Report for IMP Project

Zhaowei Zhong
11611722

I. 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.

A. 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 $(u, v) \in E$ is associated with a weight $w(u, v) \in [0, 1]$ which indicates the probability that u influences v;
- Let $S \subseteq V$ 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 $\sigma(S)$ is the influence spread of S.

Goal of IMP:

- To find a seed set S that maximizes $\sigma(S)$, subject to $|S| = k$.

Inputs:

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

B. 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.

II. Methodology

A. Notation

- G: A directed graph.
- $\sigma(S)$: the influence spread of S
- k: A predefined seed set carnality.

B. 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.

C. Model Design

1. ISE
    1) Read the raw data and configuration and store them.
    2) 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.
    3) Get the results of each process, and then calculate and return the average result.
2. IMP
    1) Read the raw data and configuration and store them.
    2) Iterate through each node, and calculate the total weight of the children of that node, and then save the total weight into a list.
    3) Compare the weights in the list, and find the largest one. Return the result.

D. Detail of Algorithms

1. 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]
```

2) 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)
```

2. 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
```

2) 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
```

3. 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])
```

III. Empirical Verification

A. 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.

B. 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.

C. 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.

D. Experimental Results

1. 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

2. 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

E. 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.

IV. 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.