UNIVERSITY OF OSLO
Department of Informatics

Detection of plagiarism in computer programming using abstract syntax trees.

Master thesis

Olav Skjelkvåle Ligaarden

9th November 2007

Abstract

Plagiarism in connection with computer programming education is a serious problem. This problem is common to many universities around the world, and the University of Oslo (UoO) makes no exception. The major problem is that students plagiarize homework programming assignments written by fellow students. To deal with this situation plagiarism detection software has been developed to assess the similarity between program listings. Such software is exposed to the daunting task of minimizing the numbers of false negatives and false positives at the same time, i.e. finding the highest number of copies while avoiding those which are not. UoO uses a distributed system for delivering such assignments, called Joly. This system compares program listings by using an attribute counting metric. This is a very general metric and here I investigate whether a less general-purpose metric tuned to the particularities of programming code may perform better than the one currently being used in Joly. To this end I have developed two new structure based similarity measures which quantify the structural similarity between abstract syntax trees (AST). More specifically, I have (i) modified the standard AST representation to ease the comparison between trees, (ii) identified the most common cheating strategies employed by students, (iii) assessed the changes these strategies have on the AST structures, (iv) developed and implemented two new AST similarity measuring algorithms, ASTSIM-NW and ASTSIM-LCS, focused on uncovering plagiarism based on the most common cheating strategies leaving the most distinct AST footprints, and (v) compared the performance of the two new algorithms relative to the one being currently used in Joly. Even though the test results need to be interpreted with caution, the combined use of the two new algorithms appears to perform better in terms of false negatives and false positives. This suggests that they should be considered as candidates for complementing the current attribute counting approach in Joly and thus be exposed to more extensive testing and polishing.

Acknowledgments

First of all I would like to thank my supervisors Arne Maus and Ole Christian Lingjærde for helping me finish this master thesis. We have had a lot of interesting discussions, they have introduced me to many interesting topics in computer science and other fields, and they have given me valuable feedback on my work. I also would like to thank Stian Grenborgen, a fellow master student who worked on a related topic, which I had many interesting discussions with and that gave me useful comments on my work.

I would also like to thank my uncle Stig W. Omholt for valuable feedback on how I could improve the structure of the thesis, the text and many other things. My sister Ingeborg Skjelkvåle Ligaarden must also be mentioned. I want to thank her for reading my thesis and providing me with comments and corrections, throughout the whole period that I worked on it, and for motivating me. Two other persons also needs to be mentioned. Nicolaas E. Groeneboom for helping me when I had issues with Latex and for reading my thesis, and Kjetil Østerås, friend and former fellow master student, for introducing me to the Java Compiler Compiler parser generator and other tools. Finally, I would like to thank my family and friends for supporting and motivating me during the time I worked on the thesis.

Contents

Chapter 1

Introduction

In the ten year period from 1991 to 2001 the Department of Computer Science at Stanford University experienced more honor code[1] violations than any other department at Stanford (Roberts, 2002). The Department had for the whole period 37 percent of all reported cases at the university. Considering that only 6.5 percent of the students at Stanford were enrolled in computer science classes, this is a surprisingly high number. The majority of violations appeared to be associated with unpermitted collaboration and plagiarism in connection with homework assignments involving computer programming. Another well known example of plagiarism associated with computer programming is the cheating scandal at MIT in 1991 (Butterfield, 1991). In the beginner programming course called "Computers and Engineering Problem Solving", unpermitted collaboration in connection with the weekly hand-in computer programming assignments caused 73 out of 239 students to be disciplined for violation of the honor code.

The two examples above illustrate a serious problem common to many universities around the world, where the University of Oslo (UoO) makes no exception. Plagiarism in connection with computer programming education at the Department of Informatics (DoI) at UoO is considered to be a serious problem. Arne Maus at DoI has estimated that 10-20% of the students cheat regularly on the homework programming assignments (Evensen, 2007). Of this cheating, DoI focus on students that plagiarize the work of fellow students. Detection of this cheating is not trivial. In the beginner programming course called INF1000 - "Introduction to object-oriented programming", there are hundreds of students that hand in assignments. Manual comparison of all those assignments against each other is not practically feasible. A further complicating factor is that the assignments are often reused, so a student can copy the assignment of a past student. To deal with this problem, DoI has developed a computerized plagiarism detection system called Joly (Steensen and Vibekk, 2006) which is used routinely throughout the INF1000 course.

Joly consists of a database and algorithms for processing the data. The database contains all the assignments of past and present students. When a new assignment is submitted to the system it is inserted into the database, and the system uses an algorithm to measure the similarity between this assignment and all assignments of the same kind in the database. Joly uses an attribute counting metric algorithm developed by Kielland (2006) during his master thesis to determine a similarity score between programs. More specifically, an attribute counting metric counts different attributes found in a program, such as the number of for-loops, number of lines and so on. Based on the counts of the different attributes for two programs, the algorithm calculates a similarity score between them. The assignment is then marked as a possible copy

[1] A honor code is the university's statement on academic integrity. It articulates the university's expectations of students and faculty in establishing and maintaining the highest standards in academic work.

of another assignment if the similarity score is above a certain threshold value. In this context, the word "copy" means an instance of plagiarism.

Computerized fraud detection systems like Joly is exposed to the daunting task of minimizing the numbers of false negatives and false positives at the same time, i.e. finding the highest number of fraud cases possible while avoiding those which are not. To accuse an innocent student of cheating may have dramatic consequences and for this reason there is a constant pressure on improving systems like this. One feature of attribute counting metrics like the one Joly uses is that the context in which the attributes are used is not taken into account. Two different program listings will thus be classified as likely copies if they have the same or almost the same counts of the different attributes. This could happen even though the students have worked independently. The background motivation for this thesis is to identify, develop and assess new similarity measures that might circumvent this potential downside of simple attribute counting metrics.

Indeed, such a metric may also be used to compare natural language documents by just changing the counting attributes it counts. The use of an attribute counting metric does not require syntactically correct code. Less general-purpose metrics tuned to the particularities of programming code may potentially perform better than the attribute counting metric.

One example is the class of so-called "structure based metrics". These metrics are used to measure the structural similarity between two program listings. Verco and Wise (1996) compared different attribute counting metrics and structure based metrics on program listings written in PASCAL. The major result was that the structure based metrics performed equally well or better than the attribute counting metrics in detecting plagiarized programs. Moreover, when Whale (1990) compared a different set of attribute counting metrics and structure based metrics on program listings written in FORTRAN, he came to the same conclusion.

Motivated by the reports of Verco and Wise (1996) and Whale (1990) I have in this thesis focused on developing two structure based metric algorithms that measure the structural similarity between abstract syntax trees (ASTs) of Java listings. More specifically, I have (i) modified the standard AST representation to ease the comparison between trees, (ii) identified common cheating strategies and assessed their impact on the ASTs, and (iii) implemented two new similarity measure algorithms ASTSIM-NW and ASTSIM-LCS. ASTSIM-NW uses the Top-Down Unordered Maximum Common Subtree Isomorphism algorithm (Valiente, 2002) on the whole tree together with the Needleman-Wunsch algorithm (Needleman and Wunsch, 1970). The ASTSIM-LCS uses the Top-Down Unordered Maximum Common Subtree Isomorphism algorithm on the whole tree, except for method bodies where it uses the Longest Common Subsequence algorithm (Cormen et al., 2001).

I show on a small test set that the new algorithms together appear to perform better in terms of false negatives and false positives than the current algorithm in Joly. Even though more extensive testing needs to be done before any conclusion can be drawn, the results so far are nevertheless quite promising.

1.1 The structure of the thesis

The rest of the thesis is structured as follows:

Chapter 2 This chapter contains information instrumental for assessing the subsequent chapters. It contains information about plagiarism detection, common cheating strategies, ROC curves, ASTs, and metrics which can be used for assessing the similarities between ASTs.

Chapter 3 The modfications done to the AST is presented in this chapter. I have reduced the size of the AST by removing nodes that are not important for the structure, and modified the grammar in order to ease the comparision of ASTs.

Chapter 4 In this chapter I assess the impact of the different common cheating strategies on ASTs. Then I assess which of the cheating strategies that deserve most attention based on the number of differences which we get between the ASTs and how easy it is to use the strategy. Finally, I outline how these strategies affect the development of the new similarity measures.

Chapter 5 In this chapter, I consider measures for assessing the similarity between ASTs, giving particular attention to the cheating strategies that deserve most attention. Then I outline two new similarity measures ASTSIM-NW and ASTSIM-LCS based on the previous discussion. Finally, I give a description of ASTSIM-NW and ASTSIM-LCS.

Chapter 6 I present pseudo code implementations of ASTSIM-NW and ASTSIM-LCS and provide details about the actual implementations. Moreover, I outline how these implementations were tested.

Chapter 7 The two new measures ASTSIM-NW and ASTSIM-LCS are compared against the algorithm in Joly in this chapter. I compare the algorithms abibility to detect plagiarized programs listings. On small test sets, the two new measures appear to perform better in terms of false negatives and false positives than the algorithm in Joly. I also compare the running times of the algorithms. Here, the algorithm in Joly perform better than the new algorithms.

Chapter 8 In this chapter I discuss possible shortcommings of the new measures, alternative representations to AST of the code, ASTSIM-NW's advantages over ASTSIM-LCS and vice versa, and issues regarding the implementation of the new algorithms in Joly.

Chapter 9 The conclusion is given and suggestions to further work is presented.

Chapter 2

Background

This chapter contains information instrumental for assessing the subsequent chapters.

2.1 Detection of plagiarism

This section contains a preliminary formalization of the problem, how plagiarism detection is practiced at UoO, and common student cheating strategies.

2.1.1 Preliminary formalization of the problem

In Chapter 1 we saw that we often want to test a large number of program listings for plagiarism. A plagiarism detection program selects those program listings from the set P of program listings that are most likely to be copies of each other. We can define a function for comparing a pair of listings $(p_i, p_j) \in P \times P$, where $i \neq j$, as $s = f(p_i, p_j)$, where s is a similarity score. To a similarity score s we have to associate a threshold t. This threshold maps the pair (p_i, p_j) to a element $\hat{c}$ in the set $\{p, n\}$ of positive (possible copy) and negative discrete class labels. $\hat{c}$ is the the predicted class of the listing pair. All pairs with $\hat{c} = p$ needs to be inspected manually for further assessment and a final decision on whether it is reason to accuse one or more students of fraud. In the manual inspection the pair is mapped to an element c in the set $\{p, n\}$ of positive (copy) and negative (not copy) discrete class labels. c is the actual class of the listing pair.

2.1.2 Plagiarism detection in Joly

As mentioned, Joly uses an attribute counting metric to detect plagiarism. The algorithm became somewhat modified when it was implemented in Joly. Due to this I will in the following describe the modified algorithm and not the original one. To compare two program listings p_1 and p_2 the modified algorithm counts $N = 10$ different attributes in the two listings. For Java listings the attributes that it counts are **"void"**, **"String"**, **">"**, **"class"**, **"if("**, **"="**, **"static"**, **";"**, **","** and **"public"**. Before counting these strings, the algorithm removes the comments and the string literals in the two listings. This is done to ensure that none of the attributes are found among these elements.

For each program listing, the counts of the different attributes are stored in a vector of size N. The algorithm then finds the angle α (in degrees) between the two vectors from the expression

$$\cos\alpha = \frac{\sum_{i=1}^{N} x_i \cdot y_i}{\sqrt{\sum_{i=1}^{N} x_i^2} \cdot \sqrt{\sum_{i=1}^{N} y_i^2}}, \tag{2.1}$$

where x_i and y_i are elements in the vectors x and y of the listings p_1 and p_2, respectively. If the angle α is less than or equal to some predefined threshold t, then p_1 and p_2 are classified as possible copies.

Joly then sends an email to the teaching assistant responsible for the student(-s)[1] behind the listing(s). If the assistant after manual inspection of the listings thinks there may be a fraud situation, he sends the listings along with a report to the course administrator(s). Based on the documentation the course administrator decides whether the students should be taken in for an interview. The outcome of the questioning in the interview then determines further actions from the Department of Informatics.

2.1.3 Common student cheating strategies

When a student uses source code written by someone else, then he/she will often do modifications on this source code for the purpose of disguising the plagiarism. By using his/hers creativity and knowledge of a programming language, the student can use numerous cheating strategies to disguise plagiarism. Rather than attempting to list all possible strategies, which would be a rather hard task, we will instead focus on some of the most common ones. A list of common strategies is presented by Whale (1990). This list is widely referenced in the literature.

The list is presented below. The strategies are listed from easy to sophisticated, with respect to how easy it is to apply them. This list was originally made for FORTRAN and similar languages. We can therefore remove strategy 11, since we cannot use non-structured statements, or GOTO as it is also known as, in Java.

1. Changing comments or formatting.
2. Changing the names of the identifiers.
3. Changing the order of operands in expressions.
4. Changing the data types.
5. Replacing an expression with an equivalent expression.
6. Adding redudant code.
7. Changing the order of independent code.
8. Replacing one iteration statement with another.
9. Changing the structure of selection statements.
10. Replacing procedure calls with the procedure body.
11. Using non-structured statements.
12. Combine the copied code with your own code.

The set of strategies $S = \{1, \ldots, 12\} \setminus \{11\}$ can be divided into two subsets S_1 and S_2. A student that copies the whole program from someone else can use the strategies $S_1 = S \setminus \{12\}$, while the strategies $S_2 = S$ can be used by a student that copies parts of the program from someone else and then combines it with his/hers own work.

[1] There is the possibility that one of the programmers is a past student.

2.2 Comparing different plagiarism detection programs

Performance of two plagiarism detection algorithms can be compared by use of a Receiver Operating Characteristics (ROC) curve approach (Fawcett, 2004). In order to better understand what the ROC curve approach implies I introduce the confusion matrix concept and associated vocabulary.

2.2.1 Elucidation of the confusion matrix concept

A confusion matrix (Kohavi and Provost, 1998) contains information about actual and predicted classifications done by a classification system. Each row of the matrix represents the instances in a predicted class, while each column represents the instances in an actual class. With a confusion matrix it is easy to see if the system has mislabeled one class as another.

Given the actual and the predicted class labels for a pair of listings there are four possible outcomes. Suppose a pair is positive. If it is predicted to be positive, then it is counted as a true positive (TP); if it is predicted to be negative, then it is counted as a false negative (FN). Suppose a pair is negative. If it is predicted to be negative, then it is counted as a true negative (TN); if it is predicted to be positive, then it is counted as a false positive (FP). Given the classifier and a set of instances (the listing pairs), a two-by-two confusion matrix can be constructed representing the distribution of the set of instances. Table 2.1 shows a confusion matrix.

		Actual class c		
		p	**n**	**total**
Predicted class $\hat{c}$	**p**	True Positives (TP)	False Positives (FP)	**P'**
	n	False Negatives (FN)	True Negatives (TN)	**N'**
	total	**P**	**N**	

Table 2.1: Confusion matrix with the actual and predicted numbers of positives and negatives for a set of program listings.

In this matrix $P = TP + FN$ and $N = FP + TN$ are the numbers of actual positives and negatives in the test set, while $P' = TP+FP$ and $N' = TN+FN$ are the numbers of predicted positives and negatives in the test set. The numbers along the main diagonal represent the correct decisions made by the classifier, while the numbers off this diagonal represent the incorrect decisions. The confusion matrix can be used to assess several performance characteristics of the classifier. Some of these are given below:

The *true positive rate* (or sensitivity) is the proportion of true positives among the positives:

$$TPR = \frac{TP}{P} \tag{2.2}$$

The *false positive rate* is the proportion of false positives among the negatives:

$$FPR = \frac{FP}{N} \tag{2.3}$$

The quantity $1 - FPR$ is called the *specificity* of the test. The *accuracy* of the test is how accurate the classifier is to classify the set of instances:

$$ACC = \frac{TP + TN}{P + N} \tag{2.4}$$

2.2.2 Illustration of the ROC curve approach

Most classifiers allow the trade-off between sensitivity and specificity to be adjusted through a parameter, which in our case will be a classifier threshold. We can compare different classifiers by using a ROC curve, which is a two-dimensional graph where (FPR,TPR) is plotted for changing values of the classifier threshold. An ROC curve depicts relative trade-offs between benefits (true positives) and costs (false positives). For example, the four different confusion matrices given in Table 2.2 have ROC curve representations depicted in Figure 2.1.

TP = 63	FP = 28	91
FN = 37	TN = 72	109
100	100	200

TPR = 0.63
FPR = 0.28
ACC = 0.68

(a) Confusion matrix **A**

TP = 88	FP = 24	112
FN = 12	TN = 76	88
100	100	200

TPR = 0.88
FPR = 0.24
ACC = 0.82

(b) Confusion matrix **B**

TP = 77	FP = 77	154
FN = 23	TN = 23	46
100	100	200

TPR = 0.77
FPR = 0.77
ACC = 0.50

(c) Confusion matrix **C**

TP = 24	FP = 88	112
FN = 76	TN = 12	88
100	100	200

TPR = 0.24
FPR = 0.88
ACC = 0.18

(d) Confusion matrix **D**

Table 2.2: The four different confusion matrices A, B, C and D.

There are several points in ROC space that are worth noticing. The point $(0, 0)$ represents the strategy of never issuing a positive classification. With such a strategy there will be no false positives, but there will also be no true positives. The opposite strategy is represented by the point $(1, 1)$. With this strategy we issue positive classification unconditionally. The best point in ROC space is $(0, 1)$. This point represents perfect classification. At $(0, 1)$ all true positives are found, while no false positives are found.

One point in ROC space is better than another if it is above and to the left the first. The TPR is higher, FPR is lower, or both. In Figure 2.1 we can see that B is better than A, while A is better than both C and D. Classifiers that appears on the left-hand side of an ROC graph, near the X axis, may be thought of as "conservative". Such classifiers make positive classifications only with strong evidence, so they make few false positive errors, but they often have a low rate of true positives as well. Classifiers on the upper right-hand side of an ROC graph may be thought of as "liberal". They make positive classification with weak evidence so they classify nearly all positives correctly, but they often have a high rate of false positives as well.

The diagonal line represents the strategy of randomly guessing a class. For example, if a classifier randomly guesses that a pair is a copypair (i.e. a positive outcome) half the time, it can be expected to get half the positives and half the negatives correct. This would yield the point $(0.5, 0.5)$ in ROC space. A random classifier will produce a point that "slides" back and

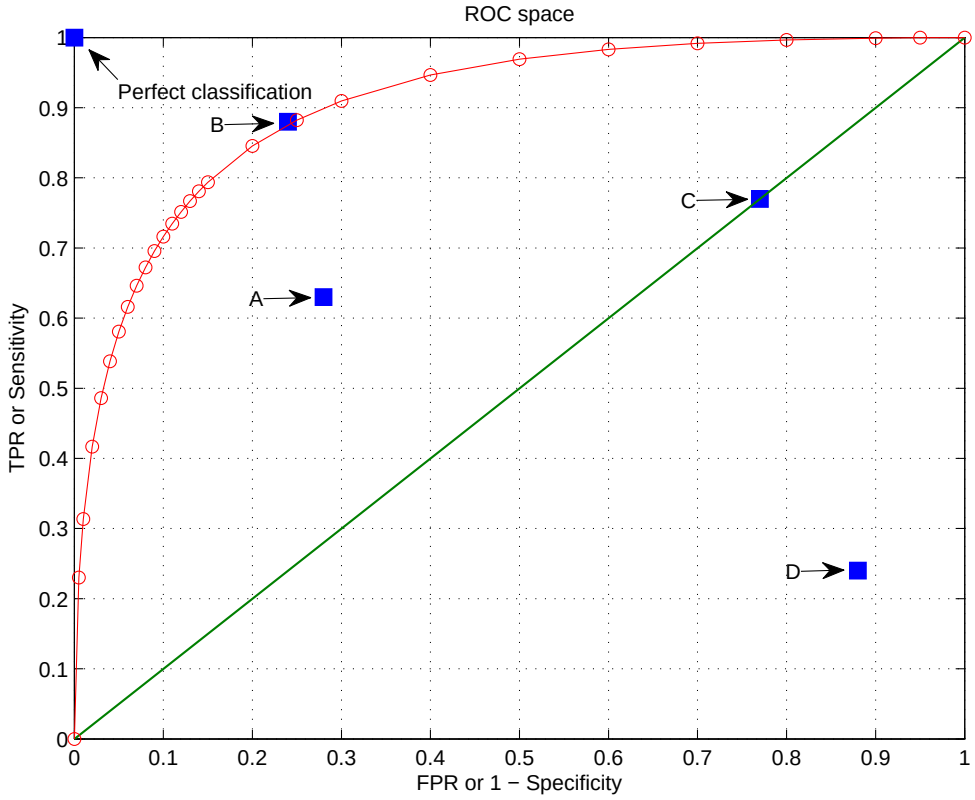


Figure 2.1: ROC points for the four confusion matrices given in Table 2.2. Cf. text for further description of the curve.

forth on the diagonal line based on the frequency with which it guesses the positive class. A classifier that appears on the diagonal line may be said to have no information about the class. To move above the diagonal line the classifier must exploit some information in the data. In Figure 2.1 the classifier C, at the point $(0.77, 0.77)$, has performance which is virtually random. C corresponds to guessing the positive class 77 % of the time.

When a classifier randomly guesses the positive class with a frequency q the classifier has the confusion matrix seen in Table 2.3. From this matrix we can calculate the TPR and the FPR, and as seen in equation 2.5, we get the same result for both rates.

$$\text{TPR} = \frac{TP}{P} = \frac{P \cdot q}{P} = q \tag{2.5}$$

$$\text{FPR} = \frac{FP}{N} = \frac{N \cdot q}{N} = q \tag{2.6}$$

A classifier that appears below the diagonal line performs worse than random guessing. In Figure 2.1, C is an example of such a classifier. A classifier below the diagonal line may be said to have useful information about the data, but it applies this information incorrectly. It is possible to negate a classifier below the diagonal line, since the decision space is symmetrical about the diagonal separating the two triangles. By negating the classifier, we will reverse all its

		Actual class c		
		p	**n**	**total**
Predicted class $\hat{c}$	**p**	$TP = P \cdot q$	$FP = N \cdot q$	**P'**
	n	$FN = P \cdot (1 - q)$	$TN = N \cdot (1 - q)$	**N'**
	total	**P**	**N**	

Table 2.3: Confusion matrix with the actual and predicted numbers of positives and negatives for a test set. This confusion matrix is made by a classifier that randomly guesses the positive class with a frequency q.

classifications. Its true positives will become false negatives, and its false positives will become true negatives. The classifier B is a negation of D.

The hyperbolic graph in Figure 2.1 is an example of a true ROC curve. This curve is made by varying the threshold of a classifier. For each threshold value, we get a new confusion matrix, and the point (FPR,TPR) is plotted in ROC space. We can compare the results of different plagiarism detection programs by making a ROC curve for each of the programs. If the curve of one of the programs lies above the curves of the other programs, then this program performs better than the other programs for all threshold values.

2.3 Outline of the AST concept

I will use an example from Louden (1997) to explain what an AST is. Consider the statement $a[index] = 4 + 2$ which could be a line in some programming language. The lexical analyzer, or scanner, collects sequences of characters from this statement into meaningful units called tokens. The tokens in this statement are: identifier (a) [identifier (index)] = number (4) + number (2).

The syntactical analyzer, or parser, receives the tokens from the scanner and performs a syntactical analysis on them. The parser determines the structural elements of the code and the relationship between these elements. The result of this analysis can either be a parse tree as seen in Figure 2.2, or an AST as seen in Figure 2.3. The internal nodes in these two trees are structural elements of the language, while the leaf nodes, in gray, represents tokens. We can see that the AST is an reduced version of the parse tree. An AST differs from a parse tree by omitting nodes and edges for syntax rules that do not affect the semantics of the program. Therefore, an AST is a better representation than a parse tree since it only contains the nodes that are necessary for representing the code.

2.4 Software for constructing AST

It is a time demanding and complicated process to develop a scanner, a parser and a tree builder for ASTs for a language such as Java. Since the generation of ASTs is only a small part of my thesis, I have used Java Compiler Compiler (JavaCC)[2] and JJTree to generate the above programs for the Java 1.5 version. JavaCC is an open source scanner and parser generator for Java. JavaCC generates a scanner and a parser in Java from a file which contains the definitions

[2]JavaCC: *https://javacc.dev.java.net/*

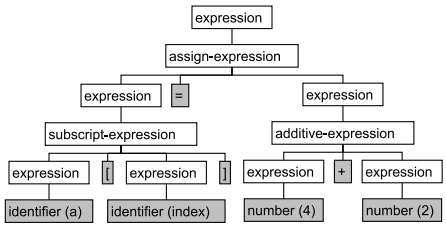


Figure 2.2: Parse tree for the code $a[index] = 4 + 2$. The internal nodes, in white, represents the structural elements of the language, while the leaf nodes, in gray, represents tokens.

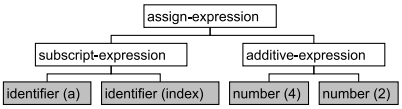


Figure 2.3: AST for the code $a[index] = 4 + 2$. The internal nodes, in white, represents the structural elements of the language, while the leaf nodes, in gray, represents tokens.

of the tokens and the grammar of a language such as Java. To generate a tree builder for the parser, JJTree together with JavaCC were used.

2.5 Assessing similarities between trees

In this section, I present different methods for assessing similarities between trees by the use of a semi-metric approach. A semi-metric is a function which defines the distance between elements of a set. In our problem we have the set P which contains the listings which we want to compare against each other. The distance function d is then defined as $d : P \times P \mapsto \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers. When we apply the function d on two listings in the set, we will get a low distance if the structural similarity is high, and a high distance if the structural similarity is low. If the structure of the two listings are identical, the distance is zero.

