Lab 7: Friend Recommendations (Breadth-First Search)

CS 3851 Algorithms

Learning Outcomes

- Describe how graph and tree structures are implemented
- List at least three engineering applications of algorithmic design and analysis.

Overview

Social media platforms like Twitter, Facebook, and LinkedIn operate on the concept of users and relationships between them. For Twitter, user relationships are asymmetric; one user can be a "follower" of another user, but it does not mean the opposite is true. For Facebook and LinkedIn, user relationships are symmetric; both users have to agree to be "friends" or connections. By following other users, a user subscribes to the content involving those users and that content is used to populate their feed.

Social media platforms make money by showing ads to users. To maximize ad impressions (and thus profit), social media platforms need to encourage users to spend as much time as possible on the platform. Users tend to be more satisfied and engaged when they are following other users who create content of interest.

With hundreds of millions or even billions of users, it's not feasible for users to simply "browse" to find other users to follow. Social media platforms employ recommendation systems that identify a small set of other users that a user may want to follow. These recommendations are based on factors such as following the same people and posting and viewing similar content. To be effective, the recommendation list needs to be small but highly accurate.

The most basic recommendation systems are based on user similarity. They compare every user with every other user, calculate a similarity score, and then output a list of the k most similar users for each user. These approaches require O(N) similarity calculations for each user or $O(N^2)$ calculations if generating recommendations for all users.

For real-world systems with billions of users, this approach is too slow. Real systems tend to employ multiple heuristics to reduce the number of comparisons. One approach is to employ the idea of "social distance." It is assumed that all pairs of people are linked by a chain of five or fewer intermediate friends (six degrees of separation). If users and their connections are stored as a graph, we can create a list of all friends within some d-degrees of separation by using a modified breath-first search algorithm. This avoids searching through all users when generating recommendations for a single user. This approach will still generate a lot of false positives, but a more computationally costly and sophisticated algorithm can then be used to perform further filtering.

In this lab, you will implement this modified breath-first search. You will then evaluate the accuracy of the predictions versus the run time for various degrees of separation.

1. Add `edge_set()` method to `DiGraph`

First, add a `set edge_set(self)` method to your `DiGraph` class. This method will loop through all of the edges, create a 2-tuple of the vertices involved in each edge for each edge, and add the tuples to a set. The set will then be returned.

2. Implement Breadth-First Search

You have been provided `graphs_stub.py` with stubs for the functions you must implement and `test_graphs.py`, which contains unit tests designed to help you check the correctness of your implementation. Copy your `DiGraph` class into `graphs_stubs.py` and rename it to `graphs.py`.

The stub file contains several useful classes and functions:

- `Vertex` – a class used to store the per-vertex information needed for breadth-first search as well as a unique name for reach vertex
- `precision` – a function used to calculate precision of friend recommendations (see below)
- `recall` – a function used to calculate recall of friend recommendations (see below)
- `load_data` – a function to load the training and testing set data.
- `bfs` – a function implementing breadth-first search
- `recommend_friends_for_user` – a function implementing a modified breath-first search (see below)
- `recommend_all_friends` – a function for make recommendations for all users using `recommend_friends_for_user` (see below)

In `graphs.py`, implement the `void bfs(G, s)` function. The method takes a DiGraph object (G) and a source vertex s.

You have been provided a Vertex class to help. The Vertex class stores the predecessor (pi), depth (d), color (color), and unique identifier (name) for each vertex. Vertex objects can be used with the DiGraph class (they are hashable).

Hint: The Python standard library contains a `deque` class that implements a double-ended queue. The methods `deque.append()` can be used to add to an item to the end of the queue, `deque.popleft()` can be used to pull an item off of the front of the queue, and `len(deque)` can be used to tell if the queue is empty.

2. Implement a Friend Recommendation Algorithm

Once you have the `bfs` function working, you should implement two more functions: `List<Vertex> recommend_friends_for_user(G, u, max_depth)` and `DiGraph recommend_all_friends(G, max_depth)`. The `recommend_friends_for_user` method takes a `DiGraph` object G, a source user u, and a max depth (`max_depth`). The `recommend_friends_for_user` performs a breadth-first search with two modifications:

1. The function should return a list of all vertices encountered during the breath-first search

2. The function should ignore all vertices with distance d from the source are greater than `max_depth`.

The `recommend_all_friends` function uses the `recommend_friends_for_user` function to recommend friends for every user in the graph G. The method records the recommendations in a new graph and returns it. The pseudocode is given below.

```
1.  RECOMMEND-ALL-FRIENDS(G, max_depth):
2.      H = DiGraph()
3.      for each vertex u in G:
4.        // find all potential friends
5.        targets = recommend_friends_for_user(G, max_depth)
6.        // remove existing friends
7.        for v in targets:
8.          if edge (v, u) not in G:
9.            // need to define edges in both directions
10.           // for an undirected graph
11.           H.add_edge(u, v)
12.           H.add_edge(v, u)
```

3. Evaluate the Algorithm

You will evaluate the algorithm on an anonymized data set of Facebook friends. The data is divided into two files: `training_set.tsv` and `testing_set.tsv`. Each file lists one pair of users per line. Each user is identified by a unique number. In a Jupyter notebook, you will need to:

1. Load the data sets using the provided function.

2. **Evaluate how the number of recommendations varies with the max depth.** Run `recommend_all_friends` on the training set with max depths from 1 to 10. The number of recommendations will be the number of edges in the resulting graph. Plot the number of edges in the resulting graph (y-axis) vs the max depth (x-axis).

3. **Evaluate the run time of `recommend_all_friends`.** Run `recommend_all_friends` on the training set with max depths from 1 to 10. Plot the run-time (y-axis) versus the max depth (x-axis).

4. **Evaluate the accuracy of the predictions.** Run `recommend_all_friends` on the training set with max depths from 1 to 10. You will use two metrics to evaluate the predictions: precision and recall. Precision indicates how many of the recommend edges are in the testing set, while recall indicates how many of the edges in the testing set are in the recommendations. You can calculate these using the provided functions.

$$Precision = \frac{number\ of\ recommended\ edges\ in\ test\ set}{number\ of\ recommend\ edges}$$

$$Recall = \frac{number\ of\ test\ set\ edges\ in\ recommended\ set}{number\ of\ test\ set\ edges}$$

You should make one plot for each metric. The max depth should be on the x-axis. The metric value should be on the y-axis.

Note: Do not be concerned if your precision is very low (<5%) and your recall is high (> 95%). This is expected.

4. Reflection

Use your lab to results to answer the following questions:

1. Does the number of recommendations plateau after a certain max depth? If so, what is the max_depth? Why do you think this happens?

2. Does precision increase or decrease as the max depth increases? Why do you think this is?

3. Does the recall increase or decrease as the max depth increases? Why do you think this is?

4. Our stated goal is to generate an initial set of recommendations which we can further filter with another algorithm. If we needed to compare all users, we would have 784 * 786 = 614,656 pairs to look at. What value would you choose for the max_depth to achieve this goal based on your analyses above?

5. Why is breadth-first search more appropriate than depth-first search for this problem?

6. Why did we need to modify the breath-first search to limit the depth?

Submission Instructions

Add your `graphs.py` file and the HTML output of your notebook to a **zip** file. Upload the zip file to Blackboard

Rubric

Followed submission instructions	10%
Passes tests	30%
Correctly ran the three analyses	30%
Plots look reasonable	10%
Answers to Reflection Questions	20%