Plagiarism Checker

CS293: Data Structures and Algorithms Lab Project

October - November 2024

1 Introduction

Hello, students! Plagiarism is a rampant issue that plagues the academia. Plagiarism is the academic malpractice of submitting non-original work (copied from other sources or from another paper) as one's own. This also applies to homework essays, documents, or code submissions, as well as projects, not just research papers.

As part of this project, you are required to implement a plagiarism checker that checks code files. This is split into three phases, one where you design a checker for **two** codes at a time (accuracy matters), the second where a bulk-scale checker is designed (efficiency matters) and the third which is somewhat of a hacking phase.

In order to avoid complicating things for this project, we have already written a parsing code that generates a **stream of integer tokens** from a `C++` file which is given to your functions. Also, we do not expect you to check for citations; just evaluate the works as such and flag them whenever necessary.

1.1 Types of Plagiarism

Plagiarism occurs in several different forms, be aware of them and do not commit any of them!

1. **Global plagiarism**: copying the entire or most of the work of someone else and claiming it as your own. This is outright cheating and typically attracts the most grave penalties.
2. **Direct plagiarism**: also known as **verbatim plagiarism**, this involves copying paragraphs or portions of someone else's work. For code files, this involves copying functions, classes or other parts of the body. Even though some part of the submission is original, this is still plagiarism unless citations are provided.
3. **Paraphrasing plagiarism**: this is very similar to the above type, except that some words or phrases are changed, and no reference or attribution to the original is provided. This is still plagiarism. The coding analogue would be changing variable names and slight order modifications, but the tokens after parsing do not get changed significantly.
4. **Self-plagiarism**: did you know that copying another of one's own work and submitting it afresh as though it is a new work, without proper attribution, is also considered plagiarism? Yes, it is equally offensive as other types, the reason being that the number of papers or submissions made does not reflect the amount of content originally created by the same person.
5. **Patchwork plagiarism**: also known as **mosaic plagiarism**, this is harder to detect, but this is a form of plagiarism wherein content or code copied from multiple sources are interwoven with original content and no citations are provided.

In this project, you will be creating robust plagiarism checkers that can detect some (or all) of the above types of plagiarism. The checkers will be tested against a variety of testcases, including some that are specially designed to bypass plagiarism checkers.

1.2 Project Structure

This project is split into three phases where you'll design, scale and improve your plagiarism checker. Make sure to read this document completely and carefully before starting the project.

The phases are designed to be somewhat independent (especially Phase 3) and the submission deadlines as well as the evaluation criteria for each phase will be separate.

Helper code is also provided for Phases 1 and 2 and files to be modified are explicitly mentioned in the instructions of those phases. Some general instructions (including commands to set up the environment) are also provided at the end of this document.

2 Phase One: Barebones Checking of Two Submissions

You will be provided **two** submissions, both of which are *vectors of integers*. Each integer is a token. Consequently, both submissions are collections of tokens in order that form the respective code after being parsed.

Do not worry about the demarcations of functions, comments, and code semantics; just look for similar patterns!

2.1 What kind of matching patterns should you detect?

Firstly, a pattern is a series of numbers in a particular order. Of course, the order matters as code without order (unless you're talking pure functional programming – even then, the order of tokens matters) is not of any use. You need to look for **matching subsequences** between the two sequences of tokens.

Of course, there is no point looking for sequences of length 1 since there are only a few individual tokens, and there are bound to be a lot of matches. The same holds for matches of very short length. For short patterns (10-20 tokens each), you need to look for *accurate matches* – all tokens are to be identical in the same order.

For larger patterns (above 30 tokens each), look for *approximate matches*: you are to consider two patterns as matches if there exists a subsequence of at least 80% the length of the longer of the two. This automatically implies that the patterns that are matched should roughly be of the same length.

2.2 Alright, what should be reported?

You are to report **five** values in order, which gives the caller an idea of the degree of match between the two code files and the most significant match.

- The zeroth return value is a flag that is 1 if the two submissions are flagged as plagiarised, and 0 otherwise
- The first value should be the total length (number of tokens) of all pattern matches detected, of lengths around 10-20. Longer pattern matches will count as multiple pattern matches, but that doesn't matter since you report the total length.
- The second value should be the length of the *longest approximate pattern match* that you were able to detect. Here, we recommend you look only at long pattern matches (30 or higher tokens); return zero otherwise.
- The third and fourth values are the start indices of the pattern you found above in the first and second files, respectively (start index of the pattern in either vector of tokens).

