Influence Maximization Problem

Project 2 Report of
CS303 Artificial intelligence

Department of Computer Science and Engineering

11812109

阮业淳

Fall 2020

Table of Content

1 Preliminaries

Influence Maximization Problem (IMP) is the problem of finding a small subset of nodes (referred to as seed set) in a social network that could maximize the spread of influence. The influence spread is the expected number of nodes that are influenced by the nodes in the seed set in a cascade manner.

This project has two computational tasks for IMPs. The first is to implement an estimation algorithm for the influence spread and the second is to design and implement a (seed set) search algorithm for IMPs.

1.1 Software & Hardware

This project is written in Python with editor PyCharm. The main testing platform are Windows 10 home Chinese version (version 10.0.19042.608) with Intel® Core™ i7-7700HQ CPU @ 2.80GHz with 4 cores and 8 threads(Python version: 3.8.3), and IMP platform(Python version: 3.6) provided by the course.

1.2 Algorithms

The Linear Threshold and Independent Cascade Models are two of the most basic and widely-studied diffusion models, in this project, they are used separately to solve the influence maximization problem.

Algorithms for solving Influence Maximization Problems include Greedy Hill Climbing, CELF, CELF++, NewGreedy, MixedGreedy, IMM, Degree, Degree Discount. IMM algorithm is used in this project.

1.3 Applications

The application scenarios of Influence Maximization Problem are very rich, including virus marketing, recommendation system, information diffusion, time detection, expert discovery, link prediction, etc.

People are most familiar with virus marketing. If a company wants to promote its own products, it hopes to select a small number of people and let them try the products for free. When the selected users (seed set) are satisfied with the products, they should recommend the products to their friends and colleagues, so that more people can understand and finally buy the products. How to find out these people to try out the products so that the final number of people to buy the products is the most important issue that the company needs to consider.

2 Methodology

A social network can be modeled as a directed graph $G = (V, E)$ with nodes in V modeling the individual in the network and each edge $(u, v) \in E$ is associated with a weight w(u, v) $\in$ [0,1] which indicates the probability that u influences v.

2.1 Notation

- G: a directed graph
- S: a small subset of nodes in a social network, referred to as seed set
- k : a predefined size of the seed set
- $\sigma(S)$: the influence spread of S
- *model*: a predefined stochastic diffusion model

2.2 Data Structure

- *graph* : a three-dimensional Python list containing the neighbor node information (node number and edge weight) of the graph node
- *seed* : a one-dimensional Python list with the serial numbers of seed nodes
- *inactive_flag* : a one-dimensional Python list that marks whether a node is inactive
- $d_{in}(v)$: the in-degree of node v

2.3 Model design

The goal of IMP is to find a seed set S that maximizes $\sigma(S)$, subject to $|S| = k$.

2.3.1 Input and Output

Input: G, k, *model*

Output: A size-k seed set S^* with the maximal $\sigma(S)$ for any size-k seed set $S \subseteq V$

2.3.2 Diffusion Process

Given the seed set S, the diffusion process unfolds in discrete rounds

- 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.
- Once a node becomes active, it will remain active till the end.
- The process stops when no more nodes get activated.

2.3.3 Stochastic Diffusion Models

- Independent Cascade (IC) model

When a node u gets activated, initially or by another node, it has a single chance to activate each inactive neighbor v with the probability proportional to the edge weight w(u, v).

Afterwards, the activated nodes remain its active state but they have no contribution in later activations

The weight of the edge (u, v) is calculated as $w(u, v) = 1/d_{in}(v)$.

- Linear Threshold (LT) model

At the beginning, each node v selects a random threshold θv uniformly at random in range [0,1].

If round $t \geq 1$, an inactive node v becomes activated if $\sum$activated neighbors $uw(u,v) \geq \theta v$

The weight of the edge (u, v) is calculated as $w(u, v) = 1/d_{in}(v)$

2.4 Details of algorithms:

In order to improve the performance, program use multiprocessing to run the iteration, and set a timeout to force quit iteration in case of timeout. The following shows only one program processing part of the algorithm.

