File name: Report.pdf
Authors: Jamie Shi (A12678538) and Sergelenbayar Tsogtbaatar (A12949154)
Date: March 13, 2017
Overview: Describes structure of our graph and disjoint set, and analyzes runtimes
Assignment #4

Graph Design Analysis

For the implementation of our graph structure, we created three classes: the Movie class, ActorNode class, and the ActorGraph class, each with its own respective .h and .cpp files.

For the Movie class, we used a vector to hold a vector of ActorNode pointers – this held all the actors that took part in the particular movie. Each Movie object would also have its own movieName (a string, representing the name of the movie), movieYear (an int, representing the year the movie was released), and movieInfo (a concatenated string of movieName and movieYear). The Movie class also had a weight of type int, which is used for Dijkstra's algorithm because Dijkstra's algorithm finds the path in the graph with the lowest weight.

For the ActorNode class, we used a vector to hold a vector of Movie pointers – this held all the movies that the current actor took part in. Each Actor object also had an actorName (string) associated with it. In addition, it had a dist (int) to represent the distance from the current actor node to the source node. It had previousActor (ActorNode*, representing the previous actor node in the path) and previousMovie (string, representing the name of the previous movie in the path), both of which are member variables used for finding paths in BFS and Dikjstra's algorithm. It also had a parent pointer (ActorNode*) and a treeSize (int), which are member variables used for the disjoint set implementation. We also made a custom less-than comparison operator, which would allow us to turn a priority queue from the default max-heap to a min-heap – this would be useful for Dijkstra's algorithm, where we want to retrieve the edges with the lowest weights first.

The ActorGraph class had three hash maps: a movieMap, which mapped a movie's information (its name and year released) to a Movie pointer, an actorMap, which mapped an actor's name to an ActorNode pointer, and a yearMap, which mapped a year (int) to a vector of Movie pointers, representing all the movies released in that particular year. The movieMap and actorMap used unordered maps because order did not matter (so its find operation has constant runtime). The yearMap used a map because we wanted to sort the years in ascending order.

In ActorGraph, we had functions loadFromFile, connectionsLoadFromFile, resetGraph, and the destructor. The loadFromFile function retrieved all the necessary information from the input file. It also gave you the option to have take into account of weighted edges where the edges are represented as movies (used for Dijkstra's algorithm), or simply unweighted edges (used for BFS).

Essentially, we created our graph by first iterating line-by-line through the input file to get the current actor and movie data, and using this information to create a forest of actor nodes that each contain a list of the movies the actor has took part in. This was achieved by making each actor node have a vector of Movie pointers, and each movie have a vector of all of the actors who starred in that movie (we use vectors to store the adjacency lists – i.e. all the movies an actor starred in and all the actors that starred in a movie – because ultimately we'd just need to traverse the elements in the vector, and because the elements in vectors are contiguous in memory the use of the vector as storage is memory-efficient.)

To prevent making actor nodes that represent actors already in the graph, and to prevent allocating more memory onto the heap that would've created duplicate movies, we used the movieMap and actorMap (both of which are unordered maps, since the order in which the actors/movies are stored in the data structure does not matter in this case). We also used movieMap and actorMap to associate the current actor's movie vector through hashes and pushing back the current movie to that vector, and getting the current movie's vector of actors through hashes and pushing back the current actor to that movie). Storing the actor nodes and movies in unordered maps is efficient because retrieving them takes on average O(1) runtime.

So to summarize, our graph comprises actor nodes stored in an unordered map. Each actor node stores a vector of Movie pointers (representing the movies this actor appeared in), and each of these movies are stored in an unordered map. Each movie also has a vector of ActorNode pointers, representing the actors that appeared in this movie.

For every line in the input file, we get the actor's name and the movie information. If the actor doesn't exist already, we initialize it and add it to the actorMap. Then we check if the movie exists or not, and if it does exist, then we get the movie from the movieMap and add that movie to the current actor's movies. If the movie does not exist, then we make a new Movie object and add that to the movieMap and the current actor's movies. Thus, after all lines have been read, our graph relationship would have been created.

