Automating Programming Assignment Marking with AST Analysis

by

Stephen Li

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis presents a novel approach to automatically mark programming assignments. We hypothesize that correct student solution ASTs will be more similar to reference solution ASTs than incorrect student solutions and that their similarities can be quantitatively measured. Our approach first preprocesses the ASTs before computing their tree edit distances. We then aggregate the student's set of edit distances from every reference solution into a final mark for the student. We have implemented our approach in our ClangAutoMarker tool. Our experiments demonstrate promising potential for reducing a human marker's workload but further refinements are needed before its accuracy can be suitable for a live classroom.

Acknowledgements

I would like to thank my adviser Patrick Lam for his encouragement, patience, and guidance throughout my undergraduate and graduate studies.

Next I would like to thank Jeff Zarnett for providing me with previous students' assignment code and marks. I would not have been able to finish my analysis without his invaluable test data.

I would also like to thank my reading committee for taking the time to read my thesis and make suggestions for improvements.

Last but not least, I would to thank my colleague Jon Eyolfson for helping me debug my program and teaching me the various nuances of Clang and the LLVM infrastructure.

Standing on the shoulders of giants

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In many computer science courses, students complete programming assignments to help them learn concepts introduced in class. In these assignments, students often need to read data from an input file, process the data according to the assignment specification, and finally write their results to an output file.

Many of these assignments are evaluated based on correctness through automated testing. To evaluate correctness, the student's output is compared against an instructor provided solution and the number of correct matches determines the student's final mark.

This approach is generally sufficient for introductory programming courses due to the simplicity of the problems. Ideally every student assignment should be reviewed by a human reader, similar to code reviews in the software industry, so that students can develop good coding styles early on in their careers. This also avoids one pitfall of input/output automated testing—ensuring students actually implemented the assignment specifications rather than getting lucky through incorrect or disallowed implementations such as using built-in libraries or generating random outputs.

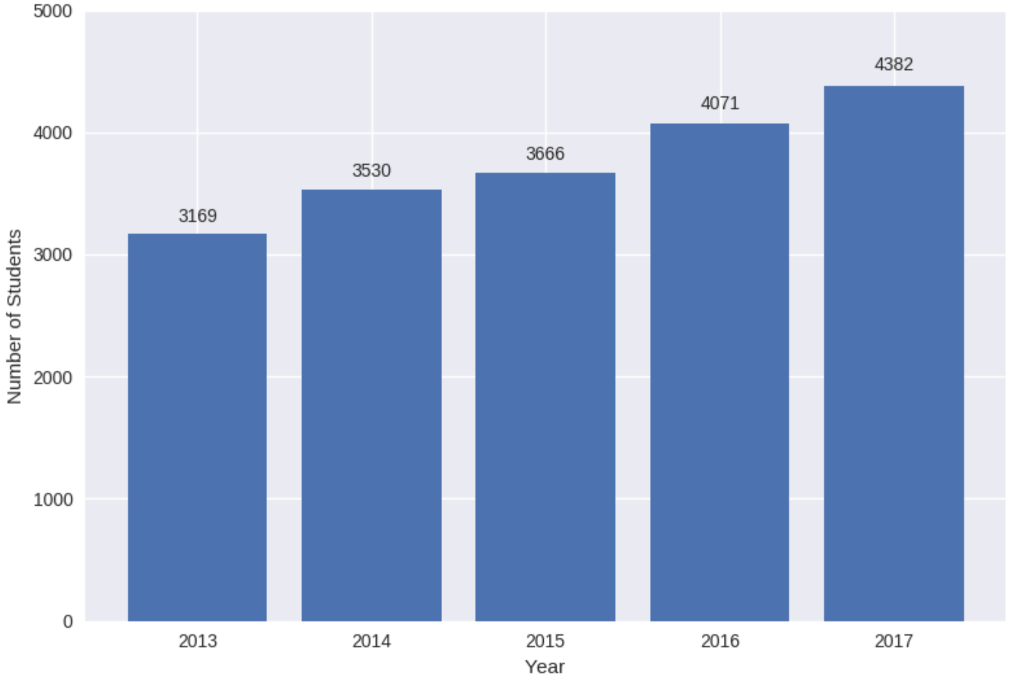


Figure 1.1: The number of students enrolled in the Computer Science, Computer Engineering, or Software Engineering programs at the University of Waterloo continues to rise over the years [18].

In recent years at the University of Waterloo, there has been a continual increase in enrollment in programming-related programs (see Figure 1.1). It is not uncommon for early-year courses to have hundreds of enrolled students. For example, the Fall 2018 offering of CS115, an introductory programming course, has over 900 enrolled students [5].

With such large classes, it becomes near-impossible to manually mark every assignment. In the past, hiring more markers was a temporary fix to this growing problem. However, this solution does not scale well because of the increased collaboration needed between markers to ensure consistency in marking. This is especially important for subjective criteria like code style that often has no “correct” answer. While marker inconsistency may be mitigated by rubrics to some extent, it still requires a deep understanding between markers.

As a result, programming assignments are generally not manually marked until the upper years where class sizes are only a couple dozen students. In rare cases, popular upper-year programming courses such as ECE459, a concurrency programming course, have hundreds of students.

Historically in ECE459, a marker would first compile and run the student’s program, aided with an automated script, to eliminate trivial failures such as failed compilations or incorrect output. The marker would then manually inspect the code to check for correct usage of concurrency constructs.

One of the most common complaints amongst past markers for this course was that manually reviewing code is extremely tedious. In addition, manual marking can be error-prone because student code can greatly vary and can be hard to reason through.

1.1 Motivating Examples

For the purpose of building a prototype automated marking tool, we focus our efforts on two assignments from ECE459 (see Figure 1.2 and Figure 1.3) [1]. In these assignments, the students have been provided with a working serial version of a program written in C. The program is a client that calls a web server using the cURL library. The web server, after a random delay, returns a random fragment of an image. The client program then repeats the process until it has received all the fragments. It then stitches them together to reconstruct the original image. Since the web server is on campus, the response time without the delay is usually under 20 ms. We add a Gaussian random delay to simulate network lag and prevent student programs from overwhelming the server. The students are tasked to rewrite the client with non-blocking IO constructs from the cURL library [2] and again with parallelism constructs from the Pthread library [12]. Since the output of all three programs, the reconstructed image, is exactly the same, only manual code inspection can ensure students have actually fulfilled the assignment specifications.

In theory, a correctly parallelized algorithm should execute faster and use more CPU cores than its non-parallel counterpart. One might think that we should be able to check correctness by limiting the run time and checking CPU usage; however, there are a number of problems with this idea.

Firstly, there were always at least a dozen students in every class with poor programming habits who could correctly use the APIs but still have horrible run times. While these students should be penalized, their remark requests and complaints often deter markers from making deductions. To further avoid student complaints, the markers often err on the side of caution and give students a generous timeout limit that even non-parallel algorithms can pass.

Secondly, if malicious students knew that we were checking for CPU usage, they could potentially create fake threads that busy-wait to run alongside the unmodified serial algorithm.

Thirdly, due to the network delay and the possibility of receiving duplicate fragments, a program could potentially take an unpredictably long time before it can reconstruct the final image.

Therefore, solely relying on automated input/output testing in addition to limiting the runtime or checking CPU usage in this course is insufficient to fully assess students' ability to correctly use the concurrency constructs.

```
int main() {
    // Initialize N non-blocking cURL calls
    for (int i = 0; i < N; i++) {
        curl_multi_add_handle();
    }

    do {
        // Make the non-blocking cURL calls
        curl_multi_perform();
        do {
            curl_multi_wait();
            curl_multi_perform(&still_running);
        } while (still_running);

        // Read the N results
        while (curl_multi_info_read()) {
            if (CURLMSG_DONE) {
                // Fetch an image fragment from remote web server
                ...
                // Write the image fragment to global image
                ...
            }
        }
    } while (!received_all_fragments);
}
```

Figure 1.2: The Non-Blocking IO assignment requires rewriting the serial program to use non-blocking calls in the cURL library. This sample solution shows the key parts of the program that a human marker would look at, such as the cURL library function calls, and their relation to control-flow constructs, such as while-loops and if-statements.

```
pthread_mutex_t mutex;

void thread_function() {
    // Fetch an image fragment from remote web server
    ...

    pthread_mutex_lock(mutex)
    // Write the image fragment to global image
    ...
    pthread_mutex_unlock(mutex)
}

int main() {
    // Initialize mutex
    pthread_mutex_init(mutex)

    // Start N threads
    for (int i = 0; i < N; i++) {
        pthread_create(thread_function);
    }

    // Wait for the N threads to finish
    for (int i = 0; i < N; i++) {
        pthread_join();
    }
}
```

Figure 1.3: The Parallel Processing assignment requires the use of parallelism provided by the Pthread library. This assignment is slightly simpler than the Non-Blocking IO assignment because the student simply has to move the key components of the serial program into an isolated function that is then passed to `pthread_create`. In this assignment, the marker has to check for the correct usage of `pthread_*` functions and ensure proper shared memory protection with mutexes.

1.2 Approach

As described in Section 1.1, the manual work for markers is straightforward and checklist-like. Our goal is to encode these steps into an automated program to reduce the time needed for marking. It is important to note that we are already automating the compilation and execution of student programs to check for trivial failures (e.g. failed compilations or segmentation faults); our ultimate goal is to further augment this step to also include automated code inspection.

We hypothesize that the ASTs for correct solutions tend to be more similar to each other than incorrect solutions. The intuition behind our idea is that, in the context of programming assignments, there are a limited number of ways to effectively use the concurrency constructs. We assume students will not purposely spend excessive and unnecessary amounts of time to obscure their code or deviate from the standard code examples from class or from online tutorials.

As a result, markers generally look at certain library function calls with relation to the program structure and skim over everything else. For example, to check for correct usage in the Parallel Processing assignment (Figure 1.3), a marker would check to see if `pthread_create` and `pthread_join` are inside loops but are not in the same loop. They would generally ignore the rest of the boilerplate code such as parameter validation and garbage collection.

From this idea, we built the ClangAutoMarker tool to automate the manual code inspection step of marking. As its name implies, this tool is built on top of the Clang and LLVM infrastructure [8]. It first leverages the Clang front-end to parse a student solution and an instructor-provided reference solution into their respective ASTs. It then post-processes these tree data structures and computes the tree edit distance[1] between the student solution's AST and the reference solution's AST. We then repeat this process for all the reference solutions; there are multiple reference solutions because most programming assignments have multiple valid approaches. Finally we normalize and aggregate the edit distances into a single mark between 0 to 100 for the student.

However in practice we would only accept an automated mark if it was a full mark of 100; we require manual review for assignments that did not receive full marks because these automated deductions do not have meaningful feedback relevant for a student, other than the fact that they were notably different from the reference solutions.