For all $p, q \in P$, the function d will be required to satisfy the following conditions:

1. $d(p, q) \geq 0$ (non-negativity) and with equality iff $p = q$
2. $d(p, q) = d(q, p)$ (symmetry)

When using d in our context, then $p = q$ in condition 1 means that p and q have identical structure. A semi-metric is similar to a metric, except that a metric d' also needs to satisfy the triangle inequality. I will define a distance function and a similarity function for most of the methods that I present in the following sections. For each similarity function that I define, there is a direct reciprocal relation to the corresponding distance function. I will only use the similarity functions in the rest of the thesis, since both functions express the same thing.

The methods to be presented work on either ordered or unordered trees. In an ordered tree, the relative order of the children is fixed for each node. The relative order of the children nodes

leads to further distinctions among the nodes in an ordered tree. We let $T = (V, E)$ be an ordered tree. The node $v \in V$ has the children set $W \subseteq V$. If a node $w \in W$ is the first child of v, then we denote that as $w = first[v]$, and if the node $w \in W$ is the last child of v, then we denote that as $w = last[v]$. Each nonfirst children node $w \in W$ has a previous sibling, denoted as $previous[w]$. Also, each nonlast children node $w \in W$ has a next sibling, denoted as $next[w]$.

Some of the methods that I present in the next sections uses the notion of a *mapping*. In our context, a mapping is used to show which nodes that correpond to each other in the two trees. The standard notation for a mapping M is given by $M : X \mapsto Y$. In this thesis, I will use the notation $M \subseteq X \times Y$, where M is a subset of a cartesian product and not a function as in the standard notation. In order for M to be a mapping from X to Y the following condition must be satisfied: if $(x, y), (x', y) \in M$ then $x = x'$.

2.5.1 Distance measures obtained by dynamic programming

The distance metrics that I present in this section uses dynamic programming (Cormen et al., 2001). This method, like the divide-and-conquer method, solves problems by combining the solutions of subproblems. In divide-and-conquer algorithms, the original problem is partitioned into independent subproblems. These subproblems are then solved recursively, and the solutions are combined to solve the original problem. Dynamic programming has an advantage over divide-and-conquer algorithms when the subproblems are not independent, that is, when the subproblems share sub-subproblems. A divide-and-conquer algorithm will in this context do unnecessary work by repeatedly solving common sub-subproblems. On the other hand, a dynamic programming algorithm solves every sub-subproblem just once and stores the result in a table. The next time the subsubproblem is encountered, we only need to look up the answer in the table.

Dynamic programming is well suited for certain optimization problems. For such problems there can be one or more possible solutions. Each solution has a value, and we want to find a solution with the optimal value. Depending on the problem, the optimal value is either a maximum or a minimum value. A solution with the optimal value is called an optimal solution to the problem. It is important to notice that we don't call this solution *the* optimal solution, since there can be several solutions that achieve the optimal value. For problems where we can find an optimal solution with dynamic programming, the problem exhibits a property called optimal substructure. A problem has an optimal substructure if the optimal solution can be built from optimal solutions to subproblems.

For each of the problems that I present in this section, with exception of tree edit distance, I will show how we can characterize the optimal substructure and how we can recursively define an optimal solution.

Longest Common Subsequences

The longest common subsequence (LCS) problem (Cormen et al., 2001) is used when we want to find the longest subsequence which is common to two or more sequences (I will only consider LCS between two sequences). LCS has many applications in computer science. It is the basis of the Unix algorithm diff, and variants of it are widely used in bioinformatics.

A subsequence of a given sequence is just the given sequence with zero or more elements left out. For example, $Z = (T, C, G, T)$ is a subsequence of $X = (A, T, C, T, G, A, T)$. If we have two sequences X and Y we say that Z is a common subsequence of X and Y if Z is a subsequence of both X and Y. And if Z is a subsequence of maximum-length we say that Z is a longest common subsequence of X and Y. For example, $Z = (T, C, T, A)$ is a longest common subsequence of the sequences $X = (A, T, C, T, G, A, T)$ and $Y = (T, G, C, A, T, A)$. This can also be given as a global alignment of X and Y, as shown below:

$$\begin{array}{cccccccccc} \mathrm{A} & \mathrm{T} & - & \mathrm{C} & - & \mathrm{T} & \mathrm{G} & \mathrm{A} & \mathrm{T} \\ - & \mathrm{T} & \mathrm{G} & \mathrm{C} & \mathrm{A} & \mathrm{T} & - & \mathrm{A} & - \end{array} \tag{2.7}$$

Characters in the two sequences that are not part of the longest common subsequence are aligned with the gap character -. In LCS the subproblems correspond to pairs of prefixes of the two sequences. Given a sequence $X = (x_1, \ldots, x_n)$, we define the *i-th* prefix of X, for $i = 0, \ldots, m$, as $X_i = (x_1, \ldots, x_i)$. For $X = (A, T, C, T, G, A, T)$, $X_4 = (A, T, C, T)$ and X_0 is the empty sequence.

For each prefix of the sequences $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_m)$, we find the LCS. There are either one or two subproblems that we must examine for the prefixes $X_i = (x_1, \ldots, x_i)$ and $Y_j = (y_1, \ldots, y_j)$. If $x_i = x_j$ then we must find a LCS of X_{i-1} and Y_{j-1}. Appending $x_i = y_j$ to this LCS yields an LCS of X_i and Y_j. If $x_i \neq y_j$ then we must solve two subproblems, which are to find an LCS of X_{i-1} and Y_j and to find an LCS of X_i and Y_{j-1}. Whichever of these two LCS's is longer is an LCS of X_i and Y_j. Since these cases exhaust all possibilities, we know that one of the optimal subproblem solutions must be used within an LCS of X_i and Y_j.

We can then define $c[i, j]$ to be the length of an LCS of the prefixes X_i and Y_j. If either $i = 0$ or $j = 0$, one of the sequences has length 0, so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula in equation 2.8.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases} \tag{2.8}$$

To use LCS on two ASTs we need two sequences of the nodes that reflects the structure of the two trees. This can be achieved by doing either a preorder or postorder traversal of the trees.

For the longest common subsequence problem between two ASTs $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ of the programs p and q, define the normalized distance function $d(p, q)$ and the normalized similarity function $sim(p, q)$ in equations 2.9 and 2.10. We call the two functions normalized since we normalize by the sum of the nodes in T_1 and T_2.

$$d(p,q) = \frac{|V_1| + |V_2| - 2 \cdot LCS(p,q)}{|V_1| + |V_2|} \tag{2.9}$$

$$sim(p,q) = \frac{2 \cdot LCS(p,q)}{|V_1| + |V_2|} \tag{2.10}$$

Needleman-Wunsch

The Needleman-Wunsch (NW) algorithm (Needleman and Wunsch, 1970) is used to perform a global alignment, with gaps if necessary, on two sequences. The algorithm is a well-known method for comparison of protein or nucleotide sequences, although today most bioinformatics applications use faster heuristic algorithms such as BLAST. If we instead want to find the best local alignment between two sequences, we could use the Smith-Waterman algorithm.

The algorithm is similar to LCS in that it finds a global alignment between two sequences, but there are some differences. Scores for aligned characters are specified by a similarity matrix S. Here, $S(i, j)$ is the similarity of characters i and j. It also uses a linear gap penalty, called d, to penalize gaps in the alignment. An example of an similarity matrix for DNA sequences is shown in 2.11.

$$S = \begin{array}{c} \\ A \\ G \\ C \\ T \end{array} \begin{array}{c} \begin{array}{cccc} - & A & G & C & T \end{array} \\ \begin{bmatrix} 10 & -1 & -3 & -4 \\ -1 & 7 & -5 & -3 \\ -3 & -5 & 9 & 0 \\ -4 & -3 & 0 & 8 \end{bmatrix} \end{array} \tag{2.11}$$

In equation (2.12) we see an optimal global alignment of the sequences AGACTAGTTAC and CGAGACGT. The score of this alignment is $S(A,C)+S(G,G)+S(A,A)+3\times d+S(G,G)+S(T,A)+S(T,C)+S(A,G)+S(C,T) = -3+7+10+3\times -5+7+-4+0+-1+0 = 1$, when using a gap penalty d which equals -5.

$$\begin{array}{cccccccccccc} A & G & A & C & T & A & G & T & T & A & C \\ C & G & A & - & - & - & G & A & C & G & T \end{array} \tag{2.12}$$

Since NW is very similar to LCS we can define $F[i, j]$ to be the score of NW of the prefixes X_i and Y_j of the two sequences X and Y. If either $i = 0$ or $j = 0$, one of the sequences has length 0. In that case, NW will produce the score $d \cdot n$, if $j = 0$, or $d \cdot m$, if $i = 0$, where n and m are the lengths of X and Y respectively. If both $i = 0$ and $j = 0$, then the score will be 0. The optimal substructure of the NW problem gives the recursive formula in equation 2.13.

$$F[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0 \\ d \cdot i & \text{if } i \neq 0 \text{ and } j = 0 \\ d \cdot j & \text{if } i = 0 \text{ and } j \neq 0 \\ max(F[i-1,j-1] + S[i-1,j-1], & \text{otherwise} \\ \quad F[i,j-1] + d, \quad F[i-1,j] + d) & \end{cases} \tag{2.13}$$

The use of Needleman-Wunsch in this thesis will be explained later.

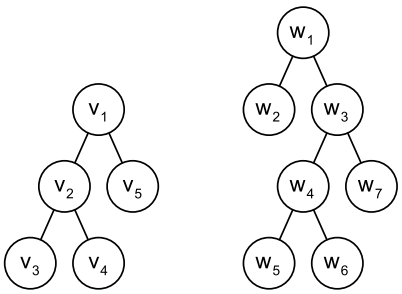
Tree edit distance

A tree T_1 can be transformed into the tree T_2 by the use of elementary edit operations, where each operation has an associated cost. The operations are: deletion of a node in a tree, insertion of a node in a tree, and substitution of a node in a tree with a node in another tree. We get a

sequence of edit operations when we transform T_1 into T_2. The cost of the least-cost sequence of transforming T_1 into T_2 is the edit distance between the two trees. There are different forms of tree edit distance. I will consider the tree edit distance method presented in Valiente (2002) for two unlabeled or labeled ordered trees. In this method the insert and delete operations are restricted to leaf nodes.

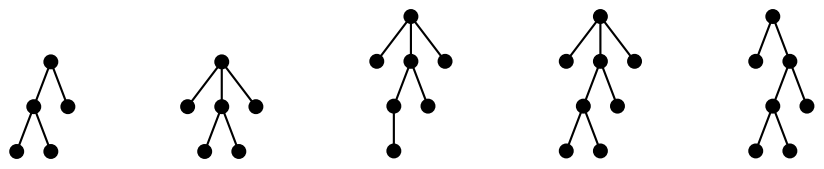
If we have two labeled ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ then T_1 can be transformed into T_2 by a sequence of elementary edit operations. Let T be a labeled ordered tree that is initially equal to T_1. At the end of the transformation $T = T_2$. The elementary edit operations on T is either the deletion from T of a leaf node $v \in V_1$, denoted by (v, λ); the substitution of a node $w \in V_2$ for a node $v \in V_1$, denoted by (v, w); or the insertion into T of a node $w \in V_2$ as a new leaf node, denoted by (λ, w).

The transformation of T_1 into T_2 is given by an ordered relation $E = e_1 e_2 \dots e_n$, where $e_i \in (V_1 \cup \{\lambda\}) \times (V_2 \cup \{\lambda\})$. In Figure 2.5 we have a transformation of T_1 in Figure 2.4a into T_2 in Figure 2.4b, where substitution of corresponding nodes is left implicit in the figure. The transformation E is given by $[(v_1, w_1), (\lambda, w_2), (v_2, w_3), (v_3, w_4), (\lambda, w_5), (\lambda, w_6), (v_4, w_7), (v_5, \lambda)]$.



(a) Tree T_1. (b) Tree T_2.

Figure 2.4: Two trees T_1 and T_2.



(a) $T = T_1$ (b) (λ, w_2) (c) (λ, w_5) (d) (λ, w_6) (e) (v_5, λ) $(T = T_2)$

Figure 2.5: The transformation of T_1 to T_2. Substitution of corresponding nodes is left implicit in the figure. We start with $T = T_1$ in 2.5a and end up with $T = T_2$ in 2.5e.

Let the cost of an elementary edit operation $(v, w) \in E$ be given by $\gamma(v, w)$. Then the cost of performing all operations in E is given by $\gamma(E) = \sum_{(v,w)\in E} \gamma(v, w)$. Let us assume that the cost of a elementary operations is $\gamma(v, w) = 1$ if either $v = \lambda$ or $w = \lambda$, and $\gamma(v, w) = 0$ otherwise. Then the transformation E in Figure 2.5 has cost equal to 4. This transformation is also a least-cost transformation, so the edit distance $\delta(T_1, T_2)$ of T_1 into T_2 is 4. It is important to point out that the edit distance $\delta(T_2, T_1)$ for transforming T_2 into T_1 equals $\delta(T_1, T_2)$. By just changing every edit operation $(v, w) \in E$ to (w, v) we get the transformation $E' = [(w_1, v_1), (w_2, \lambda), (w_3, v_2), (w_4, v_3), (w_5, \lambda), (w_6, \lambda), (w_7, v_4), (\lambda, v_5)]$ of T_2 into T_1.

For a transformation E of T_1 into T_2 it is important that the transformation is valid. First, all deletion and insertion operations must be made on leaves only. It is also important in what order the deletions and insertions are performed. For deletions, (v_j, λ) occurs before (v_i, λ) in E for all $(v_i, \lambda), (v_j, \lambda) \in E$ such that node v_j is a descendant of node v_i in T_1. While for insertions, (λ, v_i) occurs before (λ, v_j) in E for all $(\lambda, v_i), (\lambda, v_j) \in E$ such that node v_j is a descendant of node v_i in T_2. We can see in Figure 2.5 that the deletions and the insertions are performed in the correct order. The second requirement for a transformation E to be valid is that both parent and sibling order must be preserved by the transformation. This is done to ensure that the result of the transformation is an ordered tree. In a valid transformation of T_1 into T_2, the parent of a nonroot node v of T_1 which is substituted by a nonroot node w of T_2 must be substituted by the parent of node w. And, whenever sibling nodes of T_1 are substituted by sibling nodes of T_2, the substitution must preserve their relative order.

The second requirement for a transformation between two ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ to be valid is formalized by the notion of a mapping. A mapping M of T_1 to T_2 is a bijection $M \subseteq W_1 \times W_2$, where $W_1 \subseteq V_1$ and $W_2 \subseteq V_2$, such that

- $(root[T_1], root[T_2]) \in M$ if $M \neq \emptyset$
- $(v, w) \in M$ only if $(parent[v], parent[w]) \in M$, for all nonroot nodes $v \in W_1$ and $w \in W_2$
- v_1 is a left sibling of v_2 if and only if w_1 is a left sibling of w_2, for all nodes $v_1, v_2 \in W_1$ and $w_1, w_2 \in W_2$ with $(v_1, w_1), (v_2, w_2) \in M$

For the transformation E in Figure 2.5 we get the mapping in Figure 2.6. The transformation E of T_1 into T_2 is then valid, since insertions and deletions are performed on leaves only and in the correct order, and the substitutions constitute a mapping.

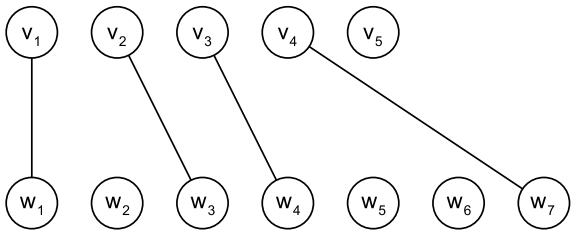


Figure 2.6: The mapping of the nodes in T_1 and T_2.

For the tree edit distance problem between two ASTs $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ of the programs p and q, define the normalized distance function $d(p, q)$ and the normalized similarity function $sim(p, q)$ as follows:

$$d(p,q) = \frac{\delta(T_1, T_2)}{|V_1| + |V_2|} \tag{2.14}$$

$$sim(p,q) = \frac{2 \cdot |M|}{|V_1| + |V_2|} \tag{2.15}$$

2.5.2 Distance measures based on tree isomorphism

When we look at isomorphism between trees, the trees can either be ordered or unordered, and they can be labeled or unlabeled. If the two trees are unlabeled or if the labels are of no importance then two trees are isomorphic if they share the same tree structure. Two labeled trees, on the other hand, are isomorphic if the underlying trees are isomorphic and if the corresponding nodes in the two trees share the same label. In this section we will look at tree isomorphism and maximum common subtree isomorphism for both ordered and unordered trees. The examples that I show for different kinds of isomorphisms will be on unlabeled trees.

Ordered and unordered tree isomorphism

Two ordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ are isomorphic, denoted by $T_1 \cong T_2$, if there is a bijective correspondence between their node sets, denoted by $M \subseteq V_1 \times V_2$, which preserves and reflects the structure of the ordered trees. M is an ordered tree isomorphism of T_1 to T_2 if the following conditions are satisfied.

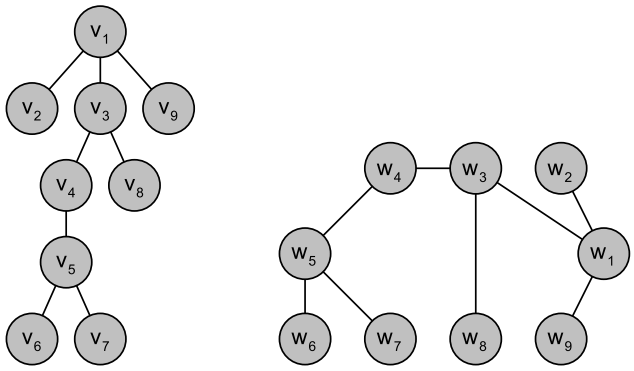
- $(root[T_1], root[T_2]) \in M$
- $(first[v], first[w]) \in M$ for all non-leaves $v \in V_1$ and $w \in V_2$ with $(v, w) \in M$
- $(next[v], next[w]) \in M$ for all non-last children $v \in V_1$ and $w \in V_2$ with $(v, w) \in M$

If there exists an ordered tree isomorphism between two ordered trees then the two trees are said to be the same tree. The two isomorphic trees can look very different, since they can be differently labeled or drawn differently. In Figure 2.7 we have two isomorphic ordered trees that are drawn differently.

In the case of unordered trees, we say that two unordered trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ are isomorphic, denoted by by $T_1 \cong T_2$, if there is a bijective correspondence between their node sets, denoted by $M \subseteq V_1 \times V_2$, which preserves and reflects the structure of the unordered trees. M is an unordered tree isomorphism of T_1 to T_2 if the following conditions are satisfied.

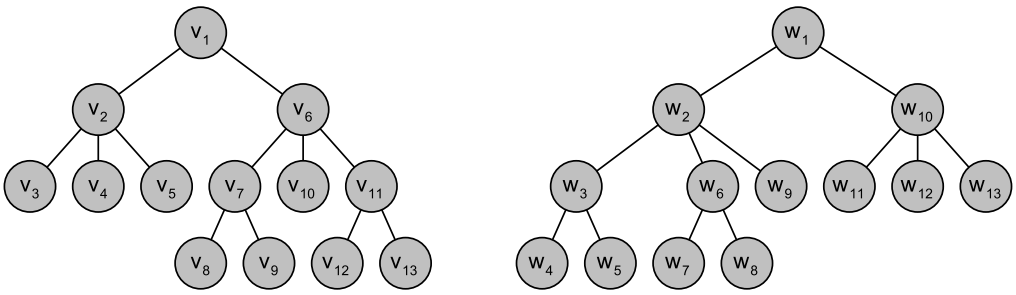
- $(root[T_1], root[T_2]) \in M$
- $(parent[v], parent[w]) \in M$ for all nonroots $v \in V_1$ and $w \in V_2$ with $(v, w) \in M$

We can see that the mapping M is less strict for unordered trees than for ordered trees, since an unordered tree isomorphism allows permutations of the subtrees rooted at some node. In Figure 2.8 we have an example of two unordered trees T_1 and T_2 which are isomorphic.



(a) Ordered tree T_1. (b) Ordered tree T_2.

Figure 2.7: Example of isomorphic ordered trees from Valiente (2002). The nodes are numbered according to the order in which they are visited during a preorder traversal.



(a) Unordered tree T_1. (b) Unordered tree T_2.

Figure 2.8: Example of isomorphic unordered trees from Valiente (2002). The nodes are numbered according to the order in which they are visited during a preorder traversal.

For both the ordered and unordered tree isomorphism between two ASTs $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ of the programs p and q, define the normalized distance function $d(p, q)$ and the normalized similarity function $sim(p, q)$ as follows:

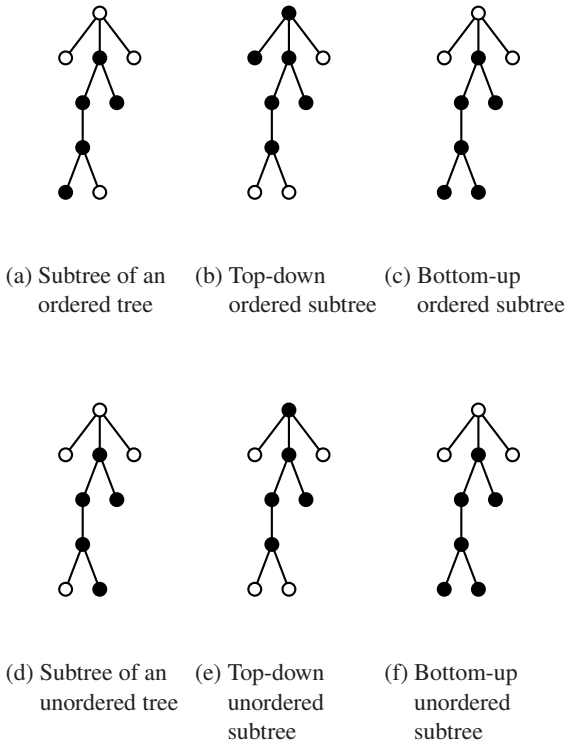
$$d(p,q) = \begin{cases} 0 & \text{if } 2 \cdot |M| = |V_1| + |V_2| \\ 1 & \text{otherwise} \end{cases} \tag{2.16}$$

$$sim(p,q) = \begin{cases} 1 & \text{if } 2 \cdot |M| = |V_1| + |V_2| \\ 0 & \text{otherwise} \end{cases} \tag{2.17}$$

Ordered and unordered maximum common subtree isomorphism

Another form of isomorphism, which doesn't require a bijection between the node sets of two trees, is the maximum common subtree isomorphism. A maximum common subtree of two ordered or unordered trees T_1 and T_2 is the largest subtree shared by both trees.

(W, S) is a subtree of a tree $T = (V, E)$, if $W \subseteq V$, $S \subseteq E$ and the nodes in W are connected. We call (W, S) an unordered subtree, or just a subtree, if T is an unordered tree. If T is an ordered tree, then (W, S) is an ordered subtree if $previous[v] \in W$ for all nonfirst children nodes $v \in W$. A common subtree of two ordered or unordered trees can either be a top-down subtree or a bottom up subtree. (W, S) is a top-down ordered or unordered subtree if $parent[v] \in W$ for all nodes different from the root, and it is a bottom-up ordered or unordered subtree if $children[v] \in W$, where $children[v]$ denote the set of children for node v, for all nonleaves $v \in W$. In Figure 2.9 we can see a subtree, a top-down subtree, and a bottom-up subtree of an ordered and an unordered tree.



(a) Subtree of an ordered tree

(b) Top-down ordered subtree

(c) Bottom-up ordered subtree

(d) Subtree of an unordered tree

(e) Top-down unordered subtree

(f) Bottom-up unordered subtree

Figure 2.9: A subtree, a top-down subtree, and a bottom-up subtree of an ordered and an unordered tree from Valiente (2002).

We define the common subtree of the two trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ as a structure (X_1, X_2, M), where $X_1 = (W_1, S_1)$ is a subtree of T_1, $X_2 = (W_2, S_2)$ is a subtree of T_2, and $M \subseteq W_1 \times W_2$ is a tree isomorphism of X_1 to X_2. A common subtree (X_1, X_2, M) of T_1 to T_2 is maximum if there is no subtree (X_1', X_2', M') of T_1 to T_2 with $size[X_1] < size[X_1']$. In Appendix A there are examples of the four different maximum common subtree isomorphisms from Valiente (2002).

For all the different maximum common subtree isomorphisms between two ASTs $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ of the programs p and q, define the normalized distance function

$d(p,q)$ and the normalized similarity function $sim(p,q)$ as follows:

$$d(p,q) = \frac{|V_1| + |V_2| - 2 \cdot |M|}{|V_1| + |V_2|} \tag{2.18}$$

$$sim(p,q) = \frac{2 \cdot |M|}{|V_1| + |V_2|} \tag{2.19}$$

2.5.3 The different distance and similarity functions

For the sake of convenience I have compiled the distance and similarity functions in the two previous sections.

Longest Common Subsequence	
$d(p,q) = \frac{\|V_1\|+\|V_2\|-2\cdot LCS(p,q)}{\|V_1\|+\|V_2\|}$	$sim(p,q) = \frac{2\cdot LCS(p,q)}{\|V_1\|+\|V_2\|}$
Tree Edit Distance	
$d(p,q) = \frac{\delta(T_1,T_2)}{\|V_1\|+\|V_2\|}$	$sim(p,q) = \frac{2\cdot\|M\|}{\|V_1\|+\|V_2\|}$
Tree Isomorphism	
$d(p,q) = \begin{cases} 0 & \text{if } 2\cdot\|M\| = \|V_1\| + \|V_2\| \\ 1 & \text{otherwise} \end{cases}$	$sim(p,q) = \begin{cases} 1 & \text{if } 2\cdot\|M\| = \|V_1\| + \|V_2\| \\ 0 & \text{otherwise} \end{cases}$
Maximum Common Subtree Isomorphism	
$d(p,q) = \frac{\|V_1\|+\|V_2\|-2\cdot\|M\|}{\|V_1\|+\|V_2\|}$	$sim(p,q) = \frac{2\cdot\|M\|}{\|V_1\|+\|V_2\|}$

Table 2.4: The different distance and similarity functions.

