Technical Writeup

NETS150 Final Project

Hanbang Wang, Jarett Schwartz, Adrian Wang

Usage of the visualizer

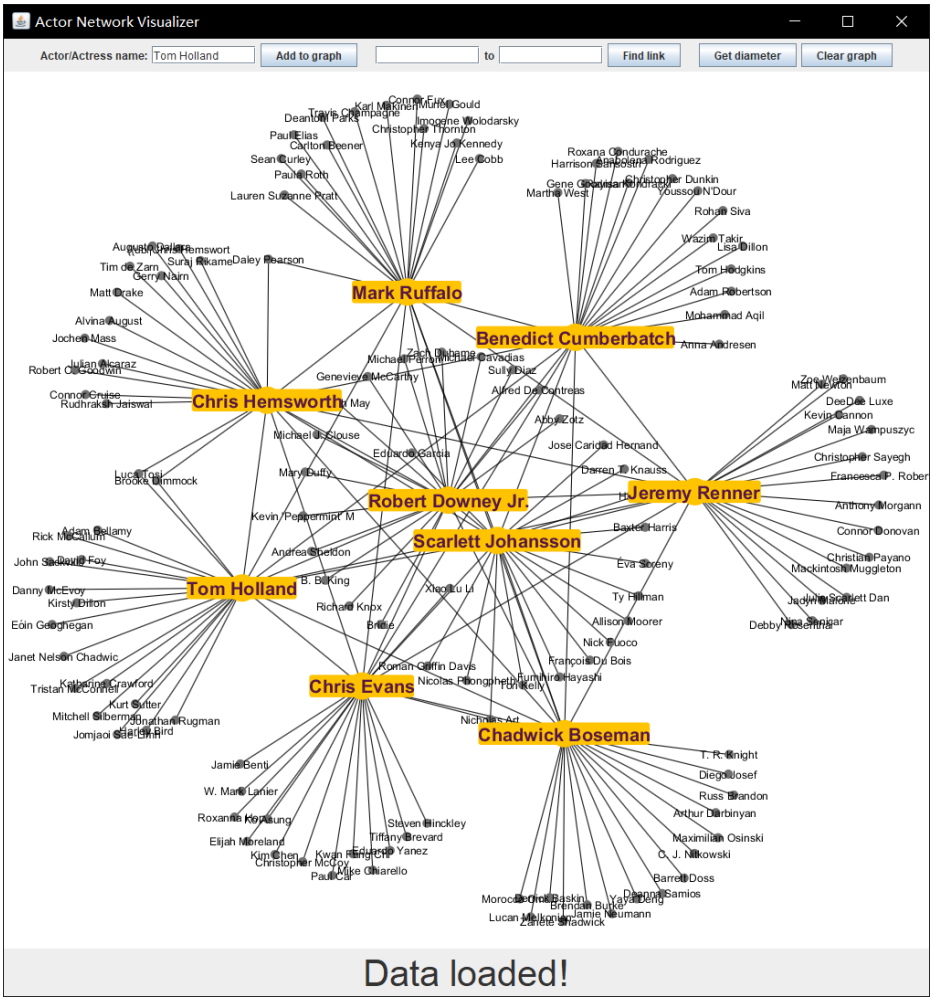
The entry point of the visualizer is `GraphVisualizer::main`. When compiling and running, make sure the libraries and the file “ui.css” are under the working directory.

The visualizer will start by downloading and processing the data. During this process, the inputs are disabled until the data is properly processed and the graph is built.