This distinction does not significantly hinder the effectiveness of our tool because, traditionally, the course staff of ECE459 perfers to have a large number of students getting full marks despite having minor or benign problems with their code.

[1] A tree T_1 can be converted to another tree T_2 through a series of steps (i.e. add, remove, or rename nodes in T_1 until it is equal to T_2). Each step also has an associated cost through a cost function F. The tree edit distance is the series of steps that would result in the minimum net cost to convert T_1 into T_2.

Chapter 2

ClangAutoMarker

The overall goal of the ClangAutoMarker tool is to take N student solution files and M reference solution files as input and generate N marks, one per student, as output.

2.1 Infrastructure

The ClangAutoMarker tool consists of two executables: a frontend and an aggregator. The frontend (Figure 2.1) is responsible for parsing student and reference solutions into their respective ASTs and computing their tree edit distances. The aggregator (Figure 2.2) is responsible for consolidating tree edit distances into a single mark for the student.

We are currently not aware of any techniques for aggregating tree edit distances between student and reference solutions into assignment marks. Therefore, we have done an exploratory analysis in Chapter 3 on various approaches we devised for the aggregator based on our intuition.

The frontend is written in C++ to leverage the LLVM infrastructure and the aggregator is written in Python; the two executables communicate through shared text files. While the aggregator could have been implemented as part of the frontend so that we have a single executable, there are a couple of engineering advantages that make Python more suitable for the aggregator.

Firstly, performing numerical calculations and generating graphs is very simple and easy to implement and debug in Python due to the availability of specialized libraries such as Numpy [4], Scikit-learn [6], and Matplotlib [3].

Secondly, as we will soon discuss in Chapter 3, the parameters and equations in our calculations are determined based on trial-and-error. Furthermore computing edit distances takes hours whereas aggregating edit distances into marks only takes seconds. As a result, the lack of need to recompile the aggregator and recompute edit distances after modifying the aggregator's equations greatly improves our productivity while developing the tool.

Thirdly, by having an external process i.e. the Python script, coordinate multiple frontend processes, we can avoid crashing an entire marking job, which could potentially take hours to complete, due to one malformed student solution or internal assertion error.

2.2 Simplifying the Clang AST

The first step of our tool is to take a student solution file and a reference solution file as input and parse them into ASTs using the Clang frontend from LLVM. After obtaining the Clang ASTs, we then simplify them into custom AST data structures.

From a conceptual point-of-view, AST simplification aims to mimic the judgment process of a human reader. For example, a human marker may

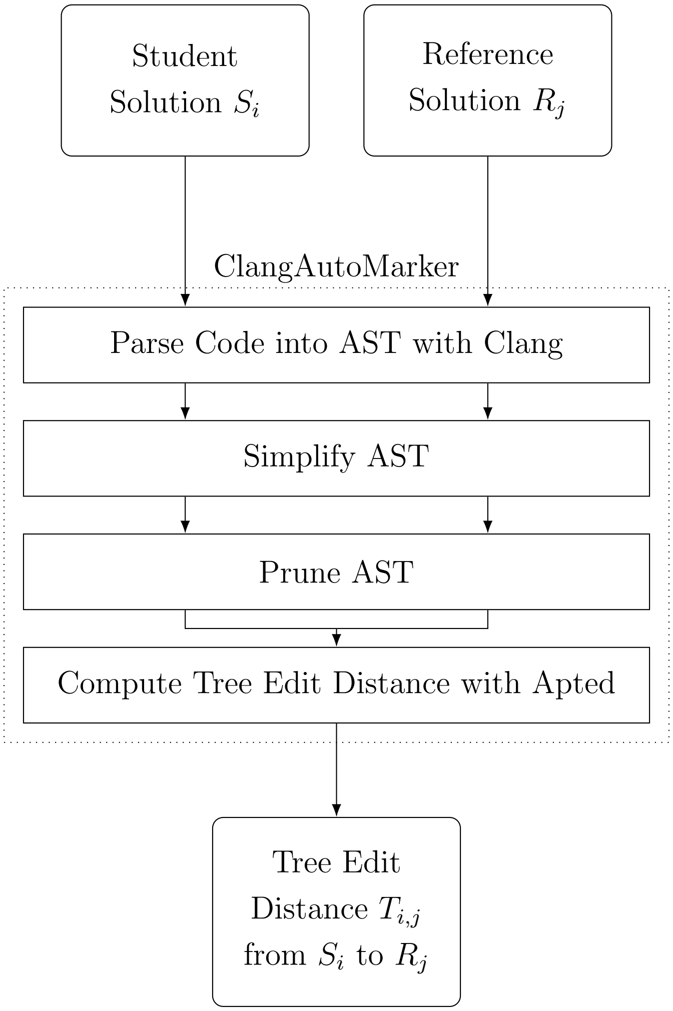


Figure 2.1: This diagram depicts the data flow inside the compiler component of the ClangAutoMarker tool. Each run of the executable takes one student solution S_i and one reference solution R_j as input. The executable compiles the solutions into ASTs, computes their tree edit distance $T_{i,j}$, and prints out the result.

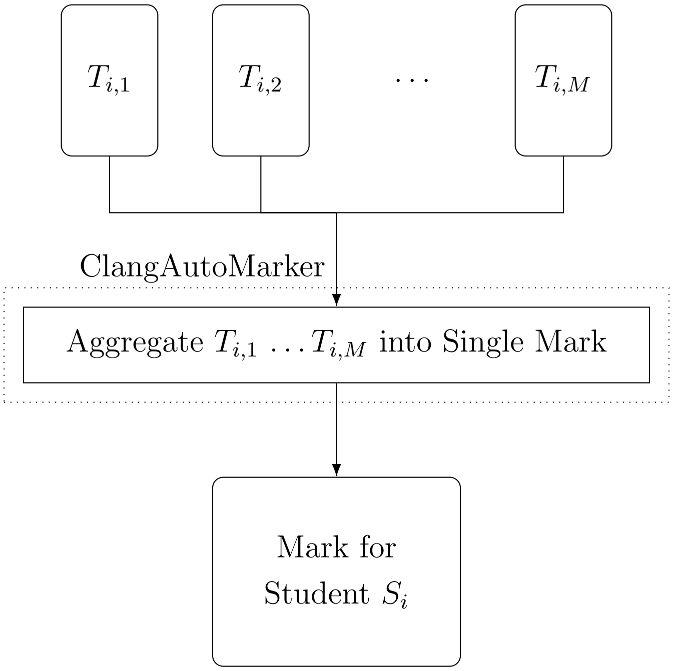


Figure 2.2: This diagram depicts the data flow inside the aggregator component of the ClangAutoMarker tool. For a given student S_i, the aggregator repeats the process in Figure 2.1 for each reference solution $R_1 \ldots R_M$; it then aggregates the results into a final mark. Since there are no data dependencies between each reference solution, the aggregator can execute a new frontend process for each reference solution in parallel.

treat a while-loop, do-while-loop, and for-loop as identical even though their exit conditions are slightly different. By canonicalizing functionally similar sub-trees, we will be able to reduce the edit distances between functionally similar code and ultimately allow them to receive similar marks.

From a practical point-of-view, AST simplification is the easiest way to use the Apted library [26] to compute tree edit distances. To use this library, we need a tree data structure that implements the library's specific interface. We determined that it is much easier to map the Clang AST to a custom AST that conforms to the expected interface than it is to modify Clang's internal data structures.

The Clang codebase is extremely large; the internal interactions and dependencies are often sparsely documented and require insider knowledge to fully understand. As a result, any modifications that we want to make to the tree may potentially break another module. Therefore, by utilizing our own simplified tree data structure, we are free to make large (often destructive) changes to the tree, without worrying about unforeseeable side effects.

Furthermore, the Clang codebase is also constantly changing; by mapping Clang's AST to our internal data structure, we are able to decouple our tool from changes in future versions of LLVM.

Finally, since we are not interested in compiling the AST to bytecode, we do not need to keep as many details. As a result, simplifying our resulting tree data structure also saves memory and computation time.

2.2.1 If Statements

In an AST, an if-statement has at least two children: the condition and the true branch, i.e. a list of statements to execute when the condition is true. The if-statement may also have an optional false branch as its third child.

```
IfStatement
  Condition
    UnaryOperator: NOT
      Variable: cond
  TrueBranch
    CallStatement: bar()
  FalseBranch
    CallStatement: foo()
```

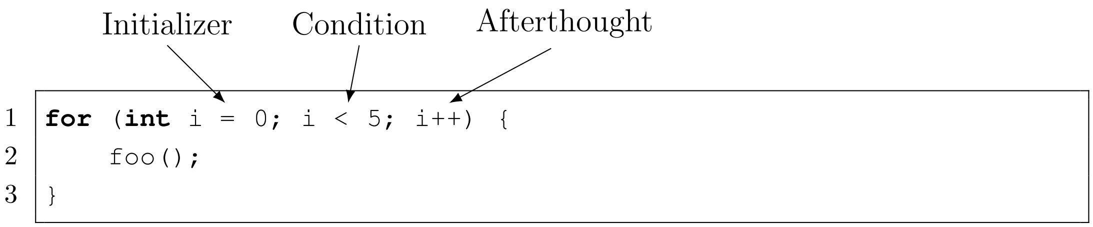
```
IfStatement
  Condition
    Variable: cond
  TrueBranch
    CallStatement: foo()
  FalseBranch
    CallStatement: bar()
```

Figure 2.3: The logical equivalent of negating the condition in an `if` statement is swapping the statements in the true branch with the statements in the false branch.

There are countless ways to declare syntactically different but logically equivalent if-statements. It is infeasible to enumerate all the possibilities to canonicalize them into a common structure. Instead, we focused our efforts on the most common types found in student assignments. From manual inspection of student code, we found that the most common difference amongst logically equivalent if-statements was due to using negation in the condition statement and swapping the true and false branches. As a result, we simplify students' if-statements by stripping out the outermost negations in the condition statement and swapping the true and false branch if there are an odd number of negations, as shown in Figure 2.3.

2.2.2 Loops

There are three types of loops in C: for-loops, do-loops, and do-while-loops. For the purpose of marking, there are no distinctions between these struc-



```
for (int i = 0; i < 5; i++) {
    foo();
}
```

```
LoopStatement
  Condition
    Operator: LessThan
      Variable: i
      Literal: 5
  Body
    CallStatement: foo()
```

Figure 2.4: A “standard” for-loop does not contain complex statements in the initializer and afterthought; it only has a variable declaration in the initializer and unary operator on the variable in the afterthought. To simplify these kinds of for-loops, we discard the initializer and afterthought because they are not important for markers.

tures; we are only interested to know that the student is repeatedly executing some code under some condition. As a result, all three types of loops are simplified into a common structure despite not necessarily being logically equivalent, shown in Figures 2.4 to 2.6.