Chapter 3

Modifications of the JavaCC ASTs

The grammar used to build the ASTs is from the JavaCC homepage[1]. For the rest of the thesis I will call this grammar the *original grammar*. In its simplest form JavaCC generates very large ASTs, since it creates nodes for all the non-terminals in the grammar. As many of the created nodes carry no structure information and just complicate the analysis, I first developed a procedure to remove these uninformative nodes. Moreover, I have sought ways to modify the original grammar to ease the comparison of ASTs.

3.1 Removal of redundant nodes

In Listing 3.1 we have a simple Java listing and in Figure 3.1a we have the abstract syntax tree of this listing.

```
public class HelloWorld {

    public static void main(String args[]){
        System.out.println("Hello World!");
    }
}
```

Listing 3.1: Class HelloWorld

In the AST in Figure 3.1a there are a lot of nodes that carry no structure information. We can see in the tree that the statement `System.out.println(...)`, starting at *Statement*, has more than half of the nodes in the tree. And of these nodes, the string argument to the method *println* is all the nodes from *Expression* to *StringLiteral*. The reason we get such a long list of nodes is that JavaCC wants to ensure precedence between different operators in an expression, but when we only have a string literal then other nodes in this list do not add any structure information to the tree.

When comparing the structure of two ASTs, we can remove nodes in the two trees that are not relevant for the comparison. Example of such nodes are nodes that are part of a list in the tree and are not the head or the tail of the list. A list starts with a node that has two or more children and end at a node that has two or more children or that is a leaf node. The nodes that we remove are highlighted with gray in Figure 3.1a. After removing these nodes we get the tree in Figure 3.1b.

[1] JavaCC grammar for Java 1.5: *https://javacc.dev.java.net/files/documents/17/3131/Java1.5.zip*

The figure also contains some nodes that are highlighted in black that are not the head or tail in a list. The node with label *Block* is not removed since this would make it harder to compare a block with a single statement and a block that contains multiple statements. We also do not remove the node with label *ClassOrInterfaceBody* since this would make it harder to compare two classes or interfaces where one contains only a single declaration while the other contains multiple declarations. The two other nodes in black are also not removed since they are labeled with information that are important when manually comparing ASTs of two programs. The node labeled with *ClassOrInterfaceDeclaration* contains the name of a class or an interface, while the node *MethodDeclarator* contains the name of a method. These nodes are not important for the structure, however by removing these it would be much harder to find the corresponding classes, interfaces and methods of two ASTs when comparing their GML files[2].

3.2 Modifications of the JavaCC grammar

I have done some modifications to the original grammar to get a better distinction between different primitive types and literals in the AST. Moreover, some modifications have been done to the grammar of loops and selection statements. The modifications are presented in the following subsections.

3.2.1 The types in the grammar

Primitive types and reference types are the two main types in the Java language. The primitive types are divided into integral types (byte, short, int, long and char), floating point types (float and double), and boolean type. The reference types are divided into class types, interface types and array types.

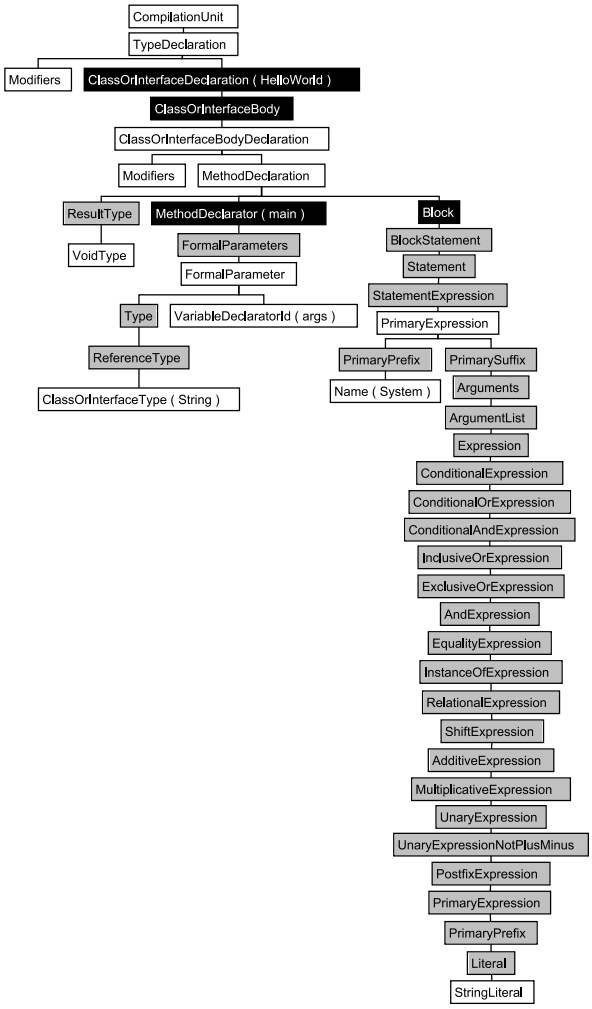
In the original grammar the rule for the primitive types is defined as seen in Listing 3.2. In this listing, "|" is used to denote choice. Since JavaCC only creates nodes for the non-terminals in the grammar, it will make no distinction between the different primitive types. All the different primitive types will then be represented with a node labeled PrimitiveType, which makes it harder to compare trees that contains different primitive types.

To make a distinction between the different types, I have changed the rule in Listing 3.2 to the rules in Listing 3.3. I have also made a further distinction between char and the rest of the integral types by making an own type for char, since a char is mostly used to represent a Unicode character. The effect of the new rules is that we create two nodes for each primitive type, instead of only one node as for the rule in Listing 3.2. For each primitive type we first create a node labeled PrimitiveType, and then we create a child of this node which is either labeled IntegerType, FloatingPointType, CharacterType, or BooleanType.

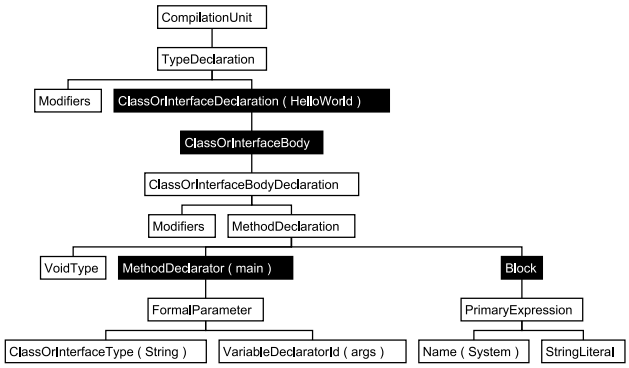
```
PrimitiveType        ::= "byte" | "short" | "int" | "long" | "float" |
                         "double" | "char" | "boolean"
```

Listing 3.2: Rule for primitive types

[2] See Section 6.4 for more information.



(a) AST of the class HelloWorld



(b) Reduced AST of the class HelloWorld

Figure 3.1: a) AST produced by JavaCC, b) reduced AST. The nodes that do not define the head or the tail of a list is highlighted in gray or black. The nodes in gray are removed, while the nodes in black are not.

```
PrimitiveType       ::= IntegerType | FloatingPointType | CharacterType |
                        BooleanType

IntegerType         ::= "byte" | "short" | "int" | "long"

FloatingPointType   ::= "float" | "double"

CharacterType       ::= "char"

BooleanType         ::= "boolean"
```

Listing 3.3: New rules for primitive types

3.2.2 The literals in the grammar

A literal is the source code representation of a value of a primitive type, the String type or the null type. In the original grammar we have the same problem for literals as for the primitive types. In Listing 3.4 we can see that JavaCC makes no distinction between literals of the different primitive types. To make a distinction between these literals, I have changed the rules for literals to the rules in Listing 3.5.

```
Literal             ::= INTEGER_LITERAL | FLOATING_POINT_LITERAL |
                        CHARACTER_LITERAL | STRING_LITERAL |
                        BooleanLiteral | NullLiteral

BooleanLiteral      ::= "true" | "false"

NullLiteral         ::= "null"
```

Listing 3.4: Rules for literals

```
Literal             ::= IntegerLiteral | FloatingPointLiteral |
                        CharacterLiteral | StringLiteral |
                        BooleanLiteral | NullLiteral

IntegerLiteral      ::= INTEGER_LITERAL

FloatingPointLiteral ::= FLOATING_POINT_LITERAL

CharacterLiteral    ::= CHARACTER_LITERAL

StringLiteral       ::= STRING_LITERAL

BooleanLiteral      ::= "true" | "false"

NullLiteral         ::= "null"
```

Listing 3.5: New rules for literals

3.2.3 The selection statements and the loops in the grammar

In a Java program a block is a sequence of statements, local class declarations, and local variable declarations within braces. In the AST the block is represented by a subtree rooted at a node with the label *Block*, where the children are the statements and declarations from the sequence.

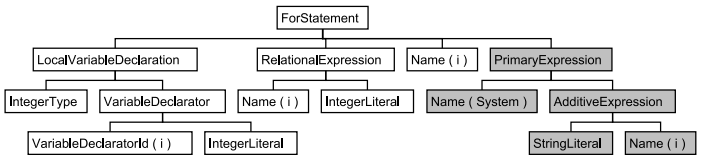
A block is often used in if-, while-, do-while-, and for-statements. These statements can also have a single statement or declaration instead of a block. In Listings 3.6 and 3.7 we have examples of two for-loops, where one contains a block while the other does not. We would say that the two for-loops are identical since they do the same thing. The problem is that we get different tree representations for the two for-loops. In Figures 3.2a and 3.2b we have the tree representations of the two listings, where the subtrees that differ are highlighted in both trees.

```
...
for(int i = 0; i < 10; i++)
    System.out.println("i = " + i);
...
```

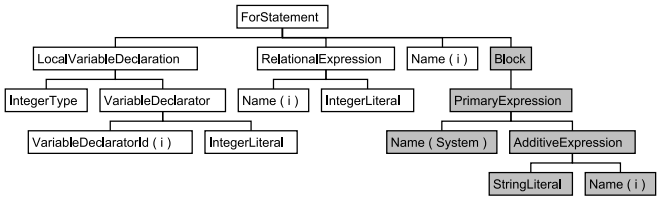
Listing 3.6: For-loop without a block

```
...
for(int i = 0; i < 10; i++){
    System.out.println("i = " + i);
}
...
```

Listing 3.7: For-loop with a block



(a) For-loop without a block.



(b) For-loop with a block.

Figure 3.2: Two for-loops, one with and one without a block. The subtrees that differ are highlighted in the two trees.

This problem is solved by always using a block in an if-, a while-, a do-while, and a for-statement. For these four statements we have the rules from the original grammar in Listing 3.8. In this listing, "<IDENTIFIER>" represents an identifier, while "+" is used to denote zero or one occurrence. In Listing 3.9 I have changed these rules and added the new rule *StatementBlock*. In the AST these four statements will now have a subtree rooted at the node *StatementBlock*. After the reduction of the AST, this node will either have a node labeled *Block* or a single statement or declaration as a child. If the child is *Block*, then the node *StatementBlock* is replaced by this node. And if the child is a single statement or declaration, then the label of the node *StatementBlock* is changed to *Block*.

```
Statement       ::= LabeledStatement | AssertStatement | Block |
                    EmptyStatement | StatementExpression() ";" |
                    SwitchStatement | IfStatement | ElseStatement |
                    WhileStatement | DoStatement | ForStatement |
                    BreakStatement | ContinueStatement | ReturnStatement |
                    ThrowStatement | SynchronizedStatement | TryStatement

IfStatement     ::= "if" "(" Expression ")" Statement ("else" Statement)+

WhileStatement  ::= "while" "(" Expression() ")" Statement

DoStatement     ::= "do" Statement "while" "(" Expression() ")" ";"

ForStatement    ::= "for" "(" ( (Type <IDENTIFIER> ":" Expression) |
                    ((ForInit)+ ";" (Expression)+ ";" (ForUpdate)+) ) ")"
                    Statement
```

Listing 3.8: Rules for the different loops and selection statements

```
StatementBlock  ::= Statement

Statement       ::= ...

IfStatement     ::= "if" "(" Expression ")" StatementBlock
                    ("else" StatementBlock)+

WhileStatement  ::= "while" "(" Expression() ")" StatementBlock

DoStatement     ::= "do" StatementBlock "while" "(" Expression() ")" ";"

ForStatement    ::= "for" "(" ( (Type <IDENTIFIER> ":" Expression) |
                    ((ForInit)+ ";" (Expression)+ ";" (ForUpdate)+) ) ")"
                    StatementBlock
```

Listing 3.9: New rules for the different loops and selection statements

We can also have a block associated with a case-label in a switch-statement. A case-label have none or more statements and/or declarations associated with it. For a case-label it is not mandatory to have the statements and/or declarations within a block when it has two or more statements and/or declarations. In this way it is different from the other four statements. For the switch-statement we have the rules from the original grammar in Listing 3.10. In this listing, "*" is used to denote zero or more occurrences. In Listing 3.11 I have changed these rules and added the new rule *SwitchLabelBlock*. In the AST each case-label will now have a subtree rooted at the node *SwitchLabelBlock*. After the reduction of the AST, this node have either no child, or a node labeled *Block* as child, or one or more statements and/or declarations as children. If it has no child, then I remove *SwitchLabelBlock* from the tree. If the child is *Block*, then the node *SwitchLabelBlock* is replaced by this node. And if the node *SwitchLabelBlock* has one or more children, then its label is changed to *Block*. We also have the special case where the node *SwitchLabelBlock* has two or more children and the first child is labeled *Block*. Then the node *SwitchLabelBlock* is replaced by the first child, and the other children becomes the children of this node.

```
BlockStatement      ::= LocalVariableDeclaration ";" | Statement |
                        ClassOrInterfaceDeclaration

SwitchStatement     ::= "switch" "(" Expression ")" "{"
                        (SwitchLabel (BlockStatement)*)* "}"

SwitchLabel         ::= ("case" Expression ":") | ("default" ":")
```

Listing 3.10: Rule for a switch

```
SwitchLabelBlock    ::= (BlockStatement)*

BlockStatement      ::= ...

SwitchStatement     ::= "switch" "(" Expression ")" "{"
                        (SwitchLabel SwitchLabelBlock)* "}"

SwitchLabel         ::= ...
```

Listing 3.11: New rule for a switch

Chapter 4

ASTs generated by the most frequent cheating strategies

When a student uses source code written by someone else, then he often modifies the code for the purpose of disguising the plagiarism. These modifications often cause differences between the ASTs of the original and the modified source code. Since I measure the similarity between ASTs, I want to assess how the AST of the modified code corresponds to the AST of the original code when using different cheating strategies.

In this chapter I provide individual source code examples of the most frequently used cheating strategies listed in Section 2.1.3. For each example I compare the ASTs of the original code and the modified code. In the comparison I identify the similarities and the differences between the two ASTs, and I identify what kind of transformations that can be used to transform the AST of the original code into the AST of the modified code. And finally, I assess which of the transformations that are most important to detect and how they affect the development of the new similarity measures.

4.1 Applying the different cheating strategies

The cheating strategies are illustrated by use of a selected part of one of the Java listings[1] *Studentregister.java*, *Suduko.java* and *ArrayParameter.java*. There will also be some code examples that are not from these listings.

4.1.1 Changing the formatting, the comments and renaming the identifiers

White spaces and comments are discarded when we build the AST, so changing the formatting and/or the comments will have no effect. Changing the names of the identifiers will also have no effect. When we change the name, we only change the value of the identifier node. We will still have the same type of node, but now with a different value. With this strategy none transformations are applied on the AST of the original code. This strategy is of no importance for the development of new similarity measures since we get no differences between the two ASTs.

[1] Java listings: *http://www.uio.no/studier/emner/matnat/ifi/INF1000/h06/programmer/*

4.1.2 Changing the order of operands in expressions

For an expression exp we can change the order of some or all the operands and get the expression exp', as long as the evaluations of the two expressions gives the same result. Listing 4.1 contains code from the method `main` in *Suduko.java*. By changing the order of some of the operands in the two expressions, we get the modified code in Listing 4.2.

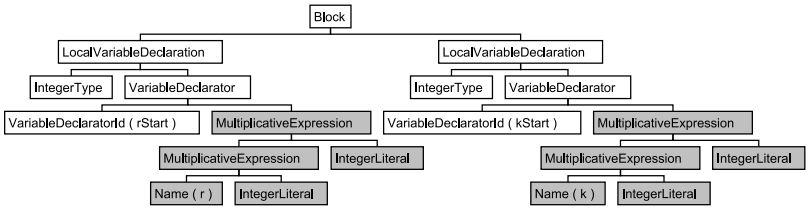
```
...
// test 3x3 felt
int rStart = (r/3) *3;
int kStart = (k/3) *3;
...
```

Listing 4.1: The code before changing the order of operands in expressions

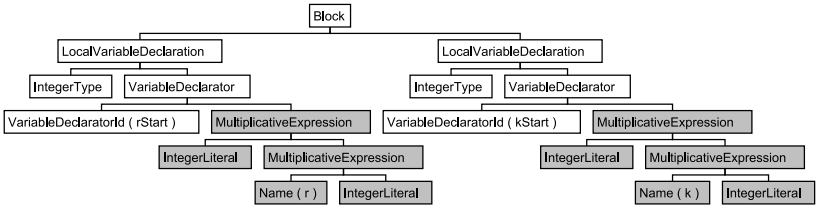
```
...
// test 3x3 felt
int rStart = 3 * (r/3);
int kStart = 3 * (k/3);
...
```

Listing 4.2: The code after changing the order of operands in expressions

The ASTs for Listings 4.1 and 4.2 are shown in Figures 4.1a and 4.1b. We can see that the children of the two subtrees rooted at *MultiplicativeExpression* in Figure 4.1a, have been rotated in Figure 4.1b. That is, the transformation that is applied in this case consists of one or more subtree rotations, where the number of rotations depends on the number of operands that switch places. This strategy produces few differences between the ASTs of two Java listings, since the transformations operates on subtrees which generally are of small sizes.



(a) AST of the code from Listing 4.1.



(b) AST of the code from Listing 4.2.

Figure 4.1: ASTs of the codes from Listings 4.1 and 4.2.

4.1.3 Changing the data types

Sometimes the data type of a variable or field can be changed to another type. Depending on the use of the variable or field, the change of the data type can require further changes in the code. The effect on the AST of replacing one data type with another depends on the data type it is changed to. If an *Integral* type is replaced with another *Integral* type there will be no differences. However, if an *Integral* type is replaced with for instance a *Class* type, then the two trees will differ. Suppose, for example, that we replace the statement `int a = 1` with `Integer a = 1`. The two ASTs are shown in Figures 4.2a and 4.2b. Observe that they differ by only one node. If we instead had changed it to `float a = 1.0f`, the ASTs would differ a bit more since the *IntegerLiteral* would be replaced with a *FloatingPointLiteral*.

(a) AST of the original code. (b) AST of the modified code

Figure 4.2: ASTs of the original and the modified code from Section 4.1.3.

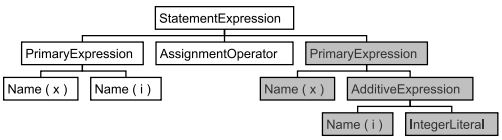
When using this strategy, the transformations depend on what we change the data type to. First, if we replace it with a type of the same kind, then no transformation is applied. Second, if we change it to a type of a different kind, then at least one node and no more than two nodes are changed. Two nodes are changed if it is part of a variable initialization and the literal that is assigned to the variable needs to be changed. And last, the change of the data type can require further changes in the tree. Here, the number of transformations and the types of transformations depend on the different changes that are necessary for making the program work. With this strategy we get few differences between the ASTs of two Java listings since the transformations operates on subtrees of small sizes.

4.1.4 Replacing an expression with an equivalent expression

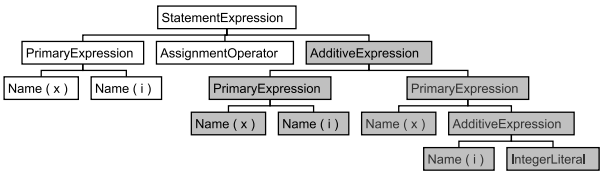
An expression can be replaced with an equivalent expression. In *ArrayParameter.java* we have the method `finnDelsummer`. In the for-loop in this method we have the expression `x[i] += x[i-1]`, which we can replace with `x[i] = x[i] + x[i-1]`. The resulting ASTs are shown in Figures 4.3a and 4.3b. For this strategy we use a transformation that replaces a subtree of an expression with another subtree, or we use a transformation that replaces the first subtree with another subtree that has the first subtree as a subtree. As see in Figure 4.3b the subtree rooted at *PrimaryExpression* in Figure 4.3a has been replaced with a subtree rooted at *AdditiveExpression*. That one subtree is replaced with another subtree is logical since one expression is replaced by another expression.

We can see that there are some similarities between the two subtrees, since the second expression uses the first expression as a sub-expression. One could expect it to be some similarity between two expressions that do the same thing, but that does not need to be the case. The two expressions can be totally different, and have no similarity at all. It can therefore be very hard to determine if it is a case of cheating or if the programmers have just written different expressions.

When using this strategy on a Java listing, we get small differences between the ASTs since the subtrees that represents expressions are of small sizes.



(a) AST of the original code.



(b) AST of the modified code

Figure 4.3: ASTs of the original and the modified code from Section 4.1.4.

4.1.5 Adding redundant code

In the method `registerStudent` in the class `Institutt` in *Studentregister.java* we have the code in Listing 4.3. If we add some statements used for debugging to this code, we get the code in Listing 4.4. Since code used for debugging is not necessary in the final version of the source code, these statements should have been removed before the code was handed in.

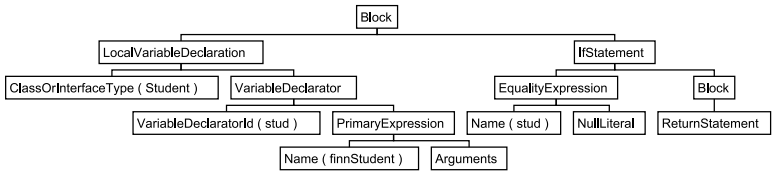
```
...
Student stud = finnStudent();

if (stud == null) { // maks antall studenter er nådd.
    return;
}
...
```

Listing 4.3: Code before adding redundant statements and variables

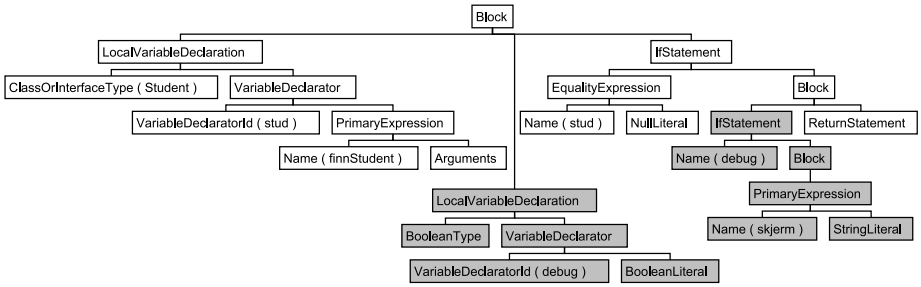
```
...
Student stud = finnStudent();

boolean debug = false;

if (stud == null) { // maks antall studenter er nådd.
    if(debug) {
        skjerm.out("DEBUG: Max number of students.");
    }
    return;
}
...
```

Listing 4.4: Code after adding redundant statements and variables

The trees for the two listings are shown in Figures 4.4a and 4.4b. In Figure 4.4b there are some subtrees that are not present in Figure 4.4a. To determine the similarity between the two trees we need to focus on the subtrees they have in common. For this cheating strategy, the amount of differences we get between ASTs of two Java listings is very much dependent upon the kind of redundant code that has been inserted. For example, the insertion of a redundant statement or local variable declaration will produce few differences, while the insertion of a redundant method or class can produce large differences.



(a) AST of the code from Listing 4.3.



(b) AST of the code from Listing 4.4.

Figure 4.4: ASTs of the codes from Listings 4.3 and 4.4.

4.1.6 Changing the order of independent code

Within a block we can change the order of independent statements and local variable declarations without changing the program's behavior. Another challenge is that several other structures are also independent in Java. In a Java listing the order of the class-, interface- and enum declarations are not important. Within these declarations there are also declarations that are independent. For instance, within a class declaration the order of initializers and constructor-, field-, method-, class- and interface declarations are not important.

In Listing 4.5 we have some field declarations from the class `Kurs` in *Studentregister.java*. If we change the order of these declarations, we get the code in Listing 4.6. For the two listings we get the trees in Figure 4.5a and 4.5b. The only difference between these trees is that we have changed the order of the children of *ClassOrInterfaceBody*. The transformation that has been applied is a rotation of subtrees. For independent statements and local variable declarations we do not necessarily need to use only this transformation. If we change the order of statements and local variable declarations within a block, then we will apply rotations. On the other hand, if we can move a statement or a local variable declaration outside the block, then we use a transformation that first removes the subtree of the statement/local variable declaration and then insert this subtree some other place in the AST.

```
class Kurs {
    String kurskode;
    Ukedag dag;
    int tid;
    ...
}
```

Listing 4.5: Code before changing the order of independent structures

```
class Kurs {
    int tid;
    Ukedag dag;
    String kurskode;
    ...
}
```

Listing 4.6: Code after changing the order of independent structures

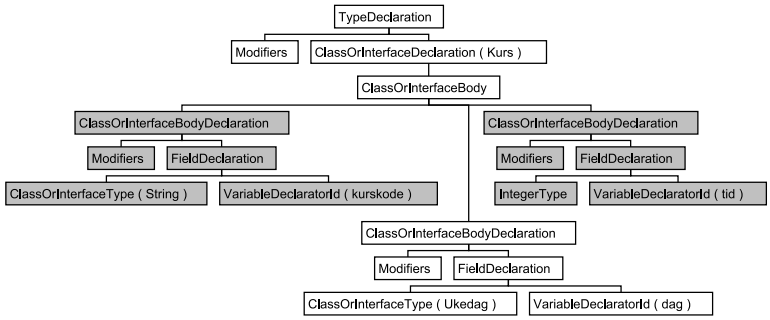
In this example only the order of some field declarations has been changed. If we instead change the order of some class declarations or method declarations, the difference between the two trees would be much bigger. The difference between the two trees depends on the kind of structures that change places. For a plagiarism detection program it is easier to detect the change of order for these declarations, than for independent statements and local variable declarations. The program *knows*, for instance, that the order of declarations within a class is not important, but within a block it is much harder to know which statements and local variable declarations correspond to each other.

