CS224W Project Write-up
Static Crawling on Social Graph

Chantat Eksombatchai
Norases Vesdapunt
Phumchanit Watanaprakornkul

Introduction

Our problem is crawling a static social graph (snapshot). Given limited resource and some methods to explore the graph with some constant cost, we want to discover as many distinct nodes and edges as possible.

This is adapted from the problem that we try to crawl twitter social graph using twitter API [1] with some restrictions such as we can't make more than 150 requests per hour. In this scenario, we want to mine real data from twitter as much as possible and forward it to do something else. Our goal is not an unbiased representative of the structure of the graph. In our problem, we also assume that there is no node/edge deletion/addition for simplicity.

We first evaluate baseline crawlers on synthetic forest fire graphs [6]: always teleport, random walk, greedy, and lottery (see Relevant Prior Work section for more information) [4]. We test each model using first our synthetic forest fire networks and finally Twitter follower data [2]. We then develop our own model inspired by discount degree heuristics [5] for influence maximization. This model tries to learn the best distribution for randomwalk from sampled networks. We also try other variances of greedy approach and mixed method: optimized greedy model, random seeds to be the starting point of greedy algorithm. Finally, because the result of teleport is the best in the beginning of the crawling process and optimized greedy gets better later on, we try the mixed method between teleport and greedy. Our experiment indicates that seeded greedy is the best model we have for twitter graph for discovering edges and teleport is the best for discovering nodes.

API

The model will follow the twitter API [1], where each function has a constant cost of one (i.e. a call to any of these functions will use up one unit of the resource).

NodeID getTeleportNode()
Return a random seed to start crawling from. This could be drawn from a uniform distribution, but in reality when we are crawling twitter, we can only pick random nodes by sampling from the twitter stream. It is highly likely that we will pick nodes with a lot of activity and thus we might need to have different weights for different nodes. Our implementation get the node based on degree. Higher degree has more change to be teleported to.

Vector<NodeID> getFollowers(node, index)
If the node has more than C followers, return C followers of the node starting from index. Otherwise, return all the followers of the node. If the index is not specified, it is set to zero.

Vector<NodeID> getFollowees(node, index)

If the node has more than C followees, return C followees of the node starting from index. Otherwise, return all the followees of the node. If the index is not specified, it is set to zero. This corresponds to GET friend request in twitter API.

Pair(int,int) getDegree(node)
Return the number of followers and followees of a node. Instead of getting the neighbors of a node in batches of C, we might choose to get its degree instead to check first if it is worth exploring or not.

Relevant Prior Work and Our Implementation

We explore the following baseline models from algorithms used in Ye et al. 2010 [4].

1. Randomly explore nodes (always teleport): We obtain all followees and followers of each node we teleport to. Here we try to imitate obtaining a random user from the actual twitter stream by obtaining a node proportional to its in-degree because users with more followers tend to be more active (post statuses more often) according to Kwak et al. 2010 [2].

2. Random walk with uniform distribution and some teleporting probability: We randomly choose to explore nodes to explore from the set of nodes we have seen but not yet crawled.

3. Greedy (from Ye et al. 2010 [4]): We explore the node with maximum in-degree (number of followers) that we have not crawled in each step. We implemented this by using a maximum heap to keep track of nodes seen but have not yet crawled. We keep extracting the maximum from the heap to obtain the node with the highest in-degree and once we crawl the node, we add newly seen nodes to our priority queue. It is easy to see that for each node, we add it into the heap once and remove it at most once and thus, the total running time to explore the whole graph is O(n log n) where n is the number of nodes in the graph.

4. Lottery (from Ye et al. 2010 [4]): This is similar to our greedy process but it is probabilistic rather than deterministic. We bias our exploration towards higher in-degree nodes. Naive implementation would take O(n) to random a node to explore in graph of n nodes. Our implementation use a binary indexed tree to sum total degree (double count) of a set of nodes in O(log n). In this way, we can do binary search to select a random node to explore the graph according to degree distribution. Therefore, the running time to explore the whole graph is O(n $\log^2$ n).

Data

We first generate synthetic data and then we will move to Twitter follower data for our final test. We generate synthetic data using the forest fire model proposed by J. Leskovec et al, 2007 [6]. We use this model because it models evolving social network graphs well with densification and shrinking diameter. It also generates graphs with power-law degree distribution. The twitter graph dataset we use (contains 41.7 million users and 1.47 billion social relations) is from http://an.kaist.ac.kr/traces/WWW2010.html made publicly available for Proceedings of the

19th International World Wide Web (WWW) Conference, April 26-30, 2010, Raleigh NC (USA) [2].