There is one caveat to this process for “non-standard” for-loops. In an AST, all for-loops have four components: an initializer, a condition, an afterthought, and the body. A standard for-loop is one that only has a variable declaration, often called the “loop counter”, in its initializer component and an unary operator in the afterthought component that modifies said loop counter. Although uncommon, it is also possible to write any number of

```
while (i < 5) {
    foo();
}
```

```
LoopStatement
  Condition
    Operator: LessThan
      Variable: i
      Literal: 5
  Body
    CallStatement: foo()
```

Figure 2.5: Since the while-loop is already the simplest form of loop, we do not need to perform any additional work to simplify it.

```
do {
    foo();
} while (i < 5)
```

```
LoopStatement
  Condition
    Operator: LessThan
      Variable: i
      Literal: 5
  Body
    CallStatement: foo()
```

Figure 2.6: A do-while-loop is essentially a while-loop except that it executes the body once before checking the condition. For the purpose of marking, we simplify it by treating it exactly the same as a while-loop.

```
for (o = initObj(); cond; free(o)) {
    foo();
}
```

```
Assignment
  Variable: o
  CallStatement: initObj()
LoopStatement
  Condition
    Variable: cond
  Body
    CallStatement: foo()
    CallStatement: free()
      Variable: o
```

Figure 2.7: We cannot discard the initializer and afterthought in non-traditional for-loops because they may contain additional non-trivial information. As a result, we place them in logically equivalent positions in the AST: the initializer above the loop and afterthought at the end of the body.

statements in either components, separated by the comma operator; we denote these cases as non-standard for-loops (Figure 2.7). When we encounter these types of for-loops, to preserve the original logic, we move the initializer above the loop node and the afterthought to the bottom of the loop body.

It is important that "standard" for-loops only modify the loop counter in the afterthought component and not inside the body. As the counter is only modified by a unary operator in the afterthought, it is not particularly interesting; therefore, we can ignore it for the purpose of marking.

The simplification shown in Figure 2.4 demonstrates that we have essentially discarded the loop counter from the AST.

2.2.3 Variable Usage

It is obvious that we cannot compare variables between different solutions by name because, with the exception of common patterns such as using `i` for counters, students working independently should not be using the same variable names. As a result, we need another method to identify variables and compare similarities across different solutions.

Recall that our ultimate goal is to compute the edit distance between two ASTs and that we want to minimize the distance between logically similar ASTs. For the purpose of marking, we would like to analyze variable usages with respect to function calls. In our system, a variable is defined to be the set of function calls that read and write to it.

After constructing the initial AST, we scan the tree for variable references. If a variable is referenced in a function call (used as a function parameter), then we record the called function and register a “read” use to the variable. If a variable is on the left-hand-side of an assignment operator from a function call or is the child of a dereference operator inside a function parameter, then we record the called function and register a “write” use to the variable. Figure 2.8 illustrates this process.

In addition, if a variable v appears on the right-hand-side of an assignment operator, then the variable(s) on the left-hand-side inherit all of the read and write uses of v. We then repeat this process until we reach a fixed-point, when every variable’s set of usages stop changing.

Although it is possible to have two different variables (different names in the original program) be considered logically equivalent in our system,

```
foo = bar(x, &y)

// foo : {write|bar}
// x   : {read|bar}
// y   : {read|bar} {write|bar}
```

Figure 2.8: The variable `foo` is on the left-hand-side of this function call so it registers a write usage from the function `bar`. The variables `x` and `y` are parameters for the function call so they register read usages from `bar`. Furthermore, since `y` is passed by reference and we do not perform interprocedural analysis, we assume the worst case and register a write usage to the variable from `bar` as well.

early experiments demonstrated that merging logically equivalent variables discards too much information from the AST and makes debugging extremely difficult. Instead, we opted to preserve the logically equivalent variable nodes in the AST but define their edit distance to be zero.

2.3 Pruning the AST

After simplification, the AST is still bloated with information unnecessary for marking. As a result, the edit distance between ASTs will result in a lot of noise and may not necessarily identify logically similar solutions. This step attempts to find as many of these unnecessary nodes as possible and delete them from the AST.

```
AssnStatement
  ImplicitCastExpr
    Variable: i
  ImplicitCastExpr
    Literal: 5
```

```
AssnStatement
  Variable: i
  Literal: 5
```

Figure 2.9: The Clang parser generates a lot of nodes unnecessary for a human marker. These nodes serve no purpose when trying to compute the edit distance between two solutions. The pruning step deletes nodes that serve no semantic meaning, such as these implicit cast expressions.

2.3.1 Useless Constructs

A human marker is generally only interested in control structures (e.g. if-statements and loops) or statements of interest (e.g. call statements). We denote nodes to be "useless" if they are inserted into the AST by the parser for syntactic purposes but serve no semantic meaning for a human marker. The most common nodes in this category include parentheses, implicit cast expressions, compound statements, etc. Figure 2.9 illustrates the pruning process for these useless nodes.

We remark that an alternative solution is to simply ignore these nodes and denote their edit distances to be zero. However, deleting them from the AST at this stage greatly reduces the computation time for work down the pipeline and makes the AST easier to read for debugging.

2.3.2 Uninteresting Functions

Many function calls such as `malloc` and `free` are not interesting to analyze. These statements are standard boilerplate code in every program and have no effect on the observable behavior of the final output. Although it is possible for these "uninteresting" functions to cause runtime exceptions such as segmentation faults, these problems can usually be detected through automated tools such as Valgrind [9]. As a result, human markers generally ignore these function calls. Likewise, our tool also deletes these uninteresting function calls from our AST.

2.3.3 Uninteresting Variables

Recall from Section 2.2.3 that we define variables to be the set of functions that reads and writes to them. We denote a variable to be "interesting" if its set of reader or writer functions includes an "interesting" function, defined on a per-assignment basis. For example, the Non-Blocking IO assignment (Figure 1.2) would denote functions from the cURL library such as `curl_multi_init()` as interesting.

We first scan the AST for variable declarations and check whether their set of reader and writer functions includes an interesting function; if so, we mark it as an interesting variable. If an interesting variable appears on the right-hand-side of an assignment operator, variables on the left-hand-side are automatically marked as interesting as well. We then repeat this process until we cannot find any new interesting variables. Finally, we delete every variable that is not marked as interesting, and their respective references, from the AST.

2.4 Computing Tree Edit Distance

Tree edit distance is formally defined as the minimum total cost of the steps needed to change from a source tree to a destination tree [11]. A “step” can either be inserting a node, deleting a node, or renaming a node. Clearly the trivial solution is to simply delete every node in the source tree and insert a copy of every node from the destination tree. The goal, however, is to find the sequence of steps that minimizes the total cost.

After preprocessing the ASTs, we are now ready to compute the tree edit distance between the student solution’s AST and the reference solution’s AST. However before we can proceed, we first need to choose a starting point for the root of our trees. In the context of assignment marking, there are two potential candidates for the tree roots.

The first choice is to root our trees at the file level. In this case, we treat the entire solution file as a single tree; functions and global variables are the immediate children of the root. The advantage of this approach is that we only need to compute the tree edit distance once. However, the disadvantage is that it is easy for logically trivial changes, such as different function orderings or function names, to significantly impact the final edit distance value.

The second choice is to root our trees at the function level. In this case, we need to compute a different tree edit distance for each matching function between the student and reference solution. For example, we would have to compute the edit distance between the tree corresponding to `main()` in the student solution to the tree corresponding to `main()` in the reference solution. We then repeat this process for each pair of matching functions between the student and reference solution.

Between these two choices, we decided the function-level approach is less prone to miscalculating tree edit distances between logically similar solutions.

Since the tree edit distance algorithm library we are using does not have a "reorder" operation, we want to avoid the case where the students' functions are in a different order than the reference solution.

2.4.1 Inlining Unexpected Functions

Before we pass the ASTs to our tree edit distance algorithm, we need to consider the potential case of unexpected helper functions created by the students. In programming assignments, some students may create helper functions to avoid duplicating code. This is problematic because the edit distance algorithm does not take the context of the nodes to edit into account.

Research in static analysis and automated marking generally focuses on single functions because inter-procedural analysis is exponentially more complex and time consuming. Luckily in our course, students historically do not use helper functions; of those that do, they mainly use "pure functions", i.e. functions that do not have any side effects and only have one return statement. In addition, emperical evidence indicate that student helper functions are rarely recursive and do not include complex logic. From this observation, we devised a simple technique (Figure 2.10) to inline these helper functions' ASTs into the caller's AST prior to computing edit distances. Should a student solution contain a more complicated helper function, their solution will be designated for manual marking.

2.4.2 Cost Model for Comparing ASTs

The last component to consider before we pass the ASTs to our tree edit distance algorithm is the cost function or model that determines the actual cost of each edit step. A cost model, given an input node, returns the "cost"

```
// int foo(int x) {
//   return x + 1;
// }
FunctionStatement
  Param x
  ReturnStatement
    OperatorStatement: Add
      Variable: x
      Literal: 1

// bar = foo(5)
OperatorStatement: Assn
  Variable: bar
  CallStatement: foo()
    Literal: 5
```

```
// bar = 5 + 1
AssnStatement
  Variable: bar
  OperatorStatement: Add
    Literal: 5
    Literal: 1
```

Figure 2.10: Our inliner first deletes the parameter list in the function root. It then replaces every reference to the parameters with the respective argument from the caller. Finally, it replaces the original function call with the child of the return statement.

to insert a copy of it into the source tree, delete it from the source tree, or rename it to a second input node.

By default, every node has an edit distance of one. Clearly, this is not ideal because some nodes should more important than others. For example, if a student forgot to call `free()` at the end of their program, they should lose a few marks (small insertion cost) whereas if they forgot to call `curl_multi_perform()`, they should lose a lot of marks (high insertion cost). Furthermore, inconsequential node differences such as parentheses should be penalized less, if at all. Therefore, we have added additional analysis to our cost model to ensure similar nodes and structures have reduced edit costs and key function calls have heavier penalties.

Variables

There are two types of variables to consider in our ASTs: external and internal variables.

External variables are variables defined outside the student or reference solution files. These are usually library constants. For example, students may choose to use `CURLE_OK` from the cURL library instead of its hard-coded value of 0 for code readability. To determine if two external variables are "equivalent", we check their types and names. We do not check for literal value because we want to encourage good coding practices such as using library defined constants. Furthermore, we do not need to worry about naming conflicts because the Clang frontend will catch them beforehand.

Internal variables are variables defined inside the student or reference solution files. These include global variables as well as scope-level variables such as loop counters and function parameters. Recall from Section 2.2.3 that variables are defined to be the set of functions that read and write to them. When comparing internal variables, we consider them to be "equivalent" and thus have zero edit cost if the intersections of their reader or writer sets are not empty.

