Data Intelligence Application

Social Influence

POLITECNICO
MILANO 1863

Scotti Andrea
Paci Marco
Dello Preite Castro Gianmarco
Valladares Stefano

April 9, 2019

Contents

Abstract

This work analyses how to maximize Social Influence. A social influence problem involves buying nodes of a network in order to maximize the reach inside the network. In this work we have analysed different sizes of networks, different policies and different algorithms, and how all of them affect the result of maximizing social influence in a given network.

1 Problem contextualization

The problem we are trying to tackle is how to maximize social influence in a given network given some constraints. These constraints are mainly regarding a budget, that is, the maximum value of nodes that can be bought inside the network. The objective is to reach the highest number of users with the given budget. This is due the fact that each node that is acquired, and is put in the Seed Set, is capable of influencing with a probability its neighbouring nodes, so all the nodes reached by an outgoing edge. This is done in a cascade fashion, so if node 2 is bought, and node 2 influences node 3 at time t=1, then node 3 may as well do the same to the nodes it can reach at time t=2. This process is over when every active node, which are the nodes influenced, attempted the influence process and terminated. The influence spread, which is usually what we want to maximize, is the number of nodes infleunced by the this process starting from a selected Seed set. Inside the graph, each node has features, and all of these features are used to build the final probability that represents an edge between two nodes. In order to fully estimate the effect that buying a node has, we have to simulate the activation of the edges in an iterative way. To do so, Monte Carlo simulations is used since is a good heuristic for the estimation of the influence spread.

2 Policies

We have introduced many different policies in order to solve one big problem of the Social Maximization problem, the time complexity. Unfortunately with increasingly large graphs, the computational complexity of the problem grows exponentially, and one of the best ways to regulate it is through different policies.

2.1 Greedy Policy

The Greedy policy is the classic policy, studied in the Kempe[1 , Maximizing the Spread of Influence through a Social Network, David Kempe, Jon Kleinberg, Eva Tardos] paper, to solve the influence maximization problem with a good approximation. This approximation is given by the fact that the function f(), that adds to a seed set another free node of the graph, is submodular, which mean that the marginal gain from adding an element to a set S is at least as high as the marginal gain from adding the same element to a superset of S. Formally, a submodular function satisfies $f(S \cup v) - f(S) \geq f(T \cup v) - f(T)$, for all elements v and all pairs of sets $S \subseteq T$

The greedy algorithm adds to the seed set, the node among the ones still free that achieves the greatest increase in the influence spread when added to the seed set. To extimate the influence spread of a set of seed node, we as most of research community use T repetitoin of MonteCarloSimulation on the network graph. Using a good T (at least 2000 to 10000 repetitions, and even more based on the number of edges of the graph) the algorithm has a good approximation of the influence spread, and can choose which node to add in the seed set. The complexity of the alghorithms is still too big to deal with, it is O(nkTb), considering a unitary budget, b is the budget (the max number of nodes that can be put in the seed set), T is the number of MonteCarlo simulation, n is the number of nodes in the graph and k the number of edges.

2.2 CELFpp Policy

To reduce the complexity, in terms of time needed to compute the algorithm, we implemented CELFpp [2 Goyal A, Lu W, Lakshmanan LV. Celf++: optimizing the greedy algorithm for influence maximization in social networks. Proceedings of the 20th international conference companion on World wide web. ACM, 2011.]. This algorithm has the same performance, in terms of nodes selection, as the greedy algorithm, but exploits even more the submodularity of the Influence Maximization problem to reduce the complexity. The complexity of the CELFpp algorithm is not easy to calculate, but it runs considerably faster than the greedy algorithm as the number of nodes to put in the seed set increase. A more complete analysis on how this algorithm works can be found in the paper.

2.3 Single Degree Discount Policy

A policy that performs extremely badly, and relies on just selecting the node with the maximum number of outgoing edges.