Evaluation

With limited cost, we evaluate crawlers by calculating the recall values for nodes and edges as follows:
recall_node = #nodes found/ #nodes in real graph
recall_edge = #edges found/ #edges in real graph

We will also compare the performance of different models more precisely by plotting the number of newly discover nodes per X requests/iterations. So for every X requests, we collect how many new nodes we discover in a pair vector and plot all of the models in the same plot. The x axis is number of requests and the y axis is the number of newly discovered nodes. This convergence plot can be use to evaluate performance even when we do not know the whole graph unlike *recall*.

Similarly, we can apply this evaluation applies to newly discover edges as well. And the performance shown in our resulting plots will distinguish models for appropriate scenarios. For instance model 1 may first discover many new nodes but the performance drops very fast as we make more requests (e.g., sharp diminishing returns). On the other hand, model 2 may not discover as many new nodes as model 1 does but its performance does not drop as much as when we make more requests. In this case, we would use model 1 if we have a very limited number of requests but would prefer model 2 if we have larger number of requests.

Initial Findings

We have completed implementing the baseline model crawlers and tested them with our synthetic forest fire graph. Our forest fire graph has has forward burning probability of 0.35 and backward burning probability of 0.32. Below is the performance comparison of baseline models:

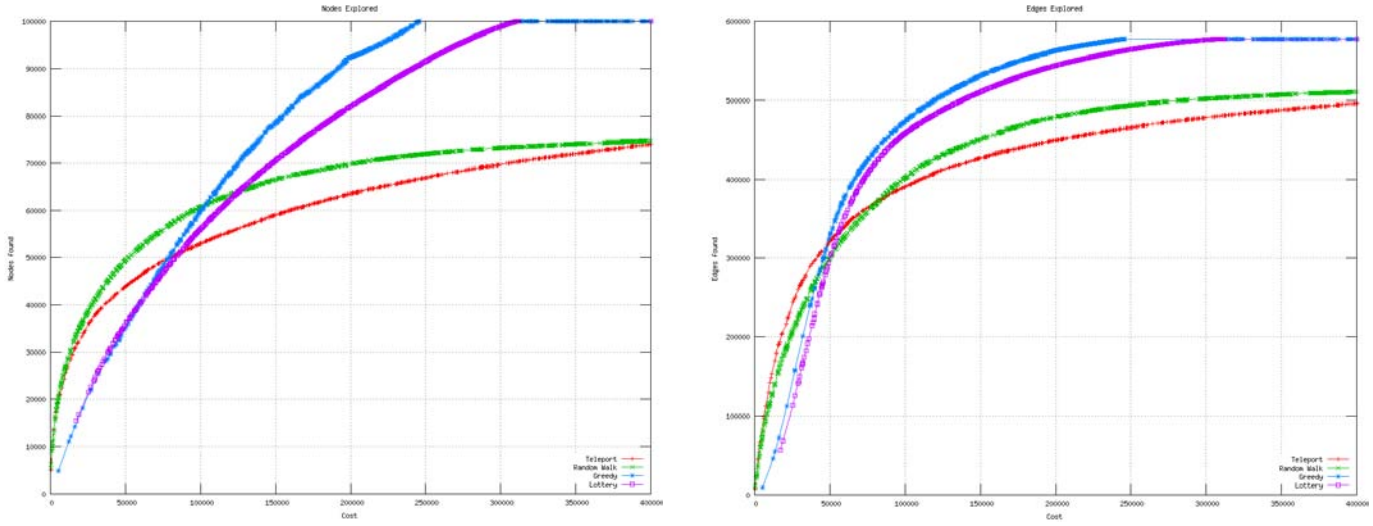


Figure 1: 100,000 Nodes

Let n be the number of nodes and m be the number of edges. Note that we use C = 50 in our experiments which means that each time we call getFollowers, the maximum size of the returned vector of follower ids is 50. This also applies to get Followees. In the actual twitter API, C is 5000 [7].

On the left, we plot the cost (number of requests) against the number of nodes found. We can see that initially random walk and teleport outperform greedy algorithm for approximately the first n requests. However, beyond that, greedy performs the best and it finds all of the nodes within 2.5n requests. Teleport slightly outperforms random walk for all number of requests. Lottery performs similarly to greedy but the growth of number of nodes found is less steep. As the number of request grows, the gap between greedy and lottery grows. Greedy discover all of the nodes roughly 0.7n requests before lottery does.

On the right, we plot the cost (number of requests) against the number of edges found. We can see that initially random walk and teleport outperform greedy algorithm for approximately the first 0.15m requests. However, beyond that, greedy performs the best and it finds all of the edges within 2.5n requests. Teleport and random walk perform essentially the same for all number of requests. Lottery performs almost identical as greedy but slightly worse. From the gap of between greedy and lottery curves, we can see that greedy discovers all of the edges approximately n requests before lottery does.

