A genome assembly graph
Project Report

Giulia Grasso

May 2020

Contents

1 Introduction

This project consists in analyzing a dataset containing data regarding DNA's segments, called **contigs**.
In order to do this, it is created a graph G, based on the given dataset, and analyzed the following aspects:

- the node degree distribution of G;
- the number of components of G;
- the component size distribution of G.

2 Unix Tools

The given database is about 7 GB, hence it is important to create a sample and do some tests on it. For this reason, it was used the Unix command **head**, which allows to take the first lines of the database. In particular, it was used with the following command:

$$head \quad -2000 \quad db.m4 \quad > \quad sample.m4$$

which stores the first 2000 lines of the database into a file called **sample.m4**.

3 Use of Git and GitHub

Git was used to track all versions of the project with daily commits.
The description of the used commits is based on *commits conventional*, with the following structure:

- **feat(FEATURE)**: feature description implemented;
- **chore(SUBJECT)**: minor improvements related to a subject;
- **doc(DIARY)**: description of the related documentation added;
- **tests(DATABASE)**: any commit about the database or sample database.

GitHub has been used to store the project in the cloud and the related git log and software versions.

4 Pre-processing

Firstly, it was observed that the given database is about 7 GB, hence it was implemented an algorithm in order to filter the database, creating a smaller database with only the data which are relevant for the project, i.e. the contigs' overlaps which are not entirely contained.

The implemented algorithm, first was tested on the sample and then, after checking its functionality, was used into the entire database, producing a file of 2.6 GB in which there are **8 millions nodes** and **22 millions edges**.

Moreover, during the pre-processing phase, the identifiers were mapped with auto-increment ID, hence there are been added two columns to the database and created a new file with only these two columns.
This file represents the adjacency list of the graph and it will be used by the class GraphIO, which was implemented for lab5.

5 Node degree distribution

It was implemented an algorithm which calculate the node degree based on the formula:

$$\frac{node\ \ degree}{max\ \ degree}$$

where the degree of a node is the related node's amount of edges, which means that:

$$P(node\_degree) = \sum \frac{edges\ of\ a\ node}{max\ \ degree}$$

Note that in the sample database, the maximum degree is **968**.

Degrees Probability	**Nodes**
0.0	7976587
0.1	61011
0.3	11738
0.2	30308
0.4	3466
0.6	301
0.5	716
0.7	210
0.8	88
0.9	28
1.0	5

Table 1: Nodes degree per degree probability in the entire database

Note that "0.0" stands for all the nodes with degree probability less than 0.0*, which is not 0.
To analyze the results of the degree distribution in the sample, it was implemented a python script to show an histogram with the node degree distribution (show_degree_dist.py).

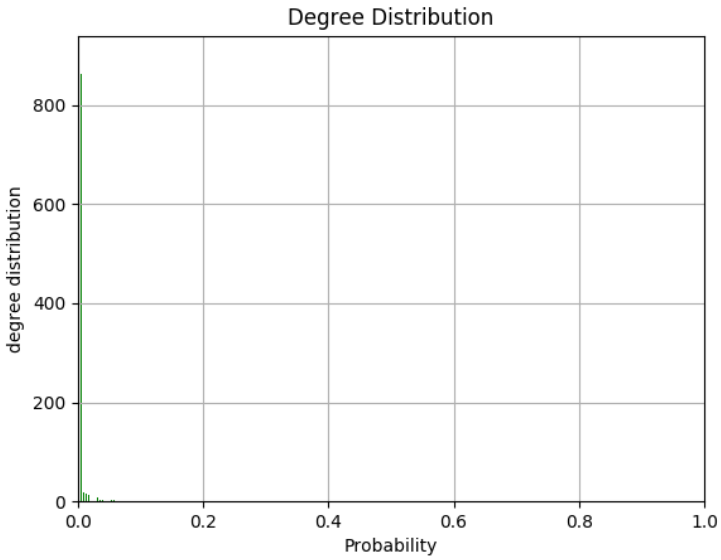

Figure 1: Histogram of degree distribution in the sample

It is possible to notice, from the histogram, that several nodes (of the sample database) have a low degree probability, which is similar to the entire database degree distribution.

6 Graph components and component size distribution

In order to compute the number of graph components and its related size distribution, there were considered the two algorithms **BFS** and **DFS**.

Based on several researches and detailed studies, it is given that the two algorithms are almost equal in terms of time and space complexity, hence the main difference is given by the structure of the graph.

To do a more accurate study for better understand which algorithm is more suitable for the given database, it was decided to apply both algorithms and

checked the time complexity and the memory that is used for both.
Using the BFS, implemented in previous lab, and a recursive version of DFS with the sample database, there were obtained the following results:

	BFS	**DFS**
time	3 ms	3 ms
memory	5424712	5424688

Using the recursive DFS with the entire database, it was found the stack-overflow error, that was solved implementing the iterative version of DFS.
Hence, applying both algorithms in the entire database, there were obtained:

	BFS	**DFS**
time	6037 ms	4903570 ms
memory	3554873344	2230251168

From these results, it can be observed that with the given database, BFS is faster than DFS but using more memory, while DFS uses less memory but it is **hugely slower**.
Thus, the number of components which is found is **961874** and the related component size per graph of all components is stored in the file *component_size_entire_DB.txt*.
Furthermore, it was implemented a python script to show the graph components size distribution in a histogram:

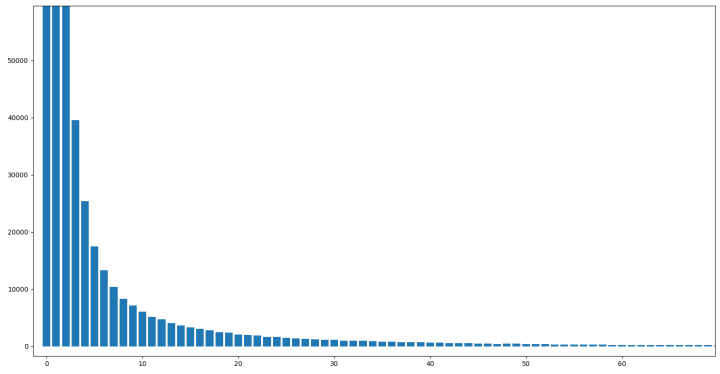

Figure 2: Histogram of graph component size distribution in the database

As a result, it can be evinced that most of the components have 1-3 as size, and, by the increasing of their node size, there is a decreasing of related graph component in the graph.