2.4.1 Influence Spread Estimation (ISE)

- Repeat IC or LT sampling N times, N=10000
- Take the average of influenced nodes over N times

```
float ISE_one_core(graph, seed, end_time):
    sum = 0
    N = 1000
    for i in range(0, N):
        one_sample_seed = IC(graph, seed) or LT(graph, seed)
        sum += len(one_sample_seed)
        if time.time() > end_time:
            return sum / (i + 1)
    return sum / N
```

```
list IC(graph, seed):
    inactive_flag = [True] * len(graph)
    for s in seed:
        inactive_flag[s] = False
    active_set = seed.copy()
    seed_list = []
    while len(active_set) != 0:
        seed_list.extend(active_set)
        new_active_set = []
        for ac_seed in active_set:
            # graph[ac_seed] = [(neighbor,weight), ...]
            for neig in graph[ac_seed]:
                if inactive_flag[neig[0]]:
                    probability = np.random.random()
                    if probability < neig[1]:
                        inactive_flag[neig[0]] = False
                        new_active_set.append(neig[0])
        active_set = new_active_set
    return seed_list
```

```
list LT(graph, seed):
    active_set = seed.copy()
    w_total = [0.0] * len(graph)
    inactive_flag = [True] * len(graph)
    for s in seed:
        inactive_flag[s] = False
    thresholds = np.random.uniform(0, 1, len(graph))
    thresholds[0] = 1
    for ac in np.where(thresholds == 0)[0]:
        if inactive_flag[ac]:
            inactive_flag[ac] = False
            active_set.append(ac)
    seed_list = []
    while len(active_set) != 0:
        seed_list.extend(active_set)
        new_active_set = []
        for ac_seed in active_set:
            for acc in graph[ac_seed]:
                w_total[acc[0]] += acc[1]
            for neig in graph[ac_seed]:
                if inactive_flag[neig[0]]:
                    if w_total[neig[0]] >= thresholds[neig[0]]:
                        inactive_flag[neig[0]] = False
                        new_active_set.append(neig[0])
        active_set = new_active_set
    return seed_list
```

The graph building from the following function *build_graph(file_name, reverse)*, parameter *file_name* is the path where save the graph information data file. If parameter *reverse* is false, the original graph will be built; otherwise, it will be a reverse graph.

```
list build_graph(file_name, reverse):
    f = open(file_name, 'r')
    graph_raw_str = f.read()
    graph_raw_arr = graph_raw_str.replace('\n', ' ').split(" ")
    it = iter(graph_raw_arr)
    node_count = int(next(it))
    edge_count = int(next(it))
    graph = [[] for _ in range(node_count + 1)]
    for x in range(edge_count):
        i = int(next(it))
        j = int(next(it))
        w = float(next(it))
        if reverse:
            graph[j].append((i, w))
        else:
            graph[i].append((j, w))
    f.close()
    return graph
```

2.4.2 Influence Maximization Problem (IMP)

The following definition and lemma are derived from other papers.

DEFINITION 1 (REVERSE REACHABLE SET [1]). *Let v be a node in V. A reverse reachable (RR) set for v is generated by first sampling a graph g from G, and then taking the set of nodes in g that can reach v. A random RR set is an RR set for a node selected uniformly at random from V .*

LEMMA 1 ([2]). *For any seed set S and any node v, the probability that a diffusion process from S can activate v equals the probability that S overlaps an RR set for v.*

Based on Lemma 1, RRsets should be found as many as possible in a limited time and then solve the seed set. The more RRsets a node is in, the more likely it is to activate more other nodes.

For IC model, RRset can be obtained by selecting a seed point and running function *IC(graph, seed)* on reverse graph.

```
list get_IC_RRsets_one_core(graph, end_time):
    R = []
    while True:
        R.append(IC(graph, random.randint(1, len(graph)-1)))
        if time.time() > end_time:
            return R
```

For IC model, RRset can be obtained by running following function on reverse graph.

```
list get_LT_RRsets_one_core(graph, end_time):
    R = []
    while True:
        RRset = set()
        current_node = seed
        while True:
            RRset.add(current_node)
            neig_count = len(graph[current_node])
            if neig_count == 0:
                break
            node_idx = random.randint(0, neig_count - 1)
            current_node = graph[current_node][node_idx][0]
            if current_node in RRset:
                break
        R.append(list(RRset))
        if time.time() > end_time:
            return R
```

Put the obtained RRsets into list *R*, and get a python dictionary through the following function to mark what RRsets each node is in. This is convenient for later calculation.

```
dict get_node_rrset_dict(R):
    node_rrset_dict = {}
    for i, RRset in enumerate(R):
        for node in RRset:
            if node in node_rrset_dict:
                node_rrset_dict[node][0] += 1
                node_rrset_dict[node][1].append(i)
            else:
                node_rrset_dict[node] = [1, [i]]
    return node_rrset_dict
```

