A comparitive study of several influence maximization algorithms

Social Network Analysis for Computer Scientists — Course paper

Christian Steennis
c.u.steennis@umail.leidenuniv.nl
LIACS, Leiden University
Leiden, Netherlands

Daniel van Heuven van Staereling
d.e.e.van.heuven.van.staereling@umail.leidenuniv.nl
LIACS, Leiden University
Leiden, Netherlands

ABSTRACT

In real-world scenarios, it might be beneficial to find the nodes in a network that maximize how much influence they have on the whole network. This is computationally expensive for bigger networks so it is important to find an algorithm that is time efficient. However, one does not want it to lose performance.

In this paper, we present a comparison of multiple different algorithms. These are some discount heuristic algorithms, MixedGreedy, influence maximization with martingales (IMM) and Global Selection Based on Local Influence (LGIM). These algorithms were originally implemented by three different papers. We aimed to find out if the performance in finding the maximum influence and runtime of the algorithms is as discussed in each individual paper compared to each other. Especially as IMM, and LGIM were developed much later than the other algorithms. Furthermore, most of the papers used lower level languages for implementing the algorithms. We looked at how the algorithms compare in a higher level network like Python with the use of simple libraries like Networkx. We wanted to find out if it is viable to use these algorithms in Python. Which after comparison to previous papers that used C++ we concluded that the simplicity of Python and NetworkX is not worth the extra runtime.

Comparison for the influence spread and running time showed us that the heuristics algorithms were the fastest as expected both for wiki-vote and email-Enron. And for wiki-vote, the best performing algorithm was mixed greedy which was not tested for email-Enron due to time constraints. The new modern algorithms (IMM and LGIM) performed quite well but they were slower by one or two orders of magnitudes than the discount algorithms. Furthermore, they still performed worse for a bigger dataset, email-Enron. For wiki-vote they performed slightly better. This performance for differently sized networks is interesting.

KEYWORDS

influence maximization, greedy algorithms, degree discount, social network analysis, network science

ACM Reference Format:
Christian Steennis and Daniel van Heuven van Staereling. 2022. A comparitive study of several influence maximization algorithms: Social Network Analysis for Computer Scientists — Course paper. In *Proceedings of Social Network Analysis for Computer Scientists Course 2022 (SNACS '22).* ACM, New York, NY, USA, 11 pages.

1 INTRODUCTION

In recent years social network analysis has made many advances in different tasks. From community detection to influence spread, the latter will be discussed in this paper. In the real world, there are many examples in social media networks where one sees a spread of information, like for example Facebook. Imagine for a moment, that someone wants to advertise a new product by showing advertisements for users on a social media page. You hope to reach as many people as possible by showing it to a certain amount of people who will then show their friends the product you are selling, which will in turn show it to their other friends and so on. Because of a limited budget, you can only show the advertisement to a small amount of people. Hence, you need to figure out which people need to see the advertisement so that the marketing reach is maximized. The concept of finding this group of people is called influence maximization. Before getting into the actual goal for this paper concerning influence maximization, we explain some concepts in more depth.

First, we clarify the context with respect to influence maximization. Furthermore, we delve into some related work where we also explain some algorithms in a basic manner. Then we can go further into the goal of this paper before going into the methodology and implementation of the algorithms.

1.1 Modelling influence maximization

The goal of influence maximization is to maximize the spread of information in a network. This information is propagated to the points of a network through the connections. We will refer to a network as a graph, the points in a graph as vertices or nodes, and the connections between nodes as edges or links. However, how do we know the spread of information? Or how do we simulate influence?

For simulating spread, we need a model. We use a cascade model to simulate influence spread. There are a multitude of different cascade models. However, this paper focuses on the Independent Cascade model [3]. In a cascade model, the information is spread like a waterfall and flows from one node to another with a probability p that is uniform for all nodes. The model starts with a set of nodes of size k. This set is called the seed set as this is where the information is 'planted'. These nodes are influenced by the information and every iteration of the model, neighboring unaffected nodes are influenced by the already informed nodes. The probability of an informed node influencing a neighbor is always p. This simulates the real world where most people do not tell all information to every friend but maybe only a few.

1.2 Algorithms

The goal of this paper is to compare different algorithms regarding information spread maximization in networks. These algorithms utilize different approaches, such as greedy algorithms (MixedGreedy), heuristic algorithms (single discount, degree discount), reverse random set methods (IMM) and local to global influence methods (LGIM). Greedy algorithms are common approaches as they are easy to implement using exploitation. Heuristics are very time efficient since they require only calculation of some features, but this makes them less robust in their performance. IMM are more modern algorithms that change up their whole approach using approximations which are backed up by different mathematical frameworks. And lastly, LGIM as an algorithm is similar to heuristics but it uses slightly different steps to hone on good solutions in a more elegant way which however also negatively affects the trade off to the runtime. The concepts of these algorithms can be complex so we only summarized them. Later the implementation is introduced in detail in section 3.3.

1.3 Related work

There are many papers about influence maximization. As we discussed in the introduction we opt to compare different algorithms that perform influence maximization to find what the current standings are for the performance of this problem. The start of this research stems from one paper that improved upon the greedy algorithm [1]. They compared their algorithm to greedy and CELF [6]. They made an improved greedy algorithm and two algorithms that used improved heuristics. The paper looked at different cascade models, independent and weighted cascade which we are not doing as we are comparing more algorithms. They used different data sets which are not available anymore.

Greedy algorithms. Greedy algorithms are often used in computer science for them being straightforward to implement. These algorithms choose the highest short term reward. Which often is not the optimal long term reward. In this context we use the fact that we can estimate the number of influenced nodes from a set S by using the aforementioned independent cascade model. We can take a set S and estimate the number of nodes that can be influenced by adding any new node v and estimate this for all nodes $v \in V$. The algorithm adds the node with the maximum influence spread. This process is repeated until set S reaches a predefined size.

