AIM

MAXIMIZING THE INFLUENCE SPREAD IN SOCIAL MEDIA USING IMPROVISED CELF ALGORITHM AND CENTRALITY MEASURES

CSE3024 – SOCIAL AND INFORMATION NETWORKS
FINAL PROJECT REPORT

(J COMPONENT)

SUBMITTED BY:
DEVA S 18BCE2103
THANUSH KUMAR P S 18BEC0548
SARATH A 18BEC0569

Abstract
Influence maximization in social networks mainly depends on selecting the set of influential users as seed set. Previous algorithms for this were greedy approach, hill climbing algorithm and CELF. In this paper we have slightly modified the CELF algorithm and we have used the centrality measures property of the nodes, which increases its efficiency And the propagation models are independent cascade and linear threshold. These algorithms are efficient for propagating the influence according to our algorithm.

Index Terms-Influence Maximization (IM), Influence Propagation, Greedy Algorithms, Sub-modularity, Social Networks.

I. INTRODUCTION

Nowadays, mostly everyone in the world are using the social network in one or the other way. So it is considered as the important media for spreading information, ideas and influence among the individuals. So here comes the concept of influence maximization, because a product developer or an event organizer cannot send the message to each and every node connected in the social network so it is necessary to select some specific users from the network who are able to influence the maximum nodes. These nodes are called influential nodes.

In viral marketing strategy, a company invites some initial users, i.e. the seed nodes, to try its new products or technologies. The company would give these initial users free samples and hope that they will give a positive feedback in social networks. By the power of word-of-mouth, these users may affect their neighbours in a social network. These affected neighbours may subsequently propagate the influence to their own neighbours, and so on. The challenge in viral marketing strategy is how to select the seed nodes to maximize return of investment.

Influence maximization in social networks is the problem of selecting a limited size of influential users as seed nodes, and these seed nodes can propagate the message throughout the network. The propagation of information can be fairly quick if the right seed nodes are selected. However, the large scale of social networks and their complicated structures made it challenging to select the right seed nodes. In this paper, we gave a whole set of solution including improving model, designing a novel seed-node selection algorithm and calculating propagation probability.

Hence our concentration is on selecting the influential nodes and designing the propagation model. The most common algorithm for selecting the seed nodes are greedy and hill climbing algorithm. Several following studies have been carried to improve the efficiency of seed-node selection algorithms. Leskovec, Krause and Guestrin proposed a nearly optimal algorithm called Cost Effective Lazy Forward (CELF) algorithm. In this algorithm, the number of nodes to be considered in each round of seed selection is greatly reduced by exploiting the sub-modularity property of models.

This algorithm scaled well to large data sets and their experiments showed that it was 700 times faster than Hill-Climbing algorithm. Our algorithm is optimized approach based on the CELF(Cost Effective Lazy Forward) algorithm .In order to further improve the naive greedy algorithm for influence maximization in social networks, which exploits the property of submodularity of the spread function for influence propagation models (e.g., Linear Threshold Model and Independent Cascade Model) to avoid unnecessary steps. Sub-modularity says the marginal gain of a new node shrinks as the set grows. Function f is sub-modular iff $f(S \cup \{w\}) - f(S) \geq f(T \cup \{w\}) - f(T)$ whenever $S \subseteq T$. The advantages of this algorithm over existing algorithms are it takes comparatively less time to identify the influential seeds in the social networks.

Since the optimization introduced in CELF is orthogonal to the method used for estimating the spread, our idea can be combined with the heuristic approaches that are based on the greedy algorithm to obtain highly scalable algorithms for influence maximization. Added to these we use the degree centrality and closeness centrality measures of the nodes added to these propagation calculations so that the most influenced seed can be selected.

II. LITERATURE REVIEW

Influence maximization was first proposed as an algorithm problem by Domingos and Matthew at 2002[1] in his study of viral marketing. Here they have formally defined the problem of maximization of influence and proposed a basic greedy algorithm. If a company's investment to a user is (I), e.g. sample or advertisement, and the expected return is (R), i.e. when user purchase product from the company, the profit (P) can be determined as P = R – I. Only when P is positive, will a company deem the user as a valuable customer.Calculation of R is different in direct marking vs. viral marketing.

In direct marketing, each user is independent from other users. A company only considers direct purchase action from a user and the user will decide his purchase action independently, not being affected by others' action or persuasion. Therefore, the most valuable user is the user who will purchase most products from the company in direct marketing. Then IM have been classified into two subcategories, one is selecting the most influential seed node and the other is the propagation models.