The resetGraph function is used for BFS to reset the graph's visited nodes, by iterating through all the actors and changing their dist, previousActor, and previousMovie member variables to their initial values. Resetting this graph does not change the structure of the graph; it merely resets the traversal and because it just iterates through all the actor nodes, it has O(V) runtime, where V is the number of actor nodes (i.e. vertices).

For the destructor, we simply iterated through the movieMap and actorMap (both unordered maps), and deleted the movie/actor accordingly. The movieMap and actorMap were traversed by using an iterator since their data can't be accessed directly by index. Maps were efficient to use in this case because the values (in this case, the Movie/Actor objects) in the maps' key-value pairs can be accessed in constant runtime (on average), allowing us to easily get and delete these Movie/Actor objects that were allocated on the heap.

Extra: Explanation for our implementation of actorconnections:

We first read all the information from the input file, getting all the pairs of actors to find connections between, and storing these pairs in a vector that stores pairs which takes two string (where the strings represent each actor's name). We also create a forest of movieless actor nodes, via calling connectionsLoadFromFile for bfs or dLoadFromFile for ufind. These actors are stored in an actorMap, which maps a string (representing the actor name) to a pointer to the ActorNode – unordered maps are used for storage in this case because given the actor name, we can find its ActorNode in constant time. We also create all the movie nodes (also stored using unordered maps

for the same reason as above) and push back all the actors who appeared in that movie to the current movie's vector of actors. Each movie uses a vector to store all the actors that appeared in it because a vector allows for fast traversal through its elements, which in this case is the actors.

In addition, we use a map, yearMap, to sort the years from earliest to latest. To explicate, this map maps a year (int) to a vector of movies (representing movies released in that particular year). Because a map by default stores its keys in ascending order, we could iterate through the map to get the years in order from earliest to latest, and then access each year's movie vector, which we could traverse to get all the movies in that year. Thus, maps in this case provide fast retrieval of year/movie information.

After we've stored all the pairs in the file, created the forest of actor nodes, sorted the years from earliest to latest, and associated all the movies with the year they appeared in, we exit the while loop.

We then create a vector of ints called “years”, where all elements are initialied to 9999 and the size of the vector is set to be the number of actor-pairs to find connections between. This is because this vector represents the earliest year that each pair of actors are connected, and in the beginning none of the actors are connected, so their connection year by default is 9999.

Then, if we're finding actor connections using bfs:
We use an outer for loop to iterate through the yearMap one at a time. In each iteration we would get all the movies released in that year, and for each movie we get all the actors that appeared in it. We add “edges” to these actors by associating them with the current movie (via pushing the movie to the current actor's vector of movies). After we've added the relevant edges to all the relevant actors for the current year, we iterate through all the pairs of actors and use BFS to see if and when a connection existed between the actors. If a connection exists we set the current index of the yearsVector to be the current connection year, indicating that this pair of actors was connected at the current connection year (and the next time we iterate through the vector of pairs, we wouldn't run BFS because we have an if statement that would make the code execute the next iteration if the connection year is already set to a value not 9999). We keep iterating through the years one at a time, incrementally adding edges (movies) to the actors and running BFS on all the pairs of actors to find connections between, until we've traversed all the years that had movies. By then our years vector should contain all the information about the earliest year a pair of actors was connected, so we iterate through this entire year vector and output the results to the out file.

If we're finding actor connections using ufind:
We also use an outer for loop to iterate through the yearMap one at a time. However, this time we do away with the concept of graphs and edges. Instead, for each year, we iterate through all the movies that appeared in that year, and do a union-by-size on all the actors that appeared in these movies. Then after all the union-ing is done, we iterate through all the pairs of actors to find connections between, and for each pair of actor nodes we call our Disjoint class's find function on them. If these two actor nodes have the same sentinel node, then they're in the same “tree” (i.e. they are connected) so we set the value at the current “year” vector's index to the current year. Thus after iterating through all the years and performing the above operations, our “year” vector should contain all the information about the earliest year in which each pair of actors is connected – so we traverse this entire years vector and output the results to the out file.