Proposed Models

Optimized Greedy

1. Fine-Grained Greedy - In the original model, we explore the node with maximum in-degree by requesting all followers and followees of that node in each step. Because this model only looks at the in-degree, we have no control over the node out-degree. This might cause problems when have nodes with low in-degree and high out-degree, or nodes with high in-degree and low out-degree. To fix this problem, we look at node A with the maximum in-degree and node B with the maximum out-degree in each step. If the value of the in-degree of A is greater than the value of the out-degree of B, we explore all the followers of A, otherwise we explore all the followees of B. The result of this improvement is shown as Greedy2 in the figure below.

2. Stepwise Greedy - We also notice that it might not be a good idea to get all the followers or followees in each step. For example, we might have some nodes with 2C + 1 followers and if we get all followers, we will have to use 3 requests for it. In this case, it might be better to get 2C followers using 2 requests and move on to other nodes. Hence, in Stepwise Greedy, every time we explore a node, we only get at most C followers or followees, minus its degree by the number of followers or followees returned and put the node back into the queue. This improvement with the previous one is shown as Greedy3 in the figure below.

3. Greedy with Pruning - When we discover each node, we have to ask for its degree which uses a cost of one and explore it using a cost of two. Because we know that our graph has a power-law degree distribution, we expect that there will be a lot of nodes with degree less than C and

thus wasting a lot of our resource asking for the nodes' degrees. A better idea might be to query for the node's followers and followees directly. If the number of followers and followee returned is less than C, we get the degree as the result, but if the number of followers and followees returned is equal to C, we do not know the degree and will have to query for the degree. Hence, we use a cost of two to explore nodes with small degree instead of a cost of three and cost of at least four to explore nodes with high degree. The result of this improvement and the previous two is shown as Greedy4 in the graph.