Kempe, Kleinberg and Tardos provided two different models Independent Cascade model and Linear Threshold model for propagation. These algorithms simulate the influence propagation in social networks. LT model is based on node-specific threshold. The threshold represents the difficulty of switching an inactive node to an active node. A larger threshold value means a node is less likely to switch its status. Then the Independent Cascade model is a dynamic cascade model based on probability theory.

Kempe proposed the first provable approximization of selecting seed nodes by using Hill-climbing algorithm. This algorithm is based on the theory of sub-modular functions. Then comes the CELF algorithm which solves the inefficiency problem of Hill-climbing. This exploits the property of submodularity of social networks by reducing the candidate nodes. A major limitation of the simple greedy algorithm is twofold: (i) The algorithm requires repeated computes of the spread function for various seed sets. The problem of computing the spread under both IC and LT models is NP-hard

As a result, Monte-Carlo simulations are run for sufficiently many times to obtain an accurate estimate, resulting in very long computation time. (ii) In each iteration, the simple greedy algorithm searches all the nodes in the graph as a potential candidate for next seed node. As a result, this algorithm entails a quadratic number of steps in terms of the number of nodes. Considerable work has been done on tackling the first issue, by using efficient heuristics for estimating the spread to register huge gains on this front. Relatively little work has been done on improving the quadratic nature of the greedy algorithm.

The most notable work is , where sub-modularity is exploited to develop an efficient algorithm called CELF, based on a "lazy-forward" optimization in selecting seeds. The idea is that the marginal gain of a node in the current iteration cannot be better than its marginal gain in the previous iterations. CELF maintains a table hu,Δu(S)i sorted on Δu(S) in decreasing order, where S is the current seed set and Δu(S) is the marginal gain of u w.r.t S. Δu(S) is re-evaluated only for the top node at a time and if needed, the table is resorted. If a node remains at the top, it is picked as the next seed. Leskovec empirically shows that CELF dramatically improves the efficiency of the greedy algorithm.

These greedy algorithms were long running, so they proposed a generic algorithm which is an stochastic optimized algorithm like stimulated annealing. This is done though multi-population competition [7]. Then various algorithms like Expansion based method, spatial based indexes, bound based method and hint based method were introduced which uses the nodes locations as well so that the companies can select the proper seeds[8].IMAX query processing was proposed, where we can distinguish within the users. In this method the social network is represent by a graph.

This uses IC model and it is suitable for target aware influence maximization. Time-sensitive algorithm, Time delay, cost are considered in this paper. Because of the monotonicity and sub modularity of this model, a greedy algorithm with (1 - 1/e) approximation ratio is produced. A learning based approach based on discrete particles warm optimization (LAPSO-IM). Linear threshold and cascade diffusion models are utilized in the approach. This is considered as the tradeoff between quality and efficiency.

Then a greedy algorithm based on local metrics have been proposed to reduce the time complexity of normal greedy algorithm, by creating a mandate vertices set instead of searching the whole vertices set which is done by evaluating the local metrics of each vertex(static and dynamic). Spread-Max consisting of two phases, 1 st where the seed nodes are identified using hierarchical reachability approach and the designated seed nodes spread infection during second phase by random walk.

Many hybrid algorithms have evolved after these like, value greed and mountain climbing algorithm which combines traditional greedy and Hill climbing algorithms which eradicates the inefficiencies of those. In the first stage, the region is numerically accumulating rapidly and is easy to activate through value-greed. In the second stage, Hill Climbing Algorithm is run to activate as many nodes as possible on the basis of the first stage.

A hybrid influence maximization that uses both PB-IM(personal based) and CB-IM(community based) is proposed to solve the micro and macro issues of IM- problem. Two strategies were proposed. The PB-CD strategy is used for influence propagation more exactly in community detection. The G-CELF strategy is best for selection of seeds from multiple community accurately. Spanning graph for maximizing the influence spread in Social Networks, a new approach based on the Independent Cascade Model (ICM) which extracts an acyclic spanning graph from the social network. This approach consists to prevent the feedback by eliminating the cycles during the determination of the seeds. Its motivations, its difference from the classical existing approach has been discussed.

III. PROPOSED WORK

Choice of algorithm

The following Algorithm describes the CELF algorithm.

We use $\sigma(S)$ to denote the spread of seed set S. We maintain a heap Q with nodes corresponding to users in the network G.

The node of Q corresponding to user u stores a tuple of the form hu.mg1,u.prev best,u.mg2,u.flagi. Here u.mg1 = $\Delta u(S)$, the marginal gain of u w.r.t. the current seed set S; u.prev best is the node that has the maximum marginal gain among all the users examined in the current iteration, before user u; u.mg2 = $\Delta u(S \cup \{\text{prev best}\})$, and u.flag is the iteration number when u.mg1 was last updated.