4.1.7 Replacing one iteration statement with another

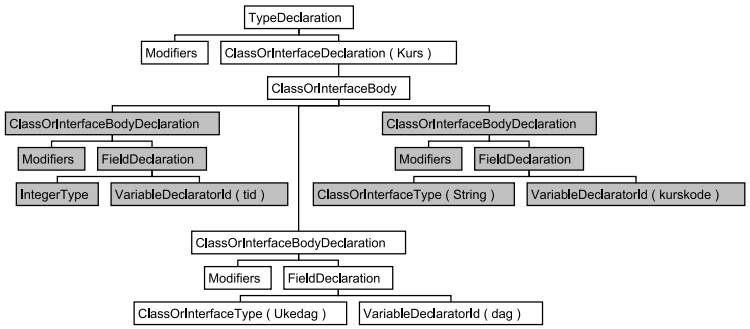
A for-loop can be replaced by a while-loop and vice versa. Consider the code in Listing 4.7 which is from the method `finnStudent` in the class `Institutt` in *Studentregister.java*. The for-loop is replaced with a while-loop as shown in Listing 4.8. The trees for the two listings are shown in Figures 4.6a and 4.6b. We can see that the two trees have the same nodes, except for the *ForStatement* node in Figure 4.6a that has been replaced with a *WhileStatement* node in Figure 4.6b. There are some structural differences between the two trees, since some of the subtrees in Figure 4.6a have been moved in Figure 4.6b.

```
...
for (int i = 0; i < antStudenter; i++) {
    if (navn.equals(studenter[i].navn)) {
        stud = studenter[i];
    }
}
...
```

Listing 4.7: Code for the for-loop



(a) AST of the code from Listing 4.5.



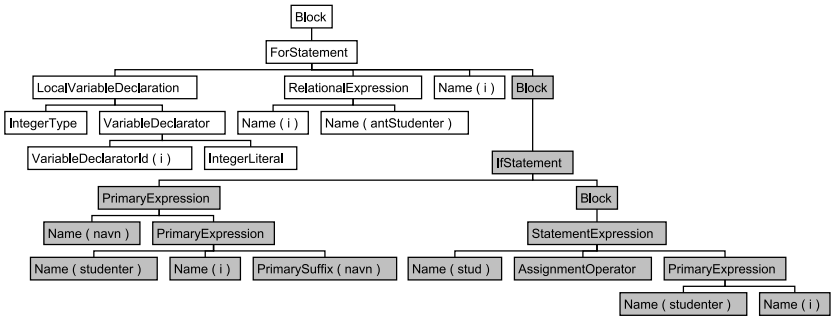
(b) AST of the code from Listing 4.6.

Figure 4.5: ASTs of the code from Listings 4.5 and 4.6.

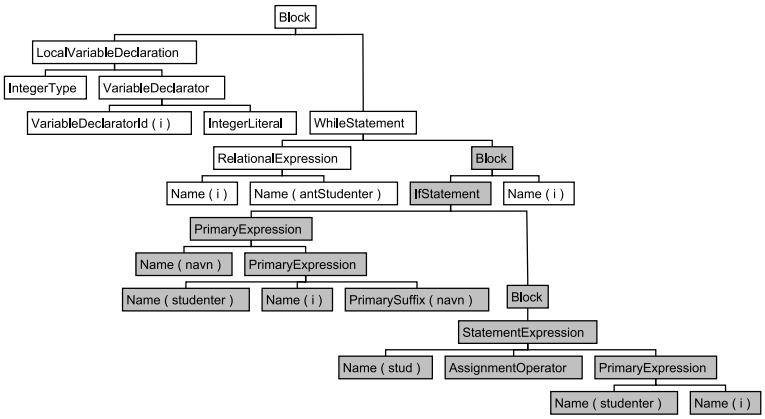
```
...
int i = 0;

while(i < antStudenter){
    if (navn.equals(studenter[i].navn)) {
        stud = studenter[i];
    }

    i++;
}
...
```

Listing 4.8: Code for the while-loop

For this strategy we use one transformation that changes the label of a node (*ForStatement* with *WhileStatement*), and two transformations that move subtrees. The subtrees that are moved are the subtrees rooted at *LocalVariableDeclaration* and at *Name (i)*, where *Name (i)* is the node that represents the increment of the variable *i*. Where we move these subtrees depends on the code that we do transformations on. In this example, the local variable declaration was moved just outside the while-loop, but we could also have moved it to another place in the method. For the node *Name (i)*, we can in this example only insert it next to the subtree rooted at *IfStatement*. This node needs to be within the subtree rooted at *Block*, but where we place it depends on when the variable *i* can be incremented.



(a) AST of the code from Listing 4.7.



(b) AST of the code from Listing 4.8.

Figure 4.6: ASTs of the codes from Listings 4.7 and 4.8.

A for-loop and a while-loop can also be replaced with a do-while-loop and sometimes with an enhanced for-loop. An enhanced for-loop can be used if we are iterating over arrays or objects from the Collection class. Listings 4.9 and 4.10 show examples of how we can rewrite the for-loop in Listing 4.7 and the while-loop in Listing 4.8 to a do-while-loop and an enhanced for-loop, respectively. We can see that we have to use an extra if-statement in each listing. This is necessary since the do-while-loop runs one or more times, while a for-loop and a while-loop runs zero or more times. For the enhanced for-loop it is necessary since some of the elements in the array *studenter* can be *null*.

```
...
int i = 0;

if(i < antStudenter){
    do {
        if (navn.equals(studenter[i].navn)) {
            stud = studenter[i];
        }

        i++;

    }while(i < antStudenter);
}
...
```

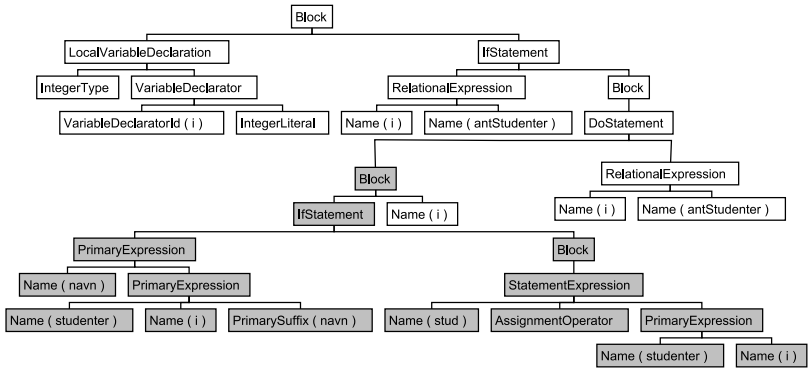
Listing 4.9: Code for the do-while-loop

```
...
for(Student student : studenter){
    if(student != null){
        if (navn.equals(student.navn)) {
            stud = student;
        }
    }
}
...
```

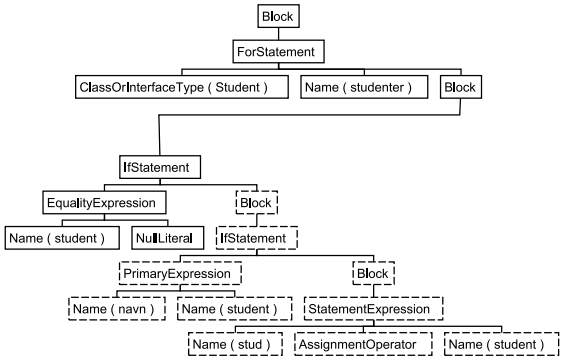
Listing 4.10: Code for the enhanced for-loop

The trees of the two listings are shown in Figures 4.7a and 4.7b. If the first if-statement had not been necessary in Figure 4.7a, then this tree would be very similar to the tree of the while-loop in Figure 4.6b. The only differences would then be the node labeled *DoStatement* and that the expression and the loop body had switched places. The tree of the enhanced for-loop in Figure 4.7b is the tree that is most different from the other trees. The subtree with broken lines in this tree, is the subtree that is most similar to the subtrees in gray in the other trees.

For both the codes and the trees, the for- and while-loop are the most similar, while the do-while-loop share some similarity with these loops. The enhanced for-loop is not so similar to the others when it comes to the code and the tree. We can become suspicious by the similarity between the code of this loop and the code of the other loops, but it is not so easy to say if it is cheating or not. The best way to find the similarity between all four trees is to look at the content of the loop body (in gray or broken lines in the trees). We can have different loops, but if they are iterating over the same code or similar code, then it is possible that they are modifications of each other. The differences we get between two ASTs by replacing a loop with another loop depend on what kind of loop the first loop is replaced with, and the sizes of the loop bodies. For the examples in this section we have, for instance, a bigger difference between the for-loop and the do-while-loop than between the for-loop and the while-loop. Also, the difference between the different loops is not so big, since the loop-body is of a small size. If a loop with a bigger loop-body was replaced with another loop, there would be a greater difference between the ASTs.



(a) AST of the code from Listing 4.9.



(b) AST of the code from Listing 4.10.

Figure 4.7: ASTs of the code from Listings 4.9 and 4.10.

4.1.8 Changing the structure of selection statements

Nested if-statements can sometimes be replaced with a sequence of if-statements. We can also use if-statements instead of a switch-statement or vice versa. In the method `menu` in the class `Institutt` in *Studentregister.java* we got the code in Listing 4.11. The switch-statement can be replaced with a sequence of if-statements which is shown in Listing 4.12. Or they can be replaced with nested if-statements which is shown in Listing 4.13.

```
...
do {
    skjerm.out(meny);
    valg = tast.inInt();
    switch (valg) {
        case 1: registrerStudent();
        break;
        case 2: finnStudent().skrivTimeplan();
        break;
        case 3: // avslutter
        break;
        default: skjerm.outln("Ukjent menyvalg");
    }
} while (valg != 3); // do-while
...
```

Listing 4.11: Code for the switch-case

```
...
do {
    skjerm.out(meny);
    valg = tast.inInt();

    if(valg == 1){
        registrerStudent();
    }

    if(valg == 2){
        finnStudent().skrivTimeplan();
    }

    if(valg > 3 || valg < 1){
        skjerm.outln("Ukjent menyvalg");
    }

} while (valg != 3); // do-while
...
```

Listing 4.12: Code for a sequence of if-statements

```
...
do {
    skjerm.out(meny);
    valg = tast.inInt();

    if(valg == 1){
        registrerStudent();
    }

    else if(valg == 2){
        finnStudent().skrivTimeplan();
    }

    else if(valg > 3 || valg < 1){
        skjerm.outln("Ukjent menyvalg");
    }

} while (valg != 3);
...
```

Listing 4.13: Code for the nested if-statements

The trees are shown in Figure 4.8a, 4.8b and 4.8c. We can see that the last two trees have the most in common. If we transform the sequence of if-statements into nested if-statements, then the right if-statement of an if-statement in the tree becomes the else-statement of this statement. If we do it the other way around, then the if-statement within the else-statement becomes the right sibling of the if-statement with the else-statement. These transformations can only be used if we test on one value and not ranges of values and conditions.

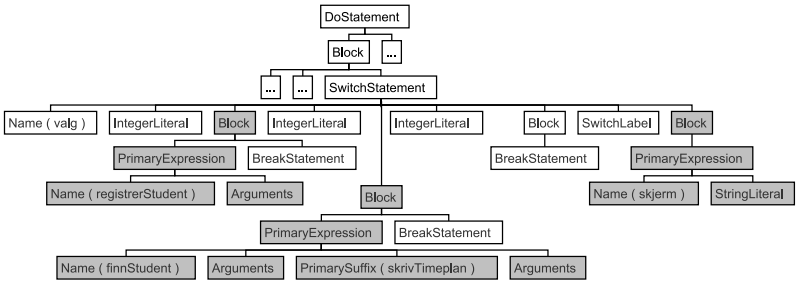
A sequence of if-statements or nested if-statements can only be transformed into a switch-statement if we test on single values. If we simplify the transformations here, we can say that we only copy the subtrees in gray from either Figure 4.8b or Figure 4.8c and use them as the bodies of the different case-labels. We can see that the first and the two last trees are not so similar, but that they have some subtrees that are identical (the subtrees in gray). We should look at these subtrees to find the similarity between the tree of a sequence of if-statements or nested if-statements and the tree of a switch-statement. The differences we get between two ASTs by replacing selection statements with other selection statements depend highly on the sizes of the subtrees in gray. The bigger these subtrees are, the bigger the difference will be.

4.1.9 Replacing procedure calls with the procedure body

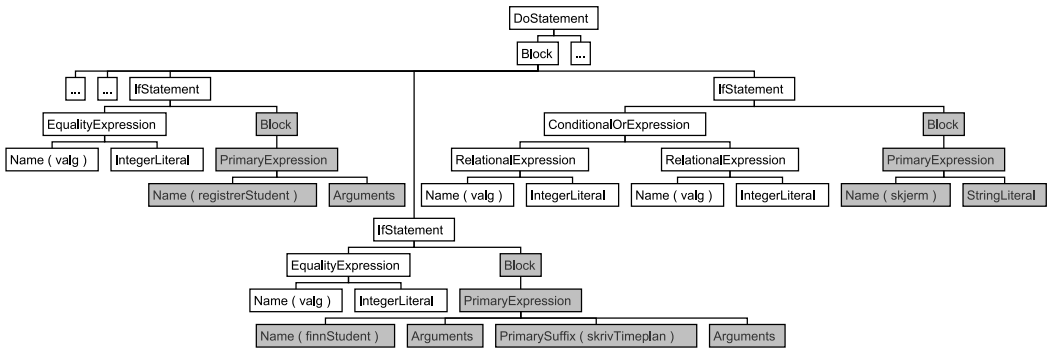
In the method `registrerStudent` in the class `Institutt` in *Studentregister.java* there is a method call to the method `finnKurs` in the same class. This is shown in Listing 4.14. If we replace the method call with the method body we get the code in Listing 4.15.

```
void registrerStudent() {
    ...
    skjerm.out("Oppgi kurskode (blank linje avslutter): ");
    String kurskode = tast.readLine();
    while (!kurskode.equals("")) {
        Kurs k = finnKurs(kurskode);
        ...
    }
}

Kurs finnKurs(String kurskode) {
    Kurs kurset = null;

    for (int i = 0; i < antKurs; i++) {
        if (kurs[i].kurskode.equals(kurskode)) {
            kurset = kurs[i];
        }
    }
    return kurset;
}
```

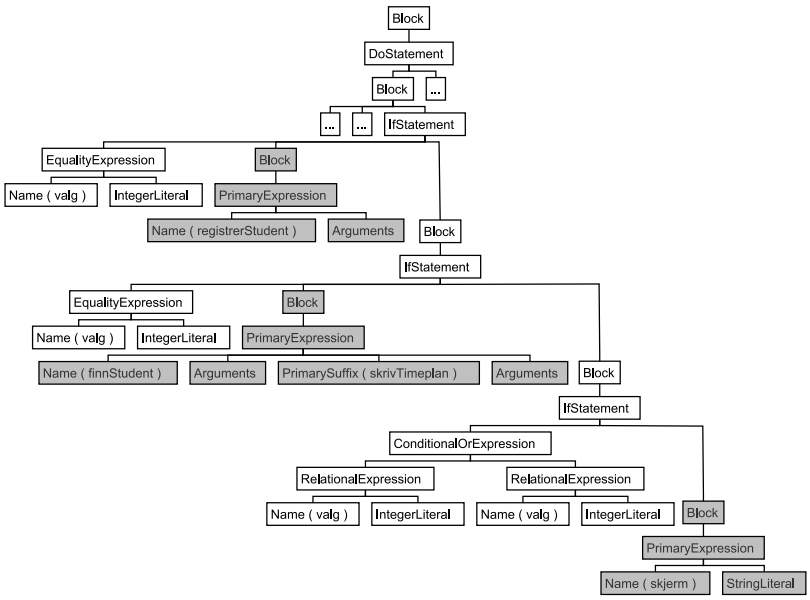
Listing 4.14: Code from Studentregister.java



(a) AST of the code from Listing 4.11.



(b) AST of the code from Listing 4.12.



(c) AST of the code from Listing 4.13.

Figure 4.8: ASTs of code from Listings 4.11, 4.12 and 4.13.

```
void registrerStudent() {
    ...
    skjerm.out("Oppgi kurskode (blank linje avslutter): ");
    String kurskode = tast.readLine();
    while (!kurskode.equals("")) {
        Kurs k = null;

        for (int i = 0; i < antKurs; i++) {
            if (kurs[i].kurskode.equals(kurskode)) {
                kurset = kurs[i];
            }
        }
        ...
    }
}
```

Listing 4.15: Code from Studentregister.java

For the two listings we get the trees in Figures 4.9 and 4.10. The transformations that can be used, depends on the method that contains the method body. In this example we have a method that returns a value. Then everything in its body, except the return statement, is moved to the place where the method call occurs, and the remaining nodes of the method are deleted. We can see in Figure 4.10 that we get new subtrees in the method body of `registrerStudent`, and that there are some similarities between the local variable declarations of the methods `registrerStudent` and `finnKurs` (denoted with broken lines). On the other hand, if the method does not return a value, then the whole method body of this method is moved to where the method call occurs, and the remaining nodes of the method are deleted. The differences that we get between two ASTs by using this strategy depends on the size of the method body that is moved. In this example the body is not so large, but for other code listings we can have much larger method bodies.

4.1.10 Combine the copied code with your own code

I will not give an example of this strategy, nor try to classify what kind of transformations that are used when transforming the AST of the original code into the AST of the modified code. The problem with this strategy is that the similarity between the two programs can be rather small. Here the similarity will depend upon how much code that is copied.

4.2 Guidelines for the development of new similarity measures

In this section, I assess which of the cheating strategies that deserve most attention based on the number of differences which we get between the ASTs and how easy it is to use the strategy. I will also outline how these strategies affect the development of the new similarity measures.

4.2.1 Strategies that are important to detect

For the sake of convenience I have made Table 4.1 on page 45 which shows how many differences we get or can get between two trees by applying the different strategies. We can see that the strategies 1 and 2 (4.1.1) are not important. For the strategies 3 - 5 (4.1.2, 4.1.3 and 4.1.4) we will normally get few differences between the two trees. This depends of course on the number of changes that are done, but since the changes are done on small subtrees the overall difference, between the two ASTs, will be small.

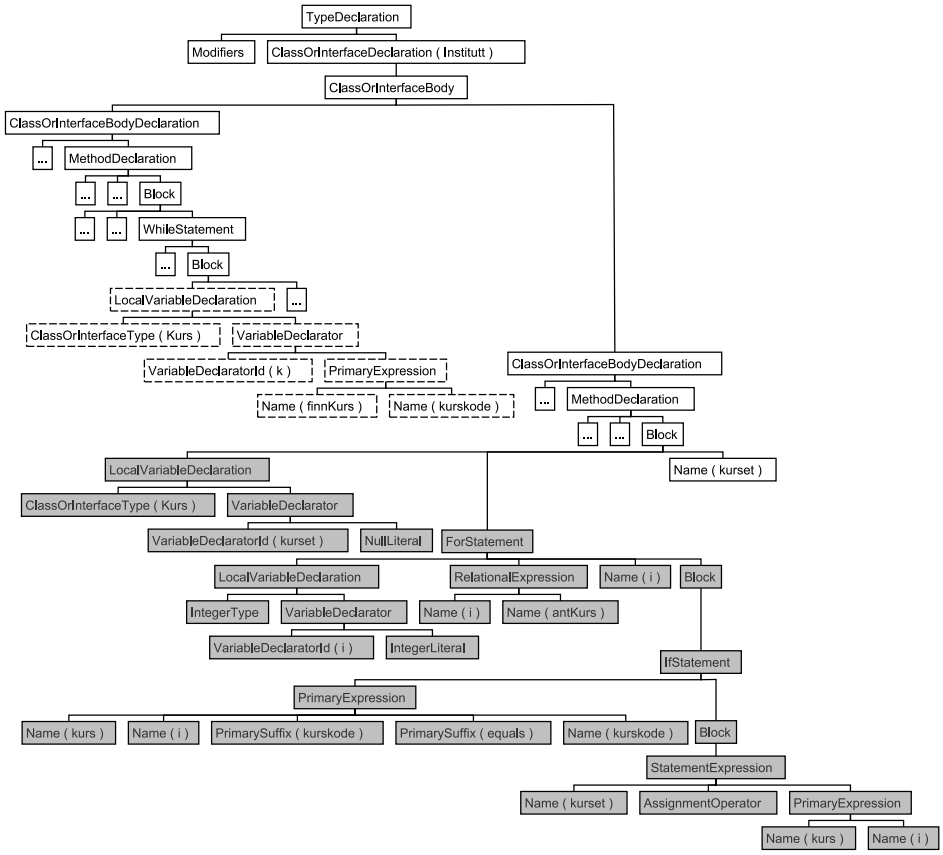


Figure 4.9: AST of the code from Listing 4.14. The subtree in broken lines contains the method call, and the subtree in gray contains the nodes that will replace the subtree in broken lines.

Only strategies 6 - 10 and 12 generate substantial differences between the two associated ASTs. For strategy 6 (4.1.5) we will not get so many differences if only statements and/or local variable declarations are added. It depends of course on how many and the sizes of the statements/local variable declarations which we add (for instance a redundant for-loop will give a greater difference than a redundant statement that prints to screen). For the student it is easiest to add statements and/or local variable declarations of small sizes, since it is easier to discover redundant code of some size. For instance a redundant class, method or statement of some size can easily be discovered by the teaching assistant when he grades the assignment. By using strategy 7 (4.1.6) we will either get few or many differences between the trees. If we change the order of independent statements, then we will not get so many differences. On the other hand, if the order of classes and/or methods are changed, we can get huge differences between the two trees. Since it is easy for the student to change the order of these structures and since the strategy can have a real impact on the differences, it will be important to discover the rotation of these subtrees of potentially large sizes. For the strategies 8 and 9 (4.1.7 and 4.1.8) the differences we get between the two trees depend on the sizes of the loop bodies for strategy 8 and the sizes of the bodies of the different selection statements for strategy 9. In Sections 4.1.7 and 4.1.8 we used bodies of small sizes, but we can have bodies of much larger sizes. When using strategy 10 the differences between the two trees depend upon the size of the method body that we move. For the student it is easiest to use this strategy for small methods, and in this case we will not

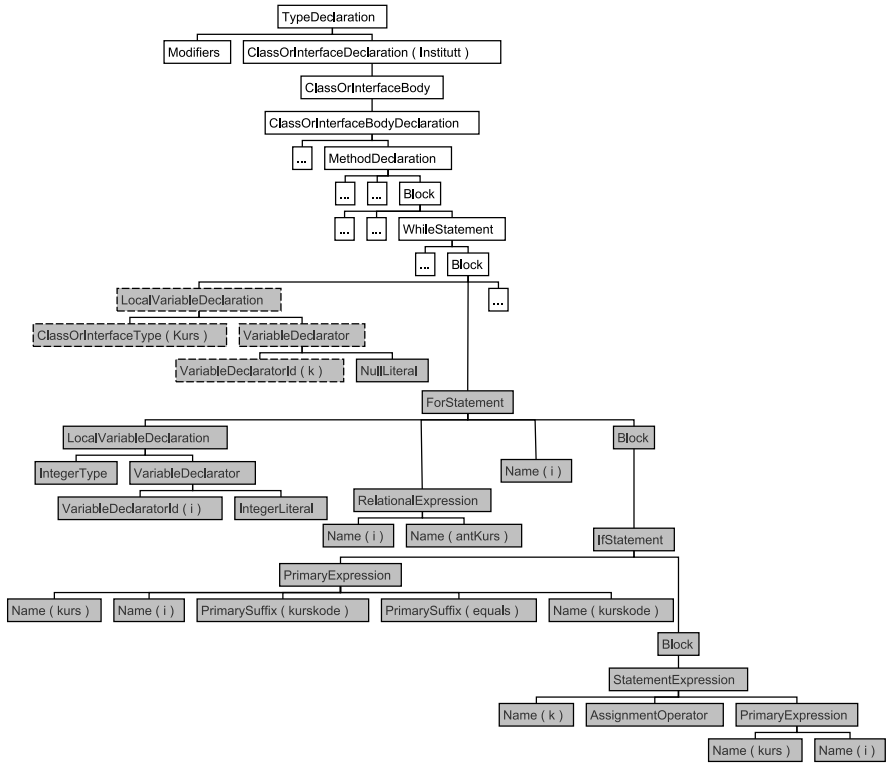


Figure 4.10: AST of the code from Listing 4.15. The subtree in gray contains nodes from the method that was called in Listing 4.14, and the subtree in broken lines show similarities between the two method bodies.

get so many differences between the two trees. A large method body can also be moved, but if many methods have method calls to this method, then all these methods need a copy of the code from that method. This will not be very practical, and it can easily be discovered by the teaching assistant when he grades the assignment. For strategy 12 (4.1.10) the differences that we get between the two trees will depend upon how much code that the student copies.