Main functions

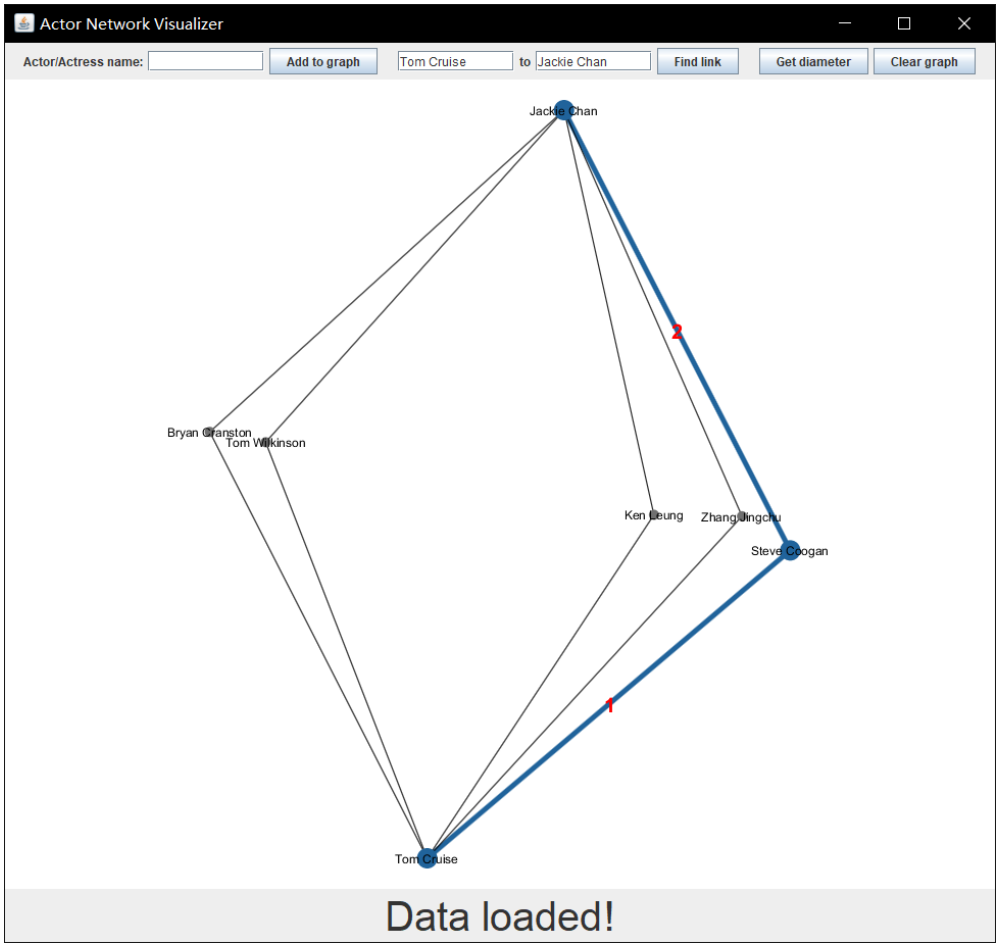
There are three basic functions of the visualizer: add an actor/actress to the visualizer, find a link between two people and display the link in the visualizer, and get the diameter of the graph and display it in the visualizer.

First, the node (actor/actress) added to the graph by the user will be displayed in a larger yellow box. When added to the graph, up to 15 of its highest-degree neighbors will also be added to the graph. If the newly added node has some edge between it and some nodes already in the visualizer, the edge will also be displayed.