The idea is that if the node u.prev best is picked as a seed in the current iteration, we don't need to recompute the marginal gain of u w.r.t $(S \cup \{\text{prev best}\})$ in the next iteration. It is important to note that in addition to computing $\Delta u(S)$, it is not necessary to compute $\Delta u(S \cup \{\text{prev best}\})$ from scratch.

More precisely, the algorithm can be implemented in an efficient manner such that both $\Delta u(S)$ and $\Delta u(S \cup \{\text{prev best}\})$ are evaluated simultaneously in a single iteration of Monte Carlo simulation (which typically contains 10,000 runs). In that sense, the extra overhead is relatively insignificant compared to the huge runtime gains we can achieve if it works out.

<u>Algorithm: Proposed Algorithm</u>

Input: G,k

Output: seed set S

1: S ← ∅; Q ← ∅; last seed = null; cur best = null.

2: for each u ∈ V do

3: u.mg1 = $\sigma(\{u\})$; u.prev best = cur best; u.mg2 = $\sigma(\{u,\text{cur best}\})$; u.flag = 0.

4: Add u to Q. Update cur best based on mg1.

5: while |S| < k do

6: u = top (root) element in Q.

7: if u.flag == |S| then

8: S ← S ∪ {u};Q ← Q − {u};last seed = u.

9: continue;

10: else if u.prev best == last seed then

11: u.mg1 = u.mg2.

12: else

13: u.mg1 = $\Delta u(S)$; u.prev best = cur best; u.mg2 = $\Delta u(S \cup \{\text{cur best}\})$.

14: u.flag = |S|; Update cur best.

15: Reinsert u into Q and heapify.

In addition to the data structure Q, the algorithm uses the variables S to denote the current seed set, last seed to track the id of last seed user picked by the algorithm, and cur best to track the user having the maximum marginal gain w.r.t. S over all users examined in the current iteration. The algorithm starts by building the heap Q initially (lines 2-4). Then, it continues to select seeds until the budget k is exhausted.

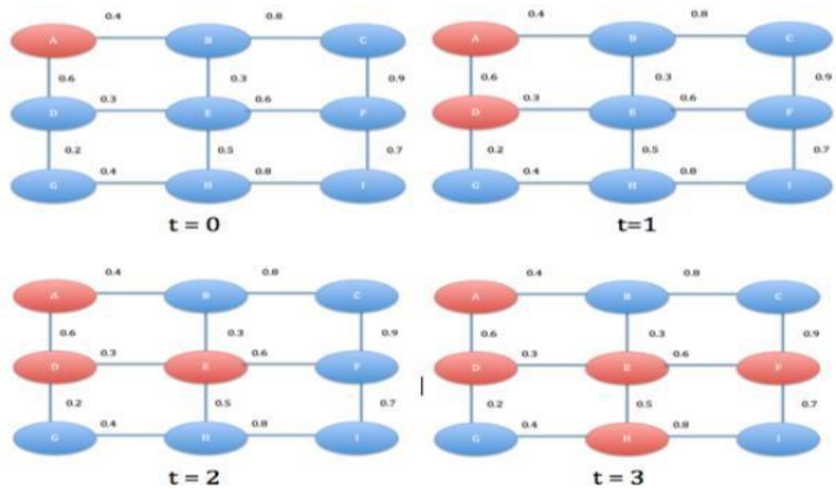
As in CELF, we look at the root element u of Q and if u.flag is equal to the size of the seed set, we pick u as the seed as this indicates that u.mg1 is actually $\Delta u(S)$ (lines 6-9). The optimization of CELF comes from lines 10-11 where we update u.mg1 without recomputing the marginal gain. Clearly, this can be done since u.mg2 has already been computed efficiently w.r.t. the last seed node picked.

If none of the above cases applies, we recompute the marginal gain of u (line 12-13). The propagation model contributes 40% and then the degree centrality measure contribute 20%, closeness centrality measure contribute 40% to the gain of each node.

We have given more weightage to closeness centrality because if the node has higher closeness centrality measure, it means that it can be able to reach many nodes in shorter time period and degree centrality is used because if the node has higher degree it is connected to many nodes.

