NETS 150: Homework 5, Part 2 - Project

Trevin Gandhi, Deepan Saravanan, Seth Bartynski

Project Description:

Our goal for this project was to develop a graph toolkit and recommendation engine. The underlying core of our project was based on a graph representation that consisted of a Graph class wrapped around Node and Edge instances. This representation could be built from reading in .txt files, as can be seen in the DataReader class. For the rest of the project, we focused on implementing two external APIs.

The first API was the GraphToolkit class. This toolkit contains a variety of graph algorithms that an end-user may want to run. The full list is: BFS, DFS, Topological Sort, Kosaraju's, Bellman-Ford Single Source Shortest Path, Floyd-Warshall All Pairs Shortest Path, Ford-Fulkerson Max Flow, Brandes' Betweenness Centrality, and Dampened PageRank (using linear algebra).

The second API was the Recommender class. This class operates on bipartite graphs in which one set of nodes represents people and the other represents items of some sort. The edges are directed from people to items and the edge weights typically represent the person's rating of that item. Our Recommender class uses a collaboration filter**,** which uses similarity metrics to find the most similar users to the input user (which we will call i). The two similarity metrics we implemented are the Pearson Correlation Coefficient and the Jaccard Similarity Coefficient. The Pearson Correlation Coefficient is a measure that takes two users and returns their covariance divided by the product of their standard deviations. This value will always be in the range [-1, 1], where 1 implies that there is complete correlation between the two users, while -1 implies there is a complete negative correlation. For the Jaccard Similarity Coefficient, let us define A to be the set of items user 1 has rated and B to be the set of items user 2 has rated. The Jaccard Similarity Coefficient then is $\frac{A \cap B}{A \cup B}$.

Once we had a set of scores for all the other users, we sorted and chose the top k of those, where k was a parameter to the function. Our previous metric summed up the values for the items that were adjacent to the k most similar users and returned those items with the highest score. However, this was not a fair representation of the rating system, and was thus inaccurate. For example, if all k users give a movie a rating of 2 out of 5, then that movie will have score 2k, a relatively high score, even though it was not really recommended by the similar users. Thus, we changed the metric to be the total weight of the ratings divided by the square root of the number of users who rated that item. This metric is more robust, and gives more of a weighted average. It also favors items that were rated by multiple users, to avoid the problem of a single 5 out of 5 rating being the best recommendation returned.

At the time of this submission, our final project will be made public on GitHub here:

https://github.com/sonicxml/RecommendationEngine, and we also have a user-facing website here: http://sonicxml.github.io/RecommendationEngine/ that includes a small writeup and links to our source and JavaDocs.

Our data can be found both in our submission and here: http://grouplens.org/datasets/movielens/

Analysis of Similarity Metrics:

We tested the accuracy our Recommendation Engine by varying the similarity metric used, the number of similar users to find, and the number of recommendations. We computed the accuracy by running through the five base and test documents provided in the MovieLens dataset, each of which are 80 - 20 splits of the whole dataset. We computed recommendations for each of the 943 users in the 80 percent base document, then checked to see whether or not those edges were present in the 20 percent test document. Finally, we computed the accuracy by dividing the number of edges found in the test document by the total number of recommendations provided. Thus, we know that the user had at least watched and rated the recommended movie. Using this method, we tested the two similarity metrics at a variety of points, where we hypothesized that Pearson would be more accurate, as it seems to be a more complicated and well-developed index.

Our results seemed to show that Jaccard was the better metric, as it had a higher percentage of nodes found in the test data than the Pearson metric, as shown in the results tables below:

Number of Similar Users	Number of Recommendations	Percentage (Pearson)	Percentage (Jaccard)
10	1	10.80	35.23
50	1	14.63	40.19
50	3	12.57	34.77
75	5	12.99	31.89

Description of Work Performed:

Trevin Gandhi:

-Built core Graph representation
-Implemented BFS
-Worked w/ Seth on PageRank
-Implemented Ford-Fulkerson
-Implemented Floyd-Warshall
-Implemented Bellman-Ford
-Worked w/ Seth on the Recommendation Engine

-Worked w/ Seth on Kosaraju's
-Test cases
-Implemented Jaccard Coefficient
-General codebase maintenance (Class structure decisions, refactoring, bug-fixing, etc.)
-General git maintenance (fixing merge conflicts, keeping our git repo clean, etc.)
-Worked w/ Seth on figuring out la4j's wonky as all hell API
-Found data online
-Worked w/ Seth on analysis portion

Deepan Saravanan:

-Implemented DFS
-Implemented Topological Sort
-Implemented Cycle Detection
-Implemented private helper methods such as stamping and getting unvisited nodes of zero in-degree
-Help with refactoring
-Javadoc comments, as needed
-Wrote Test Cases for DFS
-Wrote Test Cases for BFS
-Wrote Test Cases for Top Sort
-Composed user manual for recommendation engine

Seth Bartynski:

-Worked with Trevin on PageRank and the Recommender class
-Implemented Kosaraju's
-Added Pearson Correlation Coefficient
-Implemented Brandes'
-Javadoc comments, as needed
-Implemented DataReader class
-Implemented GUI
-Formatted data for test cases with Deepan
-Ran and wrote analysis portion
-Helped with refactoring
-Worked with Deepan on test cases
-Project spokesperson** (w/ Trevin) (at the Demo Day)