Conditions

There are often many ways to write the same or similar behaving conditions for if-statements and loops. Without relying on SAT solvers, it is difficult to know whether two condition nodes are logically similar. For the purpose of marking, we noticed that simply comparing the set of variables and function calls between two different conditions is sufficient for determining logically similar code.

We define the edit cost between two conditions to be zero if they share at least one “equivalent” variable (described in Section 2.4.2). If one condition has a function call to f, then the other condition must either contain a function call to f or a variable that has been written to by f.

Weighting Key Function Usage

Conceptually, the key aspects to look for when marking should be how students make their function calls to the assignment’s corresponding libraries. As a result, we want to penalize improper usage, i.e. increase the cost to insert the missing call statement to the student’s AST such that it matches the reference AST.

We assign each “key function” a weight on a per-assignment basis. Anywhere that requires a call to one of these key functions will use the associated weight—generally magnitudes higher than other edit costs.

Chapter 3

Conversion to Mark

At this stage in the pipeline, we have tree edit distances $T_{i,j}$ where $i \in [0, N)$ $j \in [0, M)$ for N student solutions and M reference solutions. Our next step is to convert these edit distances into N marks, one for each student. We are not aware of any established techniques for this process. As a result, we investigated several approaches based on our intuition (Section 3.2 to 3.4) and presented our results in Section 3.5.

3.1 Experiment Setup

3.1.1 Test Data

We had student code and marks from 2017 and 2018 classes to analyze. Our plan was to use our tool to generate marks for these classes and compare our automated marks with human-evaluated marks (ground-truth).

3.1.2 Cutoff Points

Recall that ClangAutoMarker's ultimate goal was to compute a student's mark, a value between 0 to 100, by comparing their solution's AST to the reference solution's AST. For a student to earn a full mark of 100 from our tool, the student would need to have structurally-identical code to the reference solution(s). However, this was essentially impossible unless the student had cheated in some form.

Assuming that students did not plagiarize current or past students, we erred on the side of caution and automatically rounded up our automated marks from a certain cutoff point. In our experiments, we tested cutoff points at 90 and 95. We chose these cutoff points from experience: in this range, the students were close enough to full marks that if they were from a human marker, the minor deductions resulting in a 90 or 95 might have essentially been due to the marker's mood and how lenient they were with minor issues. However any marks lower than 90 from a human marker were very likely to be indicative of actual errors in the student program.

We used the following formula to compute the students' final marks. To avoid confusion, we shall henceforth denote the final value to be returned to the student as "Mark" and denote the computed value from our methods in Sections 3.2 to 3.4 as "Score".

$$\text{Mark}_i = \begin{cases} \text{Score}_i & \text{if Score}_i < \text{Cutoff} \\ 100 & \text{if Score}_i \geq \text{Cutoff} \end{cases}$$

3.1.3 Effectiveness

We measured the effectiveness of ClangAutoMarker by counting how many assignments it could successfully automate, i.e. scored above the cutoff points of 90 or 95. Conversely, an assignment must be manually reviewed if we were uncertain about the mark we generated, i.e. scored below 90 or 95. There were two cases where this could happen.

The first case was if the student code had non-conventional coding patterns. It is likely that there will be a few outlier students in every class who write their code significantly different than what we normally expect in our reference solutions. Even if their solution was functionally correct, their AST would have an extremely high edit distance from our reference solutions and thus would receive an undeserved low score.

The second case was if the student program cannot be processed by our tool. This occurred if the student had syntactical errors or if the student's AST deviated from one of our assumptions and triggered an internal assertion error. For example, we assumed that all solution files have a `main` function; if this function was missing then we would not be able to perform any of our analyses and thus would require manual intervention.

To capture both cases, we designated an assignment for manual review if its score was too low or non-existent (e.g. process ended early due to assertion error). For the sake of simplicity, we deemed an automated score to be too low and thus require manual review if it was below the cutoff points previously mentioned.

3.1.4 Accuracy and False Positives

Out of the assignments that our tool could automate, we evaluated the accuracy or false positive rate of our accepted predictions. We designated an assignment to be a false positive if its mark (not the same as score) was significantly higher than what a human marker assigned to it. In practice, we tolerated an excess of up to 10 points before we considered a mark to be a false positive because, as discussed in Section 3.1.2, human markers generally could vary their evaluation by up to 10 points for trivial issues.

It is important to note that the marks we used as ground-truth also included a written report. Furthermore, our ground-truth marks also contained deductions from non-technical issues such as late submissions or plagiarism. Unfortunately, detailed historical data were not kept after the course ended. Since these unrelated deductions were included in our ground-truth, our real false positive rate might actually be lower than what we present.

3.2 Always Full Marks

Since we expected most of the class to receive full marks for the assignment, we used the trivial approach of always giving everyone full marks as our baseline to measure the accuracy and effectiveness of our other approaches. Any other marking technique must perform better than this "automatic" technique to be considered an improvement over manual marking.

3.3 Minimum Distance

We used multiple reference solutions to cover the various approaches to solving their respective assignments. A correct student solution should closely match at least one of these reference solutions and thus have an extremely small edit distance from it. Therefore, we initially assumed the student's mark should be based on the smallest edit distance to reference solutions.

Since our cost model assigned an absolute value instead of relative difference, it was impractical to compare the actual edit distance between each reference solution. Therefore, we needed to first normalize the edit distances before we could properly compare how close a student solution is to other reference solutions.

There are many approaches to normalizing values; we chose to use the MaxAbsScaler algorithm in the Sklearn Python library [7] because we believed it was the most suitable one for our purpose. For reference solution j, the algorithm normalized its edit distance to each student $T_{1,j}, T_{2,j}, \ldots, T_{N,j}$ to be between 0 and 1. This algorithm also did not shift the data and was thus able to preserve the relative distances between students.

After normalizing student i's edit distances to each of the M reference solutions, we calculated the student's score as follows:

$$\text{Score}_i = 100 \cdot (1 - \min\{T_{i,1}, T_{i,2}, \ldots, T_{i,M}\})$$

3.4 Clustering

The next idea we attempted was using clustering algorithms to group student solutions together. The goal was to put student solutions with similar edit distances, to some subset of reference solutions, in the same cluster.

Under this mindset, we treated each student as an independent "data point" and their edit distance to each reference solution as an independent "feature". Our data thus became an $N \times M$ matrix with N rows for each student and M columns for each reference solution.

We made the assumption that full-mark solutions should have very similar features and thus should be very close to each other in the same cluster. Recall that the majority of the historical student solutions in this course had received full marks. Therefore, the majority of each cluster's points should also be very close to each other as well. As a result, we believed each cluster should theoretically be "centered" closer to the full-mark solutions than the incorrect solutions; therefore we hypothesized the closer a student solution was to the cluster's center, the higher the probability that the student solution should receive full marks.

Determining Number of Clusters

The K-Means and Gaussian Mixture clustering algorithms require us to specify how many clusters we want to find in our data points (student solutions). There are no straightforward approaches to choosing this value because it depends on the input data and use-case.

If we choose too few clusters, then we risk putting too many student solutions in the same cluster despite them not being too closely related. If we choose too many clusters, then we risk not getting sufficient data to compute scores. For example, in the extreme case of N clusters, every student would become its own cluster and, according to our original hypothesis of final score being based on closeness to a cluster's center, should receive full marks.

However, there are techniques such as the "Elbow Method" [30] shown in Figure 3.1 that can be used for guidance. For our data, the Elbow Method recommended for us to use 4 clusters.

Cleaning Up Data

Although not essential, we chose to perform Principal Component Analysis (PCA) [32] on our data prior to clustering. PCA transforms our set of M features into a smaller set of linearly uncorrelated features or "components". The components are sorted in descending order of variance. In other words, the first few components theoretically capture the majority of the "information" in the source data.

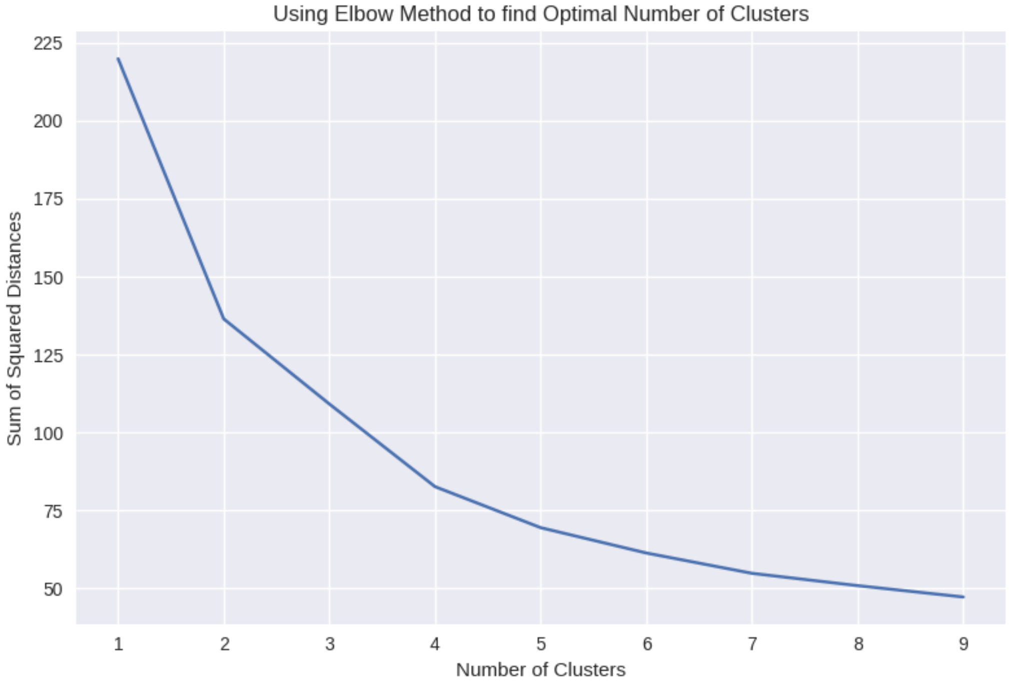


Figure 3.1: The Elbow Method runs the K-Means clustering algorithm with a range of clusters and computes the sum of squared errors. This sum measures how close the predicted clusters match the data points; the greater the error, the less clusters fit the data. This graph converges to 0 at N clusters where each point is its own cluster and thus has zero error. To find an appropriate number of clusters, we need to visually find an "elbow" point on the graph, i.e. where adding an additional cluster will not significantly reduce the error. In this graph, the elbow point is at 4 clusters.

The most obvious advantage of PCA is reducing the runtime of our clustering algorithms because it eliminates features (columns in our data matrix) that represent very little information about our data points. This technique is also useful in visualization as it allows high-dimensional data to be presented in a 2D graph while preserving the majority of the information and relationships between data points.