2.4 Degree Discount C Policy

This heuristic algorith, is an evolution of the Single Degree Discount Policy, but uses a better heuristic than relying only on the number of outgoing edges. In this algo every time a node is selected, a discount factor is inserted in all the nodes reached by the selected node. This due to the fact that if a node n is being put in the seed set, we have a possibility that the nodes reached by the the outgoing edges of n are influenced by the diffusion process, so the algorithm is less inclined to select to buy them. In most research papers this algorithm is said to perform really good in terms of influence spread (only 15%-20% worst than the greedy/CELFpp) with a negligible time complexity (few milliseconds vs years). But the performance of this algorithm is only good with graphs having some standard topology and probability on the edges. In some other cases the performace is really bad.

2.5 Random Policy

The simplest of the policies, but also probably the one that perfoms the worst is the Random Policy, it simply choses a node at random on the graph and buys it. However if the graph analyzed has a lot of similar node structure, meanig that every node has more or less the same number of outing edges and activating probabilities on these edges, the random algorithm performs more or less as good as the other algorithms. If the graph has a strange topography and some nodes are more influent than others, then this algorithm doesn't perform well.

3 Algorithm Description

There are two cases for the social influence problem, one in which the probabilities of the edges are known, and another in which they aren't and need to be learnt.

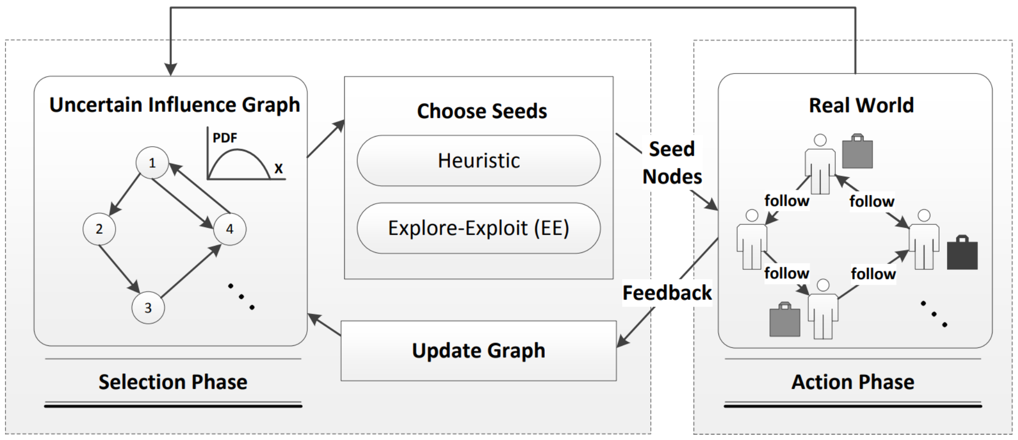
3.1 Influence Maximization Problem

In the known probabilities case, we can rely on the Influence Maximization algorithms. In the last section there will be some graphs with all the different algorithms performance put in comparison. We were not able to deeply analyse the problem because we are not able to run the CELFpp (and even worse with the greedy) when the number of nodes/edges increases too much (with a regular computer it was not possible to go further than 150).

3.2 Online Influence Maximization Problem

This scenario is the most similar to a real world problem. We are given a network, in which we know the edges and the nodes, but we are not given the activation probabilities on those edges. To solve this scenario, we can use an Explore/Exploit approach (bandit algorithms). In this way, the algorithms aim at finding the best seed selection at each iteration for a given time horizon. In the studied research, the scientific community uses a standard budget, we may call it K, that is the total budget to use. They may run the online algorithm K times selecting only a single seed, K/2 times selecting 2 seed each time ..... 1 time selecting K seed at once. We did not follow this approach, we went for a more streightforward way in which we indepedently selected K as a time horizon(number of times we can execute the algorithm), and k as the budget to use each time(number of seeds we can buy).