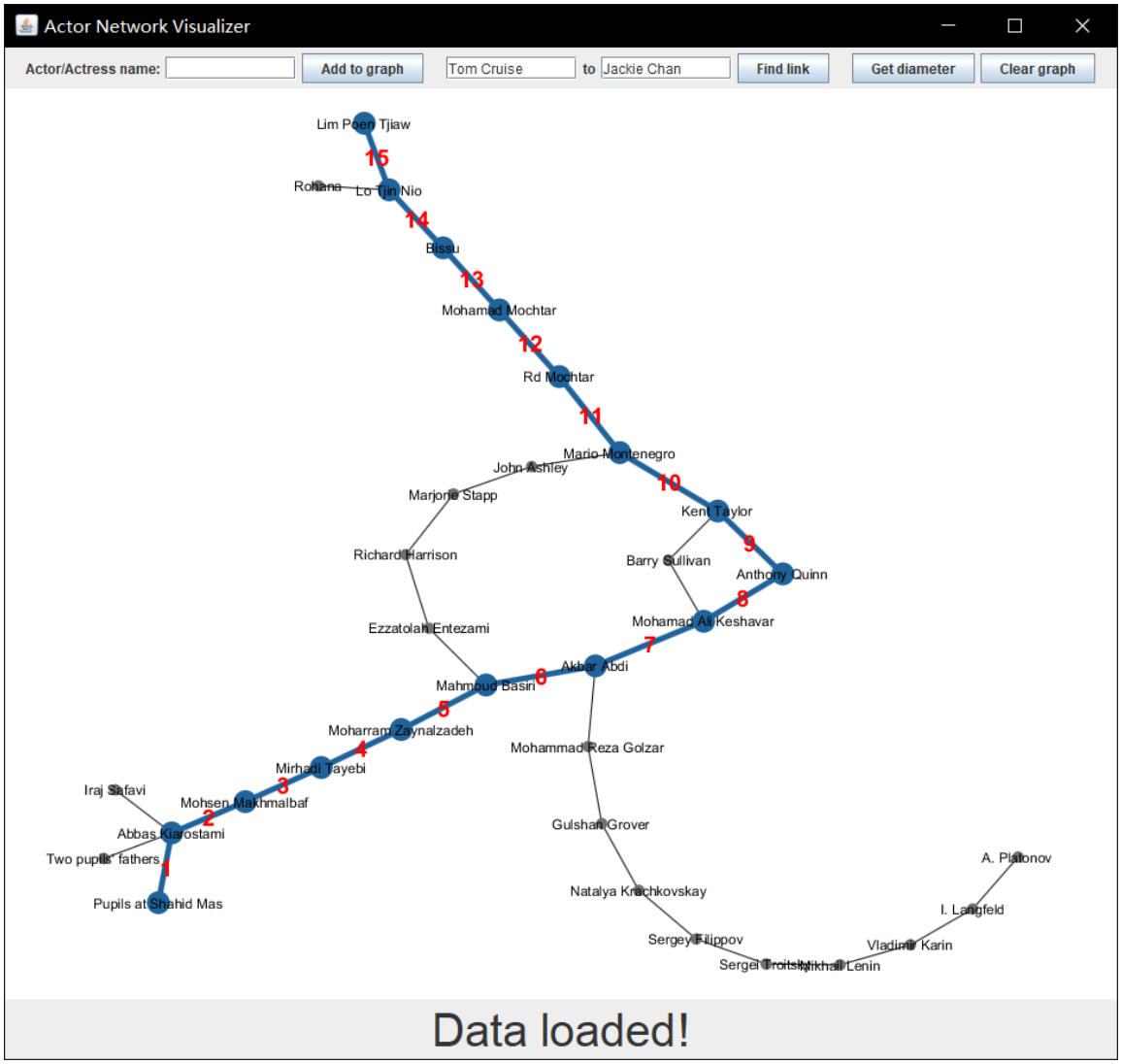


To use the Find Link function, input the starting and ending actors, and the program will find a shortest path between them. The visualizer will highlight the path in blue. The numbers of hops (distance) will also be displayed on each edge. The highlight will be

cleared when a new path is found by the visualizer. Note that neighbors' orderings will be shuffled each time, so a different shortest path could be found in subsequent runs.



The get diameter function is straightforward. The user just presses the button and a longest shortest path will be added to the visualizer. Like Find Link, each run of this function could find a new diameter.



Extras

The “clear visualizer” button will remove all nodes from the visualizer window, while not affecting the backend graph. To delete one node, right click on the node, and select “delete node” from the pop-up menu. When a node is deleted, any former neighbors that are now lone nodes will also be deleted.

By right clicking on a specific node and selecting “copy name” in the pop-up menu, you can copy the name of the actor/actress associated with a particular node.

You can click and drag any node with the left mouse button to move it around the visualizer window. Notice that after a node is moved, the graph will automatically rearrange the graph using the Barnes–Hut simulation (which is provided by the library we use).

Lastly, you can scroll to zoom in or out of a specific place on the graph (it works best with a mouse). Scrolling up will zoom in, scrolling down will zoom out, and all zooms are centered on the cursor’s current location. You can also use keyboard arrows to pan up, left, right and down.

Program Details

This program consists of 3 main classes for building the network and graph visualizer.

DataProcessing

`DataProcessing` is a class for downloading and processing the data from https://oracleofbacon.org/data.txt.bz2. This is the data set provided by The Oracle of Bacon website under its "How the Oracle of Bacon Works" (https://oracleofbacon.org/how.php) page. The data set is a txt file with each line being a JSON encoded object which includes movie's title, year, an array of cast, and so on. The data set is provided to us being compressed using bzip2 compression software.

