtained the network represented on Fig.3, where all the nodes are drawn in a circumferential alignment. The links between the nodes put in evidence the "preferential" choice of the majority of the nodes for being connected to a certain portion of them.

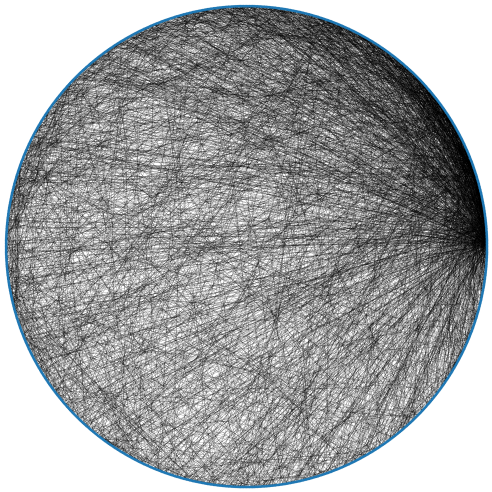

Figure 3: Preferential Attachment graph

3. Exercise 2

This part of the homework had a similar goal as Exercise 1.1. It consisted of simulating the spreading of an epidemic, following the SIR model, over a network built with the *Preferential Attachment* approach.

For this purpose, the network under analysis was obtained using the *PreferentialAttachment()* function described before, with parameters $k = 6$ and $|V| = 500$ nodes. To this graph, it was applied the *Epidemic1()* function we saw on Exercise 1.1, with parameters $\beta = 0.3$ and $\rho = 0.7$. After simulating the spreading activity for $N = 100$ times, studying the epidemic for 15 weeks all starting with an initial number of 10 infected nodes, it was obtained the results shown on Fig.4.

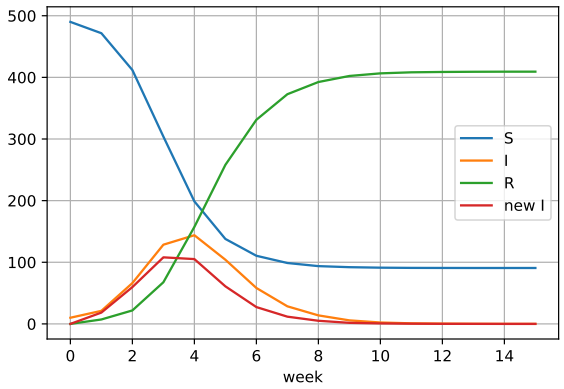

Figure 4: Average evolution of the nodes' states, for Preferential-attachment models

By comparing the results shown in Fig.4 with the ones obtained in Exercise 1.1. It can be noticed a huge difference in the number of *Recovered* nodes and the amount of nodes that have never been infected with the disease. The best and most obvious explanation for this behavior is that on the *Preferential Attachment* model, the most popular nodes are those agents who get infected more easily and also the ones who spread the disease the most. At the same time, once those nodes become *Recovered* then the spreading starts to slow down until no more nodes get infected.

4. Exercise 3

This exercise was also similar to the previous ones, analysing the spreading of a disease over a given network built using the *Preferential Attachment* model. However, here an additional action was introduced to slow down the diffusion, the vaccination. It was assumed that during each simulation, a portion of the population was vaccinated against the disease, following the distribution shown in (Eq.4), which establishes which percentage of the population should be already vaccinated per week. The idea is that once a person is vaccinated it cannot be infected and cannot infect other subjects from that week in above.

$$Vacc(t) = \\ [0, 5, 15, 25, 35, 45, 55, 60, 60, 60, 60, 60, 60, 60, 60] \tag{4}$$

For this purpose, based on the already existing function, *Epidemic1()*, it was generated a second function, *Epidemic2()* that introduces the simulation of the vaccination process. At the beginning of each week on a given simulation, it is calculated the amount of nodes to vaccinate in order for (Eq. 4) to be satisfied. This process is done by calculating the difference between the percentage regarding the current week and the week before, then converting it into absolute values considering the number of nodes on the network, and finally randomly choosing that amount of nodes from the not-yet-vaccinated nodes.