The paper [1] implements an improvement on the greedy algorithm that uses some tricks to improve the speed of the greedy algorithm, called NewGreedy. In this approach, they make a new graph in which nodes are deleted with probability 1 - p. Then, the algorithm estimates the influence for all nodes in the new graph. They try to estimate the influence spread by looking at all the descendants for a node. The nodes that have the highest average estimated spread are added to the seed. This process speeds the algorithm up significantly to 700 times from the classical greedy algorithm.

The algorithm that [1] improved on is described in the paper covering outbreak detection [6]. This paper itself improves on the basic greedy algorithm and claims to be 700 times faster. This algorithm is called CELF.

The CELF algorithm uses a concept called lazy forwarding. The first seed node is estimated by the greedy algorithm. The following nodes are added differently. For the new best node the estimate is recalculated, if its ranking does not change it is added to the seed set. Otherwise the algorithm evaluates the next best node.

The NewGreedy algorithm is more efficient in finding the estimated spread but it still has to estimate the spread for each iteration of the seed nodes. Hence, the writers of the paper combined the CELF with the NewGreedy algorithm so that the first iteration uses NewGreedy while the remaining iterations use CELF to make use of lazy forwarding. This combined algorithm is called MixedGreedy.

Discount algorithms. Discount algorithms attempt to improve the performance by selecting seed nodes in a heuristic manner, such as degree and centrality. The performance of these methods is compared to a simple greedy algorithm in [3]. Here we see that the highest degree heuristic does not score as well as a greedy algorithm. Therefore two new heuristics are proposed in [1]. These are called single discount and degree discount. The degree heuristic selects nodes based on their degree however, this method does not take already influenced nodes into account [1]. Single discount only discounts the degree of each node by one for each neighbour in the seed S. This is a simpler algorithm. Degree discount is slightly more complicated. When the propagation probability p of an IC is small the discount should be larger since even though two nodes are connected the probability of one influencing the other, or more nodes through multi-hopping, is so small. Hence, an extra term is subtracted from the degree.

Influence maximisation using Martingales. The previous works are older. As we discussed before influence maximization is important since the rise of social media where social network graphs can become huge, we see that influence maximization remains a relevant problem. Hence, new state of the art algorithms are developed. One of these algorithms is influence maximization with martingales (IMM) [9]. The algorithm is an improvement for TIM and TIM+ [10]. TIM is already an algorithm that combines approximation guarantees with practical efficiency. However, there is still a large computational cost which [9] tries to solve.

Both algorithms are a different type of algorithm that creates random sets and estimates spread based on coverage. This is a complicated concept and to fully understand it, it is necessary to read their paper. However, the main problem is that you need to create some random sets which takes the most time for these type of algorithms. This paper improved on TIM by creating a method that requires the creation of fewer random sets. In section 3.3 we explain more in depth what this means and how this works for the implementation.

TIM uses two steps in the algorithm. Parameter estimation and node selection. There are a few problems with the TIM algorithm, The algorithm requires the estimation of a lower bound for the maximum estimated influence to efficiently use node selection for finding a seed S however, the lower bound TIM finds can be much lower than is necessary. In addition the lower bound estimation is very costly. IMM uses a very similar process to TIM but changes the parameter estimation a lot, secondly the computational efficiency for lower bound estimation is improved, and thirdly the node selection is improved by use of the martingales [12].

Global selection based on local influence. The Global selection based on local influence [7], LGIM, is an algorithm that uses the local influence of nodes to select a seed set. The main idea of this algorithm is that if a node has a large influence in the network nodes influenced by this node also have a large influence. To select candidate nodes the algorithm applies a two-stage node filtering method. The first step in the two-stage process is to select a set of source nodes. These source nodes are then selected based on their influence in a two-hop region. This method is called LFV. The second step in the algorithm is to find the ancestors of these source nodes using Dijkstra's algorithm with a threshold, from these ancestors the candidates for the seed set are selected. The candidates are filtered based on their expected spread, EIOS. Then the seed is selected with the k most influential nodes from the candidate set.

The degree discount algorithm discussed in [1] is a lot faster than the greedy algorithms mentioned in the same paper. However, the proposed degree discount algorithm does not have the performance the optimized greedy algorithms have. To increase the performance and retain the speed advantages this paper [7] tries to find a good compromise between time efficiency and effectiveness. The paper focuses on the local influence values of nodes and selects the most influential nodes to generate a set of seed nodes.

2 PRELIMINARIES

2.1 Notation

variable	description
G	a representation for a full graph
n	the total amount of nodes in G
m	the total amount of edges in G
S	a set of selected nodes called a seed
k	the amount of nodes in S
R	The amount of times you run simulations in an algorithm
ε	Hyper-parameter for the IMM algorithm
l	Hyper-parameter for the IMM algorithm
G'	A graph generated by removing nodes from G with probability p
RR	A reverse random set, which is a set of all nodes connected in G' to another randomly picked node v
$\mathcal{R}$	The set of all created RR sets
V	The set of all vertices in G

Table 1: Some notation descriptions used in this paper

2.2 Research goal goal

There have been many advances in influence maximisation over the years. Algorithms that try to improve the quality of the algorithms and the time efficiency. There have been several attempts to improve the quality and/or time efficiency of already existing methods. The goal of this paper is to find out how well different algorithms compare so that we can determine the maximum influence spread in a reasonable time with decent accuracy in the future. We want to see how well the conclusions for the different discussed algorithms from their respective papers hold up when compared to different algorithms, and if these are still true when done in Python instead of C++.

3 APPROACH

3.1 Dataset

This paper evaluates the algorithms on two different datasets. The first dataset is the Wiki-vote dataset [5]. The dataset is a directed graph with 7,115 nodes and 103,689 edges. Each node represents a user that voted for another user to become an admin. The dataset contains more information but this is the only necessary information for this paper.

