**Report for IMP Project**

Zhaowei Zhong

11611722

1. **Preliminaries**

This project has two computational tasks for Influence Maximization Problems (IMPs) in social networks. The first is to implement an estimation algorithm for the influence spread and the second is to design and implement a search algorithm for IMPs.

1. ***Problem Description***

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.

**Social network and influence spread:**

* A social network is modeled as a directed graph G = (V, E) with nodes in V modeling the individual in the network and each edge (u, v) ∈ E is associated with a weight w(u, v) ∈ [0, 1] which indicates the probability that u influences v;
* Let S ⊆ V to be the subset of nodes selected to initiate the influence diffusion, which is called the seed set;
* The stochastic diffusion models specify the random process of influence cascade from S, of which the output is a random set of nodes influenced by S. The expected number of influenced nodes σ(S) is the influence spread of S.

**Goal of IMP:**

* To find a seed set S that maximizes σ(S), subject to |S| = k.

**Inputs:**

* An directed graph G = (V, E);
* A predefined seed set carnality k;
* A predefined stochastic diffusion model.

1. ***Problem Applications***

Viral marketing is one of the applications of Influence Maximization Problem, is to get a small number of users to adopt a product, which subsequently triggers a large cascade of further adoptions by utilizing Word-of-Mouth effect in social networks. Time plays an important role in the influence spread from one user to another and the time needed for a user to influence another varies.

1. **Methodology**
2. ***Notation***

* *G*: A directed graph.
* *σ(S)*: the influence spread of *S*
* *k*: A predefined seed set carnality.

1. ***Data Structure***

* Graph: A python class, used for storing the nodes of the graph and the relations between them, which contains an integer named nodes\_num, and three lists including weight, parents\_map, and children\_map;
* result: A python list, used for storing the results.

1. ***Model Design***
2. ISE
3. Read the raw data and configuration and store them.
4. Using *Independent Cascade (IC)* and *Linear Threshold (LT)* method to solve the corresponding models. In order to improve the performance, I use multiprocessing to run the iteration, and set a timeout to force quit iteration in case of timeout.
5. Get the results of each process, and then calculate and return the average result.
6. IMP
7. Read the raw data and configuration and store them.
8. Iterate through each node, and calculate the total weight of the children of that node, and then save the total weight into a list.
9. Compare the weights in the list, and find the largest one. Return the result.
10. ***Detail of Algorithms***
11. **Common Algorithms and Designs**
    1. Graph Structure: A python class, used for storing the nodes of the graph and the relations between them, which contains an integer named nodes\_num, and three lists including weight, parents\_map, and children\_map

class Graph:

'''

Graph Class

'''

nodes\_num = None

weight = None

parents\_map = None

children\_map = None

def \_\_init\_\_(self, nodes\_num, weight, parents\_map, children\_map):

self.nodes\_num = nodes\_num

self.weight = weight

self.parents\_map = parents\_map

self.children\_map = children\_map

def get\_weight(self, src\_node, dst\_node):

return self.weight[src\_node][dst\_node]

def get\_children(self, node):

return self.children\_map[node]

def get\_parents(self, node):

return self.parents\_map[node]

* 1. Load raw data from the given file into the program, and process them to the structured data:

def load\_graph(file\_name):

'''

Load Graph data

'''

graph\_file = open(file\_name, 'r')

lines = graph\_file.readlines()

graph\_file.close()

nodes\_num = int(str.split(lines[0])[0])

weight = np.zeros((nodes\_num + 1, nodes\_num + 1), dtype=np.float)

parents\_map = [[] for i in range(nodes\_num + 1)]

children\_map = [[] for i in range(nodes\_num + 1)]

for line in lines[1:]:

data = str.split(line)

src\_node = int(data[0])

dst\_node = int(data[1])

weight[src\_node][dst\_node] = float(data[2])

parents\_map[dst\_node].append(src\_node)

children\_map[src\_node].append(dst\_node)

return Graph(nodes\_num, weight, parents\_map, children\_map)

