Influence Maximization Problem

YingZhou 11610701
Computer Science and Engineering
SUSTech
11610701@mail.sustc.edu.cn

1. Preliminaries

1.1. Problem Description

The *influence maximization problem* (IMP) normally asks for k finding a small subset of k nodes (referred to as seed set) in a social network G that could maximize the spread of influence which is the expected number of nodes that are influenced by the nodes in the seed set in a cascade manner (ISE). The project includes two parts which are influence speed estimate (ISE) and influence maximization problem (IMP).

1.2. Problem Application

The *influence maxmization problem* (IMP) has been extensively studied recently due to its potential commercial value. A well-known application is the case that a company wants to spread the adoption of a new product from some initially selected adopters through the social links between users.

2. Methodology

2.1. Notation

TABLE 1: Notation

Notation	Description
$G = (V, E)$	a social network G with a node set V and an edge set E
n, m	the numbers of nodes and edges in G, respectively
k	the size of the seed set for influence maximization
R	the set of RR sets generated by IMM's sampling phase
θ	the number of RR sets in R
$F_R(s)$	the faction of RR sets in R that are covered by a node set S

2.2. Data Structure

The main data structures I used contain dictionary, queue, set and heap et. Dictionary is used to store the edges and the weight of each edge since using adjacency matrix is memory wasting if all graphs given are sparse. Queue is applied in multi-processes part for workers to get and put the jobs. Set replacing list records active vertices to save initializing time because set in Python is implemented in hash table. Finally, lazy update optimization with heap implemented in node selection part.

2.3. Model Design

Both ISE and IMP are accelerated with 8 processes. ISE is strictly implemented according to the pseudo code of tutorial lecture. The novel point about IMM is that, in a social network, a influential person starred by many people may not follow same number of people. In that case, the subgraph of converse graph is much smaller than that of original graph. It is simple and straightforward in implementation process as the description of the paper [1] is clear and specific.

2.4. Detail of Algorithm

2.4.1. Influence Spread Estimate. Given the seed set S, the diffusion processes unfolds in discrete rounds as follow: At round 0, all nodes in S are active and the others are inactive. In the subsequent rounds, the newly activated nodes will try to activate their neighbors. The process will stop when no more nodes get activated. There are two fundamental diffusion models are specified in this project including *Independent Cascade* (IC) and *Linear Threshold* (LT) .
Independent Cascade When a node u gets activated, initially or by another node, it has a single chance to activate each inactivate neighbor v with the probability proportional to the edge weight $w(u, v)$.
Linear Threshold At the beginning, each node v selects a random threshold θ_v uniformly at random in range [0,1]. Since the second round, an inactive node v becomes activated if $\sum activated\ neighbors\ uW(u, v) \geq \theta_v$. The weight of the edge (u, v) equals $\frac{1}{d_{in}(v)}$.

Algorithm 1 Independent Cascade

```
function IC
    Initialoze ActivitySet as{SeedSet}
    count ← ActivitySet.length
    while ActivitySet is not empty do
        Set newAcitivitySet as ∅
        for each seed in AcitivitySet do
            for each inactive neighbor in seed do
                Seed tries to activate neighbors with probability
                if activated then
                    Add neighbor into activate to newActicitySet
                end if
            end for
        end for
        count ← count + newActivitySet.length
        ActivitySet ← newActivitySet
    end while
end function
return count
```

Algorithm 2 Linear Threshold

```
function LT
    Initialize ActivitySet as {SeedSet}
    Randomly sample thresholds
    count ← ActivitySet.length
    while ActivitySet is not empty do
        Set newAcitivitySet as ∅
        for each seed in AcitivitySet do
            for each inactive neighbor in seed do
                Calculate the weights of activated neighbors as w_total
                if w_total ≥ neighbor.thresh then
                    Update the state as Active
                    newActivitySet
                end if
                Add neighbor into activate to newActicitySet
            end for
        end for
    end while
end function
```

2.4.2. Influence Maximazation Problem. All the details and proofs about the algorithm I used in IMP are referred Y.Tang's work [1]. I just implemented it and decided which data structures to use to make code run faster.