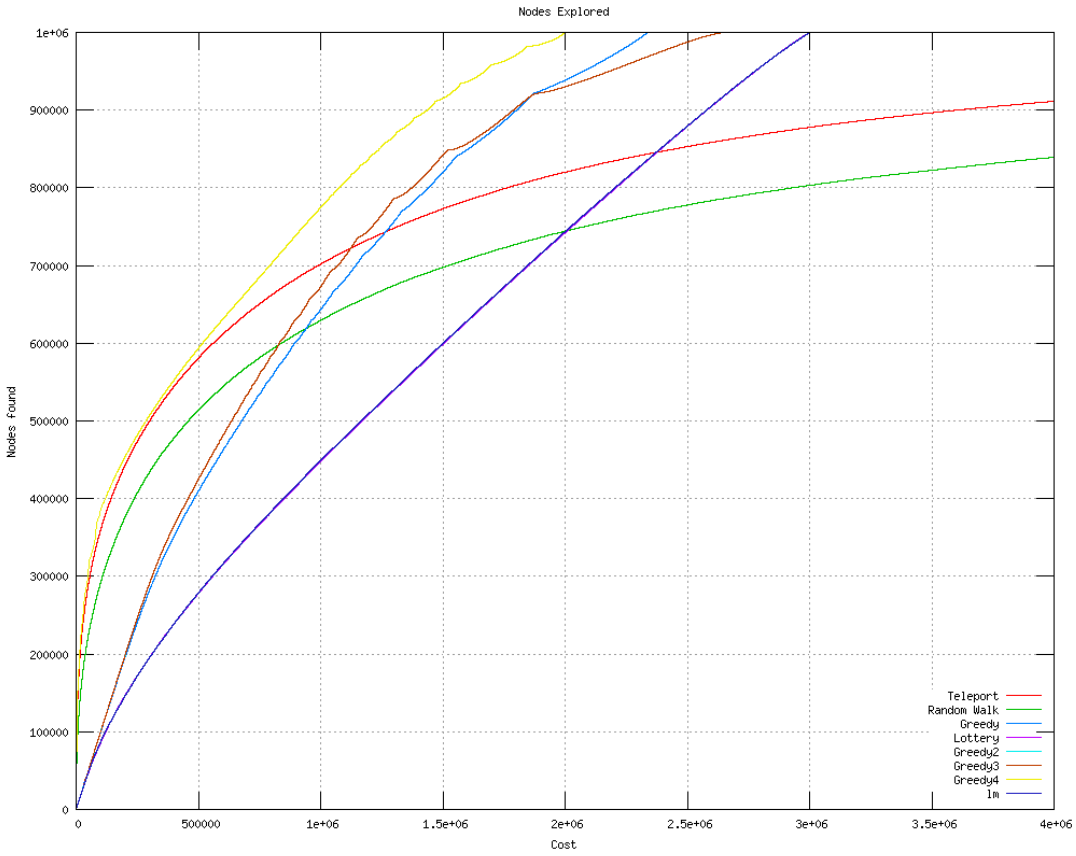
Learned Model

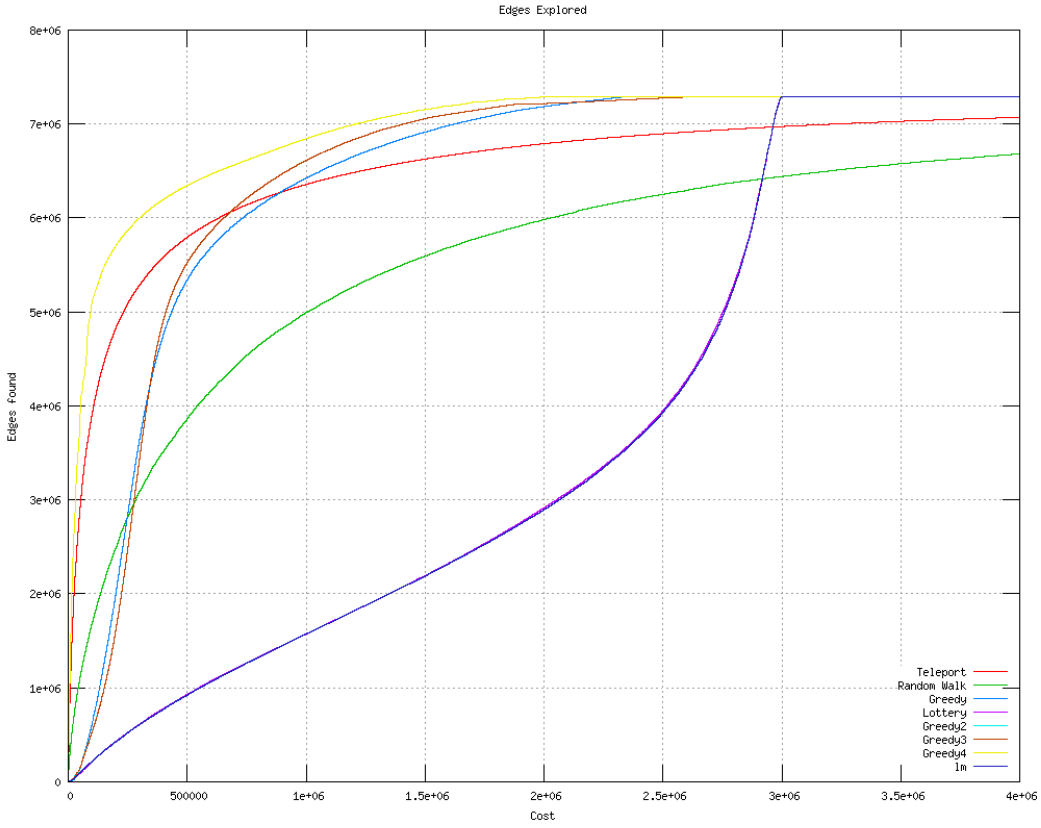
This model is very similar to Lottery approach. It chooses a node to explore randomly and ask for every edge and node adjacent to it. The only difference from Lottery is that the probability of each node to be explored is not just the uniform distribution on the degree of the node. The reason behind this is that even if the node we are going to explore have high degree, if we have already seen a lot of those adjacent edges beforehand, asking for them again would be a waste of resource. So, in this new model, we use something similar to discounted degree instead (Chen et al. 2010 [5]). Note that in crawling, we do not have the discounted degree before we crawl, so we have to settle with something similar. In addition, the distribution does not have to be uniform.

The idea of learned model is that, we will try to learn the right distribution for the random walk from randomly generated forest fire networks (which hopefully resemble real world twitter data in a smaller scale). We will find probability or score of how likely a node with a particular characteristics can get us to unseen nodes. Here, the characteristics we consider are both the edges in the real graph and the edges we already discovered. we also differentiate between being follower and being followee. (treat them as key/bucket.) For each bucket, we sum and normalize the gain (nodes and edges we discovered by exploring that node with the particular characteristics)

Experimental Results

Synthetic Networks


Nodes Explored
Nodes Found
Cost
1e+06
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
500000
1e+06
1.5e+06
2e+06
2.5e+06
3e+06
3.5e+06
4e+06
Teleport
Random Walk
Greedy
Lottery
Greedy2
Greedy3
Greedy4
lm


Edges Explored
Edges Found
Cost
8e+06
7e+06
6e+06
5e+06
4e+06
3e+06
2e+06
1e+06
0
500000
1e+06
1.5e+06
2e+06
2.5e+06
3e+06
3.5e+06
4e+06
Teleport
Random Walk
Greedy
Lottery
Greedy2
Greedy3
Greedy4
lm

Optimized Greedy

Greedy2 - We can see a small improvement from the baseline greedy, which must mean that nodes with low in-degree and high out-degree, or nodes with high in-degree and low out-degree are uncommon. We compare this version with three other variances of greedy approach: considering only in-degree(follower), considering only out-degree(followee), and considering the sum of in-degree and out-degree (the original baseline greedy). The difference of the result of all these models are negligible.