The second dataset is called email-Enron [4]. This dataset is an undirected graph with 36,692 nodes and 183,831 edges. The dataset was selected as it contains more nodes than Wiki-vote but it is undirected. That way we can test for an undirected and a directed graph with different node sizes.

3.2 Experimental details

Before getting into the algorithm implementation we mention some experimental details. It is necessary to evaluate the spread after determining a seed set using any algorithm. As mentioned before this paper uses the IC model. However, the IC model requires two parameters. These are the probability p and the number of simulations to average the spread for, mc. These values are both copied from the paper [1]. The probability is 0.01 and the number of simulations is 20,000. The number of simulations needs to be sufficiently large to rule out randomness in the simulations, and the probability needs to model the real world which usually is low.

Furthermore, the seed set is of a certain size. Both [1] and [9] use seeds size k of 1 to 50. In [1] they only compare the runtime for seed size 50 but this paper compares it for all values of k to see how well the difference in speed compares for smaller to larger values for k.

The last thing to mention is that there are a few extra parameters for the IMM algorithm which are copied from their respective paper [9]. These values are general values for the algorithm and do not need to be tuned. In the implementation the parameters will be explained further.

3.3 Algorithm implementation

We approach the problem by implementing and running several different algorithms. All algorithms are implemented in Python. Networkx was used for implementing the algorithms [2]. There are only five algorithms that we use for the experiments. However, the MixedGreedy which we explain use a combined algorithm that uses CELF and NewGreedy. Hence, we also explain those two algorithms along with greedy which CELF uses as its core component. All algorithm implementations can be found on github. [1]

Algorithm 1 GeneralGreedy(G,k,p)

```
1: Initialize $S = \emptyset$ and $R = 20000$
2: for $i = 1$ to $k$ do
3:     for each vertex $v \in V \setminus S$ do
4:         $s_v = 0$
5:         for $i = 1$ to $R$ do
6:             $s_v += |\text{SimulateIC}(S \cup \{v\})|$
7:         end for
8:         $s_v = s_v / R$
9:     end for
10:    $S = S \cup \{\arg\max_{v \in V \setminus S} s_v\}$
11: end for
12: return $S$
```

3.3.1 Original greedy. The general greedy has a simple implementation. The Pseudocode is in Alg. 1. Starting from an empty seed S the algorithm iterates and adds a node to the set until there are k number of nodes in the set S. For each node the estimated spread is estimated by simulating 20,000 times for the IC model which is the same as we discussed in the experimental details. The nodes are added greedily by selecting the maximum estimated spread by adding the node to S. The node that has the highest spread when added to S is then actually added to S. This takes a long time as for each node the model needs to estimate the spread, even worse this is repeated for all k iterations.

Algorithm 2 NewGreedy(G,k,p)

```
1: Initialize $S = \emptyset$ and $R = 20000$
2: for $i = 1$ to $k$ do
3:     Set $s_v = 0$ for all $v \in V \setminus S$
4:     for $i = 1$ to $R$ do
5:         Compute $G'$ by removing each edge from $G$ with probability $1 - p$
6:         Compute $R_{G'}(S)$
7:         Compute $|R_{G'}(\{v\})|$ for all $v \in V$
8:         for each vertex $v \in V \setminus S$ do
9:             if $v \notin R_{G'}(S)$ then
10:                $s_v += |R_{G'}(\{v\})|$
11:            end if
12:        end for
13:    end for
14:    Set $s_v = s_v / R$ for all $v \in V \setminus S$
15:    $S = S \cup \{\arg\max_{v \in V \setminus S} s_v\}$
16: end for
17: output $S$
```

3.3.2 NewGreedyIC. NewGreedy is slightly improved from the classical greedy as it has an improved method of simulating the spread. The difference with greedy is that instead of simulating the IC model for each node we make an estimate. We generate a graph G'. From the original graph G we delete any edge with probablity 1-p. This leaves us the graph G'. Then for each node v in G we look at the number of nodes that v can reach in G' so we want all the descendants from v. If there are none in G' then it should return 0. This is done by 20,000 times generating a different graph G' and estimating the spread. The highest average spread is the node selected greedily to add to S. This is slightly more efficient as the IC model does not need to simulate the spread but it still requires creating 20,000 graphs G' which takes time.

[1]https://github.com/danielvhvs/SNACS_IM_LU

Algorithm 3 CELF(G,k,p,mc)

```
1: $mg_v \leftarrow$ SimulateIC(v)
2: $Q \leftarrow$ sorted marginal gain list in descending order with corresponding node values
3: $S \leftarrow$ the first value in Q
4: $Q \leftarrow$ remove the first value of Q
5: for $1, 2, \ldots k - 1$ do
6:     $check \leftarrow$ False
7:     while not $check$ do
8:         $current \leftarrow$ first node value of Q
9:         $Q[0] \leftarrow (current, SimulateIC(S \cup \{current\}) -$
10:        $SimulateIC(S))$
11:        $Q \leftarrow$ sorted marginal gain list in descending order with
12:        corresponding node values
13:        $check \leftarrow$ True if previous top node stayed on top
14:    end while
15:    add best node from Q to S
16:    remove best node in Q
17: end for
18: return $S$
```

3.3.3 CELF. The CELF algorithm is closely related to the greedy algorithm. The first iteration is the same as greedy however after that it changes. The nodes are sorted w.r.t. the estimated spread. The node with the highest spread estimate is selected and the new spread estimate w.r.t. the seed is estimated. The spread estimate for each node is sorted again and if the node with the highest spread has not changed then it is added to the seed S. This is a big improvement over doing it like classical greedy as it requires doing the estimating process much less often.