In the `acquireData` method of `DataProcessing`, we first connect to the URL with `java.net.URLConnection`, and we use a library `org.apache.commons.compress.compressors.bzip2.BZip2CompressorInputStream` to decompress the file. Then, we prepared a class `Util.MovieInfo` for binding the JSON object. We then use `com.fasterxml.jackson.databind.ObjectMapper` to extract every JSON object into the class, and save the movie information into a list.

The `getAllCasts` method is able to extract all casts from each movie into a list of string array. Notice that in every actor in a string array has an edge between them, i.e., each string array forms a clique.

ActorsNetwork

`ActorsNetwork` is a class for all graph calculations and algorithms. It takes in a set of strings as nodes, and builds a graph based on these nodes. Inside, it will assign each node a distinct integer id as the true identification of each node. The map `nameToID` and `idToName` is used for keeping track of the mapping.

The `addEdge` method will add an undirected edge between the given node with string `u` and `v`; `getID` gets the ID with a given name as string; `getName` gets the name with a given ID as integer; `getDegree` gets the degree of the node with given ID as integer; `getSize` gets the number of nodes in the graph, with ID from 0 to size-1.

`breadthFirstSearchWithLength` implements a BFS algorithm with a starting node and returns an array of distance of the given node to other nodes. `breadthFirstSearchWithIncoming` implements a BFS algorithm with a starting and ending node and returns an array of the incoming node (parent in the BFS tree). Notice that the former will finish after traversing through the whole graph, and the latter finishes after meeting the ending node.

`shortestPath` gives the shortest path between two nodes and returns the list of nodes the path goes through. This algorithm uses BFS as in `breadthFirstSearchWithIncoming` since the graph is unweighted and undirected.

`getDiameter` gives the diameter (the longest shortest path) of the graph. This algorithm starts at a random node `p`, run `breadthFirstSearchWithLength` once to find the node `u` with largest distance from the starting node, and then run `breadthFirstSearchWithLength` again to find the node `v` with largest distance from `u`. We know from lectures (CIS 121 or NETS classes) that this will give us the diameter of

the graph. Notice that there exists a small probability that the starting node is not in the largest connected components, which would yield a result of a longest shortest path of that connected component.

The next few functions (`averageDegree`, `graphStDev`, and `actorStDevs`) are all used for statistical analysis of the whole graph. `averageDegree` calculates the average degree among all nodes through a sum of linked list lengths, then dividing by the size of the graph. `graphStDev` uses the formula for standard deviation $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$ by summing squared differences of each element from the mean, dividing by the number of elements to determine variance, then finding standard deviation of degrees (the square root of variance). `actorStDevs` makes use of these functions to find where each actor falls in the distribution, measured by positive or negative standard deviations from the mean.

`degBuckets` was used to facilitate the creation of histograms. Instead of dealing with each node and its degree individually, `degBuckets` calculated frequencies of degree intervals of a given size. For example, using an interval of 5 would find how many nodes had degree 0-4, 5-9, 10-14, and so on. Each bucket was defined by its lower bound, and the output map contained lower bounds as keys and frequencies as values. In order to allow for copy-pasting of data into Excel for visualization, empty buckets were created for intervals that contained 0 nodes. This avoided the problem of missing intervals when the map value set was printed.

`getActorStDev` was created for convenience, so that tests could gather data on specific actors without taking the time to calculate standard deviations for all actors and retrieve a specific entry from the resultant map. It functions as a single iteration of `actorStDevs`, where the difference between an actor's degree and the average is divided by the graph's standard deviation of degrees.

`sortedDegToName` was used as the first step in empirical analysis of high-degree actors. By sorting all nodes by degree and placing them in a map corresponding to their names, it was easier to pick an actor based on their degree and find additional information from the web.

GraphVisualizer

`GraphVisualizer` contains the entry point for the program. We use our own `ActorsNetwork` for algorithm, and uses the library `org.graphstream.graph` for visualization.

The method `init` will download and process the data and add all nodes to the graph class we wrote. The method `addEdge` will added an edge with given two nodes to the visualizer. `displayNeighbors` would display the node itself and up to 15 neighbors with the largest-degree on the visualizer. `findLink` will find the shortest path between the given two nodes and add it to the visualizer. `getDiameter` would get the diameter of the graph and display it on the visualizer.

The rest of the methods are related to the building of Swing components, and the code is mainly routines and pretty self-explanatory.