The framework of the problem is well explained in this picture:



On the left there is the Uncertain Influence Graph, the network used by the algorithm to select the seed to buy. From this graph then the seeds are selected, resolving the Influence Maximization Problem on the graph(see above), and are bought in the real world. From the real world our environment can get the activation cascade information, which means it can see which edges have been activated by the selected nodes. With this information, the algotrithm updates its Uncertain Influence Graph, and starts again until it reaches the end of the time horizon. This is the pseuodocode of the Online Influence Maximization Algorithms, and should be straightforward to understand:

Algorithm 1 `Framework(G, k, N)`

```
1: Input: # trials $N$, budget $k$, uncertain influence graph $G$
2: Output: seed nodes $S_n (n = 1 \dots N)$, activation results $A$
3: $A \leftarrow \emptyset$
4: for $n = 1$ to $N$ do
5:     $S_n \leftarrow \texttt{Choose}(G, k)$
6:     $(A_n, F_n) \leftarrow \texttt{RealWorld}(S_n)$
7:     $A \leftarrow A \cup A_n$
8:     $\texttt{Update}(G, F_n)$
9: return $\{S_n | n = 1 \dots N\}$, $A$
```

The problem is approached in two ways:

- In the first case we try to exploit the linear dependency of the edges with the theta parameters, in a linear regression manner, to learn the theta parameters that are common to all the edges. In this case the feature vector of the edges is known, the algorithm LinearUCB every round learns a better approximation of the theta parameters value and can in this way do a better choice of the seed to select. In this paper [3, Online Influence Maximization under Independent Cascade Model with Semi-Bandit Feedback, Zheng Wen, Branislav Kveton, Michal Valko, Sharan Vaswani] there is a more detailed explaination of how it works.

- In the second case, there is no exploit in the linear dependency of the theta parameters. In this case the Combinatoria UCB and TS algorithms exploit in a standard way the network to select the seed set. The difference between CUCB and CTS is in the way the edges probabilities on the Uncertain Influence Graph are calculated and updated.

4 Code Structure

The code is organize as follows:

- Bandit_algs Foder: Folder containing the bandit algorithms used in the Online Influence Maximization problem.
- Policies Folder: Folder containing all the different policies(algorithms to solve the Influence Maximizaztion problem) used in the project. They were all introduced in order to solve the problem of computational complexity.
- Miscellaneous: The remaining files are for generating graphs (GraphUtils) run the code (MainBandit, MainGreedy and SocialMain) and run Monte Carlo Simulations (MonteCarloSimulation and MonteCarloSimulationV2)

5 Results

5.1 Experiment setup

In every graph the probability is given by the theta parameter and the feature parameter. For every graph we these parameters are reported along with the probability to spawn an edge in the graph. We used T=3000 MC iterations

5.2 Influence Maximization

The first garph is a sparse network, with low activation probability (around 0.1)

$nodes : 150$

$budget : 22$

$edges : np.log(n_nodes)/n_nodes$

$theta : np.random.dirichlet(np.ones(n_features), size = 1)*np.random.uniform(0, 0.5)$

$features : np.random.binomial(1, 0.25, size = n_features).tolist()$

$Montecarloiterations : 3000$

The running time is also reported to better understand the time complexity issue:

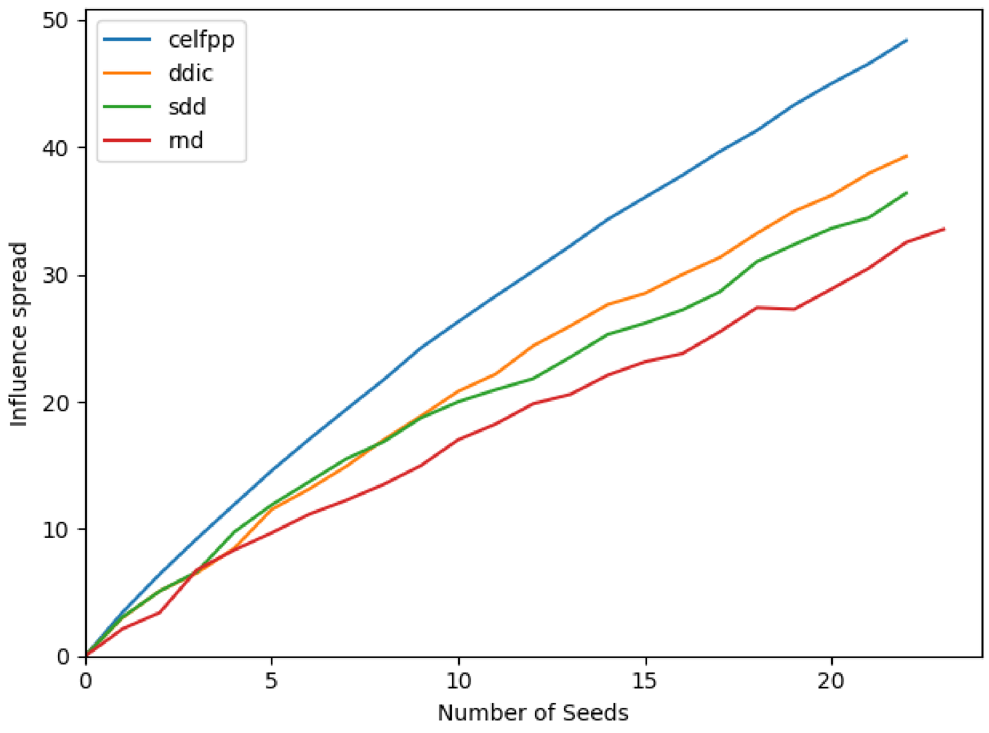
algorithm celfpp running time: 4741.0272352695465

algorithm ddic running time: 228.47204852104187

algorithm sdd running time: 227.22170042991638

algorithm rnd running time: 214.27558827400208

Note that the heuristics and random algorithms (ddic, sdd, rnd) take all that time because we use MC simulation to plot the influnce spread with every new node in the seed set, otherwise they would run in no time at all.



The second graph is still a sparse network, but with a higher activation probabilities on the edges

$nodes$: 150

$budget$: 22

$edges$: $np.log(n_nodes)/n_nodes$

$theta$: $np.random.dirichlet(np.ones(n_features), size = 1)$

$features$: $np.random.binomial(1, 0.25, size = n_features).tolist()$

$Montecarloiterations$: 3000

algorithm celfpp running time: 15547.94735622406

algorithm ddic running time: 406.5956394672394

algorithm sdd running time: 381.44613218307495

algorithm rnd running time: 350.65496921539307

celfpp optimal set of seeds: 35: 1, 10: 1, 132: 1, 80: 1, 81: 1, 6: 1, 68: 1, 12: 1, 102: 1, 43: 1, 19: 1, 103: 1, 51: 1, 92: 1, 50: 1, 83: 1, 139: 1, 2: 1, 97: 1, 24: 1, 123: 1, 36: 1

ddic optimal set of seeds: 147: 1, 149: 1, 112: 1, 32: 1, 107: 1, 111: 1, 61: 1, 67: 1, 76: 1, 117: 1, 118: 1, 66: 1, 81: 1, 87: 1, 135: 1, 50: 1, 68: 1, 4: 1, 83: 1, 113: 1, 124: 1, 9: 1

sdd optimal set of seeds: 147: 1, 149: 1, 108: 1, 112: 1, 32: 1, 107: 1, 22: 1, 82: 1, 109: 1, 111: 1, 0: 1, 56: 1, 67: 1, 117: 1, 118: 1, 76: 1, 77: 1, 90: 1, 5: 1, 47: 1, 61: 1, 87: 1