MixedGreedyIC. MixedGreedy IC is the actual algorithm we test in this paper. It is a combination of NewGreedyIC and CELF which we do not test in this paper. It starts with a seed S and then iteratively adds k number of nodes. The first node is added by estimating the spread using the NewGreedyIC but then it changes and uses CELF. This means it is just CELF but the first step changes and instead of calculating the estimate for the spread using simulations of the IC model we use the spread estimate by creating graphs G' like for newgreedyIC.

Algorithm 4 SD

```
1:  Initialize $S = \emptyset$
2:  for each vertex $v$ do
3:      Compute its degree $d_v$
4:      $dd_v = d_v$
5:  end for
6:  for $i = 1$ to $k$ do
7:      Select $u = \arg\max_v \{dd_v \mid v \in V \setminus S\}$
8:      $S = S \cup \{u\}$
9:      for each neighbor $v$ of $u$ and $v \in V \setminus S$ do
10:         for each neighbor $w$ of $v$ and $w \in S$ do
11:             $dd_v = dd_v - 1$
12:         end for
13:     end for
14: end for
15: return $S$
```

3.3.4 SingleDiscountIC. The single discount heuristic algorithm is very simple. It uses the degree as an estimate for which nodes increase the influence spread the most. Before selecting nodes for the seed S it calculates the degree for each node. The node with the maximum degree is then selected. For each neighbour of the newly selected node, the calculated degree is then decreased by one and then the process repeats until k nodes have been selected.

Algorithm 5 DDIC

```
1:  Initialize $S = \emptyset$
2:  for each vertex $v$ do
3:      Compute its degree $d_v$
4:      $dd_v = d_v$
5:      Initialize $t_v$ to 0
6:  end for
7:  for $i = 1$ to $k$ do
8:      Select $u = \arg\max_v \{dd_v \mid v \in V \setminus S\}$
9:      $S = S \cup \{u\}$
10:     for each neighbor $v$ of $u$ and $v \in V \setminus S$ do
11:         $t_v = t_v + 1$
12:         $dd_v = d_v - 2t_v - (d_v - t_v)t_v p$
13:     end for
14: end for
15: return $S$
```

3.3.5 DegreeDiscountIC. The degree discount is similar to single discount hence the implementation is very similar. The difference is that instead an extra value t is set at 0 before the seed notes are selected. After a seed node is selected instead of decreasing the degree by one there is another formula used to calculate the new degree. The formula uses the value t for the node as well as the propagation probability p. The value t is increased before adjusting the degree. Since we do not want nodes that are neighbours to selected nodes we increase t every time we see that a node v is the neighbour of the added node u. The degree of node v is decreased which will be higher as node v is more often a neighbour of u. The equation 1 is used for calculating the new degree (dd) which is based on the actual degree, t and p.

$$dd = d - 2t - (d - t)tp \tag{1}$$

Algorithm 6 node_selection(G,$\mathcal{R}$,k)

```
1:  Initialize node set S = $\emptyset$
2:  $degree_v$ = number of occurrences of v in $\mathcal{R}$
3:  for $1, 2, \ldots k$ do
4:      maxNode = node with max $degree_v$
5:      S = $S \cup maxNode$
6:      remove maxNode from degree
7:      for $RR \in \mathcal{R}$ where maxnode $\in RR$ do
8:          if RR is not seen before then
9:              for $w \in RR$ do
10:                 $degree_w = degree_w - 1$
11:             end for
12:         end if
13:     end for
14: end for
15: return $S_k$
```

Algorithm 7 Sampling(G,k,ε,l,p)

```
1:  Initialize node set R = $\emptyset$ and LB=1
2:  $\varepsilon' = \sqrt{2} \cdot \varepsilon$
3:  for $i = 1, 2, \ldots \log_2 n - 1$ do
4:      $x = n/2^i$
5:      $\lambda' =$
6:      $\theta_i = \lambda'/x$
7:      while $|\mathcal{R}| \leq \theta_i$ do
8:          Generate a new RR set based on random node and insert
            into $\mathcal{R}$
9:      end while
10:     $S_i$ = node_selection(G,$\mathcal{R}$,k)
11:     if $n \cdot F_{\mathcal{R}}(S_i) \geq (1 + \varepsilon') \cdot x$ then
12:         $LB = n \cdot F_{\mathcal{R}}(S_i)/(1 + \varepsilon')$
13:         break
14:     end if
15: end for
16: $\lambda^* =$
17: $\theta = \lambda^*/LB$
18: while $|\mathcal{R}| \leq \theta$ do
19:     Generate a new RR set based on random node and insert
        into $\mathcal{R}$
20: end while
21: return $\mathcal{R}$
```

Algorithm 8 IMM(G,k,ε,l,p)

```
1:  $l = l \cdot (1 + \log(2)/\log(n))$
2:  $\mathcal{R}$ = Sampling(G,k,$\varepsilon$,l,p)
3:  $S_k$ = node_selection(G,$\mathcal{R}$,k)
4:  return $S_k$
```

3.3.6 Martingales approach. This algorithm uses an approximation method for selecting nodes for the seed set. There are two phases. The first phase samples reverse random (RR) sets. The RR sets are bunched in one set $\mathcal{R}$. The first phase has two steps. The first phase is an iterative process of a maximum number of rounds to avoid it taking too long. A value θ_i is selected based on a formula from [9]. θ increases every iteration. After estimating θ_i a number of RR sets is generated and added to $\mathcal{R}$ such that $|\mathcal{R}|$ is bigger than θ_i. Then the node selection algorithm is applied which gives an estimate for the spread. If this spread exceeds a certain boundary value then the loop stops. The value is calculated using formulas from [9]. After the iteration stops a lower bound value for OPT is determined based on the last estimate of the spread from the $\mathcal{R}$ set. This lower bound LB is then used to calculate the actual θ which is the minimum number of RR sets there need to be in $\mathcal{R}$. So after the previous loop stops there are enough RR sets added such that $|\mathcal{R}|$ is bigger than θ.