4.2.2 How the strategies affect the development of new similarity measures

Strategy 7, used on classes, methods and other declarations, will have the greatest effect on the development of the new similarity measure algorithms. For each class we will need to measure the similarity between this class and all the other classes, and for each method we will need to measure the similarity between this method and all the other methods. This needs to be done since it can be hard to determine which classes that correspond to each other, and which methods that correspond to each other. We can also measure the similarity between all the constructors and all the field declarations, but they are not as important as the classes and methods since these structures normally have most of the nodes in the trees. It is also important that we can assess similarity between different loops (strategy 8) and between different selection statements (strategy 9) since the trees of these structures can be potentially large. These trees are often rooted at different nodes so we will need to search within the trees for similar loop bodies or similar selection statements bodies. When choosing measures, it is also important that the measure can find a good alignment between the statements and the local variables of two

Differences between the two ASTs	**Strategy**
None	1. Changing comments or formatting. 2. Changing the names of the identifiers.
Normally few	3. Changing the order of operands in expressions. 4. Changing the data types. 5. Replacing an expression with an equivalent expression.
Possibly many	6. Adding redundant code. 7. Changing the order of independent statements. 8. Replacing one iteration statement with another. 9. Changing the structure of selection statements. 10. Replacing procedure calls with the procedure body. 12. Combine the copied code with your own code.

Table 4.1: The differences we get or can get between two trees by using the different strategies.

blocks of code. This is because redundant code can have been inserted, the order of independent code can have been changed, method calls can have been replaced with method bodies, and the student can have combined his own code with the copied code.

Chapter 5

Development of new similarity measures

In this chapter, I consider measures for assessing the similarity between ASTs, giving particular attention to the cheating strategies emphasized in Section 4.2.2. The measures I consider are those listed in Section 2.5. Then I outline two new similarity measures ASTSIM-NW and ASTSIM-LCS based on the previous discussion. Finally, I give a description of ASTSIM-NW and ASTSIM-LCS.

5.1 Discussing possible measures

5.1.1 Longest Common Subsequence

To use the Longest Common Subsequence (LCS) method, defined in Section 2.5.1, we need to do a traversal of each AST. After that is done, we can find an optimal alignment between the two traversals. One problem here is that the order of different independent structures might have been changed in one of the trees. If the order of declarations such as classes, methods and so on has been changed, then we can get a low similarity score between the two trees. This is due to the fact that LCS is a method that is used to find an alignment between two ordered sequences, while our sequences contains unordered subsequences which complicates the alignment. In order to use this method, we need to find an alignment between each class declaration in the first AST and each class declaration in the second AST. By doing so we can find the class declarations that can best be aligned with each other. Further, to find the alignment between two class declarations we need to find an alignment between each method declaration in the first class and each method declaration in the second class. We can continue on if we want to find the best alignments between constructor declarations and field declarations. One plagiarism detection algorithm that uses a method similar to LCS is *Sim* (Gitchell and Tran, 1999). This algorithm is made for program listings written in *C* and it finds an optimal alignment between two token sequences. Also for this algorithm there is a need to find the best alignments between the different functions in the two listings.

The main advantage with LCS is that it can find similarities between subtrees where the labels of the root nodes differ, such as subtrees of different loops or subtrees of different selection statements. Both between the four loops in Section 4.1.7 and between the three selection statements in Section 4.1.8, LCS would find good alignments. When it comes to aligning the statements and/or local variable declarations of two blocks, LCS will find an optimal alignment between the traversals of the two subtrees rooted at the nodes labeled *Block*. This alignment is an optimal alignment between the two ordered sequences, but it does not necessary need to be the best alignment between the two subtrees. This is due to the fact that LCS can have aligned

single nodes (nodes where neither the parent of the node or children of the node are aligned with other nodes), or it can have aligned small subtrees that are part of larger subtrees, or it can have aligned the nodes in a subtree of the sequences with nodes in two or more subtrees in the other sequence. If we allow such alignments, then the ASTs of the Java listings can seem more similar than they actually are. When using this method it is important to decide which of the aligned nodes in the two sequences that should be part of the alignment between the two subtrees.

5.1.2 Tree Edit Distance

The Tree Edit Distance (TED) method, defined in Section 2.5.1, can be used directly on the trees. Since this metric works on ordered trees, it has the same problems as LCS when it comes to independent structures in the two trees. This method has been used in Sager et al. (2006) to measure the similarity between ASTs of Java listings. It has not been used to detect plagiarism, but to detect similar Java classes in software projects. The method performed well in this context, but I believe that it would perform worse in the context of plagiarism detection. In such a context the student has often used methods to disguise the similarities, while in the context of software development there is no reason for disguising the similarities between, for instance, two versions of a program.

TED will find a good tree edit distance between the ASTs of the for-loop and the while-loop in Section 4.1.7 since those trees have a very similar structure. The tree edit distance between, for instance, the while-loop and the do-while-loop will not be so good. Since the two tree structures are a bit different, the mapping (substitution) of the nodes will not be as good as for the for-loop and the while-loop, and then we will get more deletions and insertions. For the selection statements in Section 4.1.8 we have trees where the tree structures are even more different, which leads to an even higher tree edit distance. One problem with the Tree Edit Distance algorithm that I have considered in this thesis is that deletion and insertion are restricted to leaf nodes only. For other forms of Tree Edit Distance we can insert and delete internal nodes in the tree, which would be favorable when we have trees that share some similarity, such as the trees in Section 4.1.8, but that have different structures. The drawbacks of these forms of Tree Edit Distance can be an increase in complexity and that ASTs seem more similar than they actually are. When it comes to aligning the statements and/or local variable declarations of two blocks, TED will find a good alignment. TED will use the alignment of the subtrees of statements and local variable declarations in the two blocks that gives the best mapping between nodes of the two subtrees that represent the blocks.

5.1.3 Tree isomorphism algorithms

The ordered tree isomorphism, defined in Section 2.5.2, is not a good method for our problem. In order to have an ordered tree isomorphism between two trees, we need two trees with identical structure and nodes with corresponding labels. The unordered tree isomorphism, defined in Section 2.5.2, is a little bit better since it can rotate the subtrees that are children of some node. It can for instance rotate the subtrees of class declarations and the subtrees of methods within the subtrees of class declarations, but if one of the trees has just one extra node then we will have no unordered tree isomorphism between the two trees.

In the top-down ordered maximum common subtree isomorphism method, defined in Section 2.5.2, we want to find a top-down ordered maximum common subtree between two trees. For a node v to be part of this maximum common subtree there are two requirements that needs to

be satisfied: Obviously, w the parent of v needs to be part of the maximum common subtree. And, if v is not the first child of w, then the left sibling of v also needs to be part of the maximum common subtree. Consider the following problem: We have two ASTs, where the second is identical to the first tree, with exception of that we have changed the order of the different declarations within the only class declaration in the second tree. Then we will get a small maximum common subtree if many of the declarations that we try to match against each other are different. For instance, a field declaration and a method declaration will only have a couple of nodes in common. As we can see, this is not a good approach for our problem. The top-down ordered maximum common subtree isomorphism method was also used in Sager et al. (2006), where it obtained mixed results.

For the problem were we changed the order of the declarations, the top-down unordered maximum common subtree isomorphism method, defined in Section 2.5.2, would be a much better choice. With this method we would get a perfect score between the two trees, since this method can rotate the different declarations in the second tree such that they get the same order as the declarations in the first tree. This method can handle the different independent structures since it uses an unordered approach, but it will find a low similarity between the trees of the loops in Section 4.1.7 and the trees of the selection statements in Section 4.1.8. For instance when it tries to find a top-down unordered maximum common subtree between the trees of the for-loop and the while-loop in Section 4.1.7, it will stop at the nodes labeled *ForStatement* and *WhileStatement* since the node labels do not match. When it comes to aligning the statements and/or local variable declarations of two blocks, this algorithm will not find an alignment but it will find the best unordered matching between the subtrees of the different statements and local variable declarations in the two blocks. I consider it to be possible to treat most of the nodes in the trees as unordered nodes. Some of the nodes are already unordered (the roots of subtrees of class declarations, method declaration, constructor declarations, field declarations and so on), while a lot of the unordered nodes will still be treated as ordered nodes since the grammar of the Java language will lay restrictions on which nodes that can be matched against each other. The main problems with the unordered approach are unordered matching of operands in expressions where the order is important and the unordered matching of statements and local variable declarations of two blocks. Here the unordered matching of statements and local variable declarations is the most important, since these subtrees are larger than the subtrees of the operands in expressions. We should find an optimal alignment between the subtrees of the statements and local variable declaration of two blocks, instead of using an unordered matching.

We can also use the bottom-up ordered maximum common subtree isomorphism method, defined in Section 2.5.2. This method also finds a maximum common subtree, but it does it bottom-up. For a node v to be part of this maximum common subtree there is one requirement. If v is not a leaf node then all the children of v needs to be part of the maximum common subtree, and the first child of v needs to be matched against the first child of the node that v is matched to and so on. For our problem it can be hard to find a bottom-up maximum common subtree of some size, since we sooner or later will find a node z were not all the children are part of the maximum common subtree. For the bottom-up unordered maximum common subtree isomorphism method, defined in Section 2.5.2, all the children of a node v must also be part of the bottom-up subtree, but we can use an unordered matching of these children against the children of the node that v can be matched to. Here we also have the same problem as for the bottom-up ordered maximum common subtree isomorphism method. The bottom-up unordered maximum common subtree isomorphism method was also used in Sager et al. (2006), and it performed worst of all three methods.

5.2 Two new similarity measures for ASTs

The most promising method from the discussion above is the top-down unordered maximum common subtree isomorphism method. This method can find a good match between the different independent structures. One problem with this method is that it cannot find a good alignment between the statements and local variable declarations of two blocks, since it treats them all as unordered (independent) and therefore finds an unordered matching between them. Another problem is that it will find a low similarity between the trees of different loops and between the trees of different selection statements.

The first problem can be solved by using it together with the Needleman-Wunsch (NW) algorithm, defined in Section (2.5.1). NW can find an optimal alignment between the subtrees of the statements and the local variable declarations in two blocks. This approach is described in Section 5.3.2. The second problem can best be solved by using LCS. This method can also find a good alignment of statements and local variable declarations of two blocks. We saw in the discussion that TED would get a high tree edit distance between trees where the structures are very different, such as the trees of selection statements. I also consider it to be easier to implement LCS than TED.

I propose two new similarity measures called ASTSIM-NW and ASTSIM-LCS. The first measure will use the top-down unordered maximum common subtree isomorphism method for the whole tree, but instead of using an unordered matching of the subtrees of two blocks it will use the Needleman-Wunsch algorithm to find an optimal alignment between the subtrees. This measure is described in Section 5.3. The second measure will use the top-down unordered maximum common subtree isomorphism method for the whole tree, with exception of method bodies where it will use LCS. This measure is described in Section 5.4.

5.3 Description of ASTSIM-NW

In this section I give a description of ASTSIM-NW. The pseudo code for the implementation of ASTSIM-NW is given in Section 6.1.

5.3.1 Top-Down Unordered Maximum Common Subtree Isomorphism

To find the top-down maximum common subtree isomorphism of an unordered tree $T_1 = (V_1, E_1)$ into another unordered tree $T_2 = (V_2, E_2)$ we start at the root nodes of T_1 and T_2. For each node $v \in V_1$ and $w \in V_2$ we can find the maximum common subtree isomorphism between the subtree rooted at v and the subtree rooted at w as long as the labels correspond. If the labels do not correspond, then we have maximum common subtree isomorphism of size zero. If either v or w is a leaf node, then we have a maximum common subtree of size 1 or 0. Otherwise, we can construct the isomorphism between the node v and the node w by finding the maximum common subtree isomorphisms of each of the subtrees rooted at the children of node v in T_1 into each of the subtrees rooted at the children of node w in T_2.

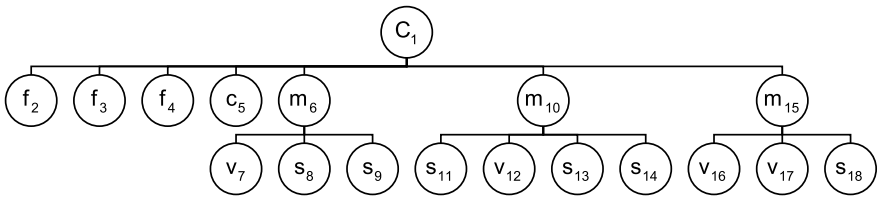
To determine the mapping of the children of v and w we will do as follows: Let p be the number of children of node v in T_1 and let q be the number of children of node w in T_1. The children of the two nodes are then denoted as $v_1, \ldots, v_p$ and $w_1, \ldots, w_q$. We then build a bipartite graph[1] $G = (\{v_1, \ldots, v_p\}, \{w_1, \ldots, w_q\}, E)$ consisting of $p + q$ vertices. If the

[1] A bipartite graph is a graph were we can partition the vertices into two disjoint vertex sets.

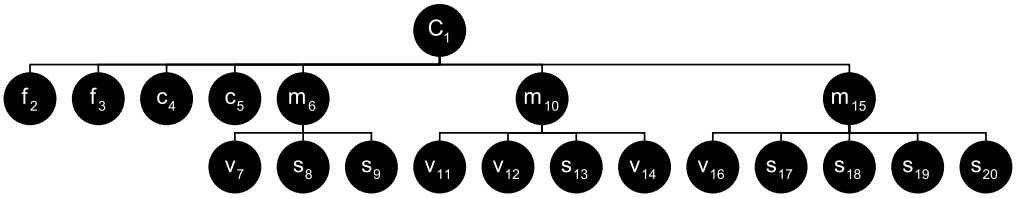
maximum common subtree of the subtree rooted at v_i and the subtree rooted at w_j has nonzero size, then $(v_i, w_j) \in E$ and that edge is then weighted by that nonzero size. The maximum common subtree of v and w has size 1 (since v and w can be mapped against each other) plus the weight of the maximum weight bipartite matching in G.

Example of a top-down unordered maximum common subtree isomorphism

I will give an example of how we can find a top-down unordered maximum common subtree isomorphism between two unordered trees. In Figures 5.1a and 5.1b we have the two simplified ASTs T_1 and T_2. For the different nodes in the trees we use the following labels: *C* is a class declaration, *f* is a field declaration, *c* is a constructor declaration, *m* is a method declaration, *v* is a local variable declaration, and *s* is a statement. The nodes are also numbered according to the order in which they are visited during a preorder traversal.



(a) AST T_1



(b) AST T_2

Figure 5.1: Two simplified ASTs.

We will first consider the problem of finding the size of the top-down unordered maximum common subtree isomorphism of T_1 into T_2. And second I will show how we can find the maximum common subtree isomorphism mapping M of T_1 into T_2. The mapping M is given by $M \subseteq B \subseteq V_1 \times V_2$. The set B contains all the nodes that can be mapped to each other, while M contains the nodes that gives the top-down common subtree with the maximum size.

We use a top-down recursive algorithm to find this isomorphism. We start with the two root nodes. To find the size of the top-down maximum common subtree isomorphism of C_1 in T_1 and C_1 in T_2, the maximum weight bipartite matching problem in Figure 5.2 is solved. Table 5.1 shows the different edge weights for this bipartite graph. An optimal solution to this maximum weight bipartite problem has weight $1+1+1+4+5+4 = 16$. The maximum common subtree of the subtree rooted at C_1 in T_1 and the subtree rooted at C_1 in T_2 then has size $1 + 16 = 17$. Before we can solve this maximum weight bipartite matching problem we have to recursively solve other maximum weight bipartite matching problems. For the nodes m_6, m_{10} and m_{15} in

T_1 we have to find maximum common subtrees with m_6, m_{10} and m_{15} in T_2, and we have to solve a maximum weight matching problem for each of these subtrees. m_6, m_{10} and m_{15} in T_1 have each 3 maximum weight matching problems that are solved as seen in Figure 5.3, 5.4 and 5.5. The weights of the different edges are omitted in the figures, since all the edges have weight 1.

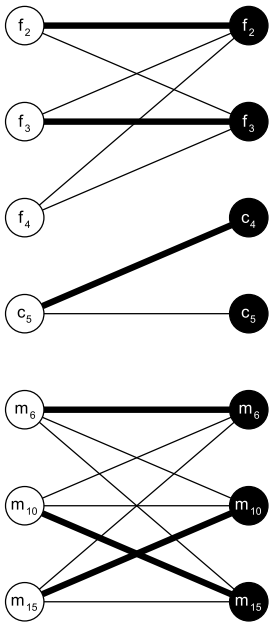
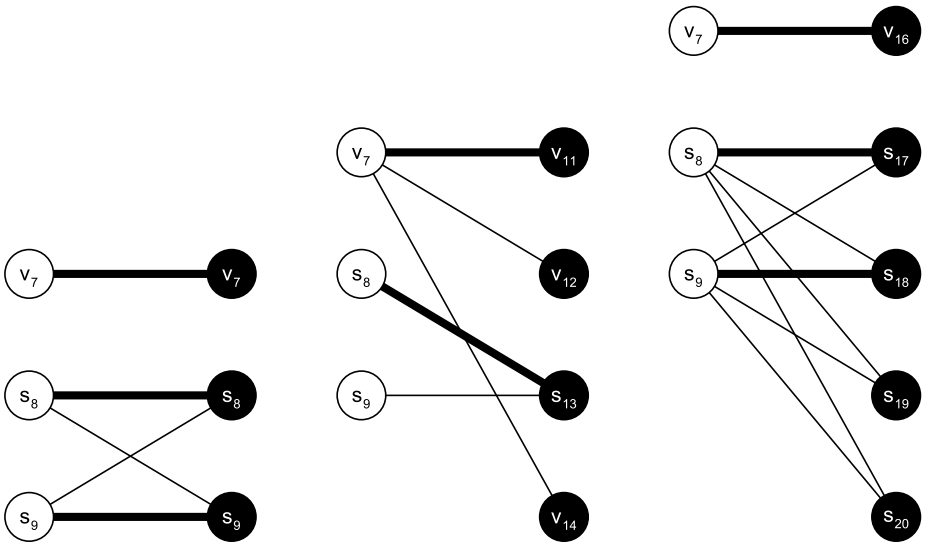


Figure 5.2: Maximum weight bipartite matching of the children of C_1 and C_1 from T_1 and T_2.

	f_2	f_3	c_4	c_5	m_6	m_{10}	m_{15}
f_2	**1**	1					
f_3	1	**1**					
f_4	1	1					
c_5			**1**	1			
m_6					**4**	3	4
m_{10}					4	4	**5**
m_{15}					3	**4**	3

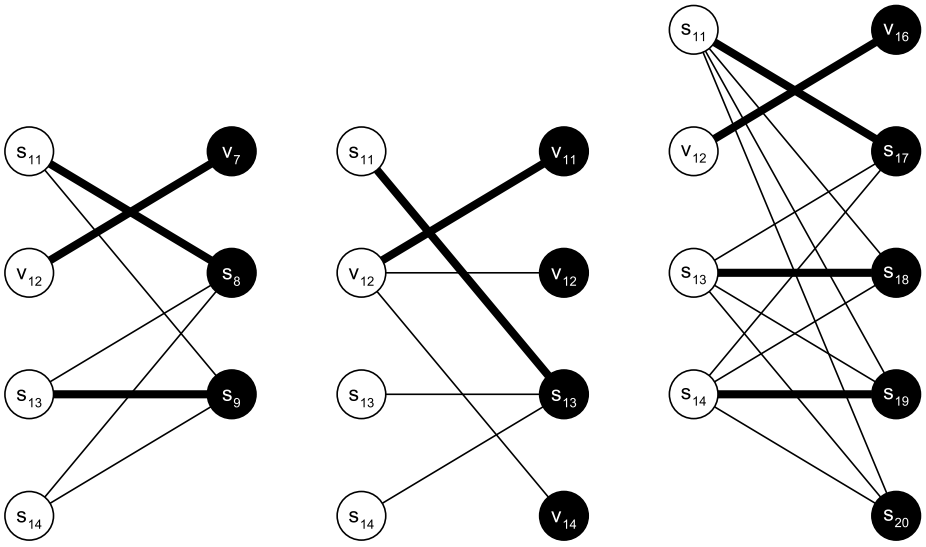
Table 5.1: The edge weights for the bipartite graph in Figure 5.2. The edges that are part of the maximum weight matching are in bold.

Here, $B \subseteq V_1 \times V_2$ contains all the solutions of the maximum weight bipartite matching problems that were solved during the top-down unordered maximum common subtree isomorphism procedure upon the unordered trees T_1 and T_2. We know that the top-down unordered common subtree between T_1 and T_2 is given by the structure (X_1, X_2, M) where $X_1 = (W_1, S_1)$ is a top-down unordered subtree of T_1 and $X_2 = (W_2, S_2)$ is a top-down unordered subtree of T_2. The top-down unordered maximum common subtree isomorphism mapping M is then given by $M \subseteq W_1 \times W_2 \subseteq B \subseteq V_1 \times V_2$.



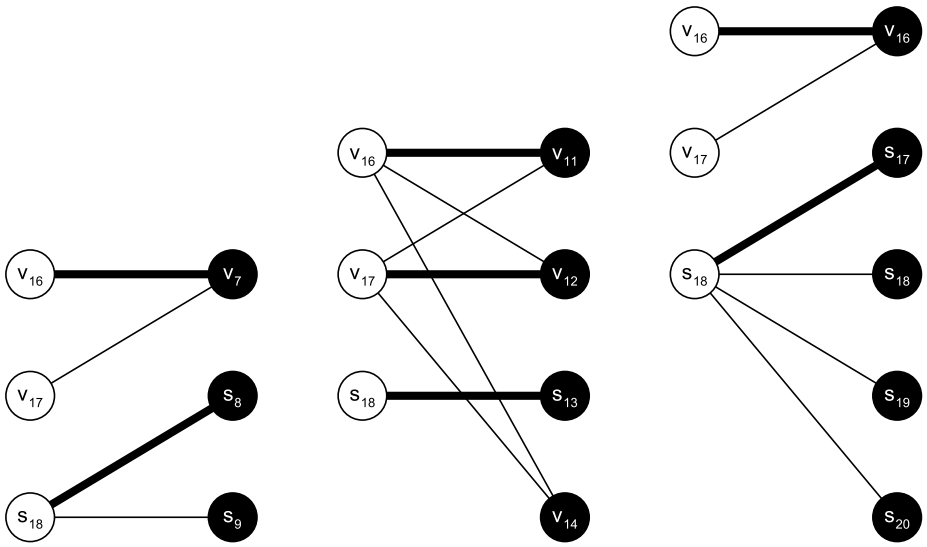
(a) m_6 - m_6 (b) m_6 - m_{10} (c) m_6 - m_{15}

Figure 5.3: Maximum weight bipartite matchings of the children of m_6 from T_1 and the children of m_6, m_{10} and m_{15} from T_2.



(a) m_{10} - m_6 (b) m_{10} - m_{10} (c) m_{10} - m_{15}

Figure 5.4: Maximum weight bipartite matchings of the children of m_{10} from T_1 and the children of m_6, m_{10} and m_{15} from T_2.



(a) m_{15} - m_6 (b) m_{15} - m_{10} (c) m_{15} - m_{15}

Figure 5.5: Maximum weight bipartite matchings of the children of m_{15} from T_1 and the children of m_6, m_{10} and m_{15} from T_2.

To find the mapping M we will do as follows: First we map the $root(T_1)$ with $root(T_2)$ if and only if $(root(T_1), root(T_2)) \in B$. If $(root(T_1), root(T_2)) \in M$, we will map the other nodes during a preorder traversal of T_1. Each nonroot node $v \in V_1$ is mapped to the unique node $w \in V_2$ if and only if $(v, w) \in B$ and $(parent(v), parent(w)) \in B$. Table 5.2 shows which nodes that are mapped to each other according to our example. We can for instance see that only the children of m_6 in T_1 are mapped against the children of m_6 in T_2 since $(m_6, m_6) \in B$. In Figure 5.6 we see the mapping M of T_1 into T_2.

This example shows why we need to use the Needleman-Wunsch algorithm within blocks. We can see that the method m_{10} in T_1 is matched to the method m_{15} in T_2. Within those method bodies the statement s_{11} in m_{10} is matched to s_{17} in m_{15}, while the local variable declaration v_{12} in m_{10} is matched to v_{16} in m_{15}. We cannot allow this unordered matching, since we do not know if these statements and local variable declarations are independent or not.

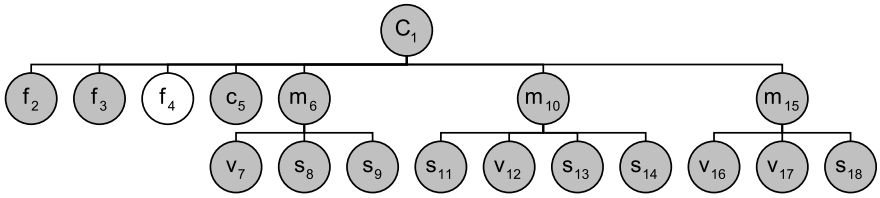
5.3.2 Needleman-Wunsch

The Top-down Unordered Maximum Common Subtree Isomorphism algorithm treats all the nodes in the two ASTs as unordered nodes. The problem with this is that not all of the nodes in the two trees can be treated as unordered nodes. This is especially true within a block where most of the statements and local variable declarations depends on other statements and local variable declarations, and can therefore not be matched with statements and local variable declarations in another block by the use of an unordered matching.

