A Genome Assembly Graph

Emilia Dunfelt

June 1, 2020

SECTION 1

Introduction

The aim of this project is to analyze a large data file containing overlap information between DNA segments. The data comes from shotgun sequencing of the Norway spruce, on which contig overlaps has been computed. By visualizing this data as a graph in which a node corresponds to a contig and an edge corresponds to a contig overlap it is possible to gain an overview of the data. In particular, it is interesting to know how many overlaps a particular contig has, that is, to find the degree of the nodes in the graph, and also to know how many and how large the overlapping parts are. So, among other things, it is of interest to know the degree distribution, the number of connected components, and the component size distribution.

SECTION 2

Method

To answer the relevant questions described in the preceding section, analyze, present and describe the data, a number of tools are used. The main programs are written in Java, with an additional Python script to present the results visually. Several Unix commands has also been used to get an overview of the problem, to run the programs, to investigate the running time and for dealing with version control via git and GitHub. A project diary has also been kept and is to be found at GitHub Pages via the GitHub repository. Next, a detailed description of the tools and methods used in the project follows.

§ 2.1. Git & GitHub. Git and GitHub have been used as a means to both publish the project diary in an easily accessible way, to share code and test files with collaborators, and to deal with version control. The GitHub repository is organized in the following way:

- /bin: directory containing executable code, ready to run
- /data: directory containing test files and data
  - /01: test files for DataTrim
  - /02: test files for GenomeIO
- /docs: directory containing html and css files for project diary and wiki
- /report: directory containing tex and pdf files for the project report
- /src: directory containing source code
- .gitignore
- README.md

There are also other directories and files that are only added to the local git repository, and not being pushed to GitHub since these files are not necessary for collaborators to have access to. Locally, there is a repository called /notes which contain the .wiki-files that contain the raw version of the project diary and wiki before these are converted to html in the /docs directory. There is also several larger data files in the /data directory that, for obvious reasons, cannot be pushed to GitHub due to their size. Finally, there is also a quite lengthy .gitignore file in the repository, that, among other things, prevents unnecessary log files from the LaTeXproject report from being pushed.

§ 2.2. Unix Tools. As a first step in the project work, the Unix commands `head` and `less` are used to determine how the data is formatted and how each column looks. After this quick investigation it becomes easier to know which the next steps are and what functionality is of interest for the final program. Among other things, we can see that the first two columns which contains the contig ids consist of really long strings which will become a memory problem later on. Another problem are the many columns that contain information that is not relevant to the problem at hand.

§ 2.3. The DataTrim Class. In order to deal with the large data file, and to trim it down to something that will be easier to work with, we need a new tool. The Java program DataTrim takes care of this crucial part, by changing the contig identifiers to integer values, removing the columns 3-5 and 9 are removed, and removing lines which represent completely overlapping contigs. This is done by looping over each line in the data file, writing the desired values to a new file, and checking for complete overlaps. Hence, at runtime, no more than a single line of the huge file is being processed, so there is no risk of the program consuming too much memory. To run this program, execute the command `java -jar datatrim.jar filename`, on the command line. This will generate a new file with the prefix 'trim_', and it takes only around four minutes to trim the given data, which can be determined by prefixing the command by the Unix command `time`.

§ 2.4. The GenomeGraph Class & DisplayData Script. After the data has been trimmed it lands at a more reasonable size of "only" 1.4GB, and we can move on to the final step of processing this information and investigating the resulting graph. A new data structure, called GenomeGraph, is created for this purpose. This structure works similar to a regular Graph data structure, but it also supports a few specific tasks that are of interest for this project in particular. Firstly, it is possible to add information about the contigs in this graph, since each node is also stored at the first place in its dedicated adjacency list. Normally this would imply that the node has an edge to itself, but the GenomeGraph class is designed in such a way that this just makes it possible to store and access information from this element. In this case, only the length of the contig is stored, but this can easily be extended should the need arise. Most methods in this class works trivially, and there are no particular remarks on time nor space complexity to be made. Note that it is possible to loop over all contigs in the graph, but this works by looping through the adjacency list so it works in linear time and does not rely on the graph being connected.