The node selection is the next step. It is already used inside of the sampling step. Node selection selects the seed based on the $\mathcal{R}$ set. It assumes that if a node occurs frequently in the sets it is also an important node. Hence the algorithm first determines the frequency of each node and then adds the highest frequency node to the seed set S. The algorithm is implemented based on [9]. It uses a greedy approach for the maximum coverage problem [11]. Hence the highest frequency node is selected. After a node is selected the degree (frequency) for all other nodes that were in the same RR sets are reduced by 1. Since they cannot activate the just added node anymore hence their importance is less. This process is iterative and stops when k nodes have been reached. The spread used for sampling is based on the fraction of nodes the seed set covers.

There are some hyperparameters the algorithm uses. These are ε and l. The same values are used from the paper [9]. ε is 0.5 and l is 1.

3.3.7 LGIM. The LGIM algorithm [7] uses the local influence of nodes to make a global selection of seed set S. The algorithm first calculates the influence it has on its neighbours up to two nodes away, based on the propagation probability. This local influence LFV is defined in Equation 2. The second step of the algorithm is to select a set of source nodes SN. This set contains the nodes with the largest local influence. The number of nodes in the source set is defined by pop.

Algorithm 9 LGIM(G,k,p)

```
1:  $lowbound = 100$
2:  $upperbound = 600$
3:  $S = \emptyset$
4:  $threshold = 1/\max_{v \in V}(degree(v))$
5:  $x = 2 * |E|/|V|$
6:  $pop = |V|/(1 + x + x^2)$
7:  if $pop \leq lowbound$ then
8:      $pop = lowbound$
9:  end if
10: if $pop \geq upperbound$ then
11:     $pop = upperbound$
12: end if
13: $SN = SelectSourceNodes(G, pop)$
14: $R, MAP = SeekAncestorMAP(G, SN, threshold)$
15: $C = FilterCandidates(G, R, MAP, k)$
16: $S = SelectSeed(G, C, MAP, k)$
17: return $S$
```

Definition (LFV): The local influence of a node within two-hop region

$$LFV(u) = 1 + \sum_{v \in N_u \setminus \{u\}} (p_{uv} + p_{uv} * \sum_{s \in N_v \setminus \{u,v\}} p_{vs}) \qquad (2)$$

The next step in the algorithm is to find the ancestors of the source nodes and to calculate the maximum active probability MAP of an ancestor R activating a source node in SN. Now that all of the ancestors of the most influential nodes are known, the algorithm filters these ancestors into a candidate set. This candidate set C has twice the size of the desired seed size k. The ancestors are filtered by calculating the global influence of every node in the ancestor set. The function that calculates this global influence is the EIOS function. This function uses the local influence to compute the expected influence of a node or set of nodes. To calculate this, Equation 3 will be used. EIOS takes the probability of a seed set S activating the earlier selected source nodes. While taking into account that the source node can also be activated by another node in the seed set. In the equation the p_{uv} and p_{nv} are the maximal active probabilities of u and n activating v respectively.
Then as a last step, the final seed set S is selected from the candidate set. For each of the candidates, the benefit of selecting the candidate into the seed set is considered. The global influence of the current seed set with and without the candidate node is compared.

Definition (EIOS): The expected benefit of nodes influenced by the node set S

$$EIOS(S) = \sum_{u \in S} \sum_{v \in SN} p_{uv} * LFV(v) * \prod_{n \in S \setminus \{u\}} (1 - p_{nv}) \qquad (3)$$

Algorithm 10 SelectSourceNodes(G,pop)

```
1: $SN = \emptyset$
2: for $i = 1$ to $pop$ do
3:     $SN = SN \cup \arg\max_{v \in V}\{LFV(v)\}$
4: end for
5: return $SN$
```

Algorithm 11 SeekAncestorMAP(G,SN,threshold)

```
1: $R = \emptyset$
2: $MAP = \emptyset$
3: for each $v \in SN$ do
4:     $maxAP = \emptyset$
5:     $Q = priorityQueue()$
6:     $maxAP[v] = 1$
7:     for each $v \in V$ do
8:         $visit[v] = False$
9:     end for
10:    Add $(v, -maxAP[v])$ to $Q$
11:    while $|Q| > 0$ do
12:        $u, v = Q.pop()$
13:        if $visit[u] == True$ then
14:            continue
15:        end if
16:        $visit[u] = True$
17:        for each $n \in Predecessors(u)$ do
18:            $new\_ap = maxAP[u] * p_n u$
19:            if $new\_ap > threshold$ then
20:                $R = R \cup n$
21:                if $new\_ap > maxAP[n]$ and $visit[n] == False$ then
22:                    $maxAP[n] = new\_ap$
23:                    Add $(n, -maxAP[n])$ to $Q$
24:                end if
25:            end if
26:        end for
27:    end while
28:    $MAP[v] = maxAP$
29: end for
30: return $R, MAP$
```

Algorithm 12 FilterCandidates(G,k,p)

```
1: $C = \emptyset$
2: for $i = 1$ to $2 * k$ do
3:     $C = C \cup \arg\max_{v \in R \setminus C} \{EIOS(v)\}$
4: end for
5: return $C$
```

Algorithm 13 SelectSeed(G,k,p)

```
1: $S = \emptyset$
2: $Q = priorityQueue()$
3: $T = \emptyset$
4: for each $v \in C$ do
5:     $inf = EIOS(\{v\})$
6:     $T[v] = 0$
7:     Add $(v, -inf)$ to $Q$
8: end for
9: $v = Q.pop()$
10: $S = S \cup \{v\}$
11: for $i = 2$ to $k$ do
12:     for $j = 1$ to $2 * k$ do
13:         $v = Q.pop()$
14:         if $T[v] == i$ then
15:             $S = S \cup \{v\}$
16:             break
17:         else
18:             $S_v = EIOS(S \cup \{v\}) - EIOS(S)$
19:             Add $(v, -S_v)$ to $Q$
20:             $T[v] = i$
21:         end if
22:     end for
23: end for
24: return $S$
```

3.4 Experiments

Experimental setup. The experiments are performed on all the following algorithms.