To use PCA, we need to specify how many components or features to keep. Similar to the Elbow Method to determine the number of clusters, we can look at the graph of cumulative explained variance shown in Figure 3.2 to estimate an appropriate number of components. For our data, the graph recommended for us to use 6 components.

3.4.1 K-Means

The first clustering algorithm we tried was K-Means [22]. It iteratively tries to group unlabeled data points with similar features together by minimizing the mean distance between each point in each group. Figure 3.3 visualizes the groups labeled by the K-Means algorithm on the 2017 class for the Non-Blocking IO assignment.

The K-Means algorithm constructs each group to minimize the average distance from the group's centroid to every point in its corresponding group. Since the majority of students in our data set received full marks, we assumed the majority of the data points in each group should also represent full-mark solutions. As a result, the volume of full-mark solution points should draw the centroid of each group towards "regions" representing higher marks. Based on this idea, we calculated a student's score based on their distance to their corresponding group's centroid; the closer a student was to their group's centroid, the higher their probability of receiving a high score.

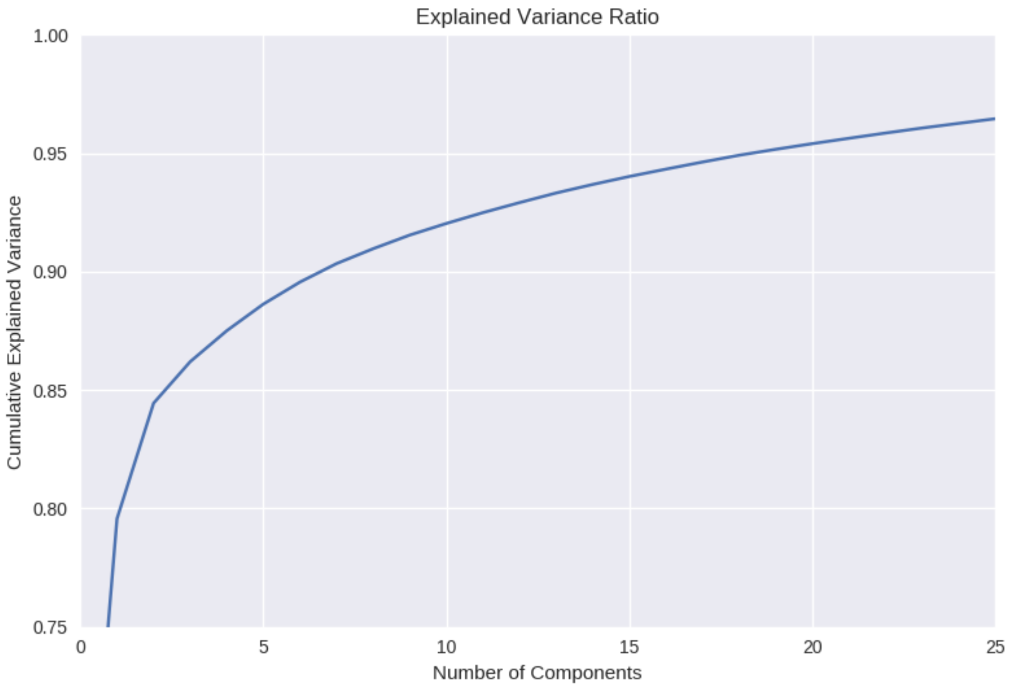


Figure 3.2: The explained variance conceptually represents the amount of information in the original data that each component captures. The first component captures almost 80% of the original information; the second component captures another 5%; and so on. This graph show the cumulative explained variance that each additional component captures. A rule-of-thumb for PCA is to choose the number of components at an "elbow point" where adding an additional component will not capture significantly more information. In this graph, the elbow point is at 6 components.

Let P_i denote the point representing student i and C_i denote the centroid of the student's group. The distance from centroid d_i is defined as:

$$d_i = \|P_i - C_i\|$$

As mentioned in Figure 3.3, the scales in the axes have no tangible interpretations. Therefore, the distances from centroid d_i we calculated for each point also had no meaningful interpretation. As a result, we tried to evaluate each point's centroid distance relative to all other points. In other words, we calculated each student's score based on their performance relative to the rest of their class.

Let μ denote the mean and σ denote the standard deviation of centroid distances of every student. We calculated student i's score as follows:

$$\text{Score}_i = 100 - 20 \cdot \frac{|d_i - \mu|}{\sigma}$$

This equation deducted up to 20 points for each standard deviation of centroid distance a student was from their corresponding group's centroid. We chose to deduct 20 points per standard deviation arbitrarily after experimenting with different values. The main idea was to deduct marks based on how far the solution was from the centroid, where we assumed majority of the full mark solutions were located.

3.4.2 Gaussian Mixture

The second clustering algorithm we tried was Gaussian Mixture [14]. Similar to the K-Means algorithm, it also groups unlabeled data points but with different criteria. As its name suggests, it assumes the points are randomly distributed following a Gaussian distribution. It iteratively tries to assign

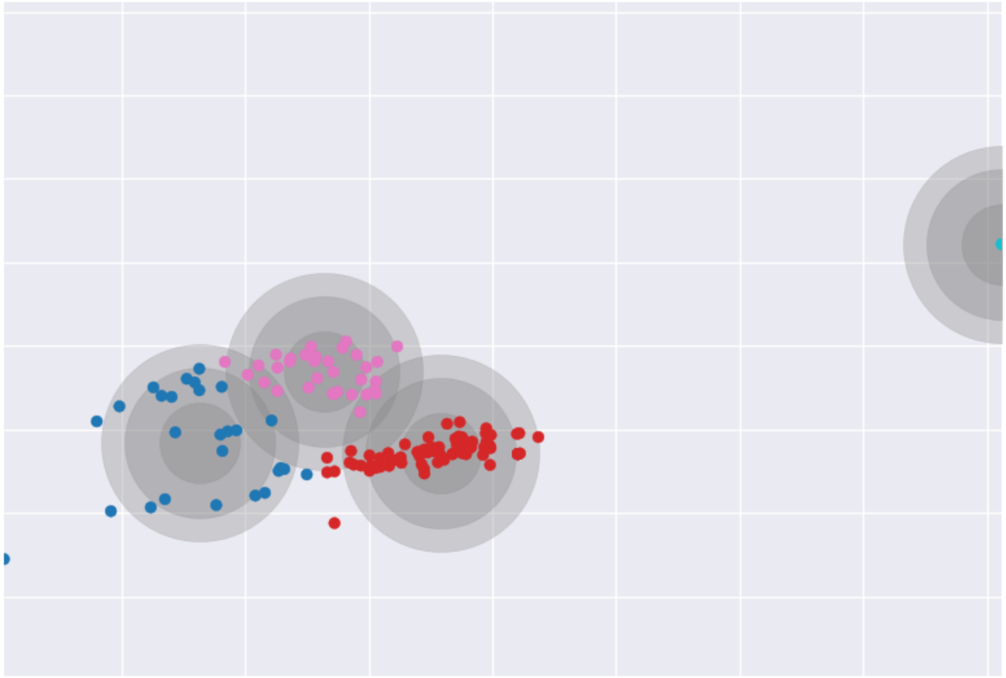


Figure 3.3: Each data point represents a student solution to the Non-Blocking IO assignment from the 2017 class. It is important to note that this is a 2D projection of a multidimensional data set. Since we preprocessed the data with PCA, the first two dimensions presented here already capture the majority of the data variance. The scales in the axes are omitted because they have no tangible interpretations; they are meant for interpreting relative distances. The K-Means algorithm assigned each point a group, represented by their corresponding colour. Each set of concentric circles represent an arbitrary distance from the centroid of each group.

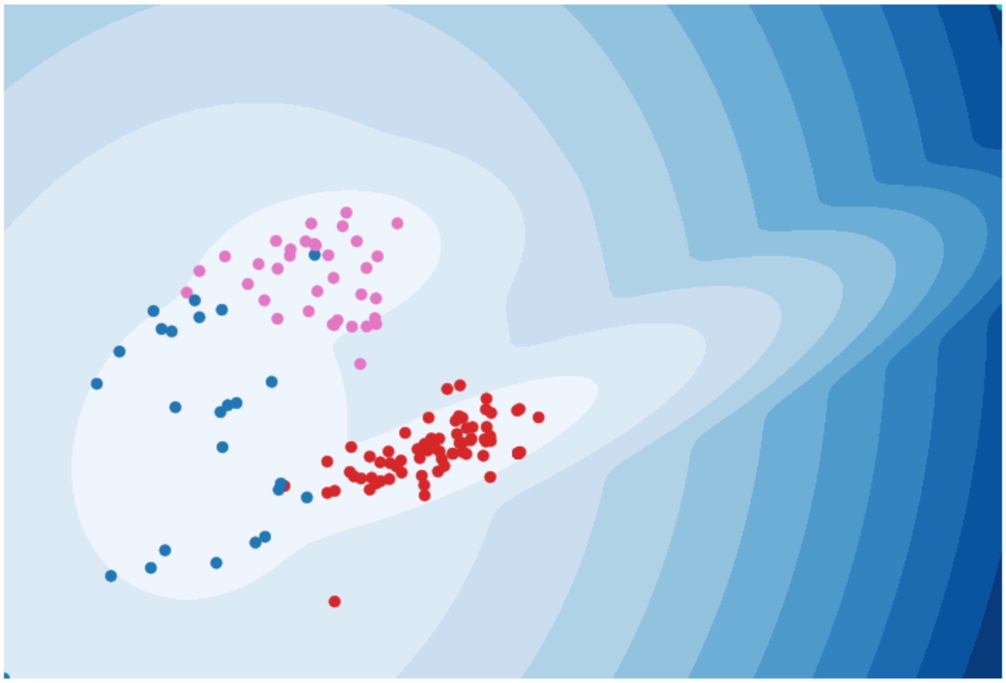


Figure 3.4: This is a probability contour graph of our data points grouped by the Gaussian Mixture algorithm. Points closer to the center of the contours, i.e. lighter areas, have a higher probability of belonging to their corresponding group.

groups to maximize the likelihood of each data point belonging to their assigned groups. Likewise, we hypothesized that the closer a point (student solution) is to the centroid, in this case the central probability contour, the higher the probability of the student receiving a high score. In practice, we designated each student's score to be equal to the probability of them belonging to their assigned cluster, returned from the Gaussian Mixture algorithm.

3.4.3 HDBSCAN

The third and last clustering algorithm we experimented with was HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise) [23]. Similar to the previous two algorithms, it tries to group unlabeled data points with its own set of criteria. As its name implies, it tries to group points based on spatial density to ensure the resulting groups meet some density parameter.