Ensure that in the first value, you do not double-count patterns, i.e., all patterns caught should be present as accurate matches in both files, and no two of them should overlap in either file. This not only takes care of direct/global plagiarism but also of patchwork plagiarism – when the offender copies lots of sections from the other work and intertwines them with original content.

Looking for approximate matches as well (for the longest match) ensures that additions/modifications of a few statements are still caught (variants of paraphrasing plagiarism), and the very fact that you look at tokens and not code files directly ensures that merely changing of variable names does not evade your detector.

For the flag (zeroth value), try to be as accurate as possible. Ensure that your code correctly identifies pairs of files that are plagiarized and those that are not. You are free to decide the threshold for the number of short and approximate pattern matches that should be caught for a pair of files to be flagged as plagiarised, but ensure that it is reasonable and avoids too many false positives and negatives. This is important for phase 3.

2.3 Your task

You are supposed to modify and submit **only** the file `match_submissions.hpp`, implementing the provided method `match_submissions` provided, and return an `std::array<int, 5>`, containing whatever is mentioned above. Do not write print, read/write file or log statements.

Write code as efficiently as you can since some submissions (a few thousand tokens) will take a long time. It is okay if you miss some small pattern matches or if values slightly differ from expected. Grading will be relatively lenient, and you will not be penalized for small differences in the length or start indices of the longest pattern match detected, provided these are approximately around the expected values.

3 Phase Two: A Full-Fledged Plagiarism Checker

Alright, now comes the challenging part! The method of pairwise checking of submission files is not scalable to real-life inputs, particularly when a large number of students submit codes in real-time and expect results for plagiarism checks as soon as possible. We need something more efficient.

3.1 The Input and Output API

Real-life codebases utilize *modularity* and *object-oriented programming*. In other words, rather than simply passing around pointers, arrays, and strings to lose global functions, there are multiple *objects* that interact with each other using special methods.

The methods provided by an object for other objects to use (to write/overwrite onto, get something done from, or simply read data from) itself are known as its *API*s. They describe how a given object works (common to all objects that are *instances* of a particular *class*), encapsulate and hide the internal working from other objects without hampering their functionality. How the object is implemented internally is abstracted out, but other objects can assume the correctness of all the APIs, from creation to all methods to destruction.

In this case, there are **four** classes of objects: one for students, one for professors, one for your checker, and one for submissions (this is technically a *plain old datatype*, or just an aggregation of multiple objects – the code file name, a pointer to the student who submitted and a pointer to the professor).

Note: `struct submission_t` has the code file as a string (representing its file name), and *not the stream of tokens* as in phase one. Meaning, you will have to instantiate a `tokenizer_t` object and call the `get_tokens()` method on it to get the tokens.

You need not worry about the working of the classes `student_t` and `professor_t`, except for the fact (given) that they provide a method (no return value) called `flag_student` or `flag_professor`. This is to alert them of a plagiarised submission (pass the submission pointer as an argument).

You also need not worry about the main method, the way professor/student objects are instantiated, the way they prepare submissions, and what they do if they are alerted for plagiarism. Nor should you worry about detailed reports. Do not print log statements; just accept inputs and call the flag API when needed.

3.2 Your class `plagiarism_checker_t`

The APIs provided by your class include its constructor with a set of submissions (again, tokenized; you get tokens as integers). This emulates the real world when past submission code files are added first after the plagiarism checker is ready to check for the plagiarism of new ones against those as well.

There is then the method `add_submission` that provides a pointer to the submission. Remember, this is how other objects (students and professors) interact with your checker, so this should be intact. They will certainly get irritated if they are to wait for hours just to add a submission and get it checked. So, you must use **parallel processing/multithreading**, with all the heavy lifting occuring in background.

Further, each submission added has a **timestamp**. That is *not provided as an argument* explicitly, but you are expected to use the `chrono::time_point` method to store the time in which the submission is provided. This is essential since a submission that copies from a very old submission should be flagged while the older one should not. We recommend you to record the timestamps *before* tokenizing the file, to avoid delays while parsing the file; this also ensures that the timestamp is as close to the actual submission time as possible.

Speaking of being flagged, you need to use the APIs of student and professor classes. Just like we want this asynchronous behavior, you can assume that our implementations of the student and professor class will respond immediately on being flagged and will do whatever they do after that.