- MixedGreedy (mix of NewGreedy and CELF)
- SingleDiscount
- DegreeDiscount
- IMM
- LGIM

One side note to make. The MixedGreedy was very resource intensive. Therefore, we did not run the algorithm for the email-Enron dataset. We did still run it for the wiki-vote dataset.

To start there are some important parameters for the experiments. Each algorithm returns a seed S of size k. The seed size k is adjusted from 1 to 50 to compare performance on different k's. This paper looks at the time it takes to run each algorithm for each k. After the seed is found, simulations on the independent cascade model are performed $mc = 20,000$ times to average the randomness out. The average spread is then used as the performance of the algorithm. The independent cascade model uses probability $p = 0.01$ from [1]. Other hyper-parameters are mentioned already for each algorithm before.

4 RESULTS

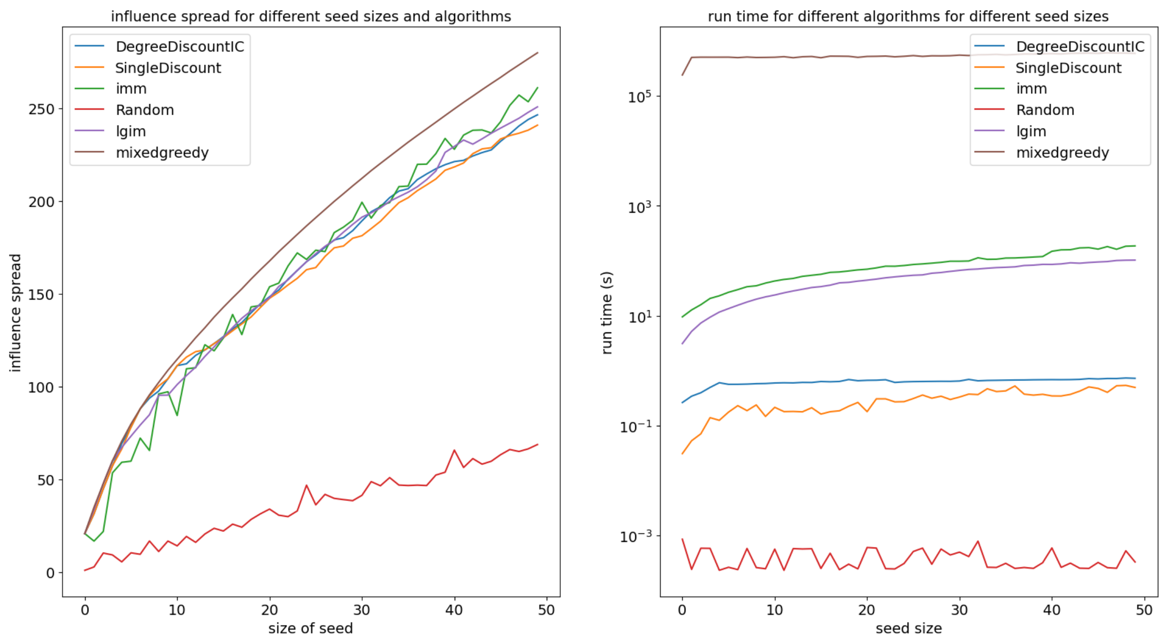

Figure 1: The results for the Wiki-vote data for all algorithms.

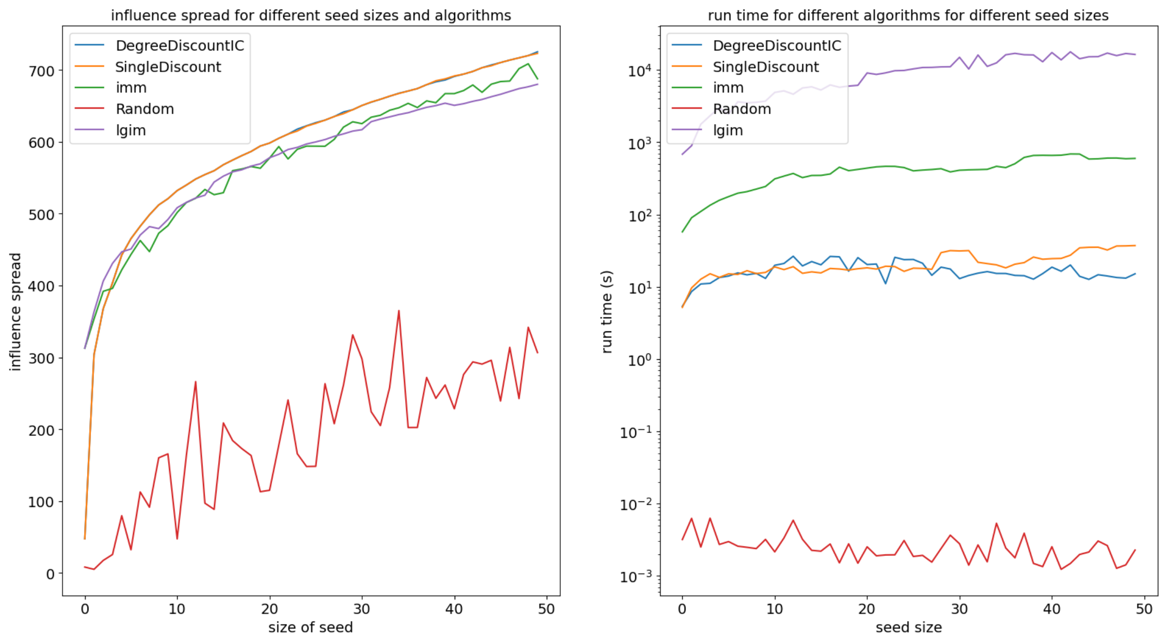

Figure 2: The results for the email-Enron data for all algorithms except newgreedy