Propagation models:

IC model: the figure depicts the propagation of the message in IC model where t is time



<u>Algorithm</u>

Input:graph[v][v],seeds[k]

Output:influnce_num

1: influences=seeds

2: queue=influences

3: whilequeueisnotnull:

4: node= queue.pop()

5: foriingraph[node]:

6: random_num=random.random()

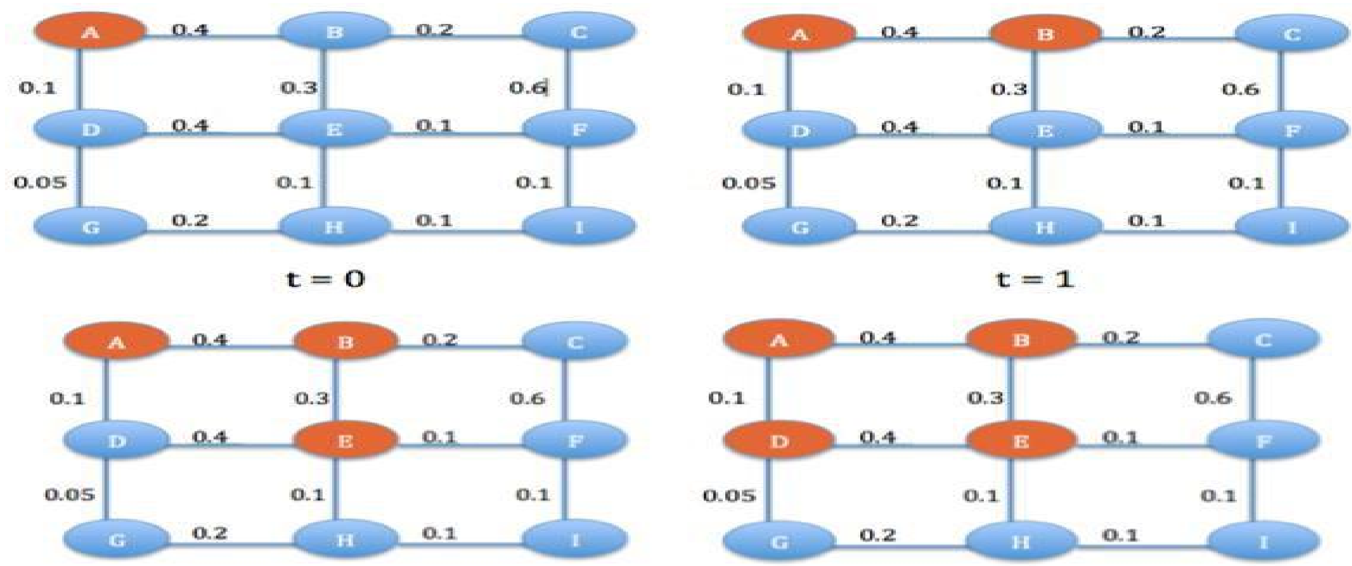
7: ifrandom_num<= graph[node][i]:

8: influences.append(i)

9: influence_num=length(influences)

10:returninfluence_num

LT model: the figure depicts the propagation of the message in IC model where t is time