What does need a note on running time, however, is `getComponents` and its helper method `DFS`, which is used to return the number of connected

components and their sizes. It uses a recursive depth-first-search approach to iterate to the graph and find the connected components. This means that it has time complexity $\mathcal{O}(|V| + |E|)$, where V is the vertex set and E is the edge set of the graph. Hence it runs in linear time proportional to the size of the graph, which is good in our case with the large data file. One point to consider however, is the space complexity, and the number of recursive function calls that needs to be made. We need to store an arraylist of size $|V|$ to keep track of which nodes have been visited. To solve the problem of the many needed recursive calls, which is not supported by the JVM, it is possible to append the flas -Xss4g when running it from the command line.

DFS is a reasonable choice for this purpose since it works through the graph by "exploring" how far it is possible to reach from a given contig. When it is not possible to go any further, we have reached the end of the component and retrieve a component. Since we also keep track of which nodes have been visited, we can simply move on to the next one and run the algorithm one more time.

The `GenomeGraph` class is used in the class `GenomeIO` which is used to actually read a trimmed file and then return an even more trimmed down text file, the result of which can be piped to the Python script `DisplayData` that displays this information graphically. The `GenomeIO` class works by reading the file line by line, add the contigs to the Graph data structure, and add the overlaps. Again, this also works with only one line at a time performing trivial operations which works in linear time.

After the information has been added to the data structure, its methods can be used to determine the degree of each node in the graph, along with the components and their sizes, so its time complexity depends on the implementation of `GenomeGraph`.

Finally, there is the Python script `DisplayData` which also takes a file and reads it line by line, of which there are only four this time. After running `GenomeIO` we have a file in which the first line corresponds to the number of contigs in the graph, the second line corresponds to the degrees of each contig, the third to the number of components, and the final line corresponds to the size of each component. This means that the second and fourth line of this graph are very long. The script reads the lines and uses a counter to count how many times a specific degree/component size occurs so that these can be displayed graphically.

To run both `GenomeIO` and `DisplayData` run the command `-Xss4g -Xmx4g -jar genomeio.jar filename | python DisplayData.py`. This will display a png-file with the results and also save these results in the current directory. All in all, it takes about four minutes to run this command, which also can be tested using the Unix command `time` as before.

SECTION 3

Results

The results of running the program are summarized in Figure 1. As it turns out, there are many contigs with large degrees, and similarly, there are many components that are very large, but the number of such contigs and components is largely insignificant compared to the number of contigs with smaller degree and components of smaller size. Hence it is almost impossible to get a good look at these contigs in an informative way. However, as we can see in the figure, the absolute majority of contigs have a degree less than 10, and the size of most components is less than 10. There is around 2.5 million contigs with degree 1, and 500,000 components with only two contigs.

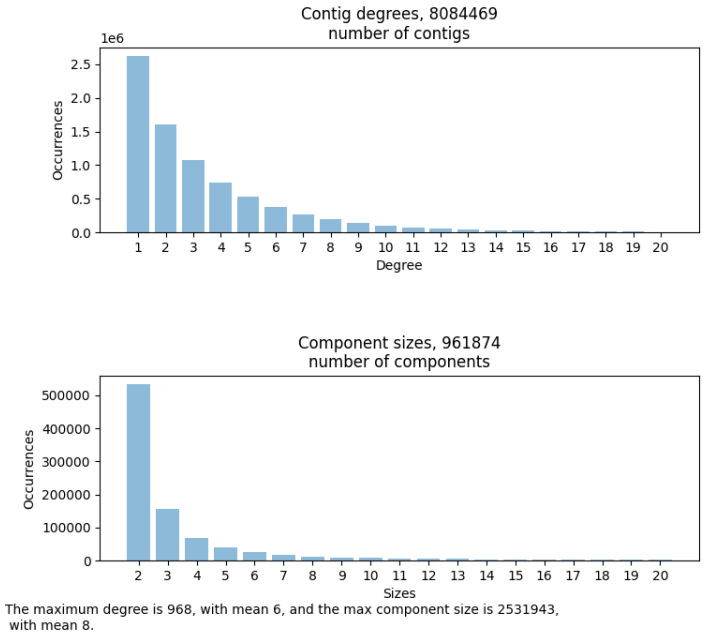

Figure 1: Overview of the contig degrees and component sizes.

The largest contig degree is 968, and the larges component has over 2.5 million contigs. We also see that the mean degree is 6, while the mean component size 8, which confirms that most values are small with a few larger ones that increase the value.