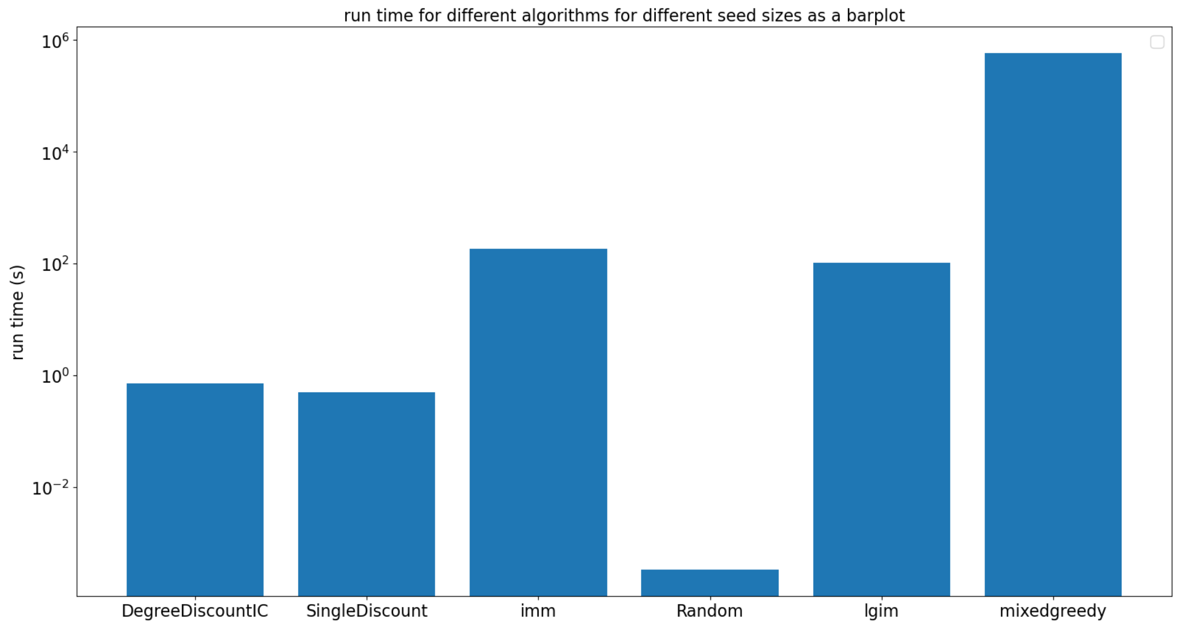

Figure 3: The runtime for seed size of k=50 for the Wiki-vote data for all algorithms

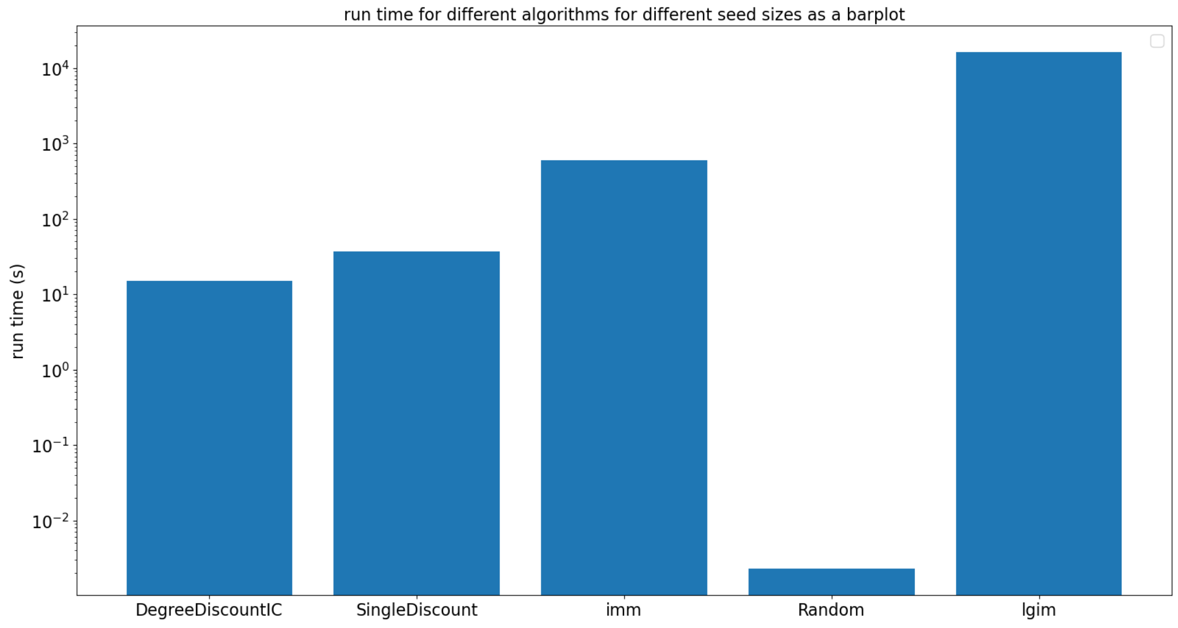

Figure 4: The runtime for seed size of k=50 the email-Enron data for all algorithms except newgreedy

First, we look at Fig. 1 for the results. This figure shows the performance and running time for the wiki-vote data. The figure shows that the performance for MixedGreedy is the best as expected. The other four algorithms perform about the same. IMM and LGIM seem to perform slightly better for the higher k values.

The runtime for wiki-vote is as expected. The fastest algorithms are the discount algorithms, after that are IMM and LGIM, and lastly by a significant margin is MixedGreedy. The time for LGIM seems to be slightly faster than IMM. Similarly, the runtime for single discount is slightly faster than degree discount. Single discount requires less computation so this is expected.