Greedy3 - Surprisingly, we cannot barely improvement from Greedy2, which means that nodes of these type must be very rare. Further testing showed that the order of nodes with degree greater than C that we explore has a negligible effect on the performance of our algorithms. This result suggests that when users reach a level of popularity (more than C followers or followees), they all yield equal information gain to our crawler. Therefore, regardless of the order, crawling users those are popular enough will yield similar new discovered nodes and edges.

Greedy4 - We see a big improvement which is expected because there are a lot of nodes with degree less than C. Thus, we are cutting a lot of unnecessary cost which resulted in an improved model that is overwhelms all the previous versions of greedy as well as other baselines.

Learned Model

There are many ways to configure the learning model. We configure them through our experiment, trying to optimize the model on forest fire graph.

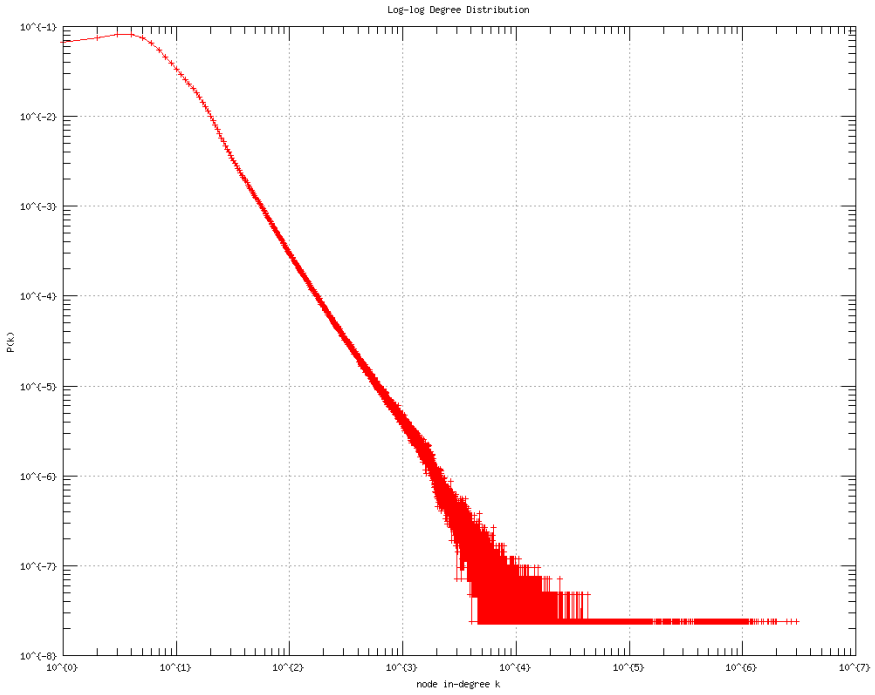
- the number of graphs, the type of graph, and the size of graph used to train. From our experiments, we choose to train the model on forest fire graph because it resembles the real world graph the most. The number of graphs we train the model on does not affect the result much if it is large enough (at least 100). Similarly, the size of the graph that we train on does not matter if it is big enough. However, it should not be too smaller than the graph we crawl on. When we test on 1000000 nodes graph, training on 100 nodes graph is good enough. But training 10 nodes graph will give us the worst crawler comparing to the baseline models.
- the score for the unknown (score for node with characteristics that have never seen before. This is prior). The higher this value, the closer the model will resemble Lottery approach. We use 1.0
- weights between node and edge (used when we compute score for the distribution.) We give nodes and edges equal importance, so both weights are 1.
- keys for characteristics that we use to learn the model. We use the number of followers and followees of the node in both the input graph and the discovered graph founded by the crawler. We also found that bucketizing these numbers does not help.

