Melvin Wijaya

Jesslyn Hernawan

Graph Design Analysis

Checkpoint Report:

The new classes that we wrote are Graph.h, Graph.cpp, and pathfinder.cpp. We decided to create a friend class for the Node (vertices) class inside the Graph class to make our implementation easier as we do not have to have implement the public getter and setter methods for accessing the member of the Node class. To represent the graph, we went with an approach where graph is represented with adjacency list. We did not define a new structure for the edges as we keep all the neighbor edges of the nodes as a vector inside the class member themselves. Apart from that, to differentiate each node in the graph, we give them index and save the actor names inside the member class. This is done because we want to be able to have direct access to the pointer that points to each node inside a vector when we pass this vector into the method where we will run BFS to find the shortest path from a source to a vertex.

Final Report:

Since our checkpoint code submission got timed out for the last test for building the graph from the movie_casts.tsv, we decided to change the entire design to a new one. This is the fix that we do to make the entire graph building faster.

Files that we create for this final submission are the following:

-Graph1.h

-Graph1.cpp

-pathfinder1.cpp

-actorconnections.cpp

Inside the class Graph1, we now have two different friend Node class, NodeActor and NodeMovieYear, to allow us to represent the graph. Unlike in the checkpoint submission, where we created adjacency list as the private field inside every vertex, we now do not create that kind of representation so that the build of graph would not take too long. In this new design, we use unordered_map of vertices as a private member in the Graph class where each key is the name of the actor from the input file like movie_casts.tsv and the element is the list of movies (we use vector to save this list) where that actor appears. Furthermore, we also use unordered_map to store each movie and year as the key with the list of actors (we use vector to save this list) who play in that movie as the element. We change to this new implementation because it will allow us constant time to access each actor and each movie they play in, as well as to access a movie and get the list of actors who play in that movie. But more importantly, by having this new design, we do not have to create adjacency list for each vertex. Thus, building the graph in this new design is way much faster.

To be able to execute the pathfinder problem, we create a new class called WeightedEdge as a friend class of Graph class to represent the edge with weight between vertices. This weighted edge saves the name of a vertex's neighbors with the weight to reach that neighbor. This Weighted Edge is used only for allowing us to run the dijkstra's algorithm on our graph. Not only that, we also write the WeightedEdgeComp so that we can get the lowest weight in our priority queue.

To be able to execute the actorconnections problem, we create a class called nodePQ as a friend class of Graph class so that we can keep track with the year of a movie and its list of actors inside the priority_queue where the earliest movie has the highest priority. We create the nodePQComp to make the priority_queue works this way. Moreover, we create the class DisjointSet to allow us to solve the actorconnections problem with disjoint set. The private members of this class are two unordered_map where the first one is to save a vertex as the key and its parent as the element. The second one is to save a set as the key and its size as the element. We choose unordered_map because accessing element in a hash table is constant time.

Overall, we decided to keep the other classes apart from the Graph1 class as friend classes so that we do not have to crate the setter and getter method which will make the code really long and hard to read.

Actor connections running time:

100 different actor pairs:

BFS Runtime: 109 seconds

Union find Runtime: 111 seconds

100 same actor pairs:

BFS Runtime: 109 seconds

Union find Runtime: 110 seconds

1. Which implementation is better and by how much?
   - From our runtime results, BFS performed faster on both the same actor pair and different actor pair files by a small amount. The runtime number in seconds are written above.

2. When does the union-find data structure significantly outperform BFS(if at all)?
   - In theory, the union-find data structure should always outperform BFS, especially when we have a large file input. For smaller data inputs, their number should come close to each other. This is caused by union-find modifying the structure of the tree every time it is called resulting in a much faster find operation compared to the BFS.

3. What arguments can you provide to support your observations?
    - In our observation, the BFS outperformed the union-find in both same actor pair and different actor pair input when in reality the union-find should be the one outperforming BFS in both of the tests. The results of our observation contradicts our understanding of the BFS and union-find running time. An explanation of why our results differ from the expected outcome may lie in our implementation of the union-find data structure and function. Our implementation may not have been efficient enough resulting in a longer runtime compared to BFS. Even with our inefficient implementation of the union-find data structure, its runtime differ only by 1 or 2 seconds and came close to beating the BFS. If the union-find data structure were implemented correctly and efficiently, its runtime should overall outperform the BFS and significantly outperform it for large inputs.