The advantage of HDBSCAN is that it does not require us to specify the number of clusters we want to find. Instead, we specify the minimum number of points in each cluster P and the algorithm will dynamically group points such that the resulting clusters have at least P points. This gives us more flexibility in our parameters because choosing the number of clusters is harder than choosing the minimum number of points per cluster. We do not know the proper the number of clusters and thus must estimate it through an ad-hoc procedure. However, we do have a stronger conceptual understanding of the number of points per cluster: since we know there are only a limited number of approaches to solve an assignment and we know the majority of the students will follow similar approaches, either through collaboration or coincidence due to limited unique approaches, we know the P value must be a significant fraction of the total number of students.

However, the disadvantage of this algorithm is that it is not guaranteed to be able to label every point; outlier points that cannot satisfy the group's density criteria are discarded as noise. The higher the minimum cluster size parameter P, the more points will be unlabeled. Ideally we want our groups to be as dense as possible so that we know the grouped points are extremely similar to each other. Ultimately, it is a balance between how many points we can label (automation rate) and how dense the resulting groups are.

Figure 3.5 shows how the algorithm labels our data set based on varying minimum cluster sizes. A cluster size of 5 is able to achieve 3 groups, which is close to the 4 groups set for the previous two algorithms. However as we can see from the graph, the 3 groups are slightly sparse. As a result, we chose $P = 10$ because it is able to label most of the points without sacrificing too much density; at higher cluster sizes, we do not seem to significantly increase our groups' densities.

Once we were satisfied with our parameters, we ran the HDBSCAN algorithm and designated each student's score to be equal to the "strength" of the student's corresponding group prediction returned from the algorithm.

3.5 Experimental Results

Before we present our results, we will first discuss the possible outcomes from our tool in Table 3.1. Recall that we round up the predicted scores from our algorithms if the student scores above certain cutoff points. After rounding, we check if their true mark from historical data is within 10 points of 100 (to account of variance in human marker leniency). Ideally, we would like to cover the full range of marks. However because our tool does not provide meaningful feedback for deductions, other than the fact that the student's solution structure was notably different from our reference solutions, we decided to focus our efforts on optimizing the accuracy for scores above cutoff points. As a result, our tool's purpose essentially becomes "flagging" correct solutions; any solutions not flagged will be designated for manual marking.

Ultimately, we would like to maximize the number of predicted scores above cutoff points while minimizing the number of false positives. The more assignments that are predicted to score above our cutoffs, the less manual work will be required from human markers. While most students would not complain about undeservedly receiving full marks, too many false positives (and student knowledge thereof) would diminish the formative value of the assignments. That is, if students were aware of a high false positive rate

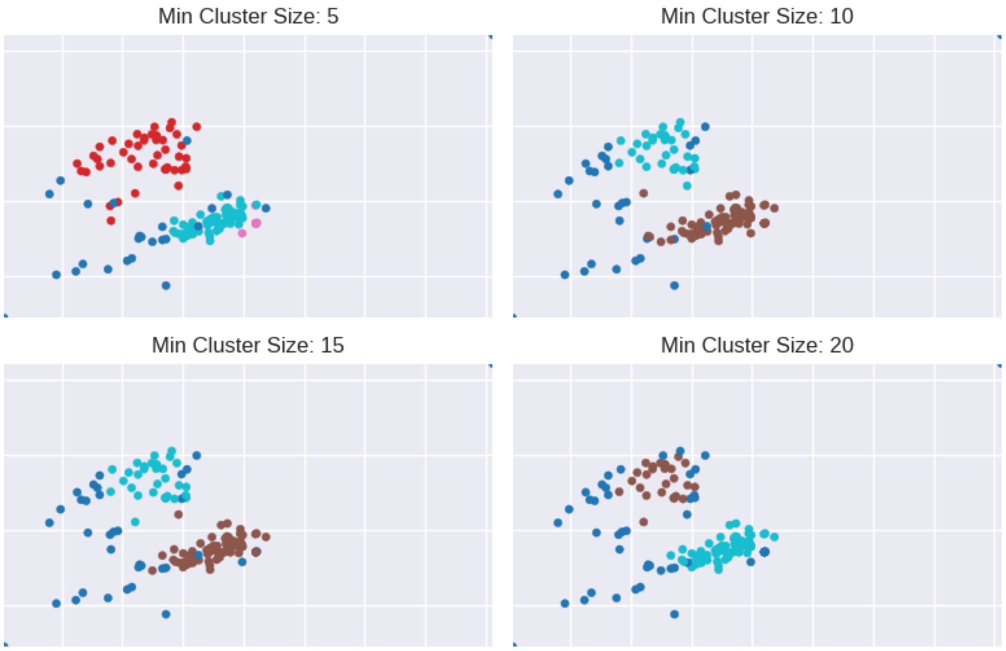


Figure 3.5: These graphs show our data points clustered with HDBSCAN using different minimum cluster size parameters. The higher the minimum size, the denser the resulting clusters. Furthermore, higher cluster sizes also result in more outlier points marked as noise (dark blue) as well as fewer total clusters. Cluster size of 5 has three different groups whereas cluster sizes of 10 and higher only have two different groups.

		True Mark	
		< 90	≥ 90
Automated Score	$<$ Cutoff	Correct Prediction	False Negative
	$\geq$ Cutoff	False Positive	Correct Prediction

Table 3.1: This table presents the possible outcomes from our tool's predictions. Without meaningful feedback for deductions, only automated scores above our cutoff points (which will later be rounded up to full marks) are relevant for us. As a result, the relevant outcomes for our tool lie in the bottom row of this table.

due to assigning full marks, we are concerned about complacency or apathy towards completing their assignments.

Figure 3.6 presents our experimental results. Since Always Full Marks did not perform any analysis, it would not encounter any processing errors and thus would always be able to automate every assignment. Clearly this was the maximum possible value and could not be exceeded; the goal for the other algorithms was to automate as many assignments as possible. While near-100% automation rate was unrealistic due to potential processing errors, we hoped to be able to automate at least half the assignments. The false positives for Always Full Marks was simply the students that did not receive full marks. Therefore, we required the other algorithms to achieve a better false positive count than this trivial approach to be considered an improvement.

Firstly, the Minimum Distance algorithm was able to achieve slightly fewer false positives than Always Full Marks with an acceptable number of automated assignments. With the exception of Non-Blocking IO in 2018,

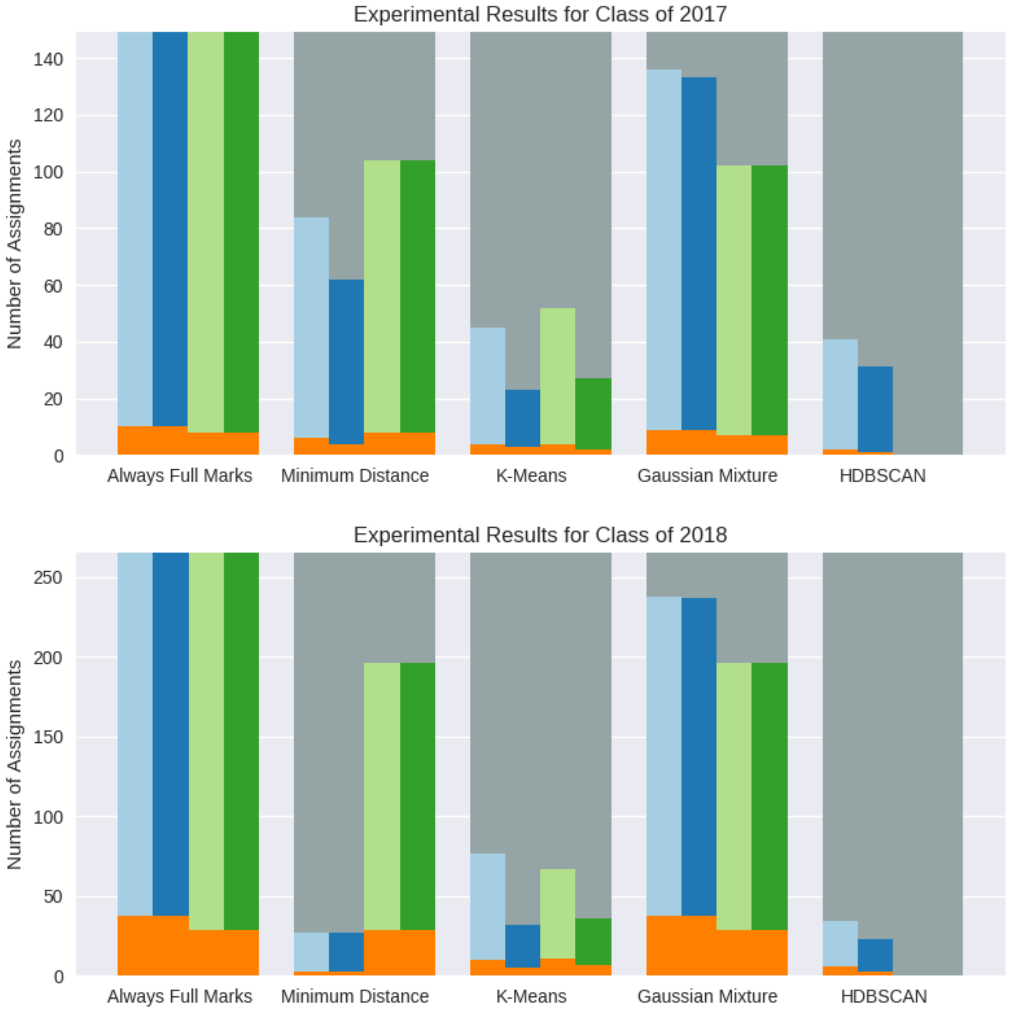


Figure 3.6: These graphs break down the outcomes of the assignments for each class. The grey bars represent the number of assignments that must be manually marked; the light blue and dark blue bars represent the number of automated Non-Blocking IO assignments at the 90 and 95 cutoffs, respectively; similarly, the green bars represent the Parallel Processing assignment; and finally, the orange bars represent the number of automated assignments deemed to be false positives.

this algorithm was able to automate almost half of the classes, which would greatly reduce the time needed for manual marking.

Secondly, the K-Means algorithm was able to achieve even fewer false positives than Minimum Distance; conversely, it automated even fewer assignments. However, we note that at the 90 cutoff, K-Means was still able to automate approximately a third of the assignments, which was a non-trivial reduction in manual work for human markers.

Thirdly, the Gaussian Mixture algorithm was able to achieve the highest automation rates out of all of our proposed algorithms. However, we note that this algorithm appeared to be strictly worse than the baseline Always Full Marks because its number of false positives was on par with the baseline despite not being able to automate as much.

Fourthly, the HDBSCAN algorithm was able to achieve the lowest number of false positives; conversely, it automated the fewest assignments. In addition, it appeared to be unable to flag any correct Parallel Processing solutions for the 2018 class; since this was exclusive to one class and not both, we believe this may have been due to minor assignment specification changes over the years and that our reference solutions were not up-to-date.