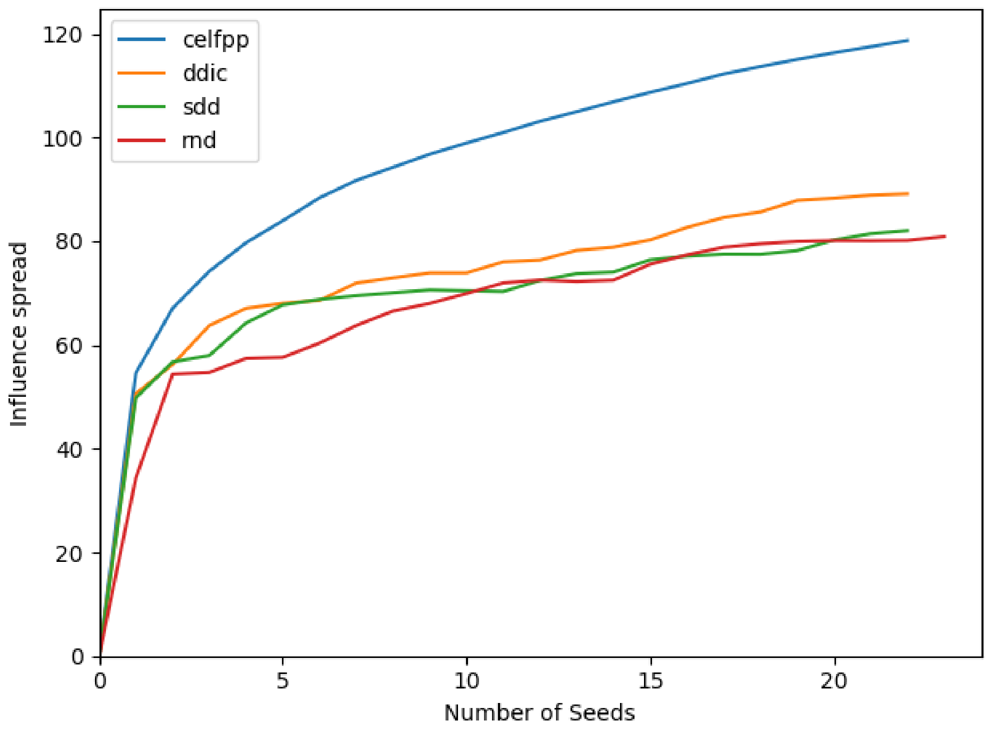
rnd optimal set of seeds: 29: 1, 104: 1, 140: 1, 48: 1, 128: 1, 94: 1, 142: 1, 18: 1, 127: 1, 76: 1, 136: 1, 120: 1, 117: 1, 100: 1, 98: 1, 41: 1, 62: 1, 66: 1, 57: 1, 114: 1, 75: 1, 13: 1

influence spread: [0, 54.575, 67.03966666666668, 74.19966666666669, 79.68966666666667, 83.93566666666666, 88.37266666666667, 91.723, 94.234, 96.75966666666666, 98.91499999999999, 100.95233333333334, 103.11400000000002, 104.96466666666667, 106.878, 108.68366666666668, 110.39366666666666, 112.22799999999998, 113.65899999999999, 115.04233333333332, 116.32166666666664, 117.49199999999998, 118.70399999999997]

influence spread: [0, 50.605, 56.20066666666668, 63.72766666666667, 67.04066666666664, 68.02233333333332, 68.58499999999998, 71.92899999999997, 72.92466666666665, 73.88199999999998, 73.87833333333333, 76.01066666666667, 76.35933333333334, 78.23066666666668, 78.85033333333332, 80.23033333333338, 82.63766666666668, 84.56366666666669, 85.63333333333335, 87.85966666666668, 88.29599999999998, 88.86800000000002, 89.13133333333336]

influence spread: [0, 49.774333333333324, 56.716666666666676, 57.952333333333335, 64.21933333333334, 67.71666666666667, 68.75566666666668, 69.493, 70.03599999999999, 70.59433333333332, 70.43666666666668, 70.289, 72.427, 73.74933333333331, 74.04133333333333, 76.40933333333334, 77.11233333333331, 77.49466666666666, 77.47000000000001, 78.18466666666667, 80.21233333333332, 81.43133333333334, 82.02533333333332]

influence spread: [0, 34.32433333333333, 54.349666666666664, 54.67966666666666, 57.39633333333333, 57.592666666666666, 60.356, 63.74133333333334, 66.515, 67.99866666666667, 69.90033333333334, 71.94500000000001, 72.50699999999999, 72.22233333333334, 72.47766666666666, 75.56466666666667, 77.35666666666667, 78.83466666666666, 79.496, 79.94466666666668, 80.102, 80.06766666666667, 80.13366666666667, 80.88066666666668]



