Influence Maximization Problem

Hao Zheng 11610127
Department of Computer Science and Engineering
Southern University of Science and Technology
11610127@mail.sustc.edu.cn

1. Preliminaries

This project is implementing a greedy algorithm to solve Influence Maximization Problem (IMP), which belongs to the class of NP-hard problem.

1.1. Problem Description

Influence maximization is the task of finding a set of seed nodes in a social network such that the influence spread of these seed nodes based on certain influence model is maximized, with the number of seeds limited.

1.2. Problem Application

One of the application is advertising strategy, in which each node is a potential customer and each edge is the influence of this person to the others. The k seeds limitation then represent the number of person who will be advertised. In order to find the most valuable set of person to advertise on, IMP algorithm can be applied in this scenario.

1.3. Algorithm

The algorithm used in this project is Cost-Effective Lazy Forward Selection(CELF), which is an improved greedy algorithm. In order to model the spread of influence, which is the σ function, the influence spread estimation is also implemented . As for the detail of algorithms, I will explain them in the next section.

2. Methodology

2.1. Notation

G	the graph
V	all the nodes
E	all the edges
k	the limited number of seeds
S	all the seeds
D	the difference in influence after adding it into seeds set
σ	influence spread estimation function

2.2. Data Structure

- graph: implemented by adjacent list with set
- adj: list, each element represents set of the outer nodes
- reverse_adj: list, each element represents set of the inter nodes

2.3. Model Design

Formally, with the limitation of k seeds, given a graph $G = (V, E)$, a function $\sigma(S, G)$ which is the number of influenced nodes by seeds S in a graph G, IMP need to find the seeds S^* such that $S^* = argmax_{S \in V, |S| \leq k} \sigma(S, G)$.

CELF is a greedy algorithm that make full advantage of submodular property in the σ function. Although it's relatively slower than some state-of-art sketch-based approach, namely PMIA form Microsoft[1], it's convenient to implement and friendly to understand, while having ability to search for a good enough result efficiently in medium data set. Basically, the core of CELF algorithm can be described as following:

1) Constructing a candidate list that hold D. It'll be initialized as the influence of the node itself.
2) When the size of S less than k:
   a. Sort the candidate list according to D.
   c. If the flag of first candidate is computed, add it into seeds set. Ignore next step and continue the loop.
   b. Update D of the first candidate in candidate list. And mark this candidate as a computed candidate.
3) Finally, return S.

2.4. Detail of algorithms

To apply less code while explaining the basic idea clearly, the "pseudocode-likely" python code is used in this section.

Before diving into the details of CELF, it's necessary to talk about Influence Spread Estimation, which is the atomic operation of the whole algorithm. There are two spread model utilized in this project—Independent Cascade (IC) and Linear Threshold (LT). The definition of these two model can be found in [2]. What should be noticed is that

the weight of each line is $\frac{1}{in-degree}$ of the activated node. As the implementation of IC and LT is roughly same, let's take LT as an example.

```
def LT():
    active_set = seeds
    pre_actived = seeds

    threshold = random threshold list
    influenced = len(seeds)
    while len(active_set):
        new_active = set()
        for active_node in active_set:
            for ina_node in inactivated node
around active_node:
                active_neighbours = the parents
of ina_node - pre_actived
                weight = len(active_neighbours)
* edge weight of this inactive node
                if random number > weigth:
                    pre_actived.add(ina_node)
                    new_active.add(ina_node)

        influenced += len(new_active)
        active_set = new_active_set

    return influenced
```

As the set operation is used in most of the time-consuming step in this implementation, it is relatively faster than normal implementation.

As for the CELF implementation, it's originated form [3]. There is two tricks in my implementation. One is that I utilize the multi-processing in the initialization of candidate list, which is really time-consuming in larger data set. The other is that I store the influence of current seeds as last_ise in order to prevent re-computing in computing influence difference step.

```
def celf():
    candidates = the ISE of each node

    seeds = set()
    while len(seeds) < k:
        computed = set()
        while true:
            sort candidates
            node, diff = candidates[0]
            if node in computed:
                seeds.add(node)
                last_ise += diff
                delete candidates[0]
                break
            else:
                candidates[0][1] = compute
D of node
                computed.add(node)

    return seeds
```

The sort operation is considered to be replaced by a priority queue, but the experiment surprisingly shows that it's slower than the sort approach, I guess that the reason is that the change of candidate list only need relatively smaller cost than keep a class of Candidate and priority queue. Thus I reserve the sort approach.

3. Empirical Verification

3.1. Dataset

I use three datasets for verification: network(V=62, E=159), NetHEPT(V=15233, E=58891), and a graph named my_net which is generated by myself which contains 1000 nodes, and 3000 edges.

3.2. Performance Measure

The performance measurement is done like that:

```
from time import perf_counter() as pc

start_time = pc()
run a celf in specific dataset
end_time = pc()

print(end_time - start_time)
```

3.3. Hyperparameters

The most important hyperparameter in my celf implementation is the times of computing ISE and get average, which determines whether I can pass the test in time or not. As my program can safely pass the network, NetHEPT, my_net with the times set with 10000, 1000 and 50, as well as noticing that the time complexity of celf is $O(BV)$ in [3], I fit a function that adjust the ISE times in testing:

$$T(V,E) = \frac{946368.048}{\sqrt{VE}} + 468.4$$

I take E into account because there's not B in this scenario, and the number of edges also affect the computing time of ISE. There's a square root because E is normally proportional to V.

References

[1] Chen, Wei, Chi Wang, and Yajun Wang. "Scalable influence maximization for prevalent viral marketing in large-scale social networks." Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2010.

[2] Kempe, David, Jon Kleinberg, and Éva Tardos. "Maximizing the spread of influence through a social network." Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2003.

[3] Leskovec, Jure, et al. "Cost-effective outbreak detection in networks." Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2007.