Project 3: Influence Maximizing Problem

Yuejian Mo 11510511
Department of Biology
Southern University of Science and Technology
Email: 11510511@mail.sustc.edu.cn

1. Preliminaries

This project is an implementations of Greedy algorithm to find out a seed set to maximize the spread of influence through a social network, based on LT(Linear Threshold) and IC(Independent Cascade) models. With the development of Internet and smart phone, more and more people are connected by social network. It is interesting to study the influence spread between different members. Ecplicaty, maximizing the spread in network with less afford takes the eyes of adversity provider and government managers. In 2003, Kempe, Kleinberg and Tardos proposal two models to represent and optimize this kind of problem. [1]

This problem, Influence Maximizing Problem(IMP) is a NP-hard. Inspired by above report, I success produce seed set by natural greedy.

1.1. Software

This project is written by Python 3.7 with editor Atom and Vim. Numpy, os, time, random, sys and argparse library are used.

1.2. Algorithm

Using LT and IC model to evaluate the spread influence of seed sets and natural greedy algorithm to search optimal seeds set.

2. Methodology

Firstly, a social network is modeled as a directed graph $G = (V, E)$ and each edge $(u, v) \in E$ is associated with a weight $w(u, w) = \frac{1}{d_{in}}$ which indicates the probability that u influences v. $S \in V$ is the subset of nodes selected to initiate the influence diffusion, which is called seeds set.

Then, I try to evaluate the spread influence with two different diffusion models.

In *Linear Threshold* model, a node v is influenced by each neighbor u according to a $weihtw_{v,u}$ such that $\sum_u w_{v,u} <= 1$. The dynamics is following. Each node v are accessed a threshold $\theta \in [0, 1]$ randomly obeyed uniform distribution at first step; this represents the weighted fraction of v' neighbors that must become active in order for v become active. In step t, each inactive node v will be active by its active neighbors w at step $t - 1$ as following

$$\sum_w w_{u,v} >= \theta_v$$

. Once no new active node is generated, maximal spread influence will obtain with given seeds set in specified graph. I notices final influence as the total number of active node σ.

In *Independent Cascade* model, I start with an initial set of active nodes S, and the process unfolds in discrete steps according to the following randomized rule. When node v first becomes active in step t, it is given a single chance to activate each currently inactive neighbor w; it succeeds with a probability $p_{v,w}$, a parameter of the system, independently of the history thus far. (If w has multiple newly activated neighbors, their attempts are sequenced in an arbitrary order.) If v succeeds, then it cannot make any further attempts to activate w in subsequent rounds. Again, the process runs until no more activation are possible.

2.1. Representation

Some main data are maintain during process: **capacity**,**graph_dm** , **graph_ct**. Others data would be specified inside functions, shortest_dist should noticed.

- **capacity**: The car's capacity, generated from input file.
- **graph_dm**: The graph of edge with demand and their demand, generated from input file.
- **graph_ct**: The graph of edge with cost and their cost, generated from input file.
- **shortest_dist**: A dictionary of shortest distance and pathway between two vertexes.

2.2. Architecture

Here list all functions in given code:

- **generateGraph**: Generate cost and demand graph from input file.
- **dijkstra**: Calculate all vertexes closest distance and pathway away from specify source.
- **genDijkstraDist**: Generate each two vertexes closest distance and pathway from **dijkstra** function.

- **better**: Second rule to choose one vertexes from two vertex.
- **pathScan**: Path-Scanning algorithm to generate serve sequence.
- **s_format**: Standard output function.
- **__name__**: Main control function.

The CARP_solver would be executed in test platform.

2.3. Detail of Algorithm

Here describes some vital functions.

- **generateGraph**: Generate global cost and demand graph from input file.

Algorithm 1 generateGraph

```
Input: $input\_file\_name$
Output:
 1: open $input\_file\_name$ as $file$ {open file and read line by line}
 2: $capacity$ ←file.$readline$
 3: {Read each line information until arrive edge information}
 4: for each edge $e$ do
 5:     split each line string into an array $line$ with 4 elements.
 6:     if ($line$[3] larger than 0) then
 7:         $graph\_dm$[($line$[0], $line$[1])]= $line$[3]
 8:         $graph\_dm[(line[1], line[0])] = line[3]$ {Only add edge to dictionary $graph\_dm$when edge has demand. Both of two direction will be created.}
 9:     end if
10:     $graph\_ct[(line[0], line[1])] = line[2]$
11:     $graph\_ct[(line[1], line[0])] = line[2]$
12: end for
```