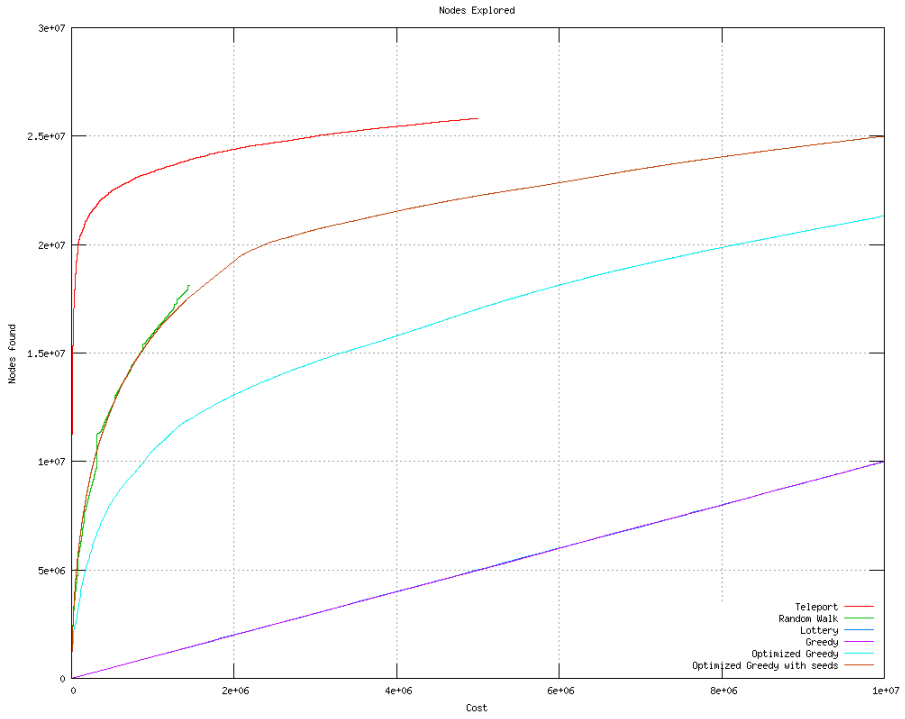
Despite its idea, this model does not perform better than Lottery approach. The node/edge discovered vs. cost curve is almost linear. The gain, newly discovered nodes and edges per cost, is almost at a constant rate. This is expected since the goal of our distribution is to avoid the

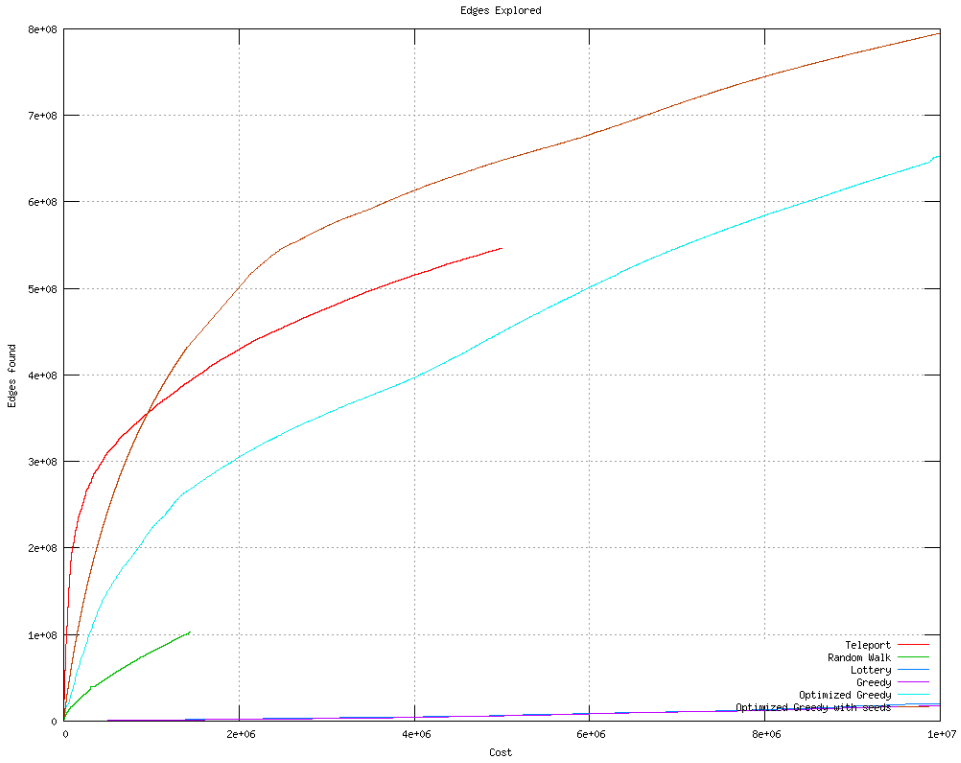
situation that the gain decreases when we have already seen many nodes and edges making it harder to discover new nodes/edges. However, the problem is that the rate is low. In forest fire graph, learned model performance is similar to Lottery approach in the beginning and the end. However, when we test on other random graphs such as Erdos-Renyi Gnp, Gnm graph, the learned model perform better than all the greedy variances due to its randomization. But since these random graphs are very different from real world graph, greedy based model is preferable. From the experiment, we decide that we will not try this on real world twitter data or iterate it. Given that the result is similar to Lottery and the learned model depend heavily on the type of graph that it train on. Forest fire resemble the real world graph to some degree, but it is still different. So, our model trained from forest fire is expected to perform worse on the real world graph.