For the epidemic simulation, the only modification done at this exercise is the implementation of the constraints regarding vaccinated nodes, not being considered at the infection process, so *m* considers only the not-yet-vaccinated nodes linked to a given individual.

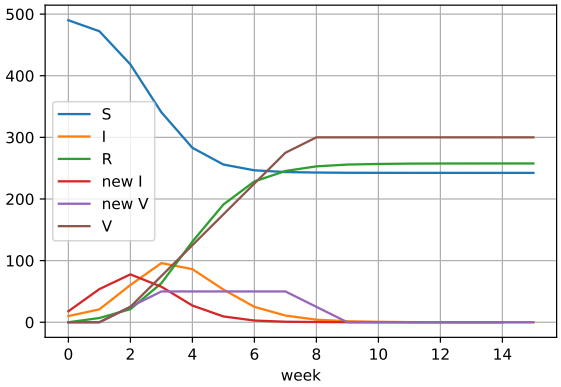

Figure 5: Average evolution of the nodes' states, for Preferential-attachment models with Vaccination

The experiment was carried on using the same parameters as in the previous exercise, run over a network of $|V| = 500$ nodes. From the obtained results, shown in Fig.5, it can be seen that indeed the average evolution of the infected nodes is being reduced by the presence of the vaccination process. In comparison with the results shown in Exercise 2, there is a noticeable reduction in the number of recovered nodes, since fewer nodes got infected during the simulations. It can also be noticed that the highest amount of newly vaccinated nodes is 50, as it was expected from (Eq.4), since the maximum difference between two consecutive weeks is 10%. Finally, at the end of all the experiments it was obtained a network with 300 vaccinated nodes, representing the 60% of the total individuals.

5. Exercise 4

For the last exercise of this homework, it was required to analyse a real-life case, the H1N1 pandemic in Sweden of 2009, considering only a representative portion of the original population, $|V| = 934$. The goal of the experiment was to try different configurations for the parameters k, β and ρ in order to obtain an infection curve as close as possible to the scaled version of the real-life case, whose values are detailed on (Eq.5).

$$I_o(t) = [1, 1, 3, 5, 9, 17, 32, 32, 17, 5, 2, 1, 0, 0, 0, 0] \tag{5}$$

For this purpose, it was implemented the function *GradientBasedSearch()*, receiving as parameters the vaccination distribution per week as in (Eq.6) and the ground truth of the weekly infections, together with the number of nodes n, and the number of weeks per simulation.

$$Vacc(t) = [5, 9, 16, 24, 32, 40, 47, 54, 59, 60, 60, 60, 60, 60, 60, 60]. \tag{6}$$

At the beginning of the algorithm it starts with an initial guess of the parameters, using $k_0 = 10$, $\beta_0 = 0.3$ and $\rho_0 = 0.9$. The idea is to define a parameter space such that $k \in \{k_0 - \Delta k, k_0, k_0 + \Delta k\}$, $\beta \in \{\beta_0 - \Delta\beta, \beta_0, \beta_0 + \Delta\beta\}$, and $\rho \in \{\rho_0 - \Delta\rho, \rho_0, \rho_0 + \Delta\rho\}$, from which to run simulations with all possible of combinations of them. At the end, selecting those values for which the lowest error is obtained and using them to initialize the parameter space of the next iteration. In order to measure the dissimilarity between the output curve and the ground truth, it was calculated the *RMSE* between both vectors, as it is shown in (Eq.7).

$$RMSE = \sqrt{\frac{1}{15}\sum_{t=1}^{15}(I(t) - I_0(t))^2} \tag{7}$$

In case of Δk, $\Delta\beta$, and $\Delta\rho$, it was decided to initialize their values as $\Delta k = \frac{k_0}{2}$, $\Delta\beta = \frac{\beta_0}{2}$, and $\Delta\rho = \frac{\rho_0}{2}$ and to reduce them by half on every new iteration. It was also decided that 4 iterations were reasonable to find the global optimum, since going further with the iterations were about to increase redundancy to the experiment. In this report it is included the evolution of the global optimum, for which it was achieved a final result of RMSE = 5.90.

References

[1] Fabio Fagnani Giacomo Como. Lecture notes on network dynamics. page 1–216, December, 2021.

