Partner1:
Name: Jiaqi Fan
PID: A12584051
Partner2:
Name: Xuanru Zhu
PID: A91064234
Date: 3/13/2017
PA4

Report

Graph Design Analysis:

Describe the implementation of your graph:

Upon the two uncompleted ActorGraph file provided we created two new classes, one is Vertex class contain Vertex.cpp,Vertex.h. The other one is Edge class that contain Edge.h and Edge.cpp. Each Vertex stands for one actor, it contains the actor information (string actor), the edges between actors so we used vector edge pointer (verctor<Edge*> edges) which is the one we used for checkpoint since we only need to find the shortest path in unweighted graph, every two actor could be connected by several movies which are both in, and we also use a hashmap (std::unordered_map<std::string, Edge*> weighted_edges) for final submission which is find the shortest path in weighted graph, since in weighted graph we will want to choose newer movies over older movies when connecting two actors. Besides these, we have the variable (int dist) the distance from the source, this is used when we implement the weighted and unweighted graph. The previous vertex in the path (Edge* prev) also used to implement the weighted and unweighted graph to track the shortest path by tracking backward. Lastly the boolean variable that stand for when searched or finished or not (bool done) in the BFS search and dijkstra's algorithm. Those are the needed elements in vertex class.

Each edge contains information about the movie that two actors share (string movie), and the movie year (int movie_year), and the weight which is the weight of the movie edge(int weight), the weight calculation will be 1 + (2015 - Y) we are using 2015 not 2017 is because the database only contain movies in 2015 and earlier. Lastly the actors that are connected by a pointer which also can be described as vertex (Vertex* vertex). For ActorGraph class we used two unordered hashmap (unordered_map<string, Vertex*> actors) and (unordered_map<string, vector<Vertex*>*> graphBuild). In (unordered_map<string, Vertex*> actors), the string is actor's name, and Vertex* is the pointer to the actor's vertex. In (unordered_map<string, vector<Vertex*>*> graphBuild) the string is the specific movie, and the vectors contain all the actors that in the specific movie. We use this vector to build the graph, to build the connections(edges) between actors.

Also, in the Vertex.h class, we have another two things which are std::unordered_set<Vertex*> connection and Vertex* connect, the first hashset contains all the connections from this actor to all other actors if they have ever been both in one movie, this is used for the actorconnections BFS search to find out whether two actors are connected. The second Vertex* connect is pretty obvious, it is used for actorconnections too, but for UnionFind, it is the pointer which points to it parent in the Up-Tree data structure. We also have (int size) to used in the UnionFind to keep track the size of the tree to see which is bigger or smaller.

Describe "Why" you choose to implement your graph this way:

Our design is really clean and easy to understand. Just by looking at it other people will know that the vertices are the actors with their information and edge is the movie with the name and year and the pointer to the coactor. It is clean because each actor is one vertex and each

movie is one edge if two vertices are connected then there is one edge going and one edge going backward. It will be easy for us to track backward when we need to find the shortest path for weighted and unweighted graph. That will save a lot of time when we prepare everything during the graph build.

For the unweighted graph(checkpoint) we used vector to store the edges, because we have to read the entire input file whatever, and for unweighted graph, we don't care about the weight of the edge and when we try to search the shortest path, every actor is only pushed into the queue once,(cause it will be marked done when pushed into the queue). So, vector does not affect the running time. For the weighted graph(final) we implement our edges using hashmap, since this time we care about the weight, we want the newest movie to connect to two actors, so two co-actors only has one edge between them, we do not want to push them into the priority queue more than once from the same actor which connected to it. And we only mark them done when popped from the priority queue. For the data structure to store Vertex, we always use the hashmap, because we can access each actor in O(1) time. The main reason that we use hashmap is that it is the fastest data structure that we could use in the C++ STL data structures. The running time of the hashmap is O(1). When we search the actors or movies that we stored in it could save more time than other data structures. The most thing that we want to optimizing is the running time of pathfinder, actorconnections that we are going to implement. We are required to have the runtime not slower than the ref solution runtime. We have not change the graph design for checkpoint since we really care about the running time efficiency at the beginning, and we added some(the hashmap) for the final djkstra's algorithm in order to maintain the running time efficiency, and our design succeeded in running time and style.

Actor Connection Running Time:

Note:

We calculated the running time based on the 'time' flag we run the program like this:

time ./actorconnection movie_casts.tsv pair.tsv outFile ufind/bfs

And there will be three time printed on the terminal real time, user time, sys time. We only want the "real time" that printed on the terminal. We recorded all the real time and take the average of it. All program are executed 10 times and we take the average on them.

Test for one actor pair:

The first argument is movie_casts.tsv and the second argument is the one we generated which is HOUNSOU, DJIMON 50 CENT. the running time described below:

bsf: 4711 millisecond ufind: 1383 millisecond

The result showed that the first connect between those 2 actors is in 2003.

Test for 100 repeated pairs:

The first argument is movie_casts.tsv and the second argument is the one we generated which is BACON, KEVIN (I), HOUNSOU, DJIMON. the running time described below:

bsf: 10311 millisecond ufind: 1268 millisecond

The result showed that the first connect between those 2 actors is in 1992.

Test for pair.tsv:

The first argument is movie_casts.tsv and the second argument is pair.tsv. the running time described below:

bsf: 14577 millisecond ufind: 1148 millisecond

Q1: Which implementation is better and by how much?

A1: Union Find is better. In different test cases the running time is different in general when we are running single pair and in movie_casts.tsv the approximate running time for ufind is more than 3 times faster than bfs. The running time is about the same for 100 repeated pairs and 100 different pairs using UnionFind. But the running time for bfs increases obviously. The running time of the UnionFind is about 9 times faster than bfs in 100 repeated pairs and 14 times faster than bfs in 100 different pairs. So, we concluded that when the number of pairs increase, the time for bfs takes longer and when the "same" property changes to "different", the running time for bfs also increases. However, the running time for ufind is about constant for any size and property of data.

Q2: When does the union-find data structure significantly outperform BFS (if at all)?

A2: All the cases that union-find data structure outperforms BFS. For significantly outperforming BFS, when there are a lot of pairs finding, the union find data structure will significantly outperforms BFS, because every time, union-find use path-compression when finding, but BFS does nothing, BFS has to search again every time. So, when there are lot of pairs to find, the union find data structure will significantly outperforms BFS.

Q3: What arguments can you provide to support your observations?

A3: In class we learned that the worst case running time for BFS is O(|V| + |E|) where |V| is all the actors in the graph and |E| means all the connection edges in the graph. In the worse case for BFS we are taking count of all the edges and vertices which is very large amount of data if the input file contain a lot of movie and a lot of actor pairs. In the other hand the worst case for union-find is O(1) with path-compression optimization and weighted union(path-compression

really does a good job). This supports our observations, the result clearly showed that union find run much faster than BFS in all kinds of large dataset.