3.3 Tips regarding the asynchronous part

You are expected to do multithreading and asynchronous programming for this phase, something you have not learnt officially (and will do in Operating Systems course in the next semester). So, we provide you some tips to achieve the same. `C++` offers `std::thread` to help you out, and threads can be stored as class members or variables as well (but cannot be copied).

In order to synchronize the threads, you are encouraged to use a `std::mutex`, commonly used around a shared object (shared among the threads of your class, invisible to students and professors). The main thread (executed

by the caller) should jsut execute the `add_submission` method and return immediately to the caller(s); the evaluation and flagging should be done by the other thread(s), which should be hidden from outside the object.

Of course, you could also use conditional variables or semaphores – any synchronization method that works is perfectly acceptable – just make sure that there are no deadlocks or data races and that it works as expected!

3.4 How do you check a submission for plagiarism?

Firstly, you need to tokenize the code file (of the submission object) into a stream of tokens. We have the method implemented for you, but you need to create an instance of class `tokenizer_t` with that file, `get_tokens()` from it and then analyze them to report plagiarism if either of the following hold:

- There is an exact match of at least one pattern of length around 75 or more.
- The number of pattern matches caught between them is 10 or more.

Even if a submission is caught for plagiarism, it should be stored in the database since other files that copy from it are still plagiarised submissions (they are definitely not original).

- If the timestamps of insertion of the submissions differ by *greater than or equal to* ***one*** *second*, then only the *later* submission is to be flagged as plagiarised. Call the corresponding `flag_student` and `flag_professor` methods both to do this. In case any of the student or professor pointers is null (student submitted alone, or professor submitted alone), then only the objects pointed to by non-null pointers are to be flagged.
- If the timestamps differ by *less than* ***one*** *second*, then *both* submissions are to be flagged (both students and both professors, except null pointers) in a similar manner.
- The original submissions passed in the constructor are not to be flagged, even if plagiarism is detected among those. Timestamps for those are treated as zero (far behind timestamps for real-time submissions).

In addition, you are also supposed to check for *patchwork plagiarism* across multiple sources: if there exists more than or equal to 20 pattern matches in total, each with a corresponding (distinct) pattern taken off any work under evaluation among all the previous works by timestamps, then it is to be flagged as plagiarised in a similar manner.

None of the older ones are to be flagged, though. Once again, the master set of all sentences is across all submissions that are added up to **one second after** the current submission is being evaluated.

3.5 Your task

As mentioned above, you are to implement the class `plagiarism_checker_t` and are supposed to modify exactly **two** files: `plagiarism_checker.hpp` and `plagiarism_checker.cpp`. Details are clearly mentioned above.

The former is the header file, where you can add your own sub-routines, class member definitions, and helper objects, provided you do not modify the public interface of your class. The latter is the file where you implement the methods (public and protected) of your class. Do not add function bodies in header files, since it will cause linker errors; just write signatures.

For phase two, in order to make detection of matches and storing or finding sentences for matches easier, two patterns are considered matches if their length is 15 or more and if they are exact matches.

Also, you can ignore approximate matches with a few tokens here and there that are different. As evident, the check becomes more lenient since you end up ditching checks for a few minor changes like an extra statement or changed variable names (made by dishonest students who try to evade checkers) for the sake of speed. Accuracy vs speed is always a trade-off in life, but here, focus on efficiency and ensure that all other checks are accurate.

Efficiency is paramount, even more so than in phase one. It is okay if you miss some matches in corner cases or look for slightly smaller match lengths or slightly larger lengths as well; testcases will not be very strict in checking whether your code exactly complies with the above guidelines all the time. Your code should not crash or result in undefined behavior; testing will be strict in *that* sense, though.

BONUS: Can you implement a working, accurate plagiarism checker that is both efficient (not naive check one by one against all paragraphs) **and** matches patterns with small modifications, like in phase one?

4 Phase Three: Gotta Hack 'em all!

Alright, now that you have implemented a (hopefully working) plagiarism checker, it is time to break some!

In this phase, you will be provided with the source codes of some of the plagiarism checkers implemented by your peers in the first phase. Your task is to find ways to **bypass** these checkers, i.e., submit pairs of code files that are incorrectly flagged by some of the checkers.

4.1 What counts as a hack?

There are two types of incorrect results a plagiarism checker can give:

- **False Positives**: When the checker flags a pair of code files as plagiarised, but they are not.
- **False Negatives**: When the checker does not flag a pair of code files as plagiarised, but they are.