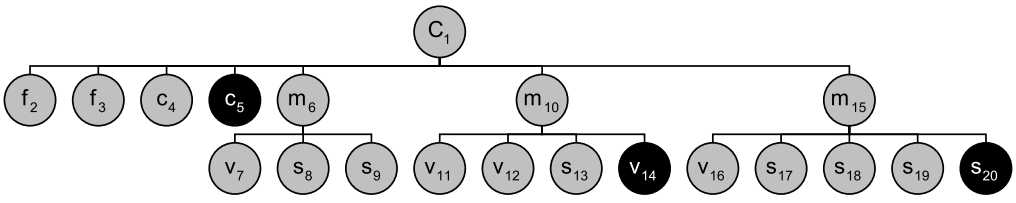
In Listing 5.1 and 5.2 we have two methods which we want to find the best match between. The two methods switch the value of two integer variables and prints the values of the two variables before and after the switch. We can see that the method bodies of the two methods

$v \in V_1$	$w \in V_2$ and $(v, w) \in B$		
C_1	$\boldsymbol{C_1}$		
f_2	$\boldsymbol{f_2}$		
f_3	$\boldsymbol{f_3}$		
f_4			
c_5	$\boldsymbol{c_4}$		
m_6	$\boldsymbol{m_6}$		
v_7	$\boldsymbol{v_7}$	v_{11}	v_{16}
s_8	$\boldsymbol{s_8}$	s_{13}	s_{17}
s_9	$\boldsymbol{s_9}$	s_{18}	
m_{10}	$\boldsymbol{m_{15}}$		
s_{11}	s_8	s_{13}	$\boldsymbol{s_{17}}$
v_{12}	v_7	v_{11}	$\boldsymbol{v_{16}}$
s_{13}	s_9	$\boldsymbol{s_{18}}$	
s_{14}	$\boldsymbol{s_{19}}$		
m_{15}	$\boldsymbol{m_{10}}$		
v_{16}	v_7	$\boldsymbol{v_{11}}$	s_{17}
v_{17}	$\boldsymbol{v_{12}}$		
s_{18}	s_8	$\boldsymbol{s_{13}}$	s_{17}

Table 5.2: The nodes that are part of the mapping M. The node v in T_1 is mapped to the node w in T_2 that is in bold. The nodes in T_1 are listed as they are visited during a preorder traversal.



(a) Tree T_1



(b) Tree T_2

Figure 5.6: The maximum common subtree isomorphism mapping M of T_1 into T_2. The mapping M is shown in gray.

differ a bit. In Listing 5.1 we print the values before and after the switch, while in Listing 5.2 we do all of the printing after the switch, but take into account that the switch has occurred.

```
public static void switchIntegers1(int a, int b){
    System.out.println("Before the switch: a = " + a + " b = " + b);

    int tmp = a;
    a = b;
    b = tmp;

    System.out.println("After the switch: a = " + a + " b = " + b);
}
```

Listing 5.1: Method switchIntegers1

```
public static void switchIntegers2(int a, int b){
    int tmp = a;
    a = b;
    b = tmp;

    System.out.println("Before the switch: a = " + b + " b = " + a);
    System.out.println("After the switch: a = " + a + " b = " + b);
}
```

Listing 5.2: Method switchIntegers2

The different statements and local variable declarations in the two method bodies are children of a node labeled *Block*. To find the maximum common subtree between subtrees rooted at the two block nodes $block_1$ and $block_2$, we first need to find the maximum common subtree of each of the subtrees rooted at the children of node $block_1$ into each of the subtrees rooted at the children of node $block_2$. When all these maximum common subtrees are found, we can find the maximum common subtree between $block_1$ and $block_2$. If we use an unordered matching approach to match the statements and the local variable declarations in the two blocks, we build a bipartite graph $G = (A \cup B, E)$, where $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, \ldots, b_m\}$ are the children sets of $block_1$ and $block_2$, respectively. Then we find the maximum weight matching for the graph G. This matching is shown in Figure 5.7. The edges that are part of the matching are in **bold**. The weights of the different edges are omitted in this figure, because all the edges from one node to its neighbors have the same weight.

The problem with this matching is that the first statement from A cannot be matched against the fourth statement of B. The fourth statement from B, b_4, depends upon statements b_1, b_2 and b_3 from B. We need to do an ordered matching of the statements instead of an unordered. In such a matching we can only use edges that do not cross other edges used in the matching. What we want to do is to find an alignment between the nodes of A and B, with gaps if necessary. There can be many different alignments, but we want to find an alignment where the sum of the weights of the edges is the maximum among all possible alignments. (5.1) is a maximum alignment between A and B. We can see that a_1 and b_4 are aligned with gaps.

$$\begin{array}{cccccc} a_1 & a_2 & a_3 & a_4 & - & a_5 \\ - & b_1 & b_2 & b_3 & b_4 & b_5 \end{array} \tag{5.1}$$

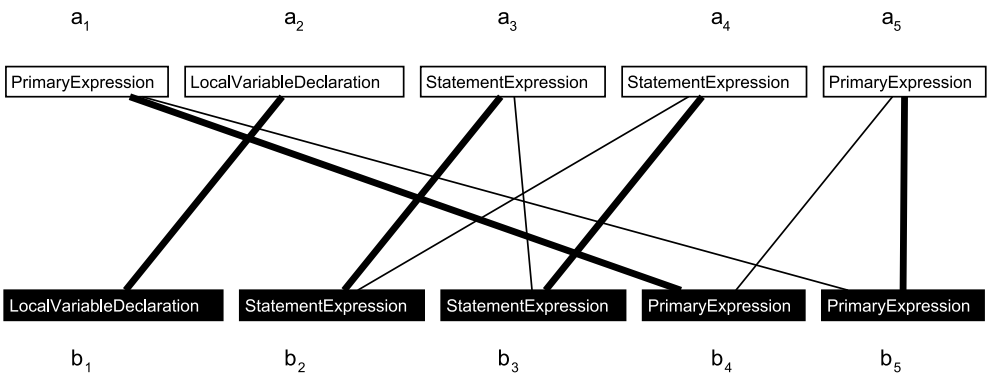


Figure 5.7: The best unordered matching between the statements and local variable declarations from the method bodies of the Listings 5.1 and 5.2.

This problem can be solved with Needleman-Wunsch, defined in Section 2.5.1. For this algorithm we use a similarity matrix S, given in (5.2), which contains the scores for aligning the different statements and local variable declarations from the set A with the different statements and local variable declarations in the set B. The scores in the matrix are the sizes of the different maximum common subtrees isomorphisms of each of the subtrees rooted at the children of node $block_1$ into each of the subtrees rooted at the children of node $block_2$. The score between the statements b_i and a_j is given by $S(i, j)$. For instance, b_1 and a_1 has a score equal to zero since there is no maximum common subtree between the subtree rooted at a_1 and the subtree rooted at b_1. There is no maximum common subtree since the node labels differ. This method also uses a linear gap penalty called d, but I will not use a gap penalty, so I set $d = 0$.

$$S = \begin{array}{c} \text{nodes} \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{array} \begin{array}{c} \begin{array}{ccccc} a_1 & a_2 & a_3 & a_4 & a_5 \end{array} \\ \begin{bmatrix} 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 4 & 0 \\ 0 & 0 & 4 & 4 & 0 \\ 7 & 0 & 0 & 0 & 7 \\ 7 & 0 & 0 & 0 & 7 \end{bmatrix} \end{array} \tag{5.2}$$

To find an optimal solution to the alignment problem we will use a $(n + 1) \times (m + 1)$ matrix denoted by F. In this matrix there is one column for each of the statements and local variable declarations in A, and one row for each of the statements and local variable declarations in B. There is also one column and one row for the gap character, here denoted by ϕ. We will fill this matrix with values according to the recursive formula defined in equation 2.13 on page 14. Each cell $F(i, j)$ in the matrix will be an optimal solution of the alignment of the first i statements and local variable declarations in B with the first j statements and local variable declarations in A.

In matrix (5.3) we can see the result from applying the recursive formula. The lower right hand corner contains the maximum score for the alignment of all the statements and local variable declarations. To find out which statements that are aligned to each other we do a backtracking in the F matrix. This backtracking starts in the lower right hand corner. From this corner we

move either diagonally, left or up. To decide where to move, we compare the value in the cell $F(i,j)$ with the three possible values it could be set to. If it equals $F(i-1,j-1)+S(i-1,j-1)$, then node b_i and a_j are aligned with each other and we move diagonally. The second possibility is that it equals $F(i-1,j)+d$. Then node b_i is aligned with a gap and we move up. And finally, the third possibility is that it equals $F(i,j-1)+d$. Then node a_j is aligned with a gap and we move to the left. We continue this procedure until we reach a cell that is either in the column or the row of the gap character.

$$F = \begin{array}{c} \text{nodes} \\ \phi \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{array} \begin{array}{c} \begin{array}{cccccc} \phi & a_1 & a_2 & a_3 & a_4 & a_5 \end{array} \\ \left[\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 5 & 5 & 5 \\ 0 & 0 & 5 & 9 & 9 & 9 \\ 0 & 0 & 5 & 9 & 13 & 13 \\ 0 & 7 & 7 & 9 & 13 & 20 \\ 0 & 7 & 7 & 9 & 13 & 20 \end{array}\right] \end{array} \tag{5.3}$$

In matrix (5.4) this backtracking is shown. When a cell $F(i,j)$ is marked with gray, then node b_i and a_j are aligned with each other. Figure 5.8 shows the best ordered matching of the statements. There we can see that all of the nodes are matched, except for a_1 and b_4.

$$F = \begin{array}{c} \text{nodes} \\ \phi \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{array} \begin{array}{c} \begin{array}{cccccc} \phi & a_1 & a_2 & a_3 & a_4 & a_5 \end{array} \\ \left[\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \boxed{5} & 5 & 5 & 5 \\ 0 & 0 & 5 & \boxed{9} & 9 & 9 \\ 0 & 0 & 5 & 9 & \boxed{13} & 13 \\ 0 & 7 & 7 & 9 & 13 & 20 \\ 0 & 7 & 7 & 9 & 13 & \boxed{20} \end{array}\right] \end{array} \tag{5.4}$$

5.4 Description of ASTSIM-LCS

In this section I give a description of ASTSIM-LCS. Since the Top-Down Unordered Maximum Common Subtree Isomorphism method has already been described in Section 5.3.1, I will only give a description of how LCS should be used. The pseudo code for the implementation of ASTSIM-LCS is given in Section 6.2.

ASTSIM-LCS uses the LCS to compare the method bodies of methods. Let $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ be two ASTs, and let $W_1 \subset V_1$ be the nodes in a method body in T_1 and $W_2 \subset V_2$ be the nodes of a method body in T_2. In order to find an alignment with a maximum score of the nodes in W_1 and W_2, we need to do a traversal of the two subtrees that represents the method bodies. We then get two ordered sequences of the nodes in W_1 and W_2. For all nodes $v_i \in W_1$ and $w_j \in W_2$, i and j denotes in which order the node is visited during the traversal. For LCS I do a preorder traversal of each method body.

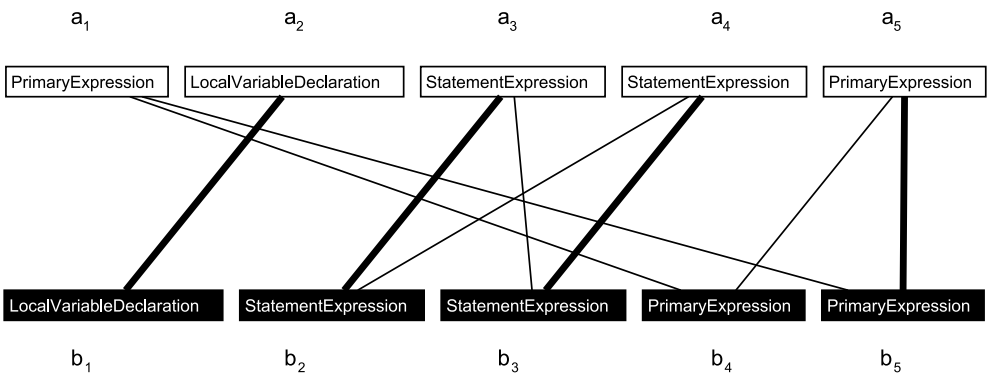


Figure 5.8: An optimal alignment between the statements and local variable declarations from the methods in listing 5.1 and 5.2.

To find an optimal solution to this alignment problem we will use a $(n+1) \times (m+1)$ matrix denoted by c. In this matrix there is one row for the gap character (row 0) and one row for each of the nodes from the preorder traversal of W_1. There is also one column for the gap character (column 0) and one column for each of the nodes from the preorder traversal of W_2. We fill this matrix with values according to the recursive formula defined in equation 2.8 on page 13. Each cell $c(i, j)$ in the matrix will be an optimal solution of the alignment of the first i nodes in preorder traversal of W_1 with the first j nodes in the preorder traversal of W_2. An algorithm for the computation of an optimal alignment is given in Algorithm 1, while an algorithm for finding the actual alignment is given in Algorithm 2.

Algorithm 1 Find an optimal alignment of the nodes in W_1 and W_2.

```
1:  for i = 0 to m do
2:      c(i, 0) ← 0
3:  end for
4:  for j = 0 to n do
5:      c(0, j) ← 0
6:  end for
7:  for i = 1 to m do
8:      for j = 1 to n do
9:          if label(v_i) = label(w_j) then
10:             c(i, j) = c(i − 1, j − 1) + 1
11:         else
12:             c(i, j) = max(d(i, j − 1), d(i − 1, j))
13:         end if
14:     end for
15: end for
```

If we use these algorithms on the method bodies of the Listings 5.1 and 5.2, we get the alignments shown in Figures 5.9 and 5.10. Taking a closer look at this alignment, we can see that there are some problems. In Figure 5.9 we have the subtree rooted at *27 StatementExpression*.

Algorithm 2 Find the actual alignment of the nodes in W_1 and W_2.

```
1: i ← m
2: j ← n
3: while i > 0 and j > 0 do
4:    if label(v_i) = label(w_j) then
5:       Align v_i with w_j.
6:       i ← i − 1
7:       j ← j − 1
8:    else if c(i, j − 1) > c(i − 1, j) then
9:       j ← j − 1
10:   else
11:      i ← i − 1
12:   end if
13: end while
```

The nodes in this subtree are aligned with nodes from two different subtrees in Figure 5.10. The nodes numbered 27-29 are from the subtree rooted at *27 StatementExpression*, while the node numbered 30 is from the subtree rooted at *PrimaryExpression*. One way to solve this problem is to only align nodes that have the same parent. We can see in Figures 5.9 and 5.10 that the nodes labeled *30 Name (tmp)* and *30 Name (a)* have different parents in the two trees. The problem with this approach is that we still can align nodes in one subtree with nodes from two or more subtrees as long as the nodes have the same parents. Another problem is that we cannot align two nodes labeled *Block* if they have different parents. As we know, loops can be transformed into other loops and selection statements can be transformed into other selection statements. In the trees of two different loops the block nodes will have different parents. We can solve these problems by always aligning block nodes and by only aligning a node v_1 with w_1 if the parent of v_1 is aligned with the parent of w_1. By using this strategy, the nodes labeled *30 Name (tmp)* and *30 Name (a)* will not be part of the alignment in Figures 5.9 and 5.10.

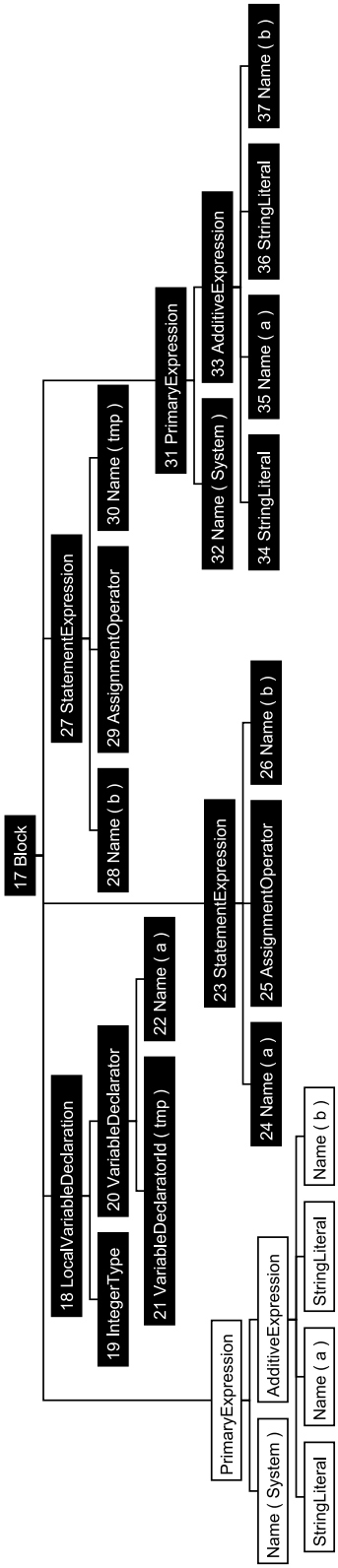


Figure 5.9: AST of the method body from Listing 5.1. The nodes that are part of the alignment are in black, and each node is numbered to show which node it is aligned with in Figure 5.10.

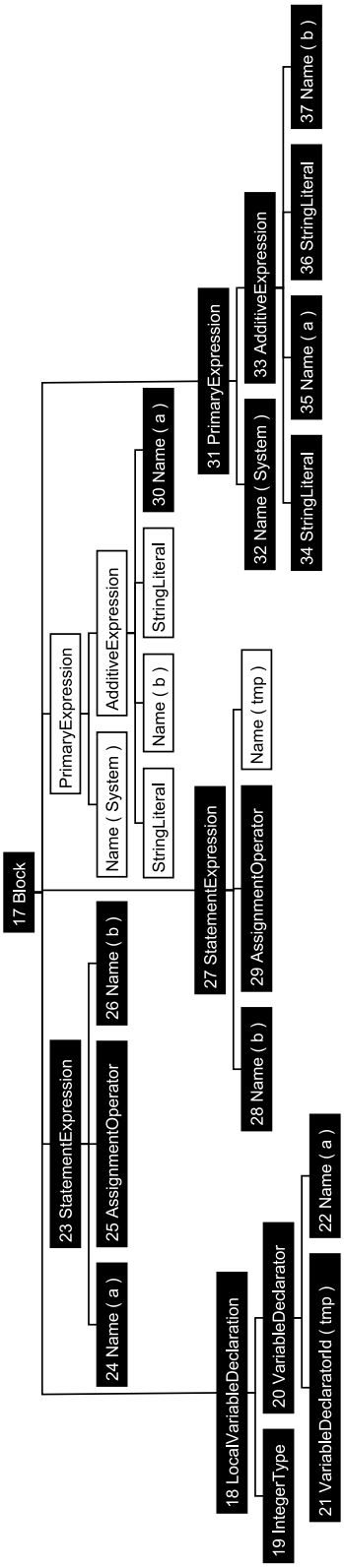


Figure 5.10: AST of the method body from Listing 5.2. The nodes that are part of the alignment are in black, and each node is numbered to show which node it is aligned with in Figure 5.9.

Chapter 6

Implementation of the new similarity measures

In this chapter I present pseudo code implementations of the two new similarity measures ASTSIM-NW and ASTSIM-LCS and provide details about the actual implementations. Moreover, I outline how these implementations were tested.

6.1 Pseudo code for ASTSIM-NW

The pseudo code for the ASTSIM-NW algorithm is given in Algorithms 3 (Part I) and 4 (Part II). The algorithm is called with the root nodes of the two trees $T_1 = (V_1, E_2)$ and $T_2 = (V_2, E_2)$, and with the Map B. The map B stores all the nodes in V_2 that a node $v \in V_1$ can be mapped to. After the top-down unordered maximum common subtree between T_1 and T_2 has been found, the map B is used to find the mapping M of this maximum common subtree.

In Algorithm 3 (Part I) we first check if the node labels correspond. If they do not, then there is no maximum common subtree between subtrees rooted at those nodes. Then we check if one or both of the nodes are leaf nodes. If one or both are, then we have a maximum common subtree of size one. If r_1 and r_2 are not leaf nodes and their labels correspond, then we can find the maximum common subtree between the subtrees rooted at those nodes. In order to find this subtree, we first need to find maximum common subtrees of all the subtrees rooted at the children of r_1 into all the subtrees rooted at the children of r_2. For the children of r_1 and r_2 we build a bipartite graph. For each of the children of r_1 and r_2 we create a vertex that is used to represent the child in the graph. We use the Map GT to map a node to a vertex, and the Map T_1G and T_2G to map a vertex to a node.

In Algorithm 4 (Part II) we use a double for-loop to find the maximum common subtrees between the children of r_1 and r_2. If there is a maximum common subtree between a subtree rooted at v_1 and the subtree rooted at v_2, then we create an edge between the vertices that represents v_1 and v_2 in the bipartite graph. The weight of this edge is the size of this maximum common subtree. When we have found all the maximum common subtrees, we find the maximum common subtree between r_1 and r_2. This is either done with Needleman-Wunsch, if r_1 and r_2 have the label BLOCK, or with a maximum weight bipartite matching algorithm. Both algorithms return the edges between the matched vertices in the bipartite graph. Then we can find the size of the maximum common subtree between r_1 and r_2 and which of the children of r_1 and r_2 that are part of this maximum common subtree. First we set $result$ to 1, since r_1

Algorithm 3 ASTSIM-NW($Node\ r_1$, $Node\ r_2$, $Map < Node, List < Node >> B$) **(Part I)**

```
1: if label(r_1) ≠ label(r_2) then
2:     return 0
3: end if
4: if isLeafNode(r_1) or isLeafNode(r_2) then
5:     return 1
6: end if
7:
8: Map < Node, Vertex > T_1G
9: Map < Node, Vertex > T_2G
10: Map < Vertex, Node > GT
11:
12: List < Vertex > U
13: for all v_1 ∈ children(r_1) do
14:     Create new Vertex v
15:     Insert v into U
16:     GT[v] ← v_1
17:     T_1G[v_1] ← v
18: end for
19:
20: List < Vertex > W
21: for all v_2 ∈ children(r_2) do
22:     Create new Vertex w
23:     Insert w into W
24:     GT[w] ← v_2
25:     T_2G[v_2] ← w
26: end for
```

and r_2 can be mapped to each other. Then we add the weight of each matched edge to $result$. Each matched edge goes from the vertices that represents the children of r_1 to the vertices that represents the children of r_2. If a child of r_1 has already been mapped to nodes in T_2, then we use the list that has been mapped to this node in B, otherwise we use a new list. The child $GT[v_2]$ of r_1 is then inserted into this list, and then the list is mapped to $B[GT[v_1]]$. When we have gone through all the matched edges, we return the size of the maximum common subtree between r_1 and r_2.

Algorithm 4 ASTSIM-NW($Node\ r_1$, $Node\ r_2$, $Map < Node, List < Node >> B$) **(Part II)**

```
1:  List < Edge > edges
2:  for all v1 ∈ children(r1) do
3:      for all v2 ∈ children(r2) do
4:          result ← ASTSIM-NW(v1,v2,B)
5:          if result ≠ 0 then
6:              Create new Edge e = (T1G[v1], T2G[v2])
7:              e.weight ← result
8:              Insert e into edges
9:          end if
10:     end for
11: end for
12:
13: List < Edge > matchedEdges
14: if label(r1) = label(r2) = BLOCK then
15:     matchedEdges ← Needleman-Wunsch(U,W,edges)
16: else
17:     matchedEdges ← MaxWeightBipartiteMatching(U,W,edges)
18: end if
19:
20: result ← 1
21: for all e ∈ matchedEdges do
22:     Vertex v1 ← source(e), where v1 ∈ U
23:     Vertex v2 ← target(e), where v2 ∈ W
24:
25:     List < Node > list
26:     if B[GT[v1]] ≠ nil then
27:         list ← B[GT[v1]]
28:     end if
29:
30:     Insert GT[v2] into list
31:     B[GT[v1]] ← list
32:     result ← result + e.weight
33: end for
34:
35: return result
```

After we have found the the size of the maximum common subtree between the root nodes of T_1 and T_2, we can find the mapping M. This procedure was explained in Section 5.3.1, so

here I will only state the algorithm. The algorithm is given in Algorithm 5. We can see that that the Map M contains the mapping between nodes in T_1 and T_2, and that the size of the mapping equals $|M| = sizeOfMapping$. The size of the mapping M is then used to calculate the similarity score between T_1 and T_2. This is done by using the similarity function defined in equation 2.19 on page 20.

Algorithm 5 Find the mapping M for ASTSIM-NW

```
1:  Map < Node, Node > M
2:  M[root(T_1)] ← root[T_2]
3:  sizeOfMapping ← 1
4:
5:  List < Node > tree_1 ← Preorder traversal of T_1
6:  for all v ∈ tree_1 do
7:    if B[v] ≠ nil then
8:      List < Node > list ← B[v]
9:      for all w ∈ list do
10:       if M[parent(v)] = parent(w) then
11:         M[v] ← w
12:         sizeOfMapping ← sizeOfMapping + 1
13:       end if
14:     end for
15:   end if
16: end for
```

6.2 Pseudo code for ASTSIM-LCS

The pseudo code for the ASTSIM-LCS algorithm is given in Algorithms 6 (Part I) and 7 (Part II). Observe that the algorithm is almost identical to the algorithm of ASTSIM-NW, given in Algorithms 3 and 4. The differences are that we use a Longest Common Subsequence algorithm to assess similarities between method bodies and that we do not use the Needleman-Wunsch algorithm anymore.

The LCS algorithm is given in Algorithms 8 (Part I) and 9 (Part II). This algorithm is implemented from the description given in Section 5.4. In Algorithm 8 we first do a preorder traversal of the two subtrees rooted at r_1 and r_2, where r_1 and r_2 are root nodes of subtrees that represents method bodies in T_1 and T_2, respectively. Then we find the maximum size of an optimal alignment between the nodes in $subtree_1$ and $subtree_2$. Observe that nodes are only aligned if they share the same label. Moreover, the parents need to share the same label or the two nodes v_1 and v_2 need to be labeled BLOCK. After we have found this maximum size, we can find the actual alignment between the two sequences $subttree_1$ and $subtree_2$.