1. **ISE**
   1. Independent Cascade (IC):

def solve\_IC(time\_budget):

model\_start\_time = time.time()

graph = global\_graph

seeds = global\_seeds

result, count = 0, 0

while time.time() - model\_start\_time < time\_budget - 3:

activity\_set = seeds

res\_count = len(activity\_set)

activited = [False] \* (graph.nodes\_num + 1)

for node in activity\_set:

activited[node] = True

while len(activity\_set) != 0:

new\_activity\_set = []

for seed in activity\_set:

children = graph.get\_children(seed)

for child in children:

if child not in activity\_set and child not in new\_activity\_set and not activited[child]:

random.seed(int(os.getpid() + time.time() \* 1e5))

rand = random.random()

if rand <= graph.get\_weight(seed, child):

new\_activity\_set.append(child)

activited[child] = True

res\_count += len(new\_activity\_set)

activity\_set = new\_activity\_set

result += res\_count

count += 1

return result, count

* 1. Linear Threshold (LT):

def solve\_LT(time\_budget):

model\_start\_time = time.time()

graph = global\_graph

seeds = global\_seeds

result, count = 0, 0

while time.time() - model\_start\_time < time\_budget - 3:

activity\_set = seeds

threshold = np.zeros(graph.nodes\_num + 1, dtype=np.float)

activited = [False] \* (graph.nodes\_num + 1)

for i in range(1, graph.nodes\_num + 1):

random.seed(int(os.getpid() + time.time() \* 1e5))

threshold[i] = random.random()

if threshold[i] == 0.0:

activity\_set.append(i)

res\_count = len(activity\_set)

while activity\_set:

new\_activity\_set = []

for seed in activity\_set:

activited[seed] = True

children = graph.get\_children(seed)

for child in children:

threshold[child] -= graph.get\_weight(seed, child)

for seed in activity\_set:

children = graph.get\_children(seed)

for child in children:

if not activited[child]:

if threshold[child] < 0:

new\_activity\_set.append(child)

activited[child] = True

res\_count += len(new\_activity\_set)

activity\_set = new\_activity\_set

result += res\_count

count += 1

return result, count

1. **IMP**

Iterate through each node, and calculate the total weight of the children of that node, and then save the total weight into a list.

def solve(graph, count, time\_limit):

weights = np.zeros(graph.nodes\_num + 1, dtype=np.float)

for node in range(1, graph.nodes\_num + 1):

time\_now = time.time()

total\_time = time\_now - start\_time

if time\_limit - total\_time <= 3:

break

children = graph.get\_children(node)

for child in children:

weights[node] = weights[node] + graph.get\_weight(node, child)

result = list(map(list(weights).index, heapq.nlargest(count, weights)))

for i in range(count):

print(result[i])

1. **Empirical Verification**
2. ***Dataset***

I use the network-seeds5-LT, network-seeds5-IC, NetHEPT-seeds50-LT, and NetHEPT-seeds50-IC to test my ISE program; and network-5-IC, network-5-LT, NetHEPT-5-IC, NetHEPT-5-LT, NetHEPT-50-IC, and NetHEPT-50-LT to test my IMP program.

1. ***Performance Measure***

At the beginning of main function, I put a start\_time = time.time() at there, and a run\_time = (time.time() - start\_time) at the last of main function. So that I can get the total running time of my program. My environment is macOS 10.15.1 with 2.9 GHz Intel Core i7-7820HQ, 16GB RAM. Python version is 3.7.5 64-bit.

1. ***Hyperparameters***

In ISE, the number of multiprocessing workers have an effect on the results. After my testing, I finally chose 8 workers, which can make the performance more effective than others. In IMP, there’s no hyperparameter or parameter which can affect my results.

1. ***Experimental Results***
2. **ISE**

|  |  |  |
| --- | --- | --- |
| **Dataset** | **Run Time** | **Result** |
| network-seeds5-LT | 58.65 | 37.010956067698956 |
| network-seeds5-IC | 58.48 | 30.446367723250958 |
| NetHEPT-seeds50-LT | 118.66 | 1456.9446532285617 |
| NetHEPT-seeds50-IC | 118.60 | 1127.61359311972 |

1. **IMP**

|  |  |  |
| --- | --- | --- |
| **Dataset** | **Run Time** | **Result** |
| network-5-IC | 1.54 | 26.3018 |
| network-5-LT | 1.45 | 31.564 |
| NetHEPT-5-IC | 1.64 | 246.3897 |
| NetHEPT-5-LT | 1.59 | 294.3795 |
| NetHEPT-50-IC | 1.53 | 1063.7519 |
| NetHEPT-50-LT | 1.68 | 1271.8607 |

1. ***Conclusion***

After the tests on my computer and the final tests on the online platform, the results finally met my expectation about the program. As my aspect, the only problem that I may improve is the IMP algorithm. In the IMP algorithm, it could calculate much deeper in the children tree, and this could make the final results much more precise.

1. **References**

[1] W. Chen, Y. Wang, and S. Yang, Efficient influence maximization in social networks, in KDD 2009.

[2] Amit Goyal, Wei Lu, Laks V. S. Lakshmanan, SIMPATH: An Efficient Algorithm for Influence Maximization under the Linear Threshold Model in IEEE 2011.