CS303 Project 2: IMP

Jiashu ZHANG 11813121
Department of Computer Science and Engineering
Southern University of Science and Technology
zhangjs2018@mail.sustech.edu.cn

1. Preliminaries

1.1. Problem Description

The goal of this problem is that, for a directed graph, a subset of all vertices is returned by the algorithm to maximize the influence spread from the seed set, given a time upper limit, a memory upper limit and a model of influence spreading.

There are two phases for this problem. The first phase is to decide the measurement of influence. In this phase, two spread models are implemented. The second phase is to calculate the seed set as described above.

The source code of this project can be found in https://github.com/wateryloo/IMP.

1.2. Applications

The application of this problem is mainly about social networking and advertising. Suppose that we are asked to publish an advertisement for a certain product to attract a set of people, but we only have the budget to send this advertisement to a fixed number of them. If people receive this advertisement, they would accept out product and influence people they know so that those would also accept the product and advertise for it.Therefore, we would like to find those with the most ability to recommend the product to people they can influence, like their friends and family. The set of people and their relationships can be regarded as a directed graph, and this problem is to find a subset of all vertices as the seed, to maximize the influence.

1.3. Environment

The hardware and software environment used for this project are as follows. For hardware, I used 2 devices. The first one is a PC on Windows10, with i7-8700 CPU and 32GB memory. The second one is a MacBook Pro 16', with i7-9700 CPU and 16 GB memory. The Python environment on both my devices are Python3.9. I used Jetbrains PyCharm as the IDE for development. I used Git and GitHub Desktop to manage this project across 2 devices.

1.4. Algorithm

In the first phase, IC and LT models are implemented. In the second phase, two algorithms are used. The first one is IMM [1] [2] with some modifications. I was told such modification by a peer, Peiqi Yin. To accelerate the node selection step of IMM and reduce memory consumption, another algorithm called VQGS is used. VQGS is proposed by a research associate professor in my lab. The related paper is not yet published, so the reference is not given. I was told this algorithm by a peer, Zhengxin You, who cooperated with the former in that paper.

2. Methodology

2.1. Notation

Here is a list of all major notations used in this report.

1) ISE: The first phase of IMP, to estimate the influence spread.
2) IMM: The algorithm used to solve IMP problem.
3) VQGS: The algorithm used to improve IMM in node selection.
4) Spread Model: The model used to measure influence spread.
5) IC: Independent Cascading spread model.
6) LT: Linear Threshold spread model.
7) G: The graph of vertices and edges.
8) S: The input seed set of ISE.
9) RR: Reverse reachable set.
10) R: The set of all reverse reachable sets.
11) $F_R(S)$: The ratio of RRs in R that have common elements with set S.
12) S_k: The output seed set of IMM.

2.2. Data Structure

The implementation is in Python, so the data structures here corresponds to data structures in Python. It is worth noticing that, even for the same pseudo-code, the choice of Python data structures significantly influence the performance. Therefore, we do not give a table of variables and data structures. Instead, for each pseudo-code, we give a table of data structures according to our implementation.

TABLE 1. IC DATA STRUCTURE

G	A list of 3-tuples, $(source, dest, weight)$.
S	A set.
$activated$	A set.
$newActivated$	A set.

In general, dictionary, set, list and tuple are used in the implementation.

2.3. Model Design

In the first phase, i.e., ISE, the model is very simple. Given the spread model, IC or LT, the problem is to simply do some calculations to output the estimation of influence. In each round of ISE execution, it outputs the number of vertices activated as the influence. Since both models involves randomization, ISE is executed multiple times for a better estimation. In summary, the first phase can be regarded as a static function. The input of this function is G, seed set and spread model. The output of this function is the estimated influence.

The second phase contains two steps. The first step is the generation of RR, where we randomly select a vertex from the graph, generate its RR, and add the generated RR to R. In basic IMM, this part is bounded by the accuracy of output. In improved version of IMM, this part is bounded by time and memory. The second step is node selection, where we continuously select vertices from the graph so that for each selected vertex v, $F_R(S_i \cup v) - F_R(S_i)$ is maximized. VQGS is used for an implementation with comparable running time as common implementation in Python with large usage of sets, and consuming much less memory.