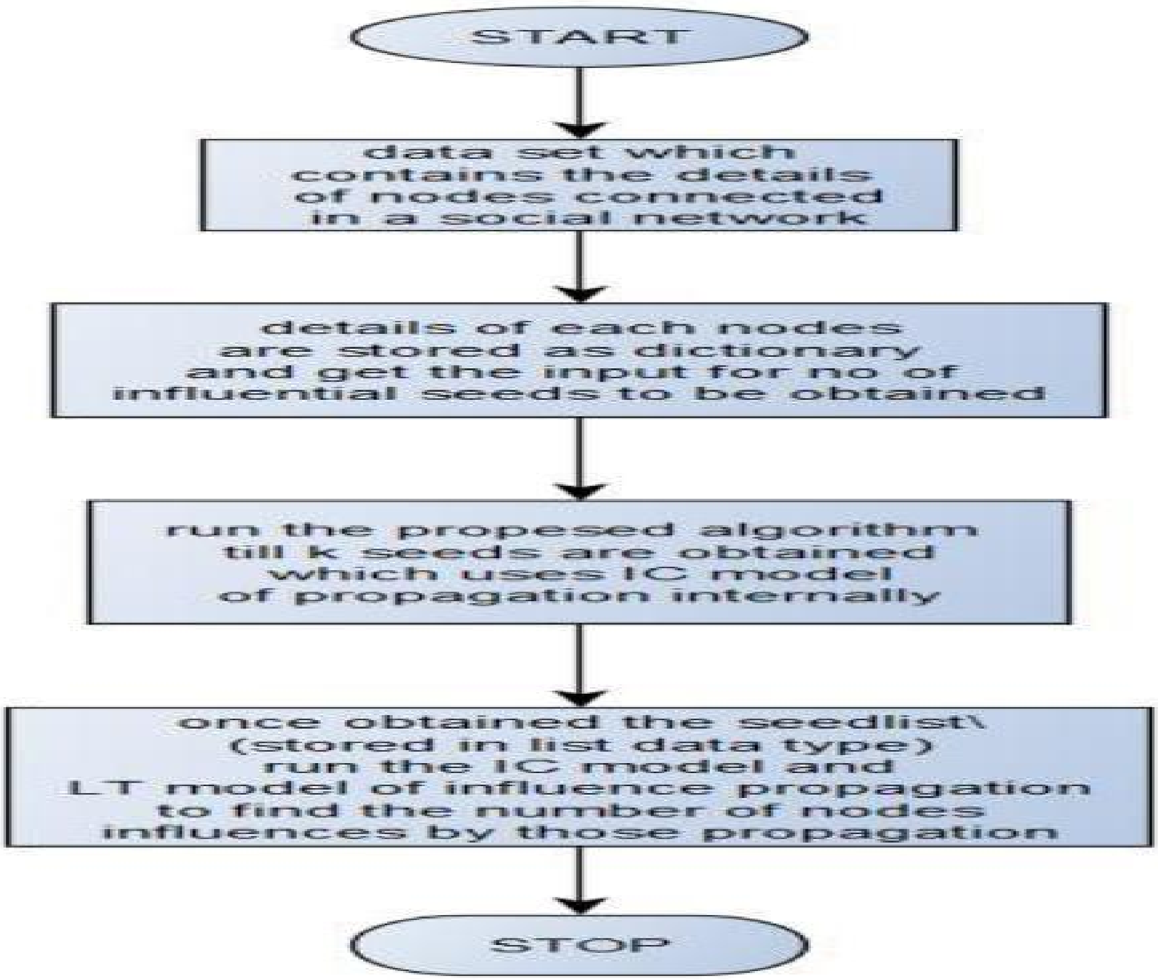
<u>**Algorithm.**</u>

Input:graph[v][v],seeds[k],in_degree[n]

Output:influence_num

```
influence=seeds
queue=influences
whilequeueisnotnull:
node=queue.pop()
foriingraph[node]:
random_num=random.random()
forjinin_degree[n]:
ifjininfluences:
value+=graph[j][i]
ifvalue>random_nun:
influences.append(i)
queue.append(i)
influence_num=length(influences)
returninfluence_num
```

B. .Design diagram



IV. EXPERIMENT RESULTS

Results for the data set with number of vertices as 50 and node probability establishment as 0.7 using Independent cascade model

```
1-graph from txt file
2-random graph
2

enter the number of vertices for Erdős-Rényi graph: 50

enter the probability for node establishment: 0.7
1753

enter the no of seeds required: 10

Propagation models:
IC - INDEPENDENT CASCADE
LT - LINEAR THRESHOLD
enter:   IC
Hence the seeds selected for highest influence are
  [2, 11, 13, 21, 10, 8, 40, 7, 18, 5]
execution time =  0.861919641494751 ms
```

Results for the data set with number of vertices as 100 and node probability establishment as 0.3 using Independent cascade model

```
1-graph from txt file
2-random graph
2

enter the number of vertices for Erdős-Rényi graph: 100

enter the probability for node establishment: 0.3
2961

enter the no of seeds required: 10

Propagation models:
IC - INDEPENDENT CASCADE
LT - LINEAR THRESHOLD
enter:   IC
Hence the seeds selected for highest influence are
  [65, 91, 7, 69, 98, 99, 21, 59, 45, 47]
execution time =  3.1425931453704834
```

Results for the data set with number of vertices as 200 and node probability establishment as 0.2 using Independent cascade model

```
1-graph from txt file
2-random graph
2

enter the number of vertices for Erdős-Rényi graph: 200

enter the probability for node establishment: 0.2
7885

enter the no of seeds required: 10

Propagation models:
IC - INDEPENDENT CASCADE
LT - LINEAR THRESHOLD
enter:   IC
Hence the seeds selected for highest influence are
  [26, 162, 125, 17, 142, 2, 48, 69, 24, 81]
execution time =  16.2159481048584
```

Results for the data set with number of vertices as 300 and node probability establishment as 0.2 using Independent cascade model

```
1-graph from txt file
2-random graph
2

enter the number of vertices for Erdős-Rényi graph: 300

enter the probability for node establishment: 0.2
18001

enter the no of seeds required: 10

Propagation models:
IC - INDEPENDENT CASCADE
LT - LINEAR THRESHOLD
enter:   IC
Hence the seeds selected for highest influence are
  [231, 29, 64, 272, 52, 77, 115, 145, 241, 279]
execution time =  67.34564590454102 ms
```