Real Twitter Data

Real twitter data contains 41,652,230 users (nodes) and 1,468,365,182 social relation (edges). We provide its degree distribution (in-degree, followers) in log-log scale below and it seems to follow power-law. In our graph, when user A follows user B, there is an edge from A to B. Therefore, a node of in-degree k means a user who has k followers.







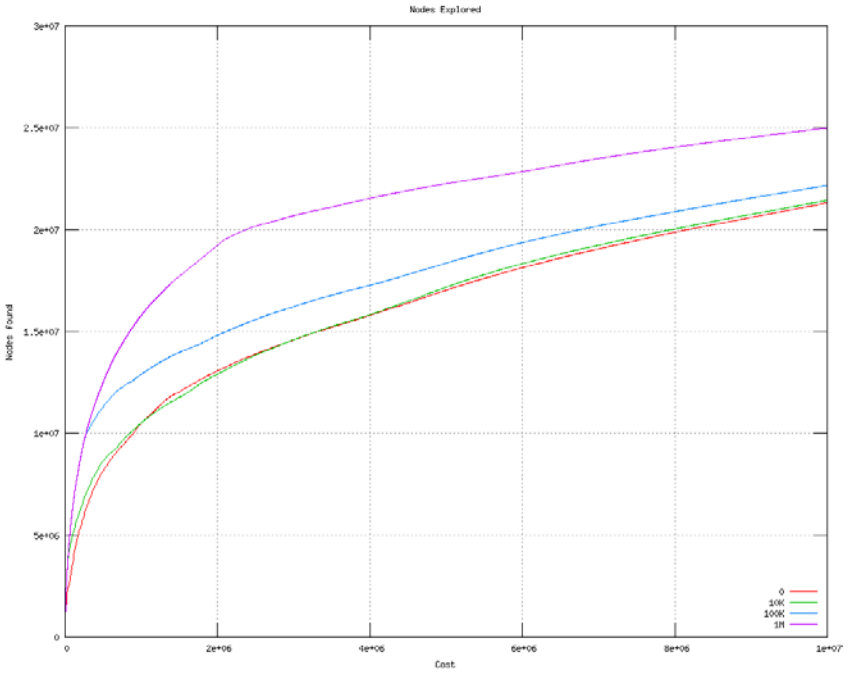
Similar reasoning as in our synthetic data results, greedy and lottery suffers from getting the degree of nodes for their heuristic and thus perform worst in discovering both nodes and edges. Our optimized greedy (Greedy4) improves significantly from greedy. Furthermore, optimized greedy with seeds (we use 1M seeds, explained in the next section) improve significantly over

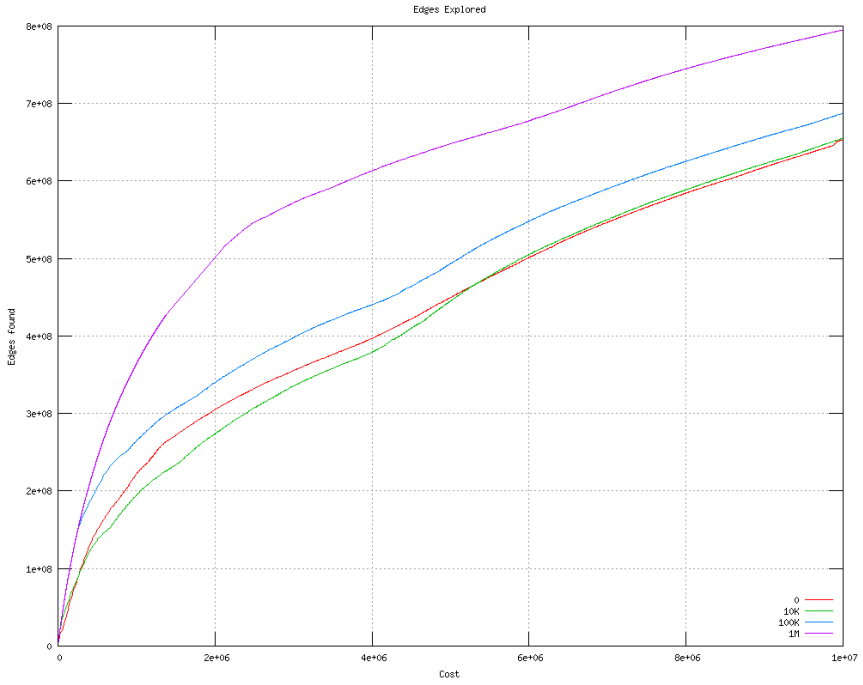
our optimized greedy and performs the best in discovering edges when we have cost greater than 1M. Teleport first discovers nodes and edges quickly and the performance deteriorates quickly (explained in the next section) but still performs the best for discovering new nodes. Although random walk can discover nodes almost as well as optimized greedy with seeds, random walk does not really discover social relations (edges) well at all.