Run the following function to get the IMP result.

```
set get_seed_set(R, node_rrset_dict, k):
    seed_set = set()
    # RRset removed from R
    RRset_flag = [True] * len(R)
    while len(seed_set) < k:
        seed_key, seed_value = max(node_rrset_dict.items(), key =
lambda x: x[1][0])
        seed_set.add(seed_key)
        for node_rrset in seed_value[1]:
            # if not removed
            if RRset_flag[node_rrset]:
                RRset_flag[node_rrset] = False
                for rrset_node in R[node_rrset]:
                    node_rrset_dict[rrset_node][0] -= 1
    return seed_set
```

3 Empirical Verification

This section is about the verification of experiments

3.1 Dataset

For ISE, I used the network-seeds5-IC, network-seeds5-LT, NetHEPT-seeds50-IC, and NetHEPT-seeds50-LT as test dataset; For IMP, I used network-5-IC, network-5-LT, NetHEPT-5-IC, NetHEPT-5-LT, NetHEPT-50-IC, NetHEPT-50-LT, NetHEPT-500-IC, and NetHEPT-500-LT as test dataset. Some of these test data are a combination of NetHEPT, network, and network_seeds provided by the course. Other tests are completed by IMP online platform.

3.2 Performance measure

For ISE, I passed all the test samples of IMP online platform. For IMP, I have tested many times on the local Microsoft Windows operating system, and through the continuous optimization of the algorithm, the influence spread has obtained a better value, which is in the forefront of the ranking of IMP online platform.

3.3 Hyperparameters

For ISE, the number of iterations *N* and calculation exit time *end_time* are the main parameters. Taking 1000 for *N* is a good value. *end_time* takes the current time plus time limit minus 2. In other words, it leaves two seconds for the program to exit to prevent running timeout.

For IMP, the main parameters are *end_time* and the number of RRsets obtained. It needs takes time to select seed nodes. For different size of graph and time limit, different parameter values are assigned. If the time limit is less than 61 seconds, the number of RRsets obtained is 360000, otherwise, it is time limit multiply 4760. If the size of the graph is greater than 30000, the *end_time* is the time limit divided by three and then the current time is added. Otherwise, the *end_time* is the time limit divided by two and the current time is added

3.4 Experimental results

3.4.1 ISE

Dataset	Run Time(s)	Result
network-seeds5-IC	1.75	30.411
network-seeds5-LT	2.16	37.169
NetHEPT-seeds50-IC	5.76	1127.023
NetHEPT-seeds50-LT	8.07	1455.327

In addition, all the IMP online platform random test samples are passed.

3.4.2 IMP

Dataset	Run Time(s)	Result
network-5-IC	11.613	30.7222
network-5-LT	11.630	37.5412
NetHEPT-5-IC	33.529	323.2786
NetHEPT-5-LT	11.674	392.975
NetHEPT-50-IC	63.177	1298.0445
NetHEPT-50-LT	37.625	1701.9953
NetHEPT-500-IC	61.289	4331.4539
NetHEPT-500-LT	55.082	5587.0572

3.5 Conclusion

My program uses multiple processes to find as many RRsets as possible, and achieves good results. Compared with the original version, I have made many detailed optimizations, such as not using shared memory, because it uses locks, which causes child processes to compete for resources and get RRsets very slowly. The relationship between the time limit and the number of RRset looking for is where my algorithm needs to be optimized.

Through this project, I learned about the IC and LT models of IMP problems, also learned some knowledge of python multi-process. I will pay more attention to these programming details in the future to make the program run more efficiently.

References

[1] Y. Tang, Y. Shi, and X. Xiao, “Influence Maximization in Near-Linear Time: A Martingale Approach,” Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 2015, doi: 10.1145/2723372.2723734.

[2] C. Borgs, M. Brautbar, J. Chayes, and B. Lucier. “Maximizing social influence in nearly optimal time,” In SODA, pp. 946-957, 2014.

[3] S. Banerjee, M. Jenamani, and D. K. Pratihar, “A Survey on Influence Maximization in a Social Network,” ArXiv Preprint ArXiv:1808.05502, 2018.