Actorconnections Running Time Comparison

1) Which implementation is better and by how much?

The Disjoint set implementation is better overall. We found the biggest runtime difference is between BFS' bfs algorithm and the disjoint set's find function. We tested the efficiency of each runtime on the given “movie_casts.tsv” file with two different pair files: one contained the same 100 actor pairs, and the other contained 100 different actor pairs. We ran ten runs for each file (the file with 100 of the same actor pairs and the file with 100 different actor pairs), for each algorithm (bfs and ufind).

The two files we tested on were:
samePair.tsv (100 same actor pairs – we've included this file in our Github repository)
pair.tsv can be found in /home/linux/ieng6/cs100w/public/pa4

We saw that for the finding actor connections using the file with 100 of the same actor pairs, bfs took on average 2935 milliseconds, whereas ufind took on average 1313 milliseconds. Thus, for the same-pairs actor file, ufind runs about 2.2 times faster than bfs.

For finding actor connections using the file with 100 different actor pairs, bfs took on average 2770 milliseconds, whereas ufind took on average 1318 milliseconds. Thus, for the different-pairs actor file, ufind runs about 2.1 times faster than bfs.

So overall, it is clear that ufind outperformed bfs for both cases – finding connections between the same pairs of actors, and finding connections between different pairs of actors.

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

The union-find significantly outperforms BFS when it tries to check for a relatively larger number of connections (say 100+) between pairs of actors.

If we're only trying to check if a connection exists between one pair of actors, the bfs algorithm is actually a little faster than ufind because the first few times we do ufind we'd have to build the up-tree (the actor nodes start out as disjoint, single nodes) by union and find, but bfs just utilizes traversal.

However, when we try to find whether a connections exists between many more pairs of actors (e.g. 100+ pairs of actors), ufind would significantly outperform BFS. It also outperforms BFS on finding the actor to be found because the nature of the union-find data structure is such that if you wanted to see if there is a connection between two actors, you only need to traverse the “up-tree” and compare their sentinel nodes. On the other hand, BFS works by checking all the neighbors of the nodes and branching out, so if you wanted to check whether there is a connection between two actors, you would need to compare every node in the graph until you find the actor to be found. In short, relative to BFS, ufind would often traverse less nodes and go through less comparisons to find the end actor node (if it exists).

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

Our BFS works by first pushing the start actor to the queue. Then we traverse “layer by layer” based on the breadth first search algorithm. In other words, we find all of the current actor node's neighbors – that is, we loop through all the movies the current actor starred in, and for each of those

movies we push all the unvisited actors to the queue. Then we pop once from the queue and repeat this search process until either the end actor is found (in which case we return true) or the queue is empty (in which case we return false because there was NOT a connection between the start and end actor). In the worst case, there is no connection between the start and end actor, so the BFS will explore all the actors and movies in the graph; therefore it has a runtime of O(V+E) where V is the number of actors and E is the number of movies

Our Disjoint set structure uses the general Union-Find functions, and it also uses path compression in Find and Union-by-Size for the Union function. These additions would allow for an overall faster runtime. As we perform more Union/Find functions, the runtime of these function should become near constant runtime. We use union-by-size, which appends the smaller subtree (i.e. less nodes) to the bigger subtree (i.e. more nodes). This was used in conjunction with path compression in find, so that as we are traversing up the node's parent pointers (until we reach the sentinel node), we store all the nodes during the path to the sentinel into a queue. After we have found the sentinel, we set all the nodes' parent in the queue to the sentinel node. This readjusting of pointers takes constant time, but traversing up the “tree” may take on average O(logn) where the node to find is the leaf node and n is the number of nodes in the entire forest. After this path compression, future find operations get faster due to the readjusting of the aforementioned parent pointers – after all, more nodes would have their parents readjusted to be the sentinel node, so finding these nodes would take much less time, close to O(1) – and the O(1) runtime for union/find operations is clearly faster than the BFS's overall runtime of O(V+E).