Seeds

We observe that teleport discovers nodes and edges very quickly at the beginning. This is because when it gets a teleport node from the API, the API usually gives more popular users (users with more followers) who tend to be more active (post statuses more often) according to Kwak et al. 2010 [2]. The API's behavior attempts to simulate obtaining users from a real twitter stream. However, as it keeps obtaining more teleport nodes, it will get redundant ones and node and edge efficiency deteriorate. We define node efficiency as the number of marginal discovered nodes per unit cost. We can calculate node efficiency by obtaining the number of newly discovered nodes at a step and divide it by the marginal cost to perform the step. Similarly, edge efficiency is defined as the number of marginal discovered edges per unit cost.

Therefore, we propose adding seeds to current optimized greedy algorithm. First we obtain teleport nodes N times to obtain seeds and then we start crawling using our optimized greedy algorithm. Below we show comparisons of our optimized greedy algorithm when we vary N.





Plots above shown that using 10K seeds is not enough improve number of nodes and edges found. Using 100K seeds improve our results slightly and using 1M seeds improve our results significantly. This is because our twitter data has a lot of nodes so when we use only 10K seeds, it is a very small fraction of total nodes. On the other hand, using 1M seeds (significant fraction of total nodes) helps us discover a significant number of good nodes and thus improve the efficiency. We pick number of seeds related to the number of nodes we expect that the network contains. From 40M nodes in twitter graph, we use 1M seeds.

Adaptive Model

We propose this model as an alternative to adding seeds to to our optimized greedy algorithm. We first crawl nodes using teleport and then once teleport performs below our node efficiency or edge efficiency threshold, we switch to our optimized greedy algorithm. We assures that teleport no longer performs well by detecting 10 executive steps that node or edge efficiency goes below our threshold before switching to optimized greedy. We observe that the slope of our optimized greedy with seeds plots (for both nodes and edges found) are roughly linear. Therefore, we can estimate the slope of each plot and estimate average node and edge efficiency. We found that the average node efficiency is 0.6945 and that the average edge efficiency is 35.6573. This means that on average, our optimized greedy discovers 0.6945 new nodes and 35.6573 new edges per request. Unfortunately, we have not finished running our adaptive crawler, but from our result so far, we expect our adaptive crawler to yield similar performance as our optimized greedy with seeds.

Difficulties

Testing on large networks is slow and the actual Twitter graph contains about 40 million users and 1.47 billion social relations which is much larger than our largest synthetic network. The

twitter graph itself is about 6GB and we plan to extract the graph and save it in binary to speed up our graph loading process. Crawlers take a very long time to run.

Conclusion & Future Work

From our experiment, we found that optimized greedy with seeds seems to be the best model to crawl a real world network. If we focus on discovering nodes only, teleport yields the best performance. However, the network is static in our problem and we only have to maximize the number of nodes or edges we found. Our only evaluation metric is recall. Real world graph is dynamic. It is always changing rapidly. So, nodes and edges that we discovered may become invalid and the precision of our crawler could drop. Therefore, we have to consider precision of our crawler as well. In dynamic crawling, apart from exploring nodes that we haven't explored yet, we need to consider explore the node that we have already seen again to maintain a good precision. Therefore, to crawl a dynamic real world graph, we need to take the refresh rate and age of each node and edge into account and modify our model accordingly.

References

[1] https://dev.twitter.com/docs/api

[2] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" In Proceedings of WWW, 2010.

[3] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou, "Walking in Facebook: A Case Study of Unbiased Sampling of OSNs," in Proc. IEEE INFOCOM, San Diego, CA, 2010.

[4] S. Ye, J. Lang, and F. Wu, "Crawling online social graphs," in Proc. 12th Asia-Pacific Web Conference, Busan, Korea, 2010, pp. 236–242.

[5] W. Chen, Y. Wang, S. Yang, "Efficient Influence Maximization in Social Networks," In Proc. KDD, 2009.

[6] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph Evolution: Densification and Shrinking Diameters," ACM Transactions on Knowledge Discovery from Data, 1(1), 2007.

[7] https://dev.twitter.com/docs/api/1/get/followers/ids