Results for the data set with number of vertices as 400 and node probability establishment as 0.2 using Independent cascade model

```
1-graph from txt file
2-random graph
2

enter the number of vertices for Erdős-Rényi graph: 400

enter the probability for node establishment: 0.2
31841

enter the no of seeds required: 10

Propagation models:
IC - INDEPENDENT CASCADE
LT - LINEAR THRESHOLD
enter:   IC
Hence the seeds selected for highest influence are
  [115, 337, 103, 152, 92, 25, 38, 211, 14, 328]
execution time =  200.6476447582245 ms
```

V. CONCLUSION

In this paper, we proposed the efficient algorithm for solving the influence maximization problem. We have modified CELF algorithm and used centrality measure property to optimize the seed selection process to find the most influential nodes as seed set. Independent cascade and linear threshold propagation models were used. We have used the enhanced algorithm for different real world datasets and obtained better results. Also the execution time is reduced while maintaining the efficiency. So clearly this algorithm solves the problem of influence maximization. Our algorithm is likely to be the scalable solution to the influence maximization problem for largescale real-life social networks.

REFERENCES

[1] Matthew Richardson and Pedro Domingos. Mining knowledge-sharing sites for viral marketing. In Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, pages 61–70. ACM, 2002.

[2] Kempe, Jon Kleinberg, Tardos KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining August 2003 Pages 137146 https://doi.org/10.1145/956750.956769

[3] Granovetter M: Threshold Models of Coolective Behaviour.The American Journal of Sociology 1978,83(6):23.

[4] Schelling TC: Micromotives and Macrobehaviour. Management science 1978.

[5] W. Yang, L. Brenner and A. Giua, "Influence Maximization in Independent Cascade Networks Based on Activation Probability Computation," in *IEEE Access*, vol. 7, pp. 13745-13757, 2019. doi: 10.1109/ACCESS.2019.2894073

[6] J. Leskovec et al. Cost-effective outbreak detection in networks. In KDD 2007.

[7] Zhang K., Du H., Feldman M.W .Maximizing Influence In A Social Network: Improved Results Using A Genetic Algorithm, Physica A, 478 (2017), Pp. 20-30

[8] Guoliang Li, Shuo Chen, Jianhua Feng, Kian-lee Tan, and Wen-syan Li. 2014. Efficient location-aware influence maximization. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (SIGMOD '14). ACM, New York, NY, USA, 87-98. DOI: https://doi.org/10.1145/2588555.2588561

[9] J. Lee and C. Chung, "A Query Approach for Influence Maximization on Specific Users in Social Networks," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 2, pp. 340-353, 1 Feb. 2015. DOI: 10.1109/TKDE.2014.2330833

[10] M. Hu, Q. Liu, H. Huang and X. Jia, "Time-Sensitive Influence Maximization in Social Networks," *2018 IEEE 18th International Conference on Communication Technology (ICCT)*, Chongqing, 2018,pp.1351-1356. doi: 10.1109/ICCT.2018.8600272

[11] Shashank Sheshar Singh, Ajay Kumar, Kuldeep Singh, Bhaskar Biswas." LAPSO-IM: A learning-based influence maximization approach for social networks ", Department of Computer Science and Engineering, Indian Institute of Technology (BHU), Varanasi, 221–005, India. DOI:https://doi.org/10.1016/j.asoc.2019.105554

[12] L. Yang, S. Ying, S. Jixi, J. Bo and W. Jianjun, "Finding High-Influence Leader Based on Local Metrics," 2013 IEEE 16th International Conference on Computational Science and Engineering, Sydney, NSW, 2013, pp. 468-475.doi: 10.1109/CSE.2013.76

[13] J. Cheriyan and G. P. Sajeev, "SpreadMax: A Scalable Cascading Model for Influence Maximization in Social Networks," 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Bangalore, 2018, pp. 1290-1296.doi: 10.1109/ICACCI.2018.8554863

[14] Y. Lin, X. Zhang, L. Xia, Y. Ren and W. Li, "A Hybrid Algorithm for Influence Maximization of Social Networks," *2019 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, Fukuoka, Japan, 2019, pp. 427-431.doi: 10.1109/DASC/PiCom/CBDCom/CyberSciTech.2019.00087

[15] : Y.Y. KO, K.J. Cho, S.W. Kim Efficient and effective influence maximization in social networks: a hybrid-approach Inf. Sci., 465 (2018), pp. 144-161

[16] Ibrahima Gaye, Gervais Mendy, Samuel Ouya, Diaraf Seck, "Spanning graph for maximizing the influence spread in Social Networks", 2015 IEEE/ACM International Conference. DOI: 10.1145/2808797.2809309