Algorithm 3 IMM

```
function IMM
    l = l · (1 + log(2)/ log(n))
    R = Sampling (G, k, ϵ, l)
    S_k = NodeSelection (R, k)
end function
```

Algorithm 4 Sampling

```
function SAMPLING(G, k, ϵ, l)
    Initialize a set R = ∅ and an integer LB = 1
    Let ϵ′ = √2 · ϵ
    for i = 1 to log_2 n do
        x = n/2^i
        θ_i = λ′/x, where λ′ is as defined in equation 1
        while ‖R‖ ≤ θ_i do
            Select a node v from G uniformly at random
            Generate an RR set for v, and insert it into R
        end while
        Let S_i = NodeSelection(R)
        if n · F + R(S_i) ≥ (1 + ϵ′) · x then
            LB = n · F_R(S_i)/(1 + ϵ′)
            break
        end if
    end for
    Let θ = λ*/LB, where λ* is as defined as in Equation 2
end function
```

Algorithm 5 NodeSelection

```
function NODESELECTION(R, k)
    Initialize a node set S_k = ∅
    for i = 1 to k do
        Identify the vertex v that maximizes F_R(S_k ∪ v) − F_R
        Insert v into S_k
    end for
end function
```

3. Empirical Verification

3.1. Dataset

TABLE 2: Datasets

Name	n	m	Type
NetHEPT	15.2K	32.2K	undirected
Epinion	39.0M	84.0K	directed
Twitter	81.3K	1.8M	undirected

3.2. Hyperparameters

$$\lambda\prime = \frac{(2+\frac{2}{3}\epsilon\prime)\cdot(\log\binom{n}{k}) + l\cdot\log n + \log\log_2 n)\cdot n}{\epsilon\prime^2} \quad (1)$$

$$\lambda^* = 2n\cdot((1-1/e)\cdot\alpha+\beta)^2\cdot\epsilon^{-2} \quad (2)$$

$$\alpha = \sqrt{l\log n + \log 2} \quad (3)$$

$$\beta = \sqrt{(1-1/e)\cdot(\log\binom{n}{k} + l\log n + log2)} \quad (4)$$

Most parameters are proposed in Y.Tang;s work [1]. The default setting of Algorithm 3 is $\epsilon = 0.1$ and $l = 1$. The maximization algorithm the paper presented provides a $(1 - 1/2 - \epsilon)$-approximate solution with at least $1 - 1/n^l$ probability. The ϵ and l setting is a trade-off between solution quality and time cost, and is set as the same as most people.

3.3. Performance Measure

The performances of accuracy and efficiency are considered. I measure the accuracy of results by comparing time and results with best performance in OJ and the efficiency presented by running in larger instances. The code is written in Python, compiled using Python 3.6.5 :: Anaconda. Only NumPy package is extra imported. Ubuntu 18.04.1, 16 processors, each processor with 2 threads, 32G RAM, Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz.

3.4. Experimental Result

TABLE 3: The performance in NetHEPT instance ($\epsilon = 0.2$)

k	diffusion model	time_avg(s)	ISE	time_OJ	OPT_OJ
5	IC	1.32	324.16	10.04	324.16
	LT	1.04	393.96	2.557	392.98
50	IC	1.37	1291.89	53.98	1298.10
	LT	1.21	1699.58	8.398	1702.00

TABLE 4: The performance in NetHEPT instance ($\epsilon = 0.1$)

k	diffusion model	time_avg(s)	ISE	time_OJ	OPT_OJ
5	IC	3.34	323.31	10.04	324.16
	LT	2.78	393.72	2.557	392.98
50	IC	3.99	1295.20	53.98	1298.10
	LT	2.91	1701.19	8.398	1702.00

TABLE 5: The performance in NetHEPT instance ($\epsilon = 0.05$)

k	diffusion model	time_avg(s)	ISE	time_OJ	OPT_OJ
5	IC	10.04	323.72	10.04	324.16
	LT	7.87	393.72	2.557	392.98
50	IC	12.34	1297.22	53.98	1298.10
	LT	8.52	1702.00	8.398	1702.00

TABLE 6: The performance in larger instances ($\epsilon = 0.1$)

Instance	k	diffusion model	time(s)
Epinion	5	IC	6.25
		LT	4.17
	50	IC	8.75
		LT	6.72
Twitter	5	IC	55.12
		LT	15.41
	50	IC	70.51
		LT	16.70

3.5. Conclusion

According to the experiment results, it proves that my code can get the same quality solutions in less time. To conclude, the algorithm proposed in [1] is creative and efficient. Not only reversing graph to solve *influence maximization problem* is proposed, but also it calculates the lower bounds of sampling size saving computation cost. The obvious disadvantage is that it still needs massive sampling which is memory consuming. This project greatly helps me to understand the extremely important role of data structure and code construction when I completed some tricky idea in my code. If there is a chance, I would like to contribute some works in *infulence maximization* problem.

References

[1] Y. Tang, Y. Shi, and X. Xiao, Influence Maximization in Near-Linear Time, in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD 15, 2015.