- **dijkstra**: generate closest distance and pathway away from source

Algorithm 2 dijkstra

```
Input: $source$
Output: $dist$, $prev$
    create vertex set $Q$
 2: create distance set $dist$
    create path set $prev$
 4: for each vertex $v$ in $graph\_ct$ do
        $dist[v] \leftarrow INFINITY$
 6:     $prev[v] \leftarrow UNDEFINED$
        add $v$ to $Q$
 8: end for
    $dist[source] \leftarrow 0$
10: while ($Q$ is not empty) do
        $u \leftarrow$ vertex in $Q$ with min $dist[u]$
12:     remove $u$ from $Q$
        for for each neighbor $v$ in $u$ do
14:         $alt \leftarrow dist[u] + graph\_ct(u, v)$
            if alt larger than $dist[u]$ then
16:             $dist[v] \leftarrow alt$
                $prev[v] \leftarrow u$
18:         end if
        end for
20: end while
    return $dist$, $prev$
```

- **genDijstraDist**: generate closest distance and pathway between two vertex into a dictionary $shortestDist$

Algorithm 3 genDijstraDist

```
Input:
Output: $shortestDist$
    create dictionary $shortestDist$
    for each edge's first vertex in $graph\_dm$ as $source$ do
 3:     $dist$, $prev$ ← dijkstra($source$)
        for each edge's first vertex in $graph\_dm$ as $target$ do
            $shortestDist[source, target] \leftarrow dist$
 6:     end for
    end for
    return $shortestDist$
```

- **better**: choose better in pointers

Algorithm 4 better

```
Input: $now$, $pre$
Output: $Ture$ or $False$
    return $graph\_dm[now] < graph\_dm[pre]$
```

- **pathScan**: Path Scanning algorithm to determine serve routes

Algorithm 5 pathScan

```
Input: $shortest\_dist$
Output: $R$, $cost$
    create successive routes $R$
    copy required edge from $graph\_dm$ to $free$
    $k, cost \leftarrow 0$
4:  while $free$ is not empty do
        $k \leftarrow k + 1$
        $cost\_k, load\_k \leftarrow 0$
        rest successive routes $R\_k$ each time
8:      serve from origin vertex: $source \leftarrow 1$
        while $free$ in not empty do
            rest shortest distance: $d \leftarrow \infty$
            rest candidate serve edge: $e\_candidate \leftarrow -1$
12:         for each edge $e$ in $free$ do
                if $load\_k + graph\_dm[e] \leq capacity$ then
                    $dist\_now \leftarrow shortest\_dist[source, e.start]$
                    if $dist\_now \leq d$ then
16:                     $d \leftarrow dist\_now$
                        $e\_candidate \leftarrow e$
                    else if $dist\_now = d \cap$ better($e$,$e\_candidate$) then
                        $e\_candidate \leftarrow e$
20:                 end if
                end if
            end for
            if $e = \infty$ then
24:             BREAK
            end if
            add $e\_candidate$ to $R\_k$
            $load\_k = load\_k + graph\_dm[e\_candidate]$
28:         $cost\_k = cost\_k + graph\_ct[e\_candidate]$
            $i = e\_candidate.end$
            remove $e\_candidate$ and its opposite edge from free
        end while
32:     add back home distance: $cost\_k = cost\_k + shortest\_dist[i, 1]$
        add $R\_k$ to $R$
        $cost = cost + cost\_k$
    end while
36: return $R$, $cost$
```

3. Empirical Verification

Empirical verification is confirmed in public test platform. Both right output format and reasonable routes are produced.

3.1. Design

Dijkstra's algorithm return successfully shortest distance and pathway between two vertex. Path-Scanning run correctly. However, Path-Scanning don't provide most optimal routes in spite of the size of test data. Function *better*

3.2. Data and data structure

Dictionaries and lists are used widely rather than matrix. Because the input graph is sparse, dictionary are always used to store graph. Global variable $graph\_dm$ and $graph\_ct$ are dictionary. Lists always store routes and edges information.

3.3. Performance

Following table show different performance with different dataset. Offline test perform at Fedora 29 with Intel® Xeon(R) CPU E5-1680 v3@3.20GHz and 32GiB memory.

Dataset	Run Time(s)	Cost
gdb1	0.65	370
gdb10	0.63	309
val1A	0.68	212
val4A	0.92	450
val7A	4	334
egl-e1-A	0.96	4201
egl-s1-A	5.02	6383

3.4. Result

Solutions are accepted for all data set test. Larger data set require more time.

3.5. Analysis

Because Path-Scanning is kind of greedy algorithm with heuristics, most of solutions are not optimal. The space cost is close to $O(n)$. First time cost $O(n^2)$ is in building Dijkstra's distance, which only run once. Path-Scanning without function $Better$ just cost $O(n)$. Total cost time most vary on function $Better$. For example, if $Better$ just return random result, it will cost less time than comparison method. Although complex and well-design $Better$ cost more time, it's solution is less-cost-routes.

Acknowledgment

Thanks TA Yao Zhao who explain question and provide general method to solve it. And I also thanks for Kebing Sun discuss algorithm and point the output formation error.

References

[1] Ke Tang, Yi Mei, and Xin Yao, "Memetic Algorithm With Extended Neighborhood Search for Capacitated Arc Routing Problems," IEEE Transactions on Evolutionary Computation, vol. 13, no. 5, pp. 1151–1166, Oct. 2009.