The actual alignment that was found between the sequences $subtree_1$ and $subtree_2$ in Algorithm 8, is not necessarily the alignment that we will use between the nodes in the subtree rooted at r_1 and the nodes in the subtree rooted at r_2. In Algorithm 9 a top-down approach is used to find an alignment between those nodes. The alignment is stored in the Map $alignment$. A node $v \in subtree_1$ can only be aligned to a node $w \in subtree_2$ if $tmp[v] = w$ and one of the following requirements are satisfied:

Algorithm 6 ASTSIM-LCS($Node\ r_1$, $Node\ r_2$, $Map < Node, List < Node >> B$) **(Part I)**

```
1: if label(r_1) = label(r_2) = BLOCK and label(parent(r_1)) = label(parent(r_2)) = METHOD-
   DECLARATION then
2:    result ← LCS(v_1,v_2,B)
3:    return result
4: end if
5:
6: if label(r_1) ≠ label(r_2) then
7:    return 0
8: end if
9: if isLeafNode(r_1) or isLeafNode(r_2) then
10:   return 1
11: end if
12:
13: Map < Node, Vertex > T_1G
14: Map < Node, Vertex > T_2G
15: Map < Vertex, Node > GT
16:
17: List < Vertex > U
18: for all v_1 ∈ children(r_1) do
19:    Create new Vertex v
20:    Insert v into U
21:    GT[v] ← v_1
22:    T_1G[v_1] ← v
23: end for
24:
25: List < Vertex > W
26: for all v_2 ∈ children(r_2) do
27:    Create new Vertex w
28:    Insert w into W
29:    GT[w] ← v_2
30:    T_2G[v_2] ← w
31: end for
```

Algorithm 7 ASTSIM-LCS($Node\ r_1$, $Node\ r_2$, $Map < Node, List < Node >> B$) **(Part II)**

```
1:  List < Edge > edges
2:  for all v_1 ∈ children(r_1) do
3:     for all v_2 ∈ children(r_2) do
4:        result ← ASTSIM-NW(v_1,v_2,B)
5:        if result ≠ 0 then
6:           Create new Edge e = (T_1G[v_1], T_2G[v_2])
7:           e.weight ← result
8:           Insert e into edges
9:        end if
10:    end for
11: end for
12:
13: List < Edge > matchedEdges
14: matchedEdges ← MaxWeightBipartiteMatching(U,W,edges)
15:
16: result ← 1
17: for all e ∈ matchedEdges do
18:    Vertex v_1 ← source(e), where v_1 ∈ U
19:    Vertex v_2 ← target(e), where v_2 ∈ W
20:
21:    List < Node > list
22:    if B[GT[v_1]] ≠ nil then
23:       list ← B[GT[v_1]]
24:    end if
25:
26:    Insert GT[v_2] into list
27:    B[GT[v_1]] ← list
28:    result ← result + e.weight
29: end for
30:
31: return result
```

Algorithm 8 LCS($Node\ r_1$, $Node\ r_2$, $Map < Node, List < Node >> B$) **(Part I)**

1: $List < Node > subtree_1 \leftarrow$ Preorder traversal of the subtree rooted at r_1
2: $List < Node > subtree_2 \leftarrow$ Preorder traversal of the subtree rooted at r_2
3: $m \leftarrow$ Size of the subtree rooted at r_1
4: $n \leftarrow$ Size of the subtree rooted at r_2
5: $Array < INT, INT > c$
6:
7: **for** $i = 0$ to m **do**
8: $c(i, 0) \leftarrow 0$
9: **end for**
10: **for** $j = 0$ to n **do**
11: $c(0, j) \leftarrow 0$
12: **end for**
13:
14: **for** $i = 1$ to m **do**
15: **for** $j = 1$ to n **do**
16: $v_1 \leftarrow$ *(i-1)-th* element in $subtree_1$
17: $v_2 \leftarrow$ *(j-1)-th* element in $subtree_2$
18: **if** (label(v_1) = label(v_2)) **and** ((label(parent(v_1) = label(parent(v_2)) **or** label(v_1) = BLOCK) **then**
19: $c(i, j) = c(i - 1, j - 1) + 1$
20: **else**
21: $c(i, j) = max(c(i, j - 1), c(i - 1, j))$
22: **end if**
23: **end for**
24: **end for**
25:
26: $Map < Node, Node > tmp$
27: $i \leftarrow m$
28: $j \leftarrow n$
29: **while** $i > 0$ **and** $j > 0$ **do**
30: $v_1 \leftarrow$ *(i-1)-th* element in $subtree_1$
31: $v_2 \leftarrow$ *(j-1)-th* element in $subtree_2$
32: **if** (label(v_1) = label(v_2)) **and** ((label(parent(v_1) = label(parent(v_2)) **or** label(v_1) = BLOCK) **then**
33: $tmp[v_1] \leftarrow v_2$
34: $i \leftarrow i - 1$
35: $j \leftarrow j - 1$
36: **else if** $c(i, j - 1) > c(i - 1, j)$ **then**
37: $j \leftarrow j - 1$
38: **else**
39: $i \leftarrow i - 1$
40: **end if**
41: **end while**

- Node v is the root node in the subtree rooted at r_1. The nodes r_1 and r_2 can be aligned with each other since their parents share the same label (METHODDECLARATION).

- The parents of v and w are aligned, with $aligned[parent(v)] = parent(w)$. This requirement ensures that there is a path of aligned nodes $aligned[v] = w, aligned[parent(v)] = parent[w], \ldots, aligned[b_1] = b_2$, where b_1 and b_2 are labeled BLOCK.

- Node v and w are labeled BLOCK, and minimum one child of v is aligned to a child of w.

If a node v can be aligned to a node w, then w is inserted into the list of nodes that can be mapped w. At the end, the result of the alignment between the nodes in $subtree_1$ and $subtree_2$ is returned.

After we have found the the size of the maximum common subtree between the root nodes of $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$, we can find the mapping M. This algorithm is given in Algorithm 10. The algorithm is similar to the algorithm given in Algorithm 5, but there are some differences. In Algorithm 5 we have the requirement that a node $v \in V_1$ can only be mapped to a node $w \in V_2$ if $M[parent(v)] = w$ and $w \in list = B[v]$. When we use the LCS algorithm, we can align two nodes $v \in V_1$ and $w \in V_2$ without aligning the parents of v and w, when both v and w are labeled BLOCK. For instance, the parent of v can be labeled FORSTATEMENT, while the parent of w can be labeled WHILESTATEMENT. Then $M[parent(v)] \neq parent(w)$, and the requirement from Algorithm 5 is no longer satisfied. In order to solve this problem, we do preorder traversals of the subtrees rooted at v and w when v and w are the root nodes in subtrees that represents method bodies. For each node $v_1 \in subtree_1$ we check if a node $w_1 \in subtree_2$ can be mapped to v_1. If that is the case, then $M[v_1] = w_1$. After haveing found the mapping M and the size $|M| = sizeOfMapping$ of this mapping, we can calculate the similarity score between T_1 and T_2. This is done by using the similarity function defined in equation 2.19 on page 20.

6.3 Actual implementation of the new similarity measures

All the algorithms have been written in the Java 1.5 language, and the actual implementation can be found at *http://olavsli.at.ifi.uio.no/thesis*.

The Top-Down Unordered Maximum Common Subtree Isomorphism algorithm has been implemented by using source code listings, written in C++ with the use of the LEDA library (Mehlhorn and Näher, 1999), from Valiente (2002). LEDA is a library, written in C++, that contains data structures and algorithms for combinatorial and geometric computing. To implement the Maximum Weight Bipartite Matching algorithm, I used source code listings from Mehlhorn and Näher (1999). Since the matching algorithm need a priority queue to do shortest-path computations in the bipartite graph, I implemented a Fibonacci heap based on a pseudo code implementation given in Cormen et al. (2001). The use of a Fibonacci heap is recommended by Mehlhorn and Näher (1999), since this heap improves the asymptotic running time of the shortest-path computation. For both the implementation of the isomorphism algorithm and the matching algorithm, I created data structures that are similar to the data structures used in LEDA.

Algorithm 9 LCS($Node\ r_1$, $Node\ r_2$, $Map < Node, List < Node >> B$) **(Part II)**

```
1:  result ← 0
2:  Map < Node, Node > alignment
3:
4:  for all v ∈ subtree_1 do
5:    if tmp[v] ≠ nil then
6:      w ← tmp[v]
7:
8:      if v = r_1 then
9:        alignment[v] ← w
10:     else
11:       p_1 ← parent(v)
12:       p_2 ← parent(w)
13:
14:       if alignment[p_1] = p_2 then
15:         alignment[v] ← w
16:       else if label(v) = BLOCK then
17:         for all v_1 ∈ children(v) do
18:           if tmp[v_1] ≠ nil then
19:             w_1 ← tmp[v_1]
20:             if parent(w_1) = w then
21:               alignment[v] ← w
22:             end if
23:           end if
24:         end for
25:       end if
26:     end if
27:
28:     if alignment[v] = w then
29:       List < Node > list
30:       if B[v] ≠ nil then
31:         list ← B[v]
32:       end if
33:
34:       Insert alignment[v] into list
35:       B[v] ← list
36:       result ← result + 1
37:     end if
38:   end if
39: end for
40:
41: return result
```

Algorithm 10 Find the mapping M for ASTSIM-LCS

```
1:  Map < Node, Node > M
2:  M[root(T_1)] ← root[T_2]
3:  sizeOfMapping ← 1
4:
5:  List < Node > tree_1 ← Preorder traversal of T_1
6:  for all v ∈ tree_1 do
7:    if v.checked = false and B[v] ≠ nil then
8:      List < Node > list ← B[v]
9:      for all w ∈ list do
10:       p_1 ← parent(v)
11:       p_2 ← parent(w)
12:       if M[p_1] = p_2 then
13:         if label(v) = label(w) = BLOCK and label(p_1) = label(p_2) = METHODDECLAR-
            ATION then
14:           List < Node > subtree_1 ← Preorder traversal of the subtree rooted at v
15:           List < Node > subtree_2 ← Preorder traversal of the subtree rooted at w
16:
17:           for all v_1 ∈ subtree_1 do
18:             v_1.checked ← true
19:             List < Node > list2 ← B[v]
20:
21:             for all w_1 ∈ list2 do
22:               if w_1 ∈ subtree_2 then
23:                 M[v_1] ← w_1
24:                 sizeOfMapping ← sizeOfMapping + 1
25:               end if
26:             end for
27:           end for
28:         else
29:           M[v] ← w
30:           sizeOfMapping ← sizeOfMapping + 1
31:         end if
32:       end if
33:     end for
34:   end if
35: end for
```

For the Longest Common Subsequence algorithm I used to pseudo code implementation given in Cormen et al. (2001) to implement it, while for the Needleman-Wunsch algorithm I used the description given in Needleman and Wunsch (1970).

6.4 Testing of the actual implementations

I cannot guarantee that the implementations do not contain errors, but the algorithms have been tested on large sets of Java listings (> 350) without errors. I have, however, found an error in the parser generated by JavaCC by using the original grammar[1]. The error is that the parser cannot parse Java listings that contains a single line comment at the end of the listing. If such a comment is found, the parser reports a parser error. During the testing of the algorithms I have removed such comments manually, but these comments should be removed automatically from listings in a future version of the plagiarism detection program.

During the testing of the algorithms I made it possible to manually check the matching between two ASTs $T_1 = (V_1, E_2)$ and $T_2 = (V_2, E_2)$. By doing so, I could assess if the matchings were reasonable. I made an option of making a file in the GML graph file format for each AST, when comparing two program listings. Then I could manually inspect the matching between the two trees by viewing the files in the yEd graph editor program[2]. In the GML graph files I used different colors for matched and unmatched nodes, and I numbered the nodes in order to see which nodes in the two trees that were matched to each other.

[1] JavaCC grammar for Java 1.5: *https://javacc.dev.java.net/files/documents/17/3131/Java1.5.zip*
[2] yEd: *http://www.yworks.com/en/products_yed_about.htm*

Chapter 7

Comparing Joly with ASTSIM-NW and ASTSIM-LCS

In this chapter I compare Joly with ASTSIM-NW and ASTSIM-LCS by using different tests. I first find a common similarity score for the three algorithms. Then I assess the algorithms' ability to discover plagiarized listing pairs by use of three different tests. Finally, I compare the running times of the three algorithms.

7.1 Finding a common similarity score for the algorithms

Joly reports degree of similarity in terms of an angle (measured in degrees), while my algorithms outputs a similarity score between 0 and 1, where a score close to 1 means that the program listings are very similar. To be able to compare Joly with ASTSIM-NW and ASTSIM-LCS I transformed Joly's similarity score into a similarity score taking on values in $[0, 1]$. This is done by use of the following function:

$$s(\varphi) = \frac{1}{1 + a \cdot \varphi} \tag{7.1}$$

Here, a is a constant describing slope of the function s and φ is the angle.

The constant a was estimated the following way: I took 368 program listings (67528 listing pairs), written by students, and found the similarity scores between them by using ASTSIM-NW and Joly. I sorted the records (listing pair, score ASTSIM-NW, angle Joly) based on the ASTSIM-NW score in decreasing order. Then I calculated the mean similarity score of ASTSIM-NW, which was found to be 0.32. A manual inspection of the list showed that a large number (1440) of the listing pairs had the score value very close to 0.32. The associated Joly scores in these 1440 listing pairs had a mean angle value of $6.447\,^\circ$. The two scoring systems was then calibrated by determination of a by use of equation 7.1. Though this calibration is quite crude, it ensures that small angles get high scores while large angles get low scores. For instance, a listing pair with $\varphi = 0.5\,^\circ$ is considered as a possible copy in (Kielland,2006). The mapping of φ to a number between 0 and 1 will then give the result $s(0.5) = 0.86$. This would also be high enough to classify the pair as a possible copy.

7.2 Comparing Joly with ASTSIM-NW and ASTSIM-LCS by the use of ROC curves

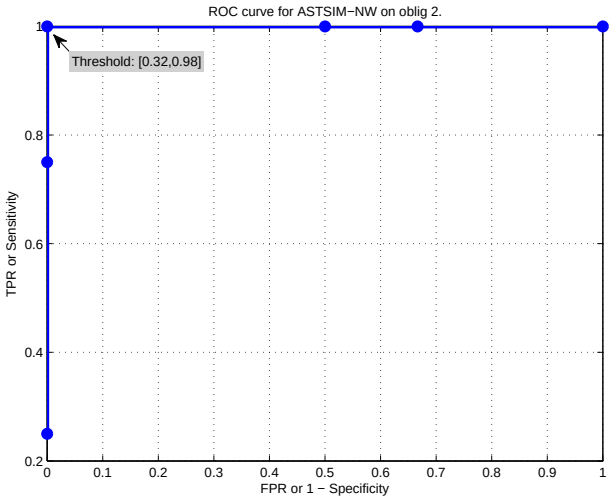
In the first test I compared the results of the three algorithms on three small sets of listings. I have run the three algorithms on mandatory programming exercises handed in by the students in the subject INF1000 - "Introduction to object-oriented programming". Only listings from the same exercises are tested against each other. The three exercises are called oblig 2, oblig 3 and oblig 4, respectively. Some basic information about them are provided in Table 7.1. Note that the number of listings in exercise oblig 2 is much smaller than those of the other two. I also want to point out that in the exercise oblig 4, the students were given a predefined structure by the course administration for solving the exercise. Observe, that I have, in each exercise set, identified the number of listing pairs that are copies. This is necessary in order to build ROC curves for the different algorithms.

No. of listings	Average size (lines / nodes)	Max size (lines / nodes)	Min size (lines / nodes)	No. of listing pairs	No. of listing pairs that are copies
OBLIG 2					
5	204 / 635	236 / 686	176 / 554	10	4
OBLIG 3					
12	425 / 1537	535 / 1847	336 / 1221	66	7
OBLIG 4					
11	479 / 1732	617 / 2236	261 / 1113	55	6

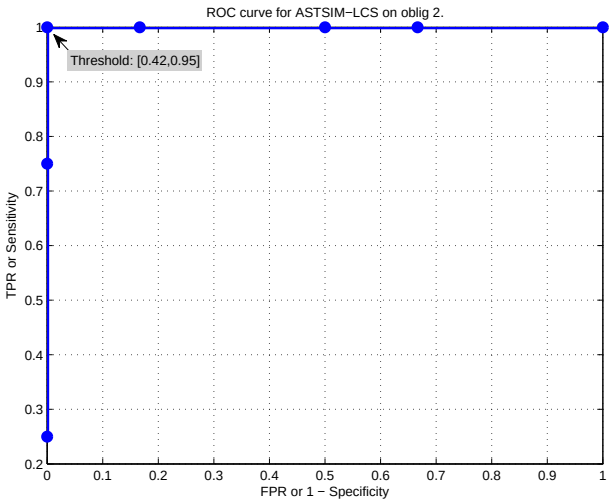
Table 7.1: Data for the three exercise sets oblig 2, 3 and 4. For average-, min- and max size I both show the numbers of lines of code and nodes in the AST.

7.2.1 Results for oblig 2

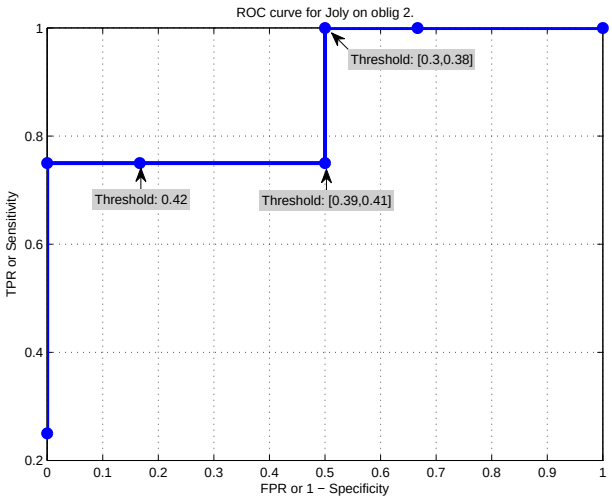
Tables 7.2 - 7.4 shows the results the three algorithms on oblig 2. We can see that both ASTSIM-NW and ASTSIM-LCS assign high scores to the listing pairs that were classified as copies during the manual inspection. Joly also assigns high scores to these pairs, with the exception of the pair 2.1 - 2.3. The ROC curves for the three algorithms are shown in Figure 7.1. We can see that both ASTSIM-NW and ASTSIM-LCS achieves a perfect classification in ROC space. ASTSIM-NW's classification is perfect for a threshold $t = [0.38, 0.92]$, while ASTSIM-LCS's has a perfect classification for a threshold $t = [0.42, 0.95]$. Joly does not obtain a perfect classification, since the last positive pair is found with a threshold $t = 0.38$. Before it finds this pair, false positives have been found for thresholds $t = [0.39, 0.42]$. I have to point out that even if ASTSIM-NW and ASTSIM-LCS achieves a perfect classification here, that does not need to be case for another set of listings. Here I have only used 10 listing pairs, which is far too few to draw a conclusion about how good the algorithms are to classify the listing pairs. In Figure 7.4a on page 84 we have the three ROC curves in one ROC graph. We can see that the ROC curves of ASTSIM-NW and ASTSIM-LCS are better than or equal to Joly's ROC curve for all threshold values. For this test set with only 10 listing pairs, we say that ASTSIM-NW and ASTSIM-LCS are better to classify the listing pairs than Joly.



(a) ASTSIM-NW



(b) ASTSIM-LCS



(c) Joly

Figure 7.1: ROC curves for the three algorithms on oblig 2.

	2.1	2.2	2.3	2.4	2.5
2.1	-	0.37	**0.92**	0.37	0.37
2.2	-	-	0.33	**0.98**	**1.00**
2.3	-	-	-	0.34	0.33
2.4	-	-	-	-	**0.98**
2.5	-	-	-	-	-

Table 7.2: Results for ASTSIM-NW on oblig 2. A score in **bold** indicates that the listing pair was classified as a copy during the manual inspection.

	2.1	2.2	2.3	2.4	2.5
2.1	-	0.40	**0.95**	0.41	0.40
2.2	-	-	0.38	**0.98**	**1.00**
2.3	-	-	-	0.39	0.38
2.4	-	-	-	-	**0.98**
2.5	-	-	-	-	-

Table 7.3: Results for ASTSIM-LCS on oblig 2. A score in **bold** indicates that the listing pair was classified as a copy during the manual inspection.

7.2.2 Results for oblig 3

Tables 7.5 - 7.7 shows the results for the three algorithms on oblig 3. We can see that both ASTSIM-NW and ASTSIM-LCS assign high scores to the listing pairs that were classified as copies during the manual inspection. Joly also assigns high scores to these pairs, with the exception of the pair 3.7 - 3.8. We can also see that Joly assigns high scores to the listing pairs 3.3 - 3.4, 3.4 - 3.9 and 3.4 - 3.12. These scores are in *italics* in Table 7.7, and the corresponding listing pairs are not copies. The ROC curves for the three algorithms are shown in Figure 7.2. Also for these listing pairs, we can see that both ASTSIM-NW and ASTSIM-LCS achieves a perfect classification in ROC space. ASTSIM-NW's classification is perfect for a threshold $t = [0.45, 0.7]$, while ASTSIM-LCS's has a perfect classification for a threshold $t = [0.49, 0.77]$. Joly does not obtain a perfect classification, since the last positive pair is found with a threshold $t = 0.55$. Before it finds this pair, many false positives are found. Joly actually finds the first false positive for a threshold $t = 0.89$, while the next one is found for a threshold $t = 0.84$. In Figure 7.4b on page 84 we have the three ROC curves in one ROC graph. We can see that the ROC curves of ASTSIM-NW and ASTSIM-LCS are better than or equal to Joly's ROC curve

	2.1	2.2	2.3	2.4	2.5
2.1	-	0.28	**0.38**	0.29	0.28
2.2	-	-	0.41	**0.91**	**1.00**
2.3	-	-	-	0.42	0.41
2.4	-	-	-	-	**0.91**
2.5	-	-	-	-	-

Table 7.4: Results for Joly on oblig 2. A score in **bold** indicates that the listing pair was classified as a copy during the manual inspection.

for all threshold values.

	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10	3.11	3.12
3.1	-	**0.99**	0.35	0.35	0.39	0.38	0.34	0.41	0.36	0.34	0.34	0.37
3.2	-	-	0.35	0.35	0.39	0.38	0.34	0.41	0.36	0.34	0.34	0.36
3.3	-	-	-	0.42	0.37	0.39	0.40	0.36	**0.75**	0.39	0.38	**0.76**
3.4	-	-	-	-	0.34	0.37	0.42	0.38	0.42	0.35	0.35	0.42
3.5	-	-	-	-	-	**0.80**	0.31	0.39	0.38	0.41	0.40	0.38
3.6	-	-	-	-	-	-	0.31	0.39	0.39	0.40	0.40	0.39
3.7	-	-	-	-	-	-	-	**0.70**	0.44	0.35	0.36	0.44
3.8	-	-	-	-	-	-	-	-	0.39	0.43	0.44	0.39
3.9	-	-	-	-	-	-	-	-	-	0.36	0.36	**0.99**
3.10	-	-	-	-	-	-	-	-	-	-	**0.98**	0.36
3.11	-	-	-	-	-	-	-	-	-	-	-	0.36
3.12	-	-	-	-	-	-	-	-	-	-	-	-

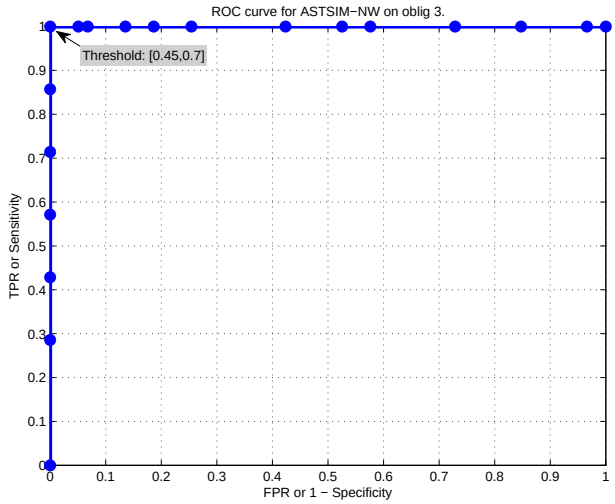
Table 7.5: Results for ASTSIM-NW on oblig 3. A score in **bold** indicates that the listing pair was classified as a copy during the manual inspection.

	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10	3.11	3.12
3.1	-	**0.99**	0.40	0.38	0.39	0.38	0.34	0.38	0.39	0.33	0.33	0.39
3.2	-	-	0.40	0.37	0.39	0.37	0.34	0.38	0.39	0.33	0.33	0.39
3.3	-	-	-	0.41	0.41	0.39	0.42	0.43	**0.77**	0.43	0.43	**0.77**
3.4	-	-	-	-	0.45	0.45	0.44	0.48	0.44	0.37	0.37	0.44
3.5	-	-	-	-	-	**0.79**	0.40	0.41	0.45	0.38	0.38	0.45
3.6	-	-	-	-	-	-	0.36	0.38	0.44	0.38	0.38	0.44
3.7	-	-	-	-	-	-	-	**0.84**	0.48	0.39	0.40	0.48
3.8	-	-	-	-	-	-	-	-	0.48	0.43	0.44	0.48
3.9	-	-	-	-	-	-	-	-	-	0.42	0.42	**0.99**
3.10	-	-	-	-	-	-	-	-	-	-	**0.98**	0.42
3.11	-	-	-	-	-	-	-	-	-	-	-	0.42
3.12	-	-	-	-	-	-	-	-	-	-	-	-

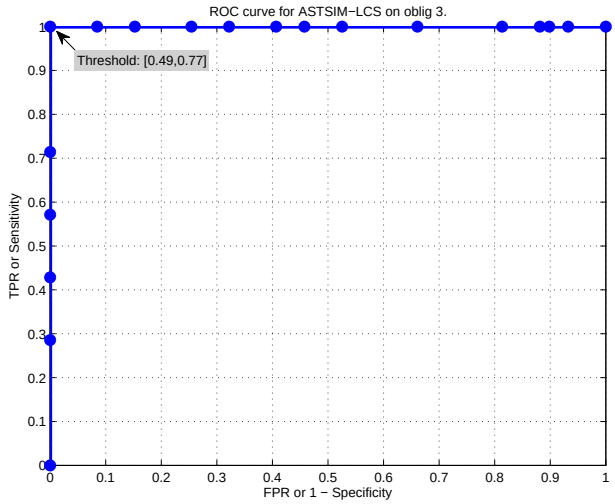
Table 7.6: Results for ASTSIM-LCS on oblig 3. A score in **bold** indicates that the listing pair was classified as a copy during the manual inspection.