Code:

```
from collections import defaultdict
import random
import time
import networkx as nx
def read_data(graph_file):
    f1 = open(graph_file, 'r')
    first_line = f1.readline().split()
    novert = int(first_line[0])
    noedge = int(first_line[1])
    graph = defaultdict(dict)
    outdeg = defaultdict(int)
    for line in f1.readlines():
        data = line.split()
        outdeg[int(data[0])] += 1
        if float(data[2])>0:
            graph[int(data[0])][int(data[1])] ={'weight': float(data[2])}
        elif float(data[2])<0:
            graph[int(data[0])][int(data[1])] ={'weight': -1*float(data[2])}
    return novert, noedge, graph, outdeg
def ICpropmodel(graph, seeds):
    inf = seeds[:]
    qu = inf[:]
    while len(qu) != 0:
        node = qu.pop(0)
        for element in graph[node]:
            if element not in inf:
                probility = random.random()
                if probility <= graph[node][element]['weight']:
                    inf.append(element)
                    qu.append(elemen
    t) noofinfl = len(inf)
    return noofinfl
def LTpropmodel(graph, seeds):
    inf = seeds[:]
    qu = inf[:]
```

```
        pre_node_record = defaultdict(float)
        threshold = defaultdict(float)
        while len(qu) != 0:
            node = qu.pop(0)
            for element in graph[node]:
                if element not in inf:

                    if threshold[element] == 0:
                        threshold[element] = random.random()

pre_node_record[element] = pre_node_record[element] + graph[node][element]['weight']
                    if pre_node_record[element] >= threshold[element]:
                        inf.append(element)
                        qu.append(elemen
        t) noofinfl = len(inf)
        return noofinfl
    def propic(graph, novert, seed_size, outdeg):
        test_count = 0
        seeds = []
        s_n_influnece = defaultdict(float)
        G=graph
        if G.is_directed():
            s = 1.0 / (len(G) - 1.0)
            degreecentrality = {n: d * s for n, d in G.out_degree()}
            G = G.reverse()
        else:
            s = 1.0 / (len(G) - 1.0)
            degreecentrality = {n: d * s for n, d in G.degree()}
        path_length = nx.single_source_shortest_path_length
        nodes = G.nodes
        closeness_centrality = {}
        for n in nodes:
            sp = dict(path_length(G, n))
            totsp = sum(sp.values())
if totsp > 0.0 and len(G) > 1: closeness_centrality[n] = (len(sp) - 1.0) / totsp
```

```
        s = (len(sp) - 1.0) / (len(G) - 1)
          closeness_centrality[n] *= s
        else:
          closeness_centrality[n] = 0.0
    while len(seeds) < seed_size:
        if len(seeds) == 0:
          for node in range(1, novert + 1):
            s_n_influnece[node] = 0
            if node in outdeg.keys():



              s_n_influnece[node] = (s_n_influnece[node] + ICpropmodel(graph,
seeds+[node]))/novert

              if not closeness_centrality[node]==0:


s_n_influnece[node]=s_n_influnece[node]*0.4+(1/closeness_centrality[node])*0.4+degree
centrality[node]*0.2
              else:

s_n_influnece[node]=s_n_influnece[node]*0.4+(0)*0.2+degreecentrality[node]*0.4
          max_seed = max(s_n_influnece, key=s_n_influnece.get)
          s_n_influnece.pop(max_seed)
          seeds.append(max_see
          d) test_count+=1
        elif len(seeds)!= 0:
          prev_best = max(s_n_influnece, key=s_n_influnece.get)
          s_n_influnece[prev_best] = 0
          marginal_profit = ICpropmodel(graph, seeds + [prev_best]) - ICpropmodel(graph,
seeds)
          s_n_influnece[prev_best] += marginal_profit
          if not closeness_centrality[prev_best]==0:
```

```
s_n_influnece[prev_best]=s_n_influnece[prev_best]*0.4+(1/closeness_centrality[prev_best
])*0.4+degreecentrality[prev_best]*0.2
            else:

s_n_influnece[prev_best]=s_n_influnece[prev_best]*0.4+(0)*0.2+degreecentrality[prev_be
st]*0.4
            current_seed = max(s_n_influnece, key=s_n_influnece.get)
            if current_seed == prev_best:
                seeds.append(current_seed)
                s_n_influnece.pop(current_seed)
            else:
                continu
    e return seeds
def proplt(graph, novert, seed_size, outdeg):
    seeds = []
    s_n_influnece = defaultdict(float)
    G=graph
    if G.is_directed():
        s = 1.0 / (len(G) - 1.0)
        degreecentrality = {n: d * s for n, d in G.outdeg()}
        G = G.reverse()
    else:
        s = 1.0 / (len(G) - 1.0)
        degreecentrality = {n: d * s for n, d in G.degree()}
```