The third graph is still a sparse network, same as experiment 2, but with a few really strong edges

$Nodes : 150$

$Budget : 22$

$edges : np.log(n_nodes)/n_nodes$

$theta : np.random.dirichlet(np.ones(n_features), size = 1)$

$features : foredgeinnetwork.edges.values() :$

$ifnp.random.binomial(0, 0.05) :$

$edge['features'] = np.random.binomial(1, 0.75, size = n_features).tolist()$

$else :$

$edge['features'] = np.random.binomial(1, 0.25, size = n_features).tolist()$

$Montecarloiterations : 3000$

algorithm celfpp running time: 14045.999668121338

algorithm ddic running time: 402.2524745464325

algorithm sdd running time: 386.236643075943

algorithm rnd running time: 371.9336190223694

celfpp optimal set of seeds: 141: 1, 131: 1, 64: 1, 134: 1, 118: 1, 88: 1, 117: 1, 51: 1, 146: 1, 68: 1, 80: 1, 148: 1, 41: 1, 67: 1, 48: 1, 71: 1, 21: 1, 4: 1, 57: 1, 46: 1, 75: 1, 19: 1

ddic optimal set of seeds: 55: 1, 34: 1, 62: 1, 133: 1, 139: 1, 117: 1, 31: 1, 95: 1, 107: 1, 131: 1, 12: 1, 15: 1, 70: 1, 91: 1, 144: 1, 58: 1, 59: 1, 65: 1, 67: 1, 88: 1, 149: 1, 47: 1

sdd optimal set of seeds: 55: 1, 34: 1, 62: 1, 133: 1, 139: 1, 123: 1, 26: 1, 72: 1, 95: 1, 117: 1, 128: 1, 1: 1, 12: 1, 15: 1, 130: 1, 131: 1, 142: 1, 145: 1, 39: 1, 47: 1, 50: 1, 88: 1

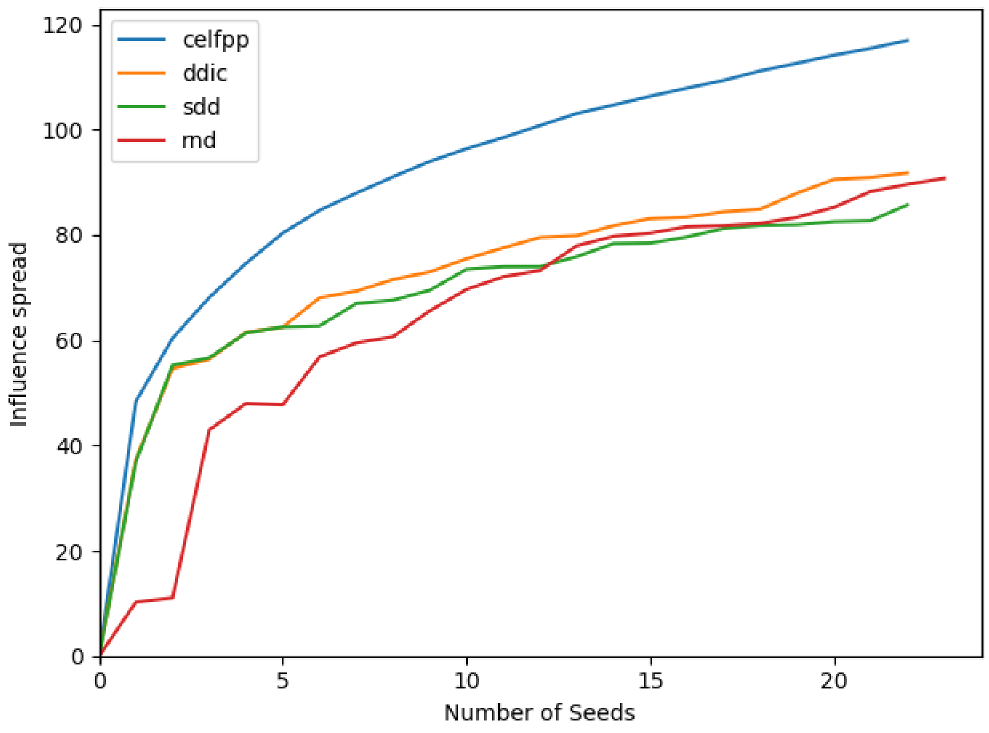
rnd optimal set of seeds: 120: 1, 64: 1, 79: 1, 103: 1, 106: 1, 32: 1, 58: 1, 61: 1, 100: 1, 128: 1, 13: 1, 88: 1, 47: 1, 25: 1, 102: 1, 60: 1, 70: 1, 77: 1, 98: 1, 51: 1, 131: 1, 122: 1

