COSC 320 – 001

Analysis of Algorithms

2022/2023 Winter Term 2

Project Topic Number: #2

Milestone 2

Group Members: Sam Garg, Samuel Street, Jimmy Ji

python for making the nice files (I made the directory Nice files just manually, but that doesn't really take much time)

Python bit for making the data given into a document that in read nicely in java

Abstract. One paragraph of your achievements in this milestone. This should be included for all milestones.

We converted the csv file we were given into one that was more easily read in java using python. Then we scanned that file into Java. Then we implemented RKA.

Dataset. Include the details of the dataset.

https://docs.google.com/document/d/1ITbNNOqqp3W39R5hJJsBu_87kOjblLezIKjNMcGPuoY/edit?usp=sharing

original data from https://ubcca-my.sharepoint.com/personal/fatemeh_fard_ubc_ca/_layouts/15/onedrive.aspx?id=%2Fpersonal%2Ffatemeh%5Ffard%5Fubc%5Fca%2FDocuments%2FCourses%2FCOSC%20320%2D2022%2D2023%20%28Jan%202023%29%2FProject%2DDataset&ga=1 provided by proff

basically we put the data into a data frame in python and then we selected the contents out of the data and then we used a very easy to use delimiter ;;; to easily separate the data in java

Implementation. Explain how you implemented the algorithm and tested it. All the subtle details should be included. This is just an explanation and you do not need to copy paste your implementation here. It can be as short as one paragraph. Include links if required.

Python is used first to clean up the csv files extracting the main text and save to individual text files, the Java is used for the rest. Our Java program takes two inputs, one is the string to be tested, and another is the directory to the folder with all the text files. The built in functions like replaceAll and split are use to clean up the strings to words only. The words are than individually hashed and saved to a arraylist. Then the program go to the folder and start reading the files in a loop, the content of each file is read and stored as a single string. This string will be cleaned and hashed just like the test string. After that hashed file will be compared to the test hash like described in the last milestone. The total number of more than three repeated words are counted, if the amount of repeated words in the test string is higher than 24% than this file will be added to a arraylist. The process is than repeated until all the files has be checked, then it output the name of all the potentially plagiarized documents.

To test this algorithm, and time part is added at the start and at the end to measure the total time taken. The loop for reading the field has also been modified to only read the set amount of files. So the amount of files compared to can easily be changed.

The full code can be found at this link: Code for Milestone2

Results. Include the plots and the interpretation of the plots as input grows. Compare it to the big O function of the running time. For example, if your algorithm runs in $O(n2)$, show the graph for the $n2$ function in same plot as well. Explain if this is what you expected, and how the implementation of your algorithm might have affected the constant values. How the choice of data structure might have affected this result?

From the last milestone we expect the time to be $O(nm)$ where n and m are the length of the test input and the number of the documents to compare to, respectively. Because the input m is constant, the only value that change is the n. This means the time should grow linearly. A few value is tested with the amount of time taken measure, and the result is as following:

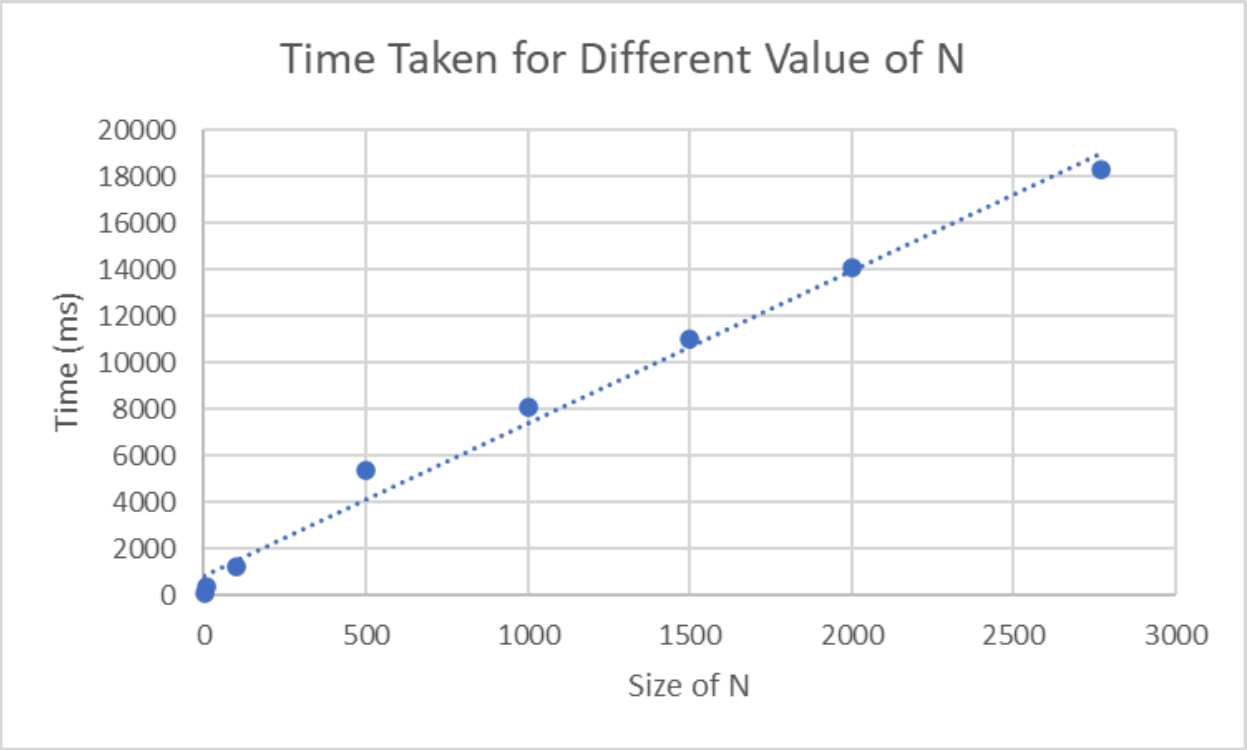

Figure 1. Plot of the time compare to the size of N

As shown in figure 1, the time growth almost exactly matched the linear best fit shown in the dotted line, so the growth is as expected.

Arraylist are used for most of the cases because we cannot know the exact size of the inputs, so we need a list that can expand if needed. The add operation is usually $O(1)$, but in the worst condition where the arraylist need to expend it could be $O(n)$ where n is the size of existing elements. This could slow the algorithm down slightly but judging by the result it does not happen enough to make a significant difference.

Unexpected Cases/Difficulties. If you encountered any issues for design and implementation of your work, include it here. Also, mention the solutions you have considered to alleviate the issue.

The data was difficult to parse in java so we made a program in python that made new files that were easier to read. Then we found some of the characters in some of the lines were not Unicode and so we decided to skip those lines in the output. Then, although this should have been expected, we did not, some of the files were empty, we skipped these files. Then we thought about grammar which Jimmy solved by only looking at letters and numbers and spaces. Then we thought about upper and lower case which Jimmy also solved by converting all of the words to lower case, that was a pretty good idea. Jimmy also fixed a lot of the errors and pretty well re-wrote ~90% of the java portion much better than how I had it only without the errors

Task Separation and Responsibilities. Who did what for this milestone? Explicitly mention the name of group members and their responsibilities.

Samuel S.:did python bit, and wrote a small portion of the java bit that was used, the thought of a couple of problems that could be faced
Jimmy J.: did 90+ percent of the final java bit and fixed the grand majority of the bugs in java, also thought of several of the possible problems that could be faced. I can say with certainty that Jimmy deserves most of the credit for this one (Samuel S.) I was working on it and I think I might have got my version to mostly work prior to the deadline, but I don't know I would have gotten all of the documents to scan properly. Also, JImmmy did the scatter plot section and implementation section.
Samuel G.: helped with the discussion for how to implement the java bit, wrote some code for java that we did not use, but did work using a brute force method, which was somewhat of a backup

One thing we did do by mistake is we made it so that if 24% of the test file was in one of the corpora of files being tested then the tested file would be considered plagiarised, we basically flipped the percent and realized it late, so the program counts the number of the document which contain >24% of the document rather than if the document contains >24% copied material. This will be an error that will be addressed in the next implementation milestone for the next method

Also, another semi serious mistake we made is that last milestone we actually separated the tasks some and this time we really, really failed to do that and we kind of all just sort of made portions of the program with massive overlap between what we all did. This will also have to be addressed