7.2.3 Results for oblig 4

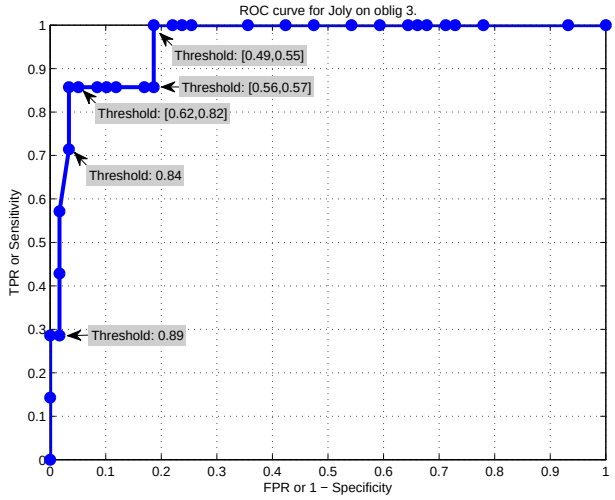
Tables 7.8 - 7.10 shows the results the three algorithms on oblig 4. We can see that both ASTSIM-NW and ASTSIM-LCS assign high scores to the listing pairs that were classified as copies during the manual inspection, with the exception of the listing pair 4.6 - 4.11. Joly also assigns high scores to these pairs, with the exception of the pairs 4.5 - 4.8, 4.6 - 4.11 and 4.8 - 4.9. The ROC curves for the three algorithms are shown in Figure 7.3. This time, both ASTSIM-NW and ASTSIM-LCS do not achieve a perfect classification in ROC space. ASTSIM-NW's finds false positives for the thresholds $t = [0.55, 0.56]$. For the threshold $t = 0.55$ it also finds the last true positive. ASTSIM-LCS's finds false positives for the thresholds $t = [0.56, 0.58]$. For



(a) ASTSIM-NW



(b) ASTSIM-LCS



(c) Joly

Figure 7.2: ROC curves for the three algorithms on oblig 3.

	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10	3.11	3.12
3.1	-	**0.93**	0.24	0.24	0.23	0.23	0.27	0.30	0.23	0.18	0.18	0.24
3.2	-	-	0.23	0.23	0.23	0.23	0.26	0.29	0.23	0.18	0.18	0.23
3.3	-	-	-	*0.89*	0.58	0.57	0.48	0.35	**0.84**	0.39	0.39	**0.88**
3.4	-	-	-	-	0.58	0.58	0.47	0.35	*0.82*	0.38	0.39	*0.84*
3.5	-	-	-	-	-	**0.92**	0.39	0.30	0.61	0.34	0.34	0.61
3.6	-	-	-	-	-	-	0.38	0.30	0.60	0.33	0.33	0.59
3.7	-	-	-	-	-	-	-	**0.55**	0.45	0.33	0.34	0.48
3.8	-	-	-	-	-	-	-	-	0.34	0.27	0.28	0.35
3.9	-	-	-	-	-	-	-	-	-	0.39	0.39	**0.86**
3.10	-	-	-	-	-	-	-	-	-	-	**0.83**	0.38
3.11	-	-	-	-	-	-	-	-	-	-	-	0.38
3.12	-	-	-	-	-	-	-	-	-	-	-	-

Table 7.7: Results for Joly on oblig 3. A score in **bold** indicates that the listing pair was classified as a copy during the manual inspection, while a score in *italics* indicates that the listing pair has a high score and has not been classified as a copy.

the threshold $t = 0.56$ it also finds the last true positive. Joly does neither obtain a perfect classification. We can see that Joly finds many false positives by using thresholds $t = [0.29, 0.48]$, before it finds the last true positive with the threshold $t = 0.29$. In Figure 7.4c on page 84 we have the three ROC curves in one ROC graph. As seen the ROC curves of ASTSIM-NW and ASTSIM-LCS are better than or equal to Joly's ROC curve for all threshold values, and also that ASTSIM-NW is a little bit better than ASTSIM-LCS since its FPR rate is lower or equal to ASTSIM-LCS's for all threshold values.

	4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9	4.10	4.11
4.1	-	**0.99**	0.42	0.36	0.41	0.32	0.42	0.44	0.34	0.41	0.33
4.2	-	-	0.42	0.36	0.41	0.32	0.42	0.44	0.34	0.41	0.32
4.3	-	-	-	**0.70**	0.45	0.38	0.56	0.54	0.42	0.56	0.42
4.4	-	-	-	-	0.45	0.30	0.50	0.44	0.41	0.49	0.36
4.5	-	-	-	-	-	0.33	0.44	**0.68**	0.55	0.43	0.34
4.6	-	-	-	-	-	-	0.37	0.41	0.24	0.37	**0.55**
4.7	-	-	-	-	-	-	-	0.55	0.42	**0.99**	0.43
4.8	-	-	-	-	-	-	-	-	**0.64**	0.54	0.44
4.9	-	-	-	-	-	-	-	-	-	0.43	0.32
4.10	-	-	-	-	-	-	-	-	-	-	0.43
4.11	-	-	-	-	-	-	-	-	-	-	-

Table 7.8: Results for ASTSIM-NW on oblig 4. A score in **bold** indicates that the listing pair was classified as a copy during the manual inspection.

7.3 Comparing the assignment of similarity scores by the three algorithms

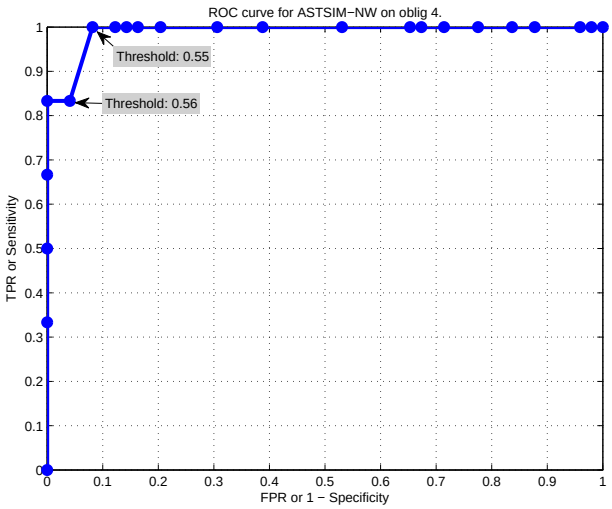
In the previous section we saw that ASTSIM-NW and ASTSIM-LCS were better than Joly to detect plagiarized listings in 3 small test sets. But even if they are better than Joly on these

	4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9	4.10	4.11
4.1	-	**0.99**	0.49	0.45	0.53	0.38	0.50	0.51	0.37	0.49	0.39
4.2	-	-	0.50	0.45	0.52	0.39	0.50	0.51	0.37	0.49	0.40
4.3	-	-	-	**0.76**	0.58	0.39	0.56	0.57	0.43	0.55	0.42
4.4	-	-	-	-	0.51	0.36	0.54	0.49	0.40	0.54	0.40
4.5	-	-	-	-	-	0.44	0.56	**0.74**	0.54	0.55	0.40
4.6	-	-	-	-	-	-	0.38	0.45	0.29	0.38	**0.56**
4.7	-	-	-	-	-	-	-	0.58	0.47	**0.99**	0.43
4.8	-	-	-	-	-	-	-	-	**0.67**	0.57	0.46
4.9	-	-	-	-	-	-	-	-	-	0.47	0.35
4.10	-	-	-	-	-	-	-	-	-	-	0.44
4.11	-	-	-	-	-	-	-	-	-	-	-

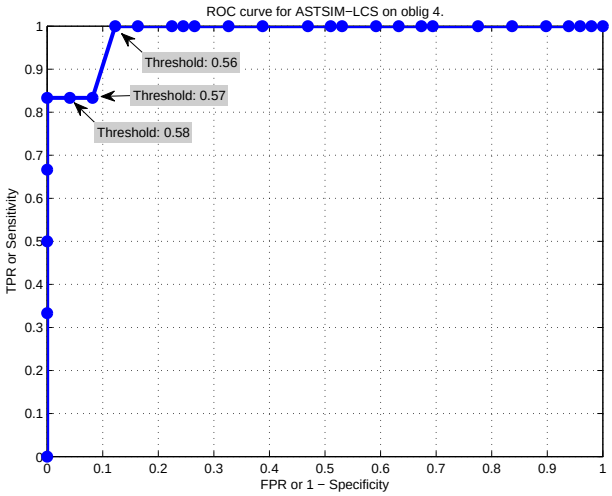
Table 7.9: Results for ASTSIM-LCS on oblig 4. A score in **bold** indicates that the listing pair was classified as a copy during the manual inspection.

	4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9	4.10	4.11
4.1	-	**0.94**	0.32	0.32	0.33	0.35	0.28	0.32	0.22	0.27	0.36
4.2	-	-	0.33	0.32	0.32	0.35	0.28	0.32	0.22	0.27	0.35
4.3	-	-	-	**0.79**	0.27	0.44	0.26	0.30	0.22	0.25	0.23
4.4	-	-	-	-	0.26	0.42	0.25	0.29	0.21	0.24	0.23
4.5	-	-	-	-	-	0.38	0.41	**0.57**	0.34	0.41	0.39
4.6	-	-	-	-	-	-	0.31	0.41	0.25	0.30	**0.29**
4.7	-	-	-	-	-	-	-	0.49	0.43	**0.88**	0.29
4.8	-	-	-	-	-	-	-	-	**0.38**	0.47	0.33
4.9	-	-	-	-	-	-	-	-	-	0.45	0.23
4.10	-	-	-	-	-	-	-	-	-	-	0.29
4.11	-	-	-	-	-	-	-	-	-	-	-

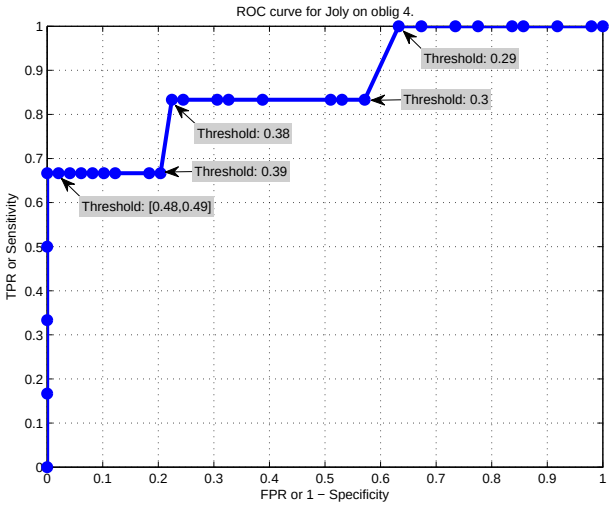
Table 7.10: Results for Joly on oblig 4. A score in **bold** indicates that the listing pair was classified as a copy during the manual inspection.



(a) ASTSIM-NW

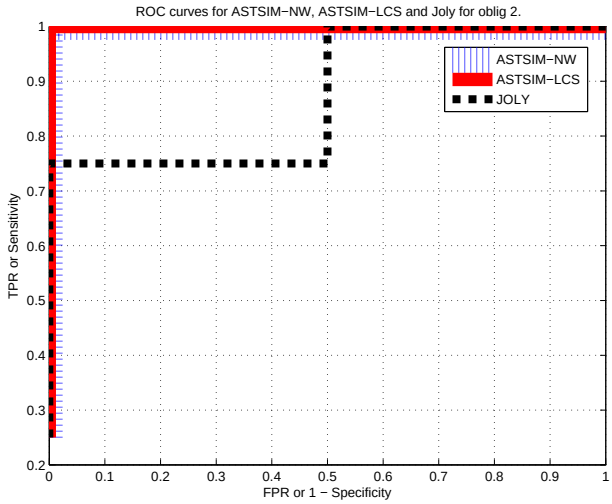


(b) ASTSIM-LCS

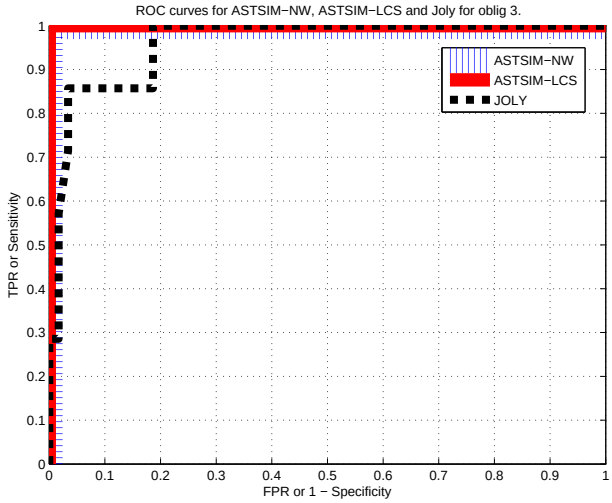


(c) Joly

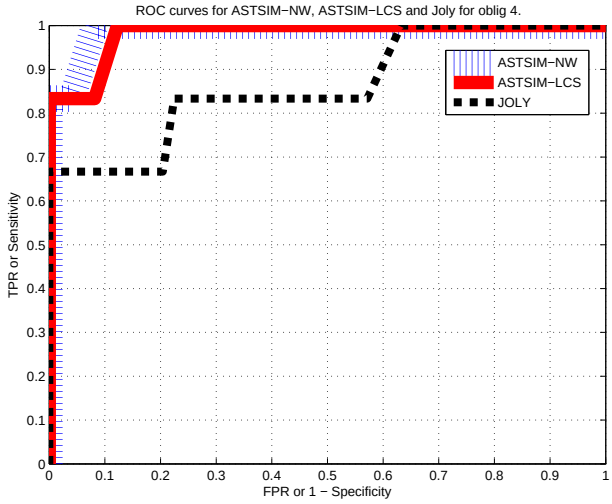
Figure 7.3: ROC curves for the three algorithms on oblig 4.



(a) Oblig 2



(b) Oblig 3



(c) Oblig 4

Figure 7.4: Comparing ROC curves for the three algorithms on the three data sets.

sets, they do not necessary need to be better on some other test sets. I will therefore do another test where I test which of Joly and ASTSIM-NW/ASTSIM-LCS that gives the most appropriate score to a listing pair. To do this I have selected 10 listing pairs where Joly finds a high score, while ASTSIM-NW/ASTSIM-LCS finds a low score and vice versa.

The listing pairs have been selected as follows: I have run the three algorithms on a set of 368 listings. Then I have sorted the results for Joly and ASTSIM-NW in decreasing order. For each of the listing pairs in the sorted lists I have found the corresponding scores for either ASTSIM-NW and ASTSIM-LCS or Joly and ASTSIM-LCS. I have then gone through the lists in decreasing order and selected listing pairs where there is a difference of 0.30 or higher between the scores of ASTSIM-NW and Joly and between the scores of ASTSIM-LCS and Joly. A difference of 0.30 is chosen to ensure that either Joly or ASTSIM-NW/ASTSIM-LCS has a much lower score than the other algorithm(-s). For each of the 20 listing pairs I have manually classified them as copies or not copies. In Table 7.11 are the results for the 20 pairs. For the pairs 1-10 ASTSIM-NW/ASTSIM-LCS have high scores, and for the pairs 11-20 Joly has high scores. We can see in Table 7.11 that ASTSIM-NW/ASTSIM-LCS assigns the most appropriate scores to the different pairs. When they assign high scores then the pairs are copies, and when they assign low scores then the pairs are not copies. On the other hand we can see that Joly does the opposite for these pairs.

Pair	ASTSIM-NW (score)	ASTSIM-LCS (score)	JOLY (score)	Diff. Joly and ASTSIM-NW	Diff. Joly and ASTSIM-LCS	Copies?
1	0.98	0.98	0.51	0.47	0.47	YES
2	0.95	0.95	0.53	0.42	0.42	YES
3	0.88	0.90	0.49	0.39	0.41	YES
4	0.86	0.89	0.46	0.40	0.43	YES
5	0.86	0.87	0.42	0.44	0.45	YES
6	0.83	0.84	0.38	0.45	0.46	YES
7	0.81	0.81	0.37	0.44	0.44	YES
8	0.80	0.79	0.47	0.33	0.32	YES
9	0.77	0.77	0.40	0.37	0.37	YES
10	0.77	0.78	0.40	0.37	0.38	YES
11	0.44	0.46	0.87	0.43	0.41	NO
12	0.49	0.47	0.83	0.34	0.36	NO
13	0.46	0.46	0.83	0.36	0.36	NO
14	0.48	0.51	0.82	0.34	0.31	NO
15	0.46	0.47	0.82	0.35	0.34	NO
16	0.37	0.40	0.82	0.45	0.42	NO
17	0.38	0.36	0.81	0.43	0.45	NO
18	0.49	0.50	0.81	0.32	0.31	NO
19	0.36	0.38	0.81	0.45	0.43	NO
20	0.47	0.43	0.81	0.34	0.38	NO

Table 7.11: The results of Joly and ASTSIM-NW/ASTSIM-LCS on at set of 20 listing pairs.

7.4 Similarity scores produced by the different cheating strategies

In the two previous sections we have seen that ASTSIM-NW/ASTSIM-LCS have assigned more appropriate scores to listing pairs than Joly in the different tests. It seems like Joly are more sensitive to the different cheating strategies than ASTSIM-NM/ASTSIM-LCS. In this section I will investigate the effect on the similarity score for the different algorithms by applying the different strategies.

I will test the effect of the strategies 3-10 from Section 2.1.3. The strategies 1 and 2 will not be considered since all of the algorithms are immune against these strategies, and 12 will also not be used. In this test I start with the listing P_0. I will then apply the different strategies one by one on this listing. We then get the listing P_i where $i = 3, \ldots, 10$. For each strategy that I apply I will measure the similarity between P_0 and P_i with the three algorithms. I will then see how the similarity score between P_0 and the other listings is affected when using more and more strategies.

It is also interesting to look at the effect of applying only one strategy. When we measure the similarity between P_0 and P_i we get the combined effect of applying the strategies 3 to i. If we on the other hand measure the similarity between P_i and P_{i-1}, then we get the effect of only using the strategy i. I consider both situations in the following.

The listing P_0 that I use is about 200 lines long and it is from the exercise oblig 2. It is found in Listing B.1 in Appendix B. In this appendix we also have the listing P_{10}, with all strategies applied. This listing is found in Listing B.2. Table 7.12 shows the different listings that were made from P_0. For each listing I have listed what kind of changes that were done. For each strategy I have done a reasonable amount of changes with respect to the size of listing P_0.

Listing	Based on	Changes applied
P_0		
P_3	P_0	The order of the operands were changed in 5 expressions.
P_4	P_3	The data types of 3 local variables were changed from int to double.
P_5	P_4	Two expressions were replaced with equivalent expressions, 3 assignments statements were replaced with one assignment statement, and two assignment statements were replaced with one assignment statement.
P_6	P_5	7 redundant statements and 3 local variable declarations were inserted.
P_7	P_6	7 independent local variable declarations and statements were moved.
P_8	P_7	Two for-loops, with a single statement, and a for-loop, with a switch-statement with 5 case labels, were changed to while-loops.
P_9	P_8	A switch-statement with 6 case labels was changed to nested if-statements.
P_{10}	P_9	A method call was replaced with the method body. The method body contains 10 lines of code.

Table 7.12: The different listings based on P_0.

7.4.1 The effect of applying more and more cheating strategies

In Table 7.13 we see the similarity scores for the different algorithms on the different pairs. Figure 7.5 shows how the similarity scores decreases by using more and more strategies and in Table 7.14 displays the vectors that Joly makes for the different listings. As expected, the three algorithms assign a perfect score when comparing P_0 with P_0, P_1 and P_2. For P_3 only Joly and ASTSIM-NW assign a perfect score.

For P_4 we can see that the score decreases a little for both ASTSIM-NW and ASTSIM-LCS, and that Joly still has a perfect score since there is no difference between the vectors of P_0 and P_4. First at P_5 that the score for Joly decreases. We can see in Table 7.14 that both the number of "=" and ";" decreases. Also, ASTSIM-NWs and ASTSIM-LCSs scores decrease, but they both have a higher score than Joly.

In Figure 7.5 we can see that the strategy of inserting redundant statements and local variable declarations has a great impact on Joly's similarity score. The score decreases from 0.8840 to 0.4780 since the vector of P_6 contains 10 more ";" than the vector of P_0. ASTSIM-NW and ASTSIM-LCS scores decrease also, but only a little. For P_7 Jolys score does not change, since there is no difference between the vectors of P_6 and P_7. We can see that the score of ASTSIM-LCS decreases more than the score of ASTSIM-NW.

In P_8 I have replaced three for-loops with while-loops. We can see that Joly's score actually increases by doing this, since the vector of P_8 contains one less ";" than the vector of P_7. For ASTSIM-NW and ASTSIM-LCS, we can see that ASTSIM-NW's score decreases more than ASTSIM-LCS, and that ASTSIM-LCS gets a higher score than ASTSIM-NW. The same thing applies to P_9 where ASTSIM-LCS's score decreases less than ASTSIM-NW's score. Joly's score also decreases for P_9, since its vector is changing. One interesting observation about the vector of P_9 is that it now contains the same number of ";" as P_0. From having 95, 95 and 94 ";" for P_6, P_7 and P_8 respectively, it has now only 88. But this does not help since the number of "if(" and "=" increases.

By replacing a method call with the method body we get a decrease in the similarity score for all three algorithms. We can see that both ASTSIM-NW and ASTSIM-LCS scores decrease by somewhat the same amount, while Joly's score decreases less than them. In the end ASTSIM-NW, ASTSIM-LCS and Joly ends up with the similarity scores 0.6306, 0.7033 and 0.3824 respectively. For Joly the listing pair P_0 - P_{10} has such a low score that we would not suspect that the programs are copies, while the similarity scores for ASTSIM-NW and ASTSIM-LCS could still indicate that the programs are copies, and then especially ASTSIM-LCS's score.

7.4.2 The effect of applying a single cheating strategy

In Table 7.15 we have the similarity scores assigned by the different algorithms for the listing pairs P_i and P_{i+1}, where $i = 0, \ldots, 9$. For the different pairs, the similarity score is plotted for each of the algorithms in Figure 7.6. In this figure we can see that all three algorithms assign a perfect score for the pairs $P_0 - P_1$ and $P_1 - P_2$, while only ASTSIM-NW and Joly assign a perfect score for the pair $P_2 - P_3$.

For the pair $P_3 - P_4$ we can see that both ASTSIM-NW and ASTSIM-LCS assign the same score, which they also do for the pairs $P_4 - P_5$ and $P_5 - P_6$. Joly assigns a perfect score to

Pair	ASTSIM-NW (score)	ASTSIM-LCS (score)	JOLY (score)	JOLY (angle)
0 - 1	1.0	1.0	1.0	0.0
0 - 2	1.0	1.0	1.0	0.0
0 - 3	1.0	0.9910	1.0	0.0
0 - 4	0.9892	0.9801	1.0	0.0
0 - 5	0.9681	0.9590	0.8840	0.3982
0 - 6	0.9423	0.9335	0.4780	3.3133
0 - 7	0.8731	0.8394	0.4780	3.3133
0 - 8	0.7538	0.7947	0.4999	3.0347
0 - 9	0.6743	0.7568	0.3983	4.5835
0 - 10	0.6306	0.7033	0.3824	4.9000

Table 7.13: The scores between the original listing and the different listing with changes.

	Program listings										
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
"void"	6	6	6	6	6	6	6	6	6	6	5
"String"	5	5	5	5	5	5	5	5	5	5	5
">"	2	2	2	2	2	2	2	2	2	2	2
"class"	1	1	1	1	1	1	1	1	1	1	1
"if("	5	5	5	5	5	5	5	5	5	11	11
"="	65	65	65	65	65	62	62	62	62	74	74
"static"	1	1	1	1	1	1	1	1	1	1	1
";"	88	88	88	88	88	85	95	95	94	88	87
","	2	2	2	2	2	2	2	2	2	2	2
"public"	1	1	1	1	1	1	1	1	1	1	1

Table 7.14: The vectors for the different listing made by Joly.

the pair $P_3 - P_4$, but for the pair $P_4 - P_5$ the score drops to 0.9205 since there is a difference between the vectors of P_4 and P_5 in Table 7.14. For the pair $P_5 - P_6$ we can see that Joly assigns a score of 0.5060. In Table 7.14 we can see that there is a big difference between the vectors of P_5 and P_6. The vector of P_6 has 10 more ";" than the vector of P_5.

When we change the order of independent statements and local variable declarations in the listing P_7, we see that ASTSIM assigns a little higher score to the pair $P_6 - P_7$ than ASTSIM-LCS (See Section 8.3 for an explanation). For the same pair, Joly achieves a perfect score. In the program P_8 I have replaced for-loops with while-loops. We can see that ASTSIM-LCS assigns a much better score for the pair $P_7 - P_8$ than ASTSIM-NW, which only assigns 0.7513 for this pair (See Section 8.3 for an explanation). Joly assigns a high score to this pair, since the vectors of P_7 and P_8 only differ with one ";".

For the pair $P_8 - P_9$ we can see that ASTSIM-LCS still assigns a better score than ASTSIM-NW, while Joly assigns a score of 0.2981. If we compare the vectors of P_8 and P_9 in Table 7.14, we can see that there is a big difference between them. The biggest difference is that the number of "if(" has doubled in P_9 compared to P_8. This has an impact on the angle between the vectors,