2.4. Detail of Algorithm

In this part, pseudo-code, and necessary explanations of algorithms are given.

Firstly, the pseudo-code of IC and LT in ISE are provided. To increase the accuracy of estimation, algorithm 1 and algorithm 2 may be run multiple times.

Algorithm 1 IC **Input:**G, S **Output:**$influenceSpread$

```
1: $activated \leftarrow S$
2: $influenceSpread \leftarrow |S|$
3: for all $v \in activated$ do
4:     activate all outgoing neighbors of $v$ in $G$ with random probability
5:     $newActivated \leftarrow$ all newly activated
6:     remove $v$ from $activated$
7:     $activated = activated \cup newActivated$
8:     $influenceSpread += |newActivated|$
9: end for
10: return $influenceSpread$
```

The data structures of algorithm 1 is in table 1.

TABLE 2. LT DATA STRUCTURE

G	A list of 3-tuples, $(source, dest, weight)$.
S	A set.
$activated$	A set.

The principle of algorithm 1 is that, for any edge (u, v) where u is activated but v is not, u has a random probability to activate v. If the probility is smaller than the weight, then v is activated. Otherwise, v is not.

Algorithm 2 LT **Input:**G, S **Output:** $influenceSpread$

```
1: $activated \leftarrow S$
2: for all inactive outgoing neighbor $n$ of $activated$ do
3:     for all $v, v \in activated \ \& \ (v, n) \in G$ do
4:         $sumOfWeights \leftarrow \sum p(v, n)$
5:     end for
6:     if $sumOfWeights > threshold(v)$ then
7:         add $n$ to $activated$
8:     else
9:         Neglect $n$ in future activation
10:    end if
11: end for
12: return $|activated|$
```

The data structures of algorithm 2 is in table 2.

The principle of algorithm 2 is that, for any vertex v that is not activated, if the sum of weights of all incoming edges u, v, where u is activated, is greater than the random threshold of v, then v is activated. Otherwise, v is not. Each vertex can be activated for at most once.

After the illustration of IC and LT in ISE, we move forward to basic IMM. There are 2 steps of basic IMM [2]. The first one is to sample some RR from G as R. This procedure is bounded by constrains generated from input and parameters. The constraints guarantee a $1 - \frac{1}{e} - \epsilon$ accuracy in $1 - \frac{1}{n^l}$ possibility, where n is the number of vertices, and l and ϵ are parameters. The second step is to generate S_k as the output from R, through node selection. The two steps are related to the spread model we choose, because different spread model prefers different seeds.

Algorithm 3 Basic IMM **Input:**$G, k, \epsilon, l, model$ **Output:**S_k

```
1: $constraints \leftarrow generateConstraints(G, k, \epsilon, l)$
2: $R \leftarrow \emptyset$
3: while $constraints$ not satisfied do
4:     $v \leftarrow randomVertex(G)$
5:     $RR \leftarrow GenerateRR(G, v, model)$
6:     add $RR$ to $R$
7: end while
8: $S_k \leftarrow BasicNodeSelection(R, k)$
9: return $S_k$
```

The data structures of algorithm 3 and algorithm 4 is table 3. Through this implementation, we are able to implement the IMM algorithm capable of produce a result

Algorithm 4 Basic Node Selection **Input:** R, k **Output:** S_k

```
1: $S_K \leftarrow \emptyset$
2: $vertexRRSetMap \leftarrow \{v : set(RR, v \in RR))\}$  ▷ The set stores ID of $RR$ that v is in.
3: $vertexFreqMap \leftarrow \{v : Freq\}$  ▷ Freq is the number of $RR$ that $v$ is in.
4: build $vertexRRSetMap$ and $vertexFreqMap$
5: while $|S_k| < k$ do
6:     iterate $vertexFreqMap$ to select vertex $v$ with the largest $freq$
7:     add $v$ to $S_k$
8:     for all $RR \in vertexRRSetMap[v]$ do
9:         for all $v_1 \in RR$ do
10:            $tempSet \leftarrow vertexRRSetmap[v_1]$
11:            remove covered $RR$ from $tempSet$
12:            $vertexRRSetMap[v_1] \leftarrow tempSet$
13:            update $vertexFreqMap[v_1]$
14:        end for
15:    end for
16: end while
17: return $S_k$
```