For this activity, you are required to *find examples of such cases* for the given checkers. Since we expect that your checkers are somewhat robust, simple operations such as just renaming the variables or changing the order of statements might not bypass most of the checkers. Likewise, two completely different files would probably not trigger false positives.

This is where your creativity comes into play. You can also use the internet to find ways to bypass plagiarism checkers and use those as an inspiration to create your own hacks specific to the provided checkers.

For simplicity, you can pick any problem from a common online judge platform (like Codeforces, LeetCode, etc.) and create a pair of code files for an accepted solution to that problem.

4.2 What do we expect you to submit?

Just for this phase, the **source codes** of detectors (some phase one submissions) along with a *Google Form* will be released at a later date. You will have to submit your hacks one-by-one through the Google Form.

For each hack, you will have to provide the following details:

- List of labels of the detectors you are trying to bypass
- Type of the hack - false positive or false negative
- Link to the problem statement - we will submit the code files on this link to check solutions' correctness
- A pair of `C++` files which trigger an incorrect result for the checker(s). These are the code files that will be submitted on the platform for correctness checking.
- A brief, yet complete explanation of how you bypassed the checkers.

You can (and are encouraged to) submit *multiple hacks* for the same checker(s) provided they are unique and not just slight modifications of the same hack. Hence it is important to provide a brief but clear *explanation* of how you bypassed the checker(s).

We would be passing your submissions through more sophisticated plagiarism checkers to verify if your hacks are actually what they claim. Hence, you cannot submit two completely different files and claim a false negative (or vice versa). Only incorrect behavior of the checker(s) will be considered as a successful hack.

4.3 Scoring for Phase three

Accepted hacks for checkers that are bypassed by multiple teams will be awarded fewer points. However, unique and creative hacks might be awarded bonus points. Hence it is advantageous to submit multiple hacks.

This phase is completely *independent* of the first two phases (apart from the provided checkers). You can work on this phase even if you have not successfully completed the first two phases. No helper code or testcases will be provided, since none are needed for this phase.

Note that the hacks you submit should be your own work. Please do not share your ideas with other teams.

BONUS: Heard of the famous MOSS plagiarism checker? Can you bypass that as well? Submit your hacks for MOSS in the same format as above with the label `"MOSS"` if you can get a low similarity score (below 20%) for plagiarised code files.

5 General Instructions

Below are some things you should have in mind while working on the **first two** phases.

5.1 Getting the libraries to work

We will be relying heavily on the famous `clang` compiler, as well as on the LLVM toolchain, for the parsing of code files into streams of tokens. We (and the LLVM devs) have done the tokenizing, but you should be able to run your code (which relies on that) on your machine.

- For **Linux**: Run this: `sudo apt install clang libc++-dev libclang-dev`. We highly recommend you use the latest software, a.k.a. Ubuntu 24.04 or Fedora 40. If you happen to use trashy operating systems like Windows and do not dual boot, use WSL 2.0 and do the same.
- For **MacOS**: Run `brew install llvm && brew link --force --overwrite llvm`. If you use an Intel MacBook, change the (Makefile) include/library directories from `/opt/homebrew/...` to `/usr/local/...`
- There will be 2 versions of Makefiles provided, one for MacOS and one for Linux. Choose the appropriate Makefile. There will not be any use of CMake, but you should be able to understand Makefiles. Run commands `make <target> -f macos.mak` or `make <target> -f linux.mak` to compile your code.

5.2 What's allowed and what's not?

You are allowed unrestricted access to `C++ STL` library, no questions asked. However,

- For either phase, do not use `C++20` modules or `#include`-s of non-STL files (even extra files you write). Nor should you add custom Makefiles; that will break our autograder.
- You should not add the include `bits/stdc++.h` in ANY of your files. This is a terrible practice frowned upon in the industry for the simple reason that many compilers, including the famous `clang`, do not support that header. Include STL headers one by one; it is also faster in compile time.
- Do not add `using namespace std;`, especially on header files; the reason why this is considered a bad practice is that it results in name clashes between STL and custom methods/libraries.

5.3 Evaluation

The exact rubrics are not going to be shared until after the submission deadline. However, you should be aware that part of the grading will be based on autograder and testcases; the other component being manual grading. Ensure that your code compiles on both `g++` and `clang++` compilers and on multiple platforms the way it is given (you should not modify any files other than the three files you are expected to submit). Otherwise, we cannot guarantee any marks for the testcases component.

