Yiming Cai
Rizwan Khan PA4 Report - 6/3/2017 CSE 100 Spring 2017

PA4 Report

Graph Design Analysis

First, we considered the ways in which we can create an easily traversable graph structure. In a graph, we will need 3 components: the graph itself, the nodes that the graph contain, and the edges that each node holds. For this assignment, we need to create an undirected graph, but it turns out that having the nodes storing directed edges is much easier for traversal than storing undirected edges, so we did that in exchange for some memory overhead. We did not check for duplicate edges between two nodes in order to optimize runtime (and it turns out to be useful for the creation of UnionFinder later on).

Our innermost tier class would be **MovieName**, which is used to create instances on the heap that simply stores the name and year of unique movies. Then, we have ActorNode and ActorEdge on the same next innermost tier. Each **ActorNode** instance stores a vector (or array) of pointers to ActorEdge instances, representing the outward-pointing edges of that node. It also stores a string, which holds the name of the actor, and serves as an identifier under the assumption that each actor has a different name. Each ActorNode instance also stores some public variables such as 'prev', 'dist' and 'done' to support graph traversal. **ActorEdge** instances each hold a pointer to the destination ActorNode instance, and a pointer to a MovieName instance to signify the connection between these two particular nodes. All the above instances are allocated on the heap so that they will not be deallocated when a stack frame is deallocated.

On the outermost tier is the **ActorGraph** instance, which stores the pointers to all unique nodes inside a set (which uses a BST underneath). A set is used because it does not allow duplicates, and we can use the 'insert' method to see if duplicate nodes are already inserted, though this is only possible after writing a custom comparator class. For memory management, ActorGraph instance also uses a vector to store the pointers of all MovieName instances, so that when the ActorGraph is deleted, it can delete the MovieName instances too.

Three other class, ActorPath, MovieList, and MovieGraph, are not considered part of the graph. Rather, they are helper classes that make the creation of the ActorGraph and path traversals easier. **MovieList** is used to store a set of pointers to MovieGraph instances, each of which contains a unique MovieName pointer (MovieName instances are, in fact, created in MovieGraph instances). Each **MovieGraph** instance stores a list of pointers to ActorNode instances, who represents the actors that have taken part in that particular movie. When all the actors and movies are loaded, each MovieGraph instance will generate enough edges to create a 'complete graph' among the list of ActorNode instances it stores. This means that, every ActorNode becomes directly connected to every other ActorNode instances inside the MovieGraph. At the end of LoadFromFile function of ActorGraph, we used this functionality of MovieGraph instances to connect all the lone ActorNode instances into a proper graph. Each edge generated by this instance will store a pointer to the same MovieName instance of this graph to preserve information of that particular connection. **ActorPath** instances are used to conveniently store paths found in the graph via the findPath function of ActorGraph. Each ActorPath instance stores a pointer to the starting node of the path, and a list of edges that constructed the path. It also helps in printing out the paths to an output filestream in a standardized format.

We have constructed our graph using these classes to maximize the efficiency or graph creation without impeding the efficiency of graph traversals. While the memory usage can be huge depending on the amount of information stored in the input file, it has made sure that all information is losslessly stored, and that no unnecessary duplicate instances are being stored at the same time.

Actor connections running time

1. Which implementation is better and by how much?

The Disjoint Set implementation very easily beats BFS for any number of pairs. The following table will display the runtime of each implementation. Note that all tests are done on Vocareum with movie_casts.tsv as the input file for fair comparison. All runtime units are in seconds.

Number of Pairs	Runtime of BFS (bfs)	Runtime of Disjoint Set (ufind)
5	0.814	0.552
100	2.387	0.584
1000	16.365	0.570

Note that the Disjoint Set implementation is faster with a test file of 1000 pairs than with 100 pairs. This is most likely because, when performing more searches, more path compression is done. This results in faster up-tree traversal since more path compression means a fatter, shorter up-tree. The scale of change in the runtime of each disjoint set is very small, most likely because most of the runtime is consumed for reading the input files, which requires disk access, and thus the number of pairs computed do not affect the overall runtime by too much. In comparison, the scale of change is much more significant for the BFS implementation.

2. When does the union-find data structure significantly outperform BFS (if at all)?

The union-find implementation significantly outperform BFS when the number of pairs grows to over a hundred. The greater the number of pairs the program is required to find, the greater the difference in runtime of the two implementations.

3. What arguments can you provide to support your observations?

BFS requires the traversal of multiple nodes, possibly the entire connected graph to conclude that a connection does not exist. Therefore, BFS has an average runtime of O(|E|) for each check, where |E| stands for the number of edges in the graph. In comparison, the Disjoint Set implementation only requires the traversal of a few nodes up the up-tree. Since we have designed the up-tree structure of each disjoint set to be as efficient as possible by reducing its height to a minimal, the traversal needed to confirm whether two nodes are connected or not takes a very short time. It has an estimate average runtime of O(log*|E|) to O(1), since most nodes are directly connected to their sentinel node.