influence spread: [0, 48.35033333333333, 60.37566666666667, 68.07766666666666, 74.58666666666664, 80.32566666666665, 84.65766666666664, 87.9303333333333, 91.01099999999995, 93.91466666666662, 96.33066666666662, 98.45499999999996, 100.75599999999997, 103.0313333333333, 104.64399999999996, 106.31499999999997, 107.85466666666663, 109.33033333333331, 111.13233333333334, 112.58066666666666, 114.10833333333332, 115.41166666666668, 116.91]

influence spread: [0, 37.32933333333335, 54.623000000000005, 56.395666666666656, 61.50766666666667, 62.44700000000002, 68.03300000000004, 69.29799999999997, 71.4906666666667, 72.94333333333329, 75.44666666666663, 77.51733333333334, 79.50966666666667, 79.83133333333338, 81.72133333333336, 83.0926666666667, 83.37266666666669, 84.36699999999996, 84.89599999999999, 87.93166666666667, 90.52899999999997, 90.89533333333334, 91.7283333333334]

influence spread: [0, 36.88266666666669, 55.249999999999986, 56.65533333333334, 61.382666666666694, 62.50800000000002, 62.69400000000002, 66.93699999999998, 67.57433333333331, 69.45866666666673, 73.4473333333333, 73.952, 73.96499999999996, 75.85433333333333, 78.318, 78.4246666666667, 79.59133333333331, 81.20533333333334, 81.78833333333331, 81.9083333333333, 82.51433333333333, 82.68333333333337, 85.68666666666671]

influence spread: [0, 10.225000000000001, 11.058666666666664, 42.949333333333335, 47.97066666666667, 47.69133333333333, 56.821, 59.489333333333335, 60.659666666666666, 65.535, 69.619, 72.01800000000003, 73.218, 77.911, 79.71166666666667, 80.326, 81.529, 81.74533333333332, 82.174, 83.339, 85.22433333333333, 88.23566666666667, 89.58500000000001, 90.733]



5.3 Online Influence Maximization

For this section we do not have many experimental results. The complexity of the algorithms is too high fora common computer to run in an acceptable time (It is the same as the IM algorithms, multiplied by the number of experiments, the time horizon, the number of algorithms to run, and the complexity to run the simulation of the activation cascade). Every run in this scenario is mediated on a number of experiments, due to the volatility of the data in the real world scenario, given by the activation cascade.

In this first example, the parameters are:

$nodes : 10$

$budget : 2$

$timehorizon : 50$

$montecarlo_simulatin : 2000$

$experiment : 20$

$edges : np.log(n_nodes)/n_nodes$

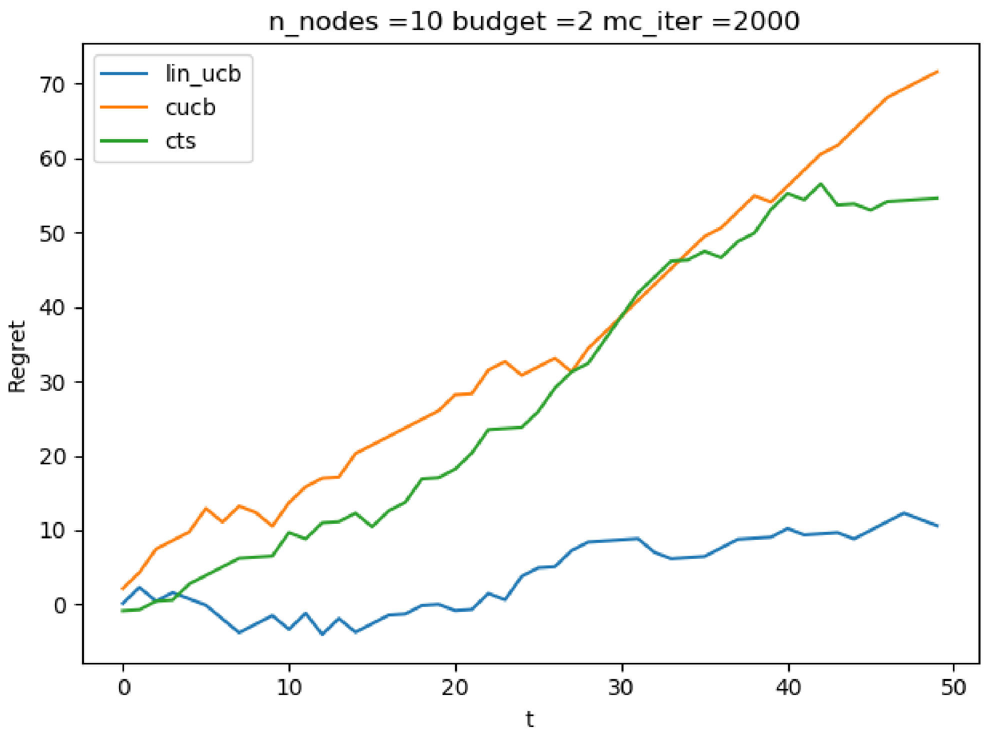
$theta : np.random.dirichlet(np.ones(n_features), size = 1)$

$features : foredgeinnetwork.edges.values() :$

$ifnp.random.binomial(0, 0.05) :$

$edge['features'] = np.random.binomial(1, 0.75, size = n_features).tolist()$
$else:$
$edge['features'] = np.random.binomial(1, 0.25, size = n_features).tolist()$

In the experimental results, it may seems strange that the regret is negative for the LinearUCB algorithm. But it is not, that happened due to the few runs of the experiment that had been done, so when sampling the influence cascade from the real enviroment, it may happen that when selecting a seed of nodes, the cascade process gives an higher results than the optimal one calculated by the Influence Maximization algorithms ran on the Real network. With more experiments and a good mediation, this problem can be eliminated.



In the second example, the paramters are:
$nodes : 20$
$budget : 4$
$timehorizon : 200$
$Montecarloiterations : 500$(MC is 2000 in the calculation of the optimum influence spread value, 500 in the selection of the best seeds in the uncertain graph)
$experiment : 50$
$edges : np.log(n_nodes)/n_nodes$

$theta : np.random.dirichlet(np.ones(n_features), size = 1)$
$features : edge['features'] = np.random.binomial(1, 0.25, size = n_features).tolist()$
Unfortunately we are not able to provide results for this example currently, as the running time of the algorithm is going to be close to 10 days on a normal computer.

6 Conclusions

This research project has the objective to analyze how various algorithms perform in the Influence Maximization and in the Online Influence Maximization problems. We implemented almost all the known algorithms for the Influence Maximization, and analyzed their performance with different kinds of graph. Using the code is easy to run even more experiments, changing the parameters of the algorithms(MC iteration, number of nodes, features) and the network topology (just modify the file GraphUtils.py). Then we implemented what was requested by the project assignment for the Online Influence Maximization problem, the Linear UCB which exploits the linearity of the edges' feature, and the more simple Contextual UCB and TS.

Everything was implemented but poorly tested, due to the too high complexty of the algorithms. If we could have the possibility to use powerful machines to do the simulations, we could achieve way better results and run our algorithms on bigger and way more complex networks. Our poor results, however, gave us enough reasons to think that all the code implemented is correct and works well (that's sure for the Influence Maximization, almost sure for the OIM).