Finally, one common observation for all the graphs was that increasing the cutoff point decreased both the number of automated assignments and false positives. This was expected because a higher cutoff enforced a higher confidence in our prediction, i.e. the predicted score must be within 5 points of 100 to be considered correct instead of 10 points. Naturally, with a stricter requirement, the number of assignments that we could automate and the number of false positives also decreased.

3.6 Shortcomings

None of our proposed algorithms were able to achieve any meaningful reduction from Always Full Mark's number of false positives. As a result, we investigated potential sources of error that may have been been caused by our assumptions.

3.6.1 Too Much Noise in Edit Distance

One of the underlying assumptions that our tool made was that the edit distance we compute must accurately reflect the closeness of two program files. Although we tried encapsulating structural differences into a tree edit distance and ensuring logically similar code had low edit distances, the edit difference was still ultimately a scalar value that discarded potentially useful information.

When we manually inspected a couple of assignments marked as false positives (assignments that were automated to higher marks than they had earned from human markers), we noticed some of them had glaring issues that were not penalized as heavily relative to the overall edit distance. For example, one student did not use the Pthread library at all for the Parallel Processing assignment and thus should had failed. However, our tool only penalized an edit distance of 100 for inserting the appropriate `CallStatement` nodes whereas final edit distances were normally between 500 to 800. As a result, this catastrophic failure went undetected as it got treated as a small outlier. This problem might had been caught if the penalty was magnitudes higher (e.g. 10,000). However, this might also obscure other issues because it would be magnitudes larger than other penalties.

Overall, we believe the single scalar value has too much noise to make accurate predictions. There are several avenues to pursue to improve this step: prune more unnecessary nodes from the ASTs, adjust the cost model penalties for key function calls, or generate a multi-dimensional matrix to represent program differences and to be used as input to our aggregator.

3.6.2 Uncertain Ground-Truth

There was some uncertainty in our ground-truth for the students' actual marks. Historical mark breakdowns were not archived and thus the marks we used include non-technical deductions such as issues with the student's written report, late submissions, or plagiarism. In addition, there was a discrepancy in marker leniency between the two years, as seen in Figure 3.7. As a result, the "true" student marks that we used to measure our accuracy were not as reliable as we originally assumed.

To verify whether the uncertainties in our ground-truth had a significant impact on our results in Section 3.5, we randomly selected 20 students from each class and remarked their assignments to ensure accuracy and consistency. Tables 3.2 and 3.3 present the automation and false positive rates of our sampled students with their re-evaluated marks. Since we changed the ground-truth, only the false positive rate should change. However we still present the automation rate to verify that our random sample is representative of their entire class. This was indeed the case as we see that the automation rate was roughly identical to the automation rates of the entire class presented Figure 3.6; Gaussian Mixture had the highest automation rate while the others fell behind with HDBSCAN as the lowest.

When we compare the false positive rates, we see that Gaussian Mixture (the algorithm we initially observed to be strictly worse than the baseline Always Full Marks) had a slightly better result than the baseline for both classes. In the 2017 class, Gaussian Mixture had zero false positives whereas the baseline had two false positives. Due to our small sample size and the

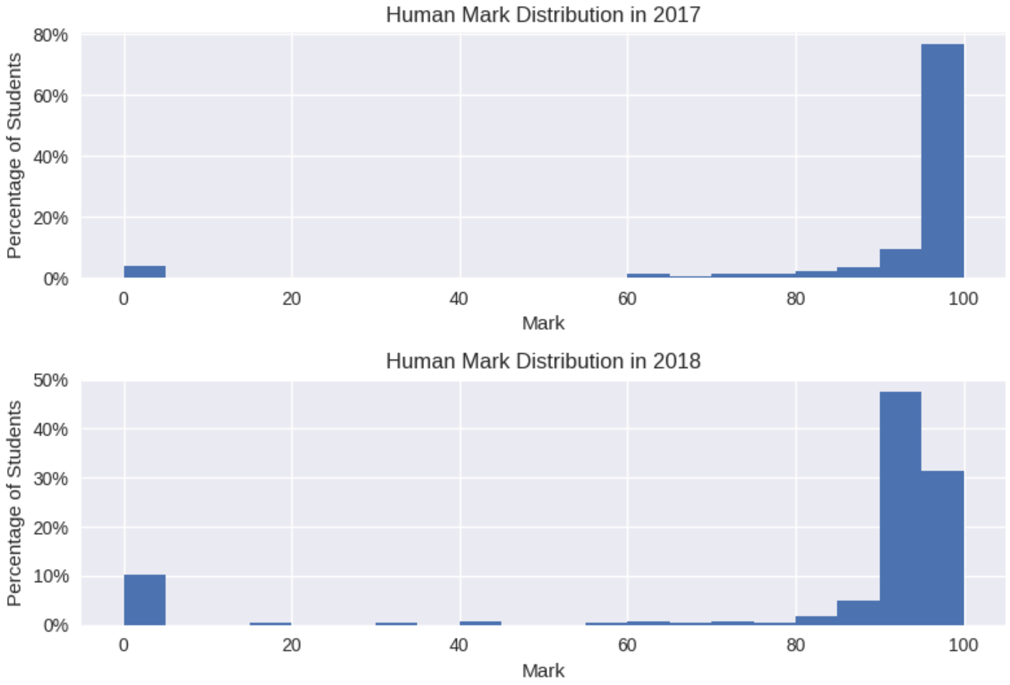


Figure 3.7: There is a discrepancy in marker leniency between 2017 and 2018. Since markers change every semester, the two classes had different markers. In the 2017 class, about 80% of the class received a 95 or higher from the marker whereas in the 2018 class, only 35% of the class received a 95 or higher from the marker.

fact that we were only able to automate 40% of the students, this might had simply been a coincidence. However when we look at the 2018 class, we see that Gaussian Mixture was able to automate 70% of the students with 21% false positive rate compared to Always Full Marks at 35% false positive rate.

Despite having a lower false positive rate than our baseline, 21% is still too high to be considered usable in a live-classroom environment. Nonetheless, this sample has demonstrated promising potential in our tool to perform better than blindly assigning everyone full marks. Further refinements to our tool may eventually lead to a false positive rate low enough to be considered usable in a regular classroom.

3.7 Summary

One common pattern we observe is the tradeoff between number of automated assignments and false positives. The more assignments that we want to automate, the more false positives we will get.

The choice of algorithm is ultimately a decision for the user to make: if we want to prioritize saving time, then we should use Gaussian Mixture; conversely, if we want to prioritize minimizing the number of false positives, then we should use K-Means or Minimum Distance.

Table 3.2: Automated Marks With New Ground-Truth for Class of 2017

	90 Cutoff		95 Cutoff	
	Automated	False Positives	Automated	False Positives
Always Full Marks	20	2	20	2
Minimum Distance	5	0	5	0
K-Means	2	0	2	0
Gaussian Mixture	8	0	8	0
HDBSCAN	0	-	0	-

Table 3.3: Automated Marks With New Ground-Truth for Class of 2018

	90 Cutoff		95 Cutoff	
	Automated	False Positives	Automated	False Positives
Always Full Marks	20	7	20	7
Minimum Distance	1	0	1	0
K-Means	2	0	2	0
Gaussian Mixture	14	3	14	3
HDBSCAN	0	-	0	-

Chapter 4

Related Works

Writing programs is one method used in computer science education to reinforce and assess practical concepts taught in class. Automating the manual marking process has been a widely studied topic due to the need for quick feedback turnaround and, by removing or minimizing the human element, maximizes objectivity and consistency. This chapter explores several tools in this field; since there are too many tools to count, we will mainly focus on tools designed for interoperability rather than ones designed for niche domains or specific assignments.

4.1 Input/Output Marking

The earliest published example we found of automated programming assignment marking dates back to 1989 by Isaacson and Scott [19]. Their approach is one of the most common automated marking techniques used in practice today: they compile each student's program (if possible), execute the students' programs with input files, and compare the programs' outputs with expected output files. This process is straightforward and easy-to-understand for testing program functionality. There are many subsequent works [13, 17, 20, 24, 28] that further enhance their process. These tools ultimately all follow the same three steps.

4.2 Fill-In-The-Gap Marking

Another increasingly common assignment style and marking technique is called "fill-in-the-gap". As this name implies, these assignments provide students with gaps in instructor-provided template code to fill prior to compilation and analysis. For example, an instructor might provide students with a method signature for a binary search algorithm and task them to implement the algorithm itself. Lieberman [21] has stated that this assignment style is the most effective way for beginner programmers to apply newly-taught concepts. Due to the rigid nature of these assignment specifications and extremely small analysis space, many static analysis techniques have been developed to analyze these gaps.

The Environment for Learning to Program (ELP) marking platform developed by Truong et al. [31] is an example of the fill-in-the-gap assignment style used in practice. After a student submits their assignment, their tool compiles the student program and extracts the relevant "gap" from the resulting AST for later analysis. Similar to our tool, they also perform some AST normalization prior to analysis in order to minimize the amount of variety among functionally-similar programs. After extracting the gaps, they perform analysis such as checking variable states and invariants before/after gaps and metrics such as line-length and cyclomatic complexity.

OverCode developed by Glassman et al. [16] is another marking platform based on the idea of fill-in-the-gap style Python programming assignments in Massive Open Online Courses (MOOC). They first clean up student programs such as normalizing variable names. They then cluster student solutions based on line-by-line comparison such that programs with similar lines are in the same clusters. The resulting clusters are presented to instructors to manually provide feedback and scores. We note that their approach was mainly made possible due to the simplicity of Python and of the assignment specification as well as the massive amount of submission data available to analyze. In offline classes such as the class we analyzed with only hundreds of students, rather than tens of thousands, it is unlikely their approach will be able to extract meaningful-sized clusters for analysis.

The disadvantage of these types of assignments and their corresponding tools is that they are mostly used in beginner programming classes rather than upper-year classes, such as the one analyzed in this thesis. Furthermore, they do not score student assignments beyond input/output testing; however, it is theoretically possible for them to apply a score based on their static analysis results such as applying deductions for high cyclomatic complexity. Nonetheless, advanced programming assignments such as our Non-Blocking IO and Parallel Processing assignments are more open-ended and generally consist of significantly larger non-divisible methods that are not as easily analyzable by the aforementioned tools.

4.3 AST-Based Marking

There are many prior works exploring the idea of automatically comparing student solutions to reference solutions at the solution level using ASTs. One common element found in all of these works is the need for normalizing ASTs by collapsing functionally-similar substructures into canonical forms prior to analysis. This is essential to reducing the number of distinct solutions to analyze and avoiding falsely penalizing functionally-similar code. In addition to the field of automated program marking, normalizing AST substructures is widely studied in other fields such as plagiarism and clone detection [10, 15].