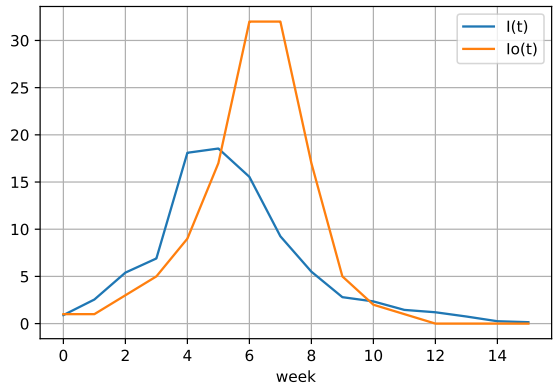

Figure 6: $\beta = 0.30$, $k = 5$, $\rho = 0.60$. RMSE = 8.00

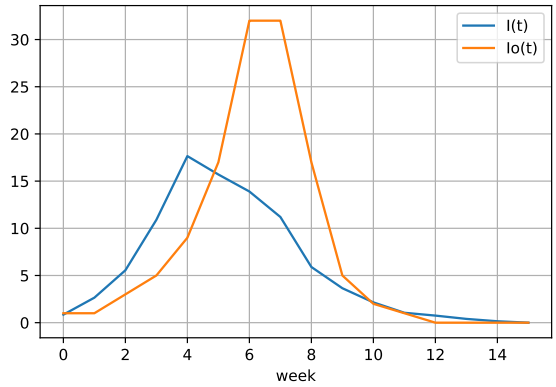

Figure 7: $\beta = 0.22$, $k = 7$, $\rho = 0.60$. RMSE = 7.93

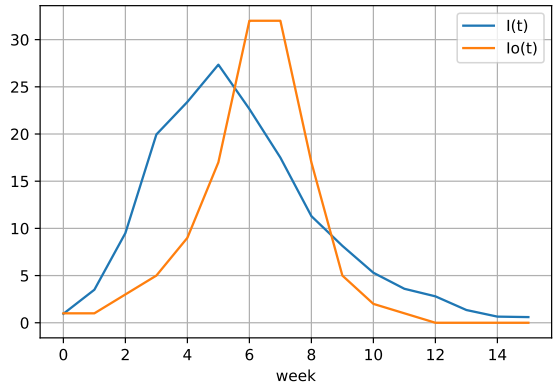

Figure 8: $\beta = 0.20$, $k = 8$, $\rho = 0.52$. RMSE = 7.72

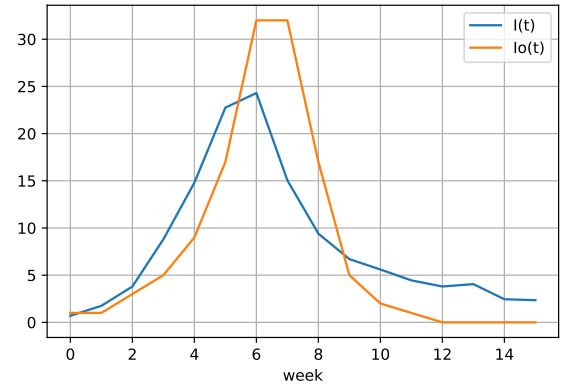

Figure 9: $\beta = 0.20$, $k = 7$, $\rho = 0.50$. RMSE = 5.90

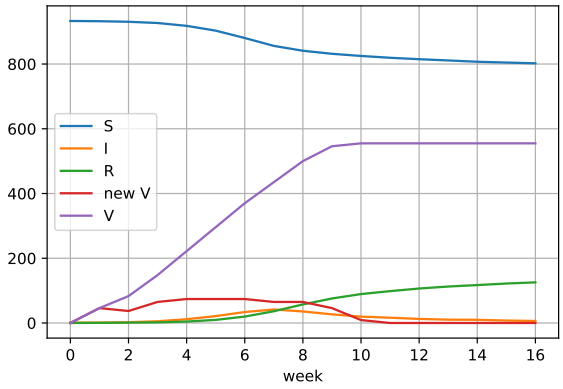

Figure 10: $\beta = 0.20$, $k = 7$, $\rho = 0.50$. RMSE = 5.90