If we then compare the results for email-Enron we see a slight difference. First of all, the performance of the discount algorithm compared to IMM and LGIM is now better. Before they compared equally and sometimes worse but here it seems clear that the performance is better. Which is interesting as the dataset has more nodes and edges. The performance difference between IMM and LGIM is still the same. IMM performs slightly better but by an insignificant amount.

The runtime shows more interesting parts. Degree discount is now sometimes faster than single discount. This is interesting as generally degree discount should perform more computation and hence be slower. What is also unexpected is that now suddenly LGIM is slower than IMM and also by a much larger amount of time than Wiki-vote.

For all algorithms, we can see that the performance is significantly greater than random selection which is good to see.

The bar graphs in Fig. 3 and Fig. 4 show the time for comparison for just the seed size of k=50. This is the same data as for the line graphs. However, it shows the differences a little better. The IMM and LGIM algorithm take about an order of 10 to the power two seconds for wiki and for Enron this is about 10^3 for IMM and 10^4 for LGIM. For the discount algorithms, it goes from about 1 second for wiki-vote to 10 seconds for email-Enron. MixedGreedy is significantly slower at already 10^6 seconds for wiki-vote.

5 DISCUSSION AND CONCLUSION

Influence maximization is a difficult problem. It is possible to create algorithms that find large influence spread. This is really good for smaller networks. However, for larger networks this is not the only thing one cares about. For a larger network you run into the problem of runtime. As people strive to make new algorithms they sometimes only care about performance improvement and not always about time efficiency. Or they forgot about older algorithms that performed really well just because the performance is worse even though their time efficiency is really good. In this paper we have looked at new state of the art algorithms and compared them to some older well performing algorithms, greedy and heuristic approaches.

What was unexpected was the performance of the algorithms. As the discount algorithms use simple heuristics we expected that they would perform worse. However, they don't show major inferiority to the greedy algorithms which are more complex, and they actually perform better in or equal in some cases to the state of the art algorithms like LGIM and IMM.

Furthermore, the experiments were done using implementations in Python. Compared to [1] and [9] which both used C++. We used different datasets, as NetHEPT and NetPHY are datasets that are not accessible easily anymore. However, NetPHY has a similar number of nodes and edges to email-enron. NetPHY actually has slightly more. If we compare the running time for the discount algorithms we can see that their results performed at about 10^{-3} seconds compared to our implementation which took about 10 seconds.

Secondly, compared to the paper about martingales [9], their algorithms generally took no more than 10 seconds on the datasets they tested even for Pokec [8], which has 1.6 million nodes and 3 million edges. This compared to our implementation which for Enron took 10^3 seconds approximately.

This comparison of the implementation of the algorithms in Python compared to the paper's implementation in C++ shows the complexity of the problem. Our implementations take several orders of magnitude longer. The running time being so much longer is really jarring. Hence, even though Python might be simpler which is often why it is used for applications regarding neural networks. It is not worth it because of the time difference. c++ is just superior in this case.

Only in the case of LGIM did the original paper not use C++ but Python. This showed still a difference off an order of 100 times difference. Our implementation was using NetworkX which is a simple library but still very slow. Without knowing the implementation of the paper we cannot make any conclusions on the exact reason for the time difference. However, most other libraries often use C or C++ for their implementation while NetworkX does not. Hence, NetworkX is slow.

In conclusion, heuristic algorithms perform really well even though they are simple algorithms that have been around for longer than some state of the art algorithms. Secondly, even though Python is often used for other machine learning applications like neural networks, for this kind of problems the difference in runtime is just too much and hence c++ or other lower level languages are preferred due to their faster computation time.

REFERENCES

[1] Wei Chen, Yajun Wang, and Siyu Yang. 2009. Efficient influence maximization in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. Association for Computing Machinery, https://doi.org/10.1145/1557019.1557047, 199–208.

[2] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX.* Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[3] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the Spread of Influence through a Social Network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Washington, D.C.) *(KDD '03)*. Association for Computing Machinery, New York, NY, USA, 137–146. https://doi.org/10.1145/956750.956769

[4] Bryan Klimt and Yiming Yang. 2004. Introducing the Enron corpus.. In *CEAS*, Vol. 45. -, Language Technology Institute, Carnegie Mellon University, Pittsburgh, PA 15213, USA, 92–96.

[5] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*. Association for Computing Machinery, https://doi.org/10.1145/1753326.1753532, 1361–1370.

[6] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. 2007. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. Association for Computing Machinery, https://doi.org/10.1145/1281192.1281239, 420–429.

[7] Liqing Qiu, Xiangbo Tian, Shiqi Sai, and Chunmei Gu. 2020. LGIM: A Global Selection Algorithm Based on Local Influence for Influence Maximization in Social Networks. *IEEE Access* 8, 0 (2020), 4318–4328. https://doi.org/10.1109/ACCESS.2019.2963100

[8] Lubos Takac and Michal Zabovsky. 2012. Data analysis in public social networks. In *International scientific conference and international workshop present day trends of innovations*. DTI, Łomża, Poland.

[9] Youze Tang, Yanchen Shi, and Xiaokui Xiao. 2015. Influence maximization in near-linear time: A martingale approach. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. Association for Computing Machinery, https://doi.org/10.1145/2723372.2723734, 1539–1554.

[10] Youze Tang, Xiaokui Xiao, and Yanchen Shi. 2014. Influence maximization: Near-optimal time complexity meets practical efficiency. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. Association for Computing Machinery, https://doi.org/10.1145/2588555.2593670, 75–86.

[11] Vijay V Vazirani. 2001. *Approximation algorithms*. Vol. 1. Springer.

[12] David Williams. 1991. *Probability with martingales*. Cambridge university press, -.