Project: A genome assembly graph

Martin Evertsson

May 2020

Problem description

In this project we were asked to advise a collaborator who wanted to understand a particular data set better. In particular we were given a data set that could be presented in the form of a graph and were asked to present the:

- The Node degree distribution of G.
- The number of components of G.
- Components size distribution of G.

The data set contains information for a large set of contiqs that was compiled from several different sets of data. Each line can be represented as an edge, containing 12 lines of information for that particular edge. The most important ones are (1, 2) that contains the vertices, and (6, 7, 8, 10, 11, 12) that contains information of overlapping contigs.

Solution

Remove overlap

The data set includes information about overlapping contiqs, so the first problem to solve is to remove all overlapping contigs. This problem was solved by writing a simple awk script in unix command that removes all lines with overlapping contigs, and only prints the remaining edges without the additional columns.

Design graph

With this condensed data set the next course of action was to translate the edges into a graph. This was done in java utilizing the HashMap interface where the "key" is the numerical vertex number and the "value" is a LinkedList containing an adjacency list of that particular map key. This solution was good for this particular problem since we are working with a large data set and maps associate values with a particular key, making the time complexity much better than, for instance, an array solution to the problem. The program scans though each line in System.in and does two things, namely:

- Checks if the contiq has been encountered before. If not save it to a seperate HashMap that stores Strings as integers.
- Produce an edge for that pair of vertices.

Since the contiqs are very large strings, it is much more memory efficient to translate them into integers before putting them into the graph interface. This solution should have a time complexity of O(n) (for n lines), since all operations in the scanning loop should have a time complexity of O(1) thanks to the functionality of HashMap. In a first iteration of this problem an ArrayList was used to store the Strings, causing the problem to get a quadratic time complexity due to how finding certain elements in a List works.

Degree Distribution

Next we need functionality to calculate the degree distribution of the graph. Here we use HashMap again, thanks to its ease of use a relative efficiency. We simply iterate over all map entries in the graph map and store their value size in a new HashMap. Next we iterate over this HashMap to print its values. This method should have time complexity of O(n + m), where n is the number of vertices and m the variance in degree. However, since the variance cannot possibly be larger than the total number of vertices, the time complexity should instead be O(n) for the first loop.

Component Distribution

For the component distribution we would first need to traverse through the graph efficiently. Since the graph structure is mostly unknown I chose to write a method based on the DFS algorithm. This was mostly because DFS only require you to store the parent nodes of the current element, making DFS more memory efficient than the BFS algorithm. Also, since I had no particular interest in the shortest path problem, the DFS felt like a more reasonable choice. Additionally I chose to write an iterative DFS, due to a StackOverFlow error I got when using recursion. The connectedComponent method iterate over all vertices and saves the component sizes in a map in the same manner as the degreeDistribution method does. In the loop the method updates the map with the component size as a key, and the value as the number of components of that given size. Generally the DFS algorithm has a time complexity of $O(|V| + |E|)$, where V and E are the vertices and edges respectively.

Furthermore the number of components were calculated by iterating over the component map and adding up all values of each key.

Results

This program was piped together with the mentioned awk script and then redirected to a txt file. In figure 1 we can see that vertices with a small amount of degrees are the most common and that most of the variance occurs in the beginning of the graph data.

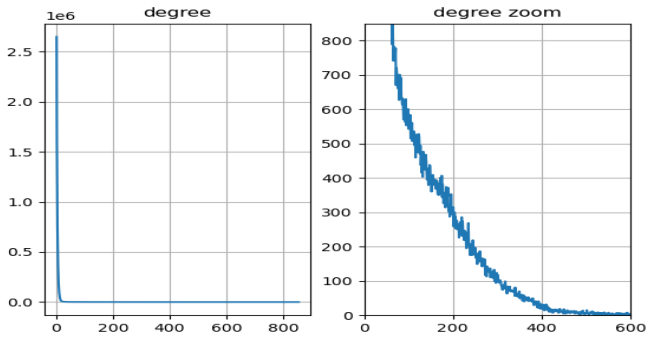

Figure 1: A figure showing degree distribution

Furthermore we can see in figure 2 that the component distribution follow the same pattern as the degree behaviour. The amount of components that consist of only two vertices is very high, with the graph plummeting quickly as the size of the components increase. This result coincides well with the results from figure 1 since if there are a large amount of components with only two vertices, it follows that there should be many vertices with only one adjacent vertex. As requested by the "collaborator" my program also found that there are a total of 961874 components in the graph.

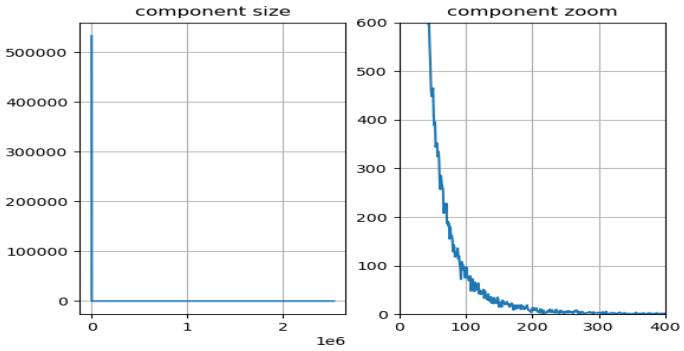

Figure 2: A figure showing degree distribution