TABLE 3. Basic IMM Data Structure

G	A list of 3-tuples, $(source, dest, weight)$.
$constraints$	Some predicates calculated in $O(1)time$.
R	A set of sets.
RR	A set.
S_k	A set.
$vertexRRSetMap$	A integer-set map.
$vertexFreqMap$	A integer-integer map.
$tempSet$	A set.

for IMP problem, with accuracy and possibility guarantee, when time and space are not limitations of execution. However, the features of IMM and its implementations also shows possibility for potential improvements. The first to be noticed is that, while $constraints$ generated by parameters ensures accuracy and possibility, it prevents us from easily run this algorithm within a time limit. The termination is determined by $constraints$ only, therefore, we can only modify the parameters to make this algorithm runs longer with better accuracy and possibility, or runs shorter with sacrificed accuracy and possibility. However, the modification of parameters is hard and highly relies on experience. A more flexible solution to IMP is still in need.

One direct solution to this problem is to change $constrains$. While previously, the $constraints$ are generated from parameters, a newer version is introduced by time limit. $timeConstraints$ is satisfied when the time for $generateRR$ is used up, and the algorithm proceeds to next step to compute the result. This improvement does not change the outline of implementation. The only modification is to replace $constaints$ with $timeConstraints$ which is basically a time counter with a time limit parameter. This improvement is introduced by a peer, Peiqi Yin.

However, another problem occurs when we introduce such modification. Sometimes, we are bounded by both time and memory, and before out time is exhausted, our memory is already used up generating RR. This problem is partly caused by the fact that in Python, set is very memory-consuming. A fast experiment shows that, for a 4-integer object, a set consumes 216 bytes, while a list and a tuple consumes 120 bytes and 72 bytes respectively. However, it is not practical to simply replace everything set to list, because set provides very fast set operations, like adding element, removing element, set union and hash-access. While list provides similar API, the efficiency is much lower.

It is easy for us to simply estimate the space of memory R takes. However, while the implementation a simple memory limit on R avoids consuming too much memory, it also undermine the performance, because with such limitation, the number of RR is reduced.

To solve both time and space problem without undermining the performance, we have to firstly know where do time and space go. The time for first step, generation of RR, can be bounded by $timeConstraints$ and is not sensitive to set or list. The time for second step, however, is very sensitive to the choice of set or list. For space, list saves space for both steps, and the second step takes more space than the first step.

Out solution is to use list for both steps, and to solve the efficiency problem in the second step. The second step is to maintaining some statistic properties, so that we can choose the vertex with the most influence on remaining RR. This part is improved by VQGS. The basic idea of VQGS is that, we can use a heap implemented in list with lazy update policy. At the beginning of each round, we re-calculate the true influence of the top vertex, and maintain the heap property. However, we only re-calculate the influence of the children of top only when the true influence of top is smaller than the influence of children before re-calculate, and we do not re-calculate the influence of vertices in the heap other than top and its two children. In this way, most re-calculation of true influence is avoided, and we do not have to go through the list every time to find the maximum, nor do we have to remove covered RR from each value of $vertexRRSetMap$. This improvement is introduced by a peer who takes this course, Zhengxin You. He cooperated with a research associate professor who proposed VQGS. The related papers are not published yet, so it is impossible for us to do citation.

Here, we give the pseudo-code of node selection with VQGS as algorithm 5, and the table of data structures as table 4.

With algorithm 5, memory usage is significantly decreased. In our python implementation, for 2GB memory limit, we can set the size of R to almost 1GB, and the memory is still enough. For more details of this algorithm, please refer to the source code in GitHub.