Singh et al. [27] developed an error-correction language for describing steps to transform an incorrect student solution into a correct reference solution. In addition to reference solutions, the instructor must also describe common student errors and the steps to correct them using the error-correction language. Their system then utilizes a constraint solver to find the minimum steps needed to correct the student solutions. Finally, it scores the student based on the steps. This is similar to our tool's cost model described in Section 2.4.2 where we associate a cost for any change to be made to the student AST.

Thorburn and Rowe's PASS tool [29] also requires additional work from the instructors prior to scoring students. Their tool requires the instructor to specify an hierarchical "solution plan" for each assignment; this is essentially a breakdown of the steps to solve a problem. For example, a merge sort problem consists of a "sort" function at the top level; it then consists of sub-components such as checking the base case, pivoting the input, and the making two recursive calls. Their tool attempts to find equivalent program components in the student code. They check for equivalence by comparing component outputs when given randomly generated inputs. Students are then scored based on the number of equivalent components found.

The main drawback for both Singh et al.'s and Thorburn and Rowe's tools is the additional work needed for instructors beyond providing reference solutions. Singh et al.'s work requires instructors to anticipate and describe common errors and corrections in a custom language. This may prove difficult as one cannot anticipate all possible errors a student may make. Thorburn and Rowe's work requires instructors to describe a solution plan by hierarchically breaking down a problem into smaller sub-problems. This may not always be possible for every assignment as some assignments may have large indivisible components.

AssignSim developed by Naudé et al. [25] was perhaps the most similar prior work to ours. Their tool directly compares solution ASTs and generates a score based on their similarities. Our works differ in how we compare the post-processed ASTs and score the student. They generate scores by directly analyzing the graphs and assigning scores based on the differing nodes and their respective neighbour nodes. On the other hand, our work utilizes a generic tree edit distance algorithm to compare the graphs and then aggregate the various edit distances into a mark. Their results were slightly better than ours when their set of reference solutions contained both high quality and low quality solutions (i.e. both high marks and low marks). However, when they only had high quality solutions, their accuracy fell to similar levels as ours. This led us to speculate that our choice of reference solutions (only high marks) may also had an influence in our results; however since the majority of past students in our data set have received full marks, we do not have sufficient data to pursue further investigation.

Chapter 5

Conclusion

This thesis presented a novel approach to automatically mark programming assignments. Our approach consists of two components: (1) we first prune a program's AST to isolate key features relevant for assignment marking; (2) we then compare a student solution to a set of reference solutions in order to generate a final mark for the student. Assignments that have received automated deductions will require manual review to provide more meaningful and individualized feedback.

This tool assumes the majority of the students will write their programs in specific patterns. We are concerned this may limit student creativity when trying to solve their assignments. However, we note that the programming assignments we tested with had rigid designs and the easiest-to-implement solutions often fall under specific patterns. Therefore we do not believe this to be a major concern.

We implemented our processes as the ClangAutoMarker tool and tested it with student submissions and marks from previous offerings of the ECE459 course at the University of Waterloo. Our initial results were not as successful as we had originally hoped. Our tool did not perform better than the baseline approach of simply always assigning full marks. However, due to the uncertainty in our ground-truth, we faithfully recollected the ground-truth data for a smaller subset of previous classes. When we reevaluated our tool with the more accurate sample, we were able to achieve a better false positive rate of 21% compared to always assigning full marks which had a false positive rate of 35%. Although this was still not very accurate, we have demonstrated that our tool has promising potential for automated marking and further improvements may make it viable for a live classroom.

References

[1] ECE 459: Programming for Performance. https://github.com/jzarnett/ece459. Accessed: 2018-08-31.

[2] libcurl - the multiprotocol file transfer library. https://curl.haxx.se/libcurl/. Accessed: 2018-08-31.

[3] Matplotlib: Python plotting. https://matplotlib.org/. Accessed: 2018-09-18.

[4] NumPy. http://www.numpy.org/. Accessed: 2018-09-18.

[5] Schedule of Classes for Undergraduate Students. http://www.adm.uwaterloo.ca/infocour/CIR/SA/under.html. Accessed: 2018-08-31.

[6] scikit-learn: Machine Learning in Python. http://scikit-learn.org/stable/. Accessed: 2018-09-18.

[7] sklearn.preprocessing.MaxAbsScaler. http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MaxAbsScaler.html. Accessed: 2018-09-25.

[8] The LLVM Compiler Infrastructure. https://llvm.org/. Accessed: 2018-08-31.

[9] Valgrind. http://www.valgrind.org/. Accessed: 2018-08-31.

[10] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.

[11] Philip Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337(1-3):217–239, 2005.

[12] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[13] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. On automated grading of programming assignments in an academic institution. *Computers & Education*, 41(2):121–131, 2003.

[14] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (methodological)*, pages 1–38, 1977.

[15] Jianglang Feng, Baojiang Cui, and Kunfeng Xia. A code comparison algorithm based on AST for plagiarism detection. In *Emerging Intelligent Data and Web Technologies (EIDWT), 2013 Fourth International Conference on*, pages 393–397. IEEE, 2013.

[16] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2):7, 2015.

[17] Colin Higgins, Tarek Hegazy, Pavlos Symeonidis, and Athanasios Tsintsifas. The CourseMarker CBA System: Improvements over Ceilidh. *Education and Information Technologies*, 8(3):287–304, 2003.

[18] Institutional Analysis & Planning. Student Headcounts. https://uwaterloo.ca/institutional-analysis-planning/

university-data-and-statistics/student-data/student-headcounts. Accessed: 2018-08-31.

[19] Peter C Isaacson and Terry A Scott. Automating the execution of student programs. *ACM SIGCSE Bulletin*, 21(2):15–22, 1989.

[20] David Jackson and Michelle Usher. Grading student programs using ASSYST. In *ACM SIGCSE Bulletin*, volume 29, pages 335–339. ACM, 1997.

[21] Henry Lieberman. An example based environment for beginning programmers. *Instructional Science*, 14(3-4):277–292, 1986.

[22] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[23] Leland McInnes, John Healy, and Steve Astels. hdbscan: Hierarchical density based clustering. *The Journal of Open Source Software*, 2(11), mar 2017.

[24] Derek S Morris. Automatic grading of student's programming assignments: an interactive process and suite of programs. In *Frontiers in Education, 2003. FIE 2003 33rd Annual*, volume 3, pages S3F–1. IEEE, 2003.

[25] Kevin A Naudé, Jean H Greyling, and Dieter Vogts. Marking student programs using graph similarity. *Computers & Education*, 54(2):545–561, 2010.

[26] Mateusz Pawlik and Nikolaus Augsten. RTED: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, 5(4):334–345, 2011.

[27] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices*, 48(6):15–26, 2013.

[28] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K Hollingsworth, and Nelson Padua-Perez. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. *ACM Sigcse Bulletin*, 38(3):13–17, 2006.

[29] Gareth Thorburn and Glenn Rowe. Pass: An automated system for program assessment. *Computers & Education*, 29(4):195–206, 1997.

[30] Robert L Thorndike. Who belongs in the family? *Psychometrika*, 18(4):267–276, 1953.

[31] Nghi Truong, Paul Roe, and Peter Bancroft. Static analysis of students' Java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*, pages 317–325. Australian Computer Society, Inc., 2004.

[32] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

Appendix A

Results of Scoring Algorithms

This appendix contains the tables presenting the results of each tree edit distance scoring method we presented in Chapter 3.

Assignments were considered "Automatable" if they scored higher than certain cutoff points. For these assignments, we analyzed their false positive rate; this was the percentage of assignments that scored higher than what they had originally earned from a human marker. Conversely, if an assignment scored lower than certain cutoff points or triggered processing errors due to unexpected AST structures or assertion failures, then it was designated for manual marking because we wanted to provide meaningful feedback to students for any deductions that they receive.

Table A.1: Always Assigning Full Marks

	2017		2018	
	90 Cutoff	95 Cutoff	90 Cutoff	95 Cutoff
Non-Blocking IO Assignment				
Total Students	149	149	265	265
Non Automatable	0	0	0	0
Below Cutoff	0	0	0	0
Processing Error	0	0	0	0
Automatable	149	149	265	265
False Positives	10	10	38	38
Parallel Processing Assignment				
Total Students	149	149	265	265
Non Automatable	0	0	0	0
Below Cutoff	0	0	0	0
Processing Error	0	0	0	0
Automatable	149	149	265	265
False Positives	8	8	29	29

Table A.2: Using Minimum Normalized Edit Distance to Obtain Mark

	2017		2018	
	90 Cutoff	95 Cutoff	90 Cutoff	95 Cutoff
Non-Blocking IO Assignment				
Total Students	149	149	265	265
Non Automatable	65	87	238	238
Below Cutoff	54	76	223	223
Processing Error	11	11	15	15
Automatable	84	62	27	27
False Positives	6	4	3	3
Parallel Processing Assignment				
Total Students	149	149	265	265
Non Automatable	45	45	69	69
Below Cutoff	0	0	0	0
Processing Error	45	45	69	69
Automatable	104	104	196	196
False Positives	8	8	29	29

Table A.3: Using K-Means Clustering to Obtain Mark

	2017		2018	
	90 Cutoff	95 Cutoff	90 Cutoff	95 Cutoff
Non-Blocking IO Assignment				
Total Students	149	149	265	265
Non Automatable	104	126	188	233
Below Cutoff	93	115	173	218
Processing Error	11	11	15	15
Automatable	45	23	77	32
False Positives	4	3	10	5
Parallel Processing Assignment				
Total Students	149	149	265	265
Non Automatable	97	122	205	239
Below Cutoff	52	77	136	170
Processing Error	45	45	69	69
Automatable	52	27	60	26
False Positives	4	2	10	4

Table A.4: Using Gaussian Mixture Clustering to Obtain Mark

	2017		2018	
	90 Cutoff	95 Cutoff	90 Cutoff	95 Cutoff
Non-Blocking IO Assignment				
Total Students	149	149	265	265
Non Automatable	18	23	40	52
Below Cutoff	7	12	25	37
Processing Error	11	11	15	15
Automatable	131	126	225	213
False Positives	9	8	35	33
Parallel Processing Assignment				
Total Students	149	149	265	265
Non Automatable	47	47	69	69
Below Cutoff	2	2	0	0
Processing Error	45	45	69	69
Automatable	102	102	196	196
False Positives	8	8	29	29

Table A.5: Using HDBSCAN Clustering to Obtain Mark

	2017		2018	
	90 Cutoff	95 Cutoff	90 Cutoff	95 Cutoff
Non-Blocking IO Assignment				
Total Students	149	149	265	265
Non Automatable	108	118	231	242
Below Cutoff	97	107	216	227
Processing Error	11	11	15	15
Automatable	41	31	34	23
False Positives	2	1	6	3
Parallel Processing Assignment				
Total Students	149	149	265	265
Non Automatable	149	149	265	265
Below Cutoff	104	104	196	196
Processing Error	45	45	69	69
Automatable	0	0	0	0
False Positives	0	0	0	0