path_length = nx.single_source_shortest_path_length nodes =
G.nodes

```
    closeness_centrality = {}
    for n in nodes:
        sp = dict(path_length(G, n))
        totsp = sum(sp.values())
        if totsp > 0.0 and len(G) > 1:
            closeness_centrality[n] = (len(sp) - 1.0) / totsp
            s = (len(sp) - 1.0) / (len(G) - 1)
            closeness_centrality[n] *= s
        else:
            closeness_centrality[n] = 0.0
    while len(seeds) < seed_size:
        if len(seeds) == 0:
            for node in range(1, novert + 1):
                s_n_influnece[node] = 0
                if node in outdeg:
                    single_node = []
                    single_node.append(node)
                    s_n_influnece[node] =( s_n_influnece[node] + LTpropmodel(graph,
single_node))/novert
                    if not closeness_centrality[node]==0:

s_n_influnece[node]=s_n_influnece[node]*0.4+(1/closeness_centrality[node])*0.4+degree
centrality[node]*0.2
                    else:

s_n_influnece[node]=s_n_influnece[node]*0.4+(0)*0.2+degreecentrality[node]*0.4
            max_seed = max(s_n_influnece, key=s_n_influnece.get)
            s_n_influnece.pop(max_seed)
            seeds.append(max_see
        d) else:
            prev_best = max(s_n_influnece, key=s_n_influnece.get)
            s_n_influnece[prev_best] = 0
            new_seeds = seeds + [prev_best]
            marginal_profit = LTpropmodel(graph, new_seeds) - LTpropmodel(graph, seeds)
            s_n_influnece[prev_best] = s_n_influnece[prev_best] + marginal_profit
            if not closeness_centrality[prev_best]==0:

s_n_influnece[prev_best]=s_n_influnece[prev_best]*0.4+(1/closeness_centrality[prev_best
```

```
])*0.4+degreecentrality[prev_best]*0.2
        else:
```

```
s_n_influnece[prev_best]=s_n_influnece[prev_best]*0.4+(0)*0.2+degreecentrality[prev_be st]*0.4
            current_seed = max(s_n_influnece, key=s_n_influnece.get)
            if current_seed == prev_best:
                seeds.append(current_seed)
                s_n_influnece.pop(current_seed)
            else:
                continu
    e return seeds
def inffind(graph, novert, seed_size, outdeg, model):
    if model == "IC":
        seeds = propic(graph, novert, seed_size, outdeg)
    else:
        seeds = proplt(graph, novert, seed_size, outdeg)
    return seeds
def calculate_average(graph, seeds, model):
    if model == "IC":
        count = 0
        total_influence = 0
        while count < 1000:
            total_influence += ICpropmodel(graph, seeds)
            count += 1
        IC_average = total_influence/count
        average_result = IC_average
    else:
        count = 0
        total_influence = 0
        while count < 1000:
            total_influence += LTpropmodel(graph, seeds)
            count += 1
        LT_average = total_influence / count
        average_result = LT_average
    return average_result
def getseeds(G, novert, seed_size, outdeg, model):
    final_seeds = []
    total_influence = 0
    final_seeds = inffind(G, novert, seed_size, outdeg, model)
    total_influence = calculate_average(G, final_seeds, model)
    print("Hence the seeds selected for highest influence are\n ", final_seeds)
```

```
choice=int(input('1-graph from txt file \n2-random graph \n')) if
choice==1:
    graph_file=input('enter the name/directory of the
    file: ') novert, noedge, graph, outdeg =
    read_data(graph_file) G = nx.DiGraph()
    G.add_nodes_from(graph)
    G.add_edges_from(((u, v, data)for u, nbrs in graph.items()for v, data in nbrs.items()))
elif choice==2:
    novert=int(input('enter the number of vertices for Erdős-Rényi graph: '))
    prob=float(input('enter the probability for node establishment: '))
    G=nx.fast_gnp_random_graph(novert, prob, seed=None, directed=True)
    noedge=nx.number_of_edges(G)
    for (u, v) in G.edges():
        G.edges[u,v]['weight'] =
        random.random()
    outdeg={}
    for i in G.nodes():
        outdeg[i]=G.out_degre
        e(i)
    print(noedge)
seed_size=int(input('enter the no of seeds required: '))
model=input('Propagation models:\nIC - INDEPENDENT CASCADE\nLT - LINEAR
THRESHOLD\nenter:      ')
start_time = time.time()
getseeds(G, novert, seed_size, outdeg, model)
print("execution time = ",time.time() - start_time,"ms")
```