Aayush Borkar 23B0944, Niral Charan 23B1045, Yash Vikas Sabale 23B1043

1 | Introduction

This report is regarding our plagiarism checker project done in a team of three students. In the first phase we utilized suffix automata for exact matching getting a time complexity of about $O(n^2 \log(k))$ where k is the number of distinct tokens possible. For approximate matchings in phase 1 we broadly considered the Levenshtein or edit distance between two approximate matching candidates. In phase 2 we proceeded to build an interface around the plagiarism checking logic developed earlier. For improved efficiency we used two threads, one dealing with submissions and timestamps only (to get precise timestamps and avoid delays related to previous submission's processing affecting the timestamp of current submission). Another thread is dedicated to the actual logic of plagiarism detection. Our class is capable of detecting both direct plagiarism and patchwork plagiarism of a file from multiple files. As for phase three one idea to try to "beat" this type of check would be to insert code blocks everywhere that do nothing (such as a++; a–;) between two lines of the original code.

2 | Phase One

For Phase one of the project we were supposed to mainly achieve two things one was exact pattern matching and the other was approximate pattern matching.

2.1 | Exact Pattern Matching

For exact pattern matching we utilized a data structure called a suffix automaton. We mainly followed the implementation here. Conceptually a suffix automata is the minimal automata that accepts all suffixes of a string S.

2.2 | Suffix Automata Implementation

Naturally all walks in this automata correspond to some sub-string of the original string. Another interesting thing is that these can be constructed in linear time. The algorithm reads the string in a stream and as it reads new characters it appropriately expands the automata. This is particularly useful because it means we can extend the automata seamlessly if needed (instead of having to build the whole thing again)

For building the automata we had a struct which corresponds to a state, with three internal variables one for the length of the current substring, map of ints to ints corresponding to the transitions (token to the transition state) and a suffix links to the next suffix of the string (which are used for more efficient traversal later).

The SuffixAutomaton Class implements the actual logic for creating this automaton with a member function for adding an int to the automata (we add the ints corresponding to the tokens one by one to make the automata).

Now suppose we want to find exact matching between two strings S and T. We began our logic by making the suffix automata for S. Now we start feeding T for each prefix of T we find the length of the longest suffix of that prefix that is in S.

This logic is done in the length_subarrays function which utilizes the suffix automata class and feeds T into it to find matching subarrays

Then each time the length goes beyond our specified length we save it into a priority queue lengthwise (storing length and start index).

Now we start building another suffix automata using the strings in the priority queue in their length order. This is to avoid duplicates being counted. If a later string matches with an earlier one then naturally we do not count it in our exact matches.

As for the time complexity of this whole procedure building the suffix automata for S is in linear time, feeding the second string T into that automata piecewise is an $O(n^2)$ procedure, similarly the checking logic can go upto $O(n^2)$ (if for example the two strings are identical). Hence our overall complexity is $O(n^2)$. Practically it seems to run quick enough to give us the result somewhat quickly

2.3 | Approximate Pattern Matching

For approximate pattern matching we utilized the Levenshtein distance between substrings. The Levenshtein or edit distance is a very natural measure since it is robust to the insertion of small strings between two matching strings corresponding to small patches of code interleaved within bigger sections of copied code. The edit distance counts the number of edits, insertions and deletions needed to convert one string into another. As for the actual algorithm we followed a dynamic programming based approach taking inspiration from this paper on similar topics.

The logic is dictated in the Levenshtein search function which finds the closest approximate match for its two inputs P and T. It takes another parameter alpha which is one minus the percentage of match we need it can be tweaked to change the percentage we with to match. The start2 is the index of P from which we start searching.

The function maintains a dynamic programming array for the Levenshtein distance and modifies it appropriately. We need to run it for all values of start2 in order to get the longest length of match.

As for the time complexity each time we run the function it takes $O(n^2)$ time for tokens of length n. While it is run for each start index hence overall the time complexity of the approximate matching is $O(n^3)$

Finally the match_submission function calls the other two functions an appropriate number of times and formats their outputs appropriately for the return vector.

3 | Phase Two

In phase two we implemented the plagiarism checker class and its methods.

3.1 | Multithreading

Taking inspiration from a few stack-overflow posts we understood the broad utilization of thread-pools and the std::mutex class.

It creates a queue which is shared between the threads. So we utilize two threads to run this application. One which deals with the time stamps of the submissions and another which deals with the tokenization, as well as the plagiarism checking logic. The first token adds the submission into the jobs queue which is processed later. This distribution ensures that the time stamps are accurate and the class is always ready to receive new submissions. The use of the mutex avoids data races i.e the modification of the same data by different threads and the issues that it could cause (the jobs queue). We also created a struct called exsubmission_t which contains the original submission_t along with the timestamps, tokens and other flag variables and a function for flagging (avoiding double flags). This struct enabled us to write more streamlined code. We kept track of the previous submissions using a vector of exsubmission_t's called submissions the check_plagiarism function maintains this submissions vector. We have a queue of exsubmissions called jobs. The add_submission function adds to this queue. Since two different threads will be accessing / modifying the queue we use mutex to avoid data races. The add_submission function adds the timestamp and then pushes the submissions into the queue. The queue keeps track of the order in which check_submission needs to be called.

3.2 | Implementation of Multithreading

One of the threads made is called worker which does the plagiarism checking logic. The thread_loop is the function called by the thread, it has a loop so it is always adding submissions to the queue for checking as they come in. We ensured that the thread sanitizer passes every time. One of the issues that we faced previously with thread sanitization is that we were not initializing some member variables in the constructor but upon fixing it the sanitization became fine.

3.3 | Implementation of check and add submissions

In the check_submission function which calls the length_subarrays function (from phase 1) first tokenizes the current submission then the current token stream is compared to all previous streams to detect plagiarism.

It has the appropriate logic, for both direct copying (greater than 10 size of direct or 75 or longer rough matching) or patchwork plagiarism (more than 20 individual plagiarisms).

Finally Based on that and the time between the submissions it flags either one or more of them

In the add_submission function we set the appropriate timestamp and then move the submission, its timestamp into the thread