Algorithm 5 VQGS Node Selection **Input:** R, k **Output:** S_k

```
1: $S_K \leftarrow list(empty)$
2: $covered \leftarrow \emptyset$
3: $vertexRRListMap \leftarrow \{v : list(RR, v \in RR)\}$ ▷ The list stores ID of $RR$ that v is in.
4: $heap \leftarrow list[vertexRRListMap]$ ▷ For each key-value pair, the comparator is length of value.
5: build $vertexRRListMap$
6: $maxHeapify(heap)$
7: while $|S_k| < k$ do
8:     while $True$ do
9:         $calculateTrueInfluence(heap.top, covered)$
10:        if $heap.top < heap.child$ then
11:            replace $heap.top$ with $heap.child$
12:            maintain heap property for $heap.top$
13:        else
14:            break
15:        end if
16:    end while
17:    $v \leftarrow heap.pop()$
18:    add $v$ to $S_k$
19:    add $v$ to $covered$
20: end while
21: return $S_k$
```

TABLE 4. VQGS NODE SELECTION DATA STRUCTURE

R	A list of lists.
RR	A list.
S_k	A list.
$vertexRRListMap$	A integer-list map.
$covered$	A set.
$heap$	A heap implemented with list.

TABLE 5. DATASET

vertex number	edge number
60,000	150,000
100,000	250,000
100,000	1,000,000

3. Empirical Verification

3.1. Dataset

Except for the data that IMP platform supply, I designed a random data generator. The generated data are as table 5.

3.2. Performance Measure

We measured the performance onf Basic IMM and IMM with VQGS. THe metrics includes running time, peak memory consumption, $|R|$ and result of estimation on platform. Our test environment is the PC mentioned above, with Python3.9, an i7-8700 CPU and 32GB memory.

TABLE 6. BASIC IMM PERFORMANCE

Dataset	k	Model	Time secs	$\|R\|$	Memory MBs
rand_100k_250K	500	IC	164	2,756,109	4742.3
rand_100k_250K	500	LT	162	2,646,249	4732.2
rand_100k_250K	100	IC	86	1,961,105	3421.8
rand_100k_250K	100	LT	69	1,606,473	2933.7
NetHEPT	500	IC	15	1,850,469	-
NetHEPT	500	LT	8	967,047	-
NetHEPT	100	IC	8	1,025,382	-
NetHEPT	100	LT	5	658,253	-
NetHEPT	50	IC	5	754,049	-
NetHEPT	50	LT	5	581,081	-

TABLE 7. IMM WITH VQGS PERFORMANCE

Dataset	k	Model	Time secs	$\|R\|$	Memory MBs
rand_100k_250K	500	IC	72	2,031,399	766.2
rand_100k_250K	500	LT	75	3,080,568	1148.4
rand_100k_250K	100	IC	69	2,078,549	817.8
rand_100k_250K	100	LT	72	3,084,016	1171.3

3.3. Hyperparameters

For basic IMM, there are two parameters, l and ϵ. I set $l = 1, \epsilon = 0.1$, which is a common practice.

For IMM with VQGS Node Selection, the only parameter $total$ is the estimated memory consumption of R in bytes. My experiments showed that the peak memory consumption is less than $2 \times total$. However, I am conservative, and I really worry about the condition of the server, so I set $total = 700,000,000$, which is approximately 667.57MB.

3.4. Experimental Results

Compare 6 and 7, we found for large dataset, IMM with VQGS does not have timeout or out-of-memory problem as Basic IMM, and is better in all dimensions. On the platform, my best records for NetHEPT 500 IC and LT are both from IMM with VQGS, which are 4331.6426 and 5587.6117 respectively.

3.5. Conclusion

The experiment shows that, while Basic IMM is able to give a result with $1 - \frac{1}{e} - \epsilon$ accuracy and $1 - \frac{1}{n^l}$ possibility, it is not bounded by time or memory. So when we are given time and memory limit, it is hard for us to exhaust the resource without breaking the limit, because the two parameters, ϵ and l, cannot be easily adjusted. IMM with VQGS simplifies the process of sampling, improved the process of node selection, and replaced some set in Basic IMM implementation with list. As a result, the running time and memory consumption is bounded automatically, and we can adjust some parameters to exhaust the resource we are given, and since generally, more RR is generated, the accuracy and possibility of result are improved as well.

References

[1] Y. Tang, X. Xiao, and Y. Shi, “Influence maximization: Near-optimal time complexity meets practical efficiency,” in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 75–86. [Online]. Available: https://doi.org/10.1145/2588555.2593670

[2] Y. Tang, Y. Shi, and X. Xiao, “Influence maximization in near-linear time: A martingale approach,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1539–1554. [Online]. Available: https://doi.org/10.1145/2723372.2723734