Moving on to the manual grading part, we will read your code and evaluate it based on its algorithm and performance. Note that writing obfuscated code or simply code that is hard to read is not only difficult for us but also difficult for your partner (heck, even yourself!) to understand or debug; **you will be penalized if your code is not readable**. Consider the following:

1. Refactor 100+ line monsters into several small functions; each one should do one simple task or call other functions. Your functions should average around 30 lines of code each.
2. Do not indent/nest your code more than 4 levels deep. This limits the complexity of the functions in terms of the cognitive load on the human reading it, and the difficulty in understanding the same.
3. Each line of code should not be more than 100 characters long (including indents). Scrolling horizontally to read long lines on a laptop – not reader-friendly!
4. Any class object, member, namespace, function, or even variables – anything that lives for longer than one single function should have names `verbose_enough` to make their purpose obvious from anywhere in the code, particularly wherever it is used. Do not abuse abbreviations.
5. Do not add comments for every line. Provided you follow points 1 - 4, most of your code should be obvious to understand. Comments at the top of the file should provide a 10000-foot view of the purpose, and comments near functions or object declarations should explain the need for them in relation to the big picture, whenever not obvious.

Try to follow these guidelines, and you should be good to go!

6 Submission Instructions

6.1 Phase one

You will have to submit a file `<rollno1>_<rollno2>_CS293_phase1.tar.gz`. If your team has three members, then submit `<rollno1>_<rollno2>_<rollno3>_CS293_phase1.tar.gz`, on *Moodle*. The deadline is **November 3rd, Sunday** EOD. On executing `tar -xvf <submission>.tar.gz`, it should create a directory with an identical name (minus the tar extension). Inside the directory should be **one** file – `match_submissions.hpp`.

6.2 Phase two

You will have to submit a file `<rollno1>_<rollno2>[_<rollno3>]_CS293_phase2.tar.gz` in *moodle*, which on opening creates a directory with identical name, which should contain **two** files – `plagiarism_checker.hpp` and `plagiarism_checker.cpp`. The deadline for this phase is **November 25th, Monday** EOD.

6.3 Phase three

The source codes required for phase three will be released by **November 7th, Wednesday** EOD. As mentioned earlier, a Google form will be shared for the submissions. You are to submit exactly one hack per form entry. There will be an option to submit the form multiple times. Any team member can submit the form. The form will contain all fields which you are required to submit (including 2 file upload fields). The deadline for this phase is **November 27th, Wednesday** EOD.

6.4 Late Submission Policy

Requests for deadline extensions will not be entertained. For each phase, there will be a **three(3)-day** late submission window. If you submit upto one day after the deadline for any phase, your score will be *80 %* of your otherwise score. This gets reduced to *70 %* and *60 %* for submissions upto two days and upto three days late, respectively. After those three days, submissions will not be accepted.

6.5 Heads up!

Since the project is on plagiarism checkers, it is reasonable to expect that we run your code through **strict plagiarism checks**. Should you choose to submit your work, it should be original. If any piece of code is lifted from the web, a comment should be added near that block as a citation. Grading will be done appropriately.

Whatever happens, do not copy code from other teams. Remember, you are better off submitting buggy code (manual grading marks) or even blank submission (zero marks) than copying from others. This is because in cases you plagiarise while building the checker, you will be sent straight to the D-ADAC.

Just like in any project, remember to keep your code base secret since if another team copies your project, you too get flagged; you definitely **do not** want to take risks when it comes to matters of academic integrity. If you must use a GitHub repository, ensure that it is a hidden repo.

As far as AI is concerned, the project is complex enough that ChatGPT cannot do the entire thing on its own. Do not use ChatGPT except for simple things like looking for syntax help. Use GitHub copilot if at all you should, sparingly (do not let it write entire functions). If you do so, ensure that every bit of code it generates is identical both in function and style to what you otherwise would have written.

If there is any bit of code that you do not fully understand or cannot explain (and it is not something super-complex that you explicitly cited in comments), then it IS very much plagiarism.

6.6 Queries and Clarifications

Any queries or clarifications should be posted on the public *Piazza* threads created for the project. We will not be answering any queries via other mediums. If you have some very specific queries, you can post them in a private *Piazza* thread visible to all instructors.

All the best!