Project

Andreas Timoudas

https://github.com/timoudas/java/tree/master/Genome

1 UNIX

To get an understanding of the data, UNIX were utilized to generate test suites and clean the data before reading it into the Graph.

To generate test-suites below was run:

```
head -n data-set.m4 > test-set.txt
```

This generates a text file with n records of data from the data-set which had approx 65M records. Further, below awk script was used to remove redundant rows in the data based on conditions written in the project description.

```
#! /usr/bin/awk -f

#awk command to wrangle data
BEGIN{}
{
    if($7-$6==$8){
    }
    else if($11-$10==$12){
   }
   else{
       print $0;
   }
}
```

This made sure that only necessary records were included in the file. When the necessary records were extracted from the dataset a new file with only the unique identifiers was generated. The reason was to later load only the identifiers and map them against integers in a HashMap, before loading the newly generated dataset with necessary nodes.

2 Git/Github

Had two workstations, so in total tree repos. This made it important to pull remote repo before starting to work and push when finished. Wasn't necessary to use branches since I'm the only developer on the repo.

3 Algorithms

The main tool used was creating an un-directed Graph to store the edges in. I did not create my own custom classes for HashMaps and ArrayList's but used Javas built-in objects to solve the problems at hand. To iterate over the Graph i used BFS. Hashmaps were used to map identifiers to integers. The Graph was built untop of a adjacent ArrayList.

4 Algorithm Characteristics

Adjacent ArrayList: This was prefered over linked-list to build the Graph ontop on beacause of simplicity of usage. Uses $O(n^2)$ memory, fast to lookup an edge $O(1)$ but slow to iterate over since neighbours are not stored compared to the linked-list.

BFS: $O(V + E)$ were V is vertices in the graph and E is edges in the graph. Since I am using adjacency list the complexity is instead $O(V^2)$ This is because we are looping over all the vertices like below:

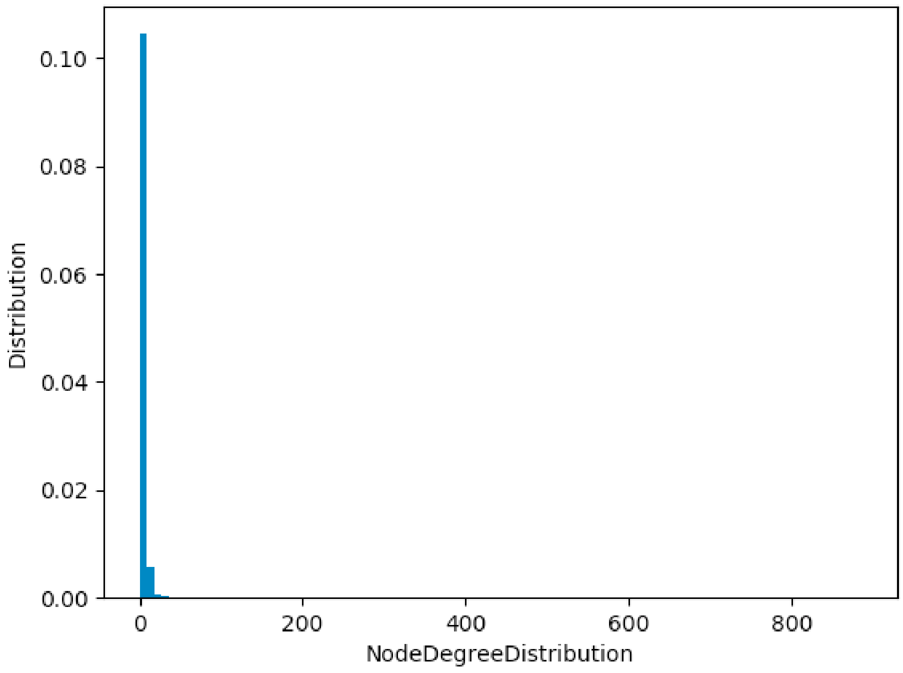
```
Queue<Integer> queue = new ArrayDeque<>();

        visited[vertex] = true;
        queue.add(vertex);

        while (!queue.isEmpty()) {
            int v = queue.poll();

            List<Integer> adj = adj_list.get(v);
            for (Integer w : adj) {
                if (!visited[w]) {
                    visited[w] = true;
                    queue.add(w);
                }
            }
        }
```

HashMaps: Great for lookup and mapping since the HasMap is in the form of $dict(key, value)$. Since all key, values were and an identifier only had to be mapped once this seemed suitable. **Other**: The biggest drawback in execution time during the development was found to be if/else statements, and checking if an element existed in a datastructure. Example:

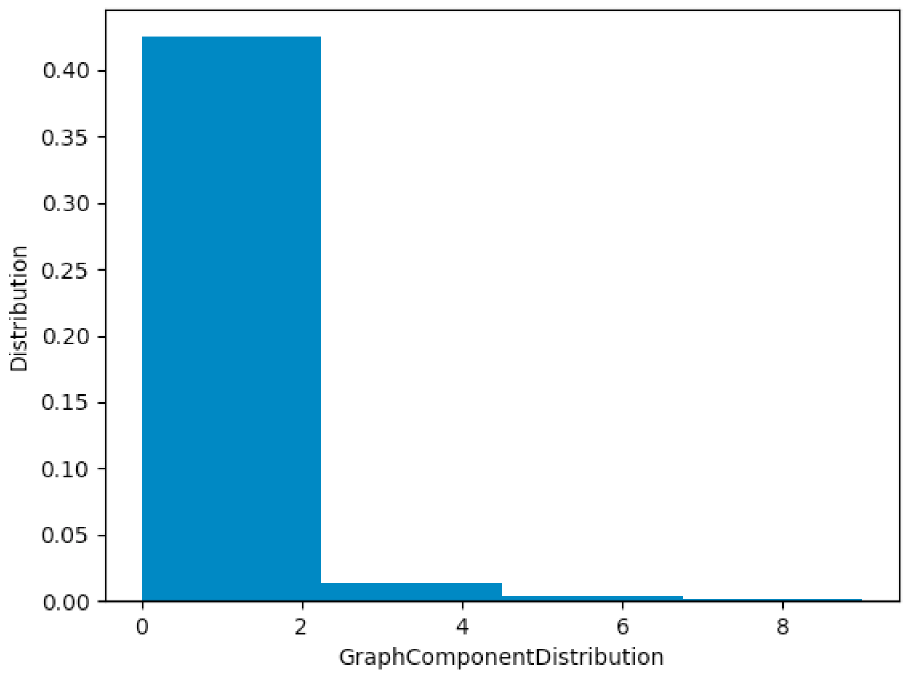
```
 for (Elem elem : ArrayList) {
    if (!OtherArrayList.contains(elem)){
    OtherArrayList.add(elem);
    }
}
```

As the ArrayList size increased the lookup-time increased and made this an impossible option to use with a dataset of 8M records. Forcing to work around problems by using UNIX and wrangling the data into digestible pieces.

- Remove unnecessary nodes
- Create a list with the unique identifiers
- Map the identifiers in a HashMap
- Load the dataset and change the identifies into their mapped values
- Build the Graph and add the nodes
- Compute node distribution, Component distribution and number of components
- Save the output to generate graphs of the output
- Profit



Node degree is quite small but there are outliers were the biggest degree is 885.



Most graph components have 0-2 edges and a samll share of the total graph components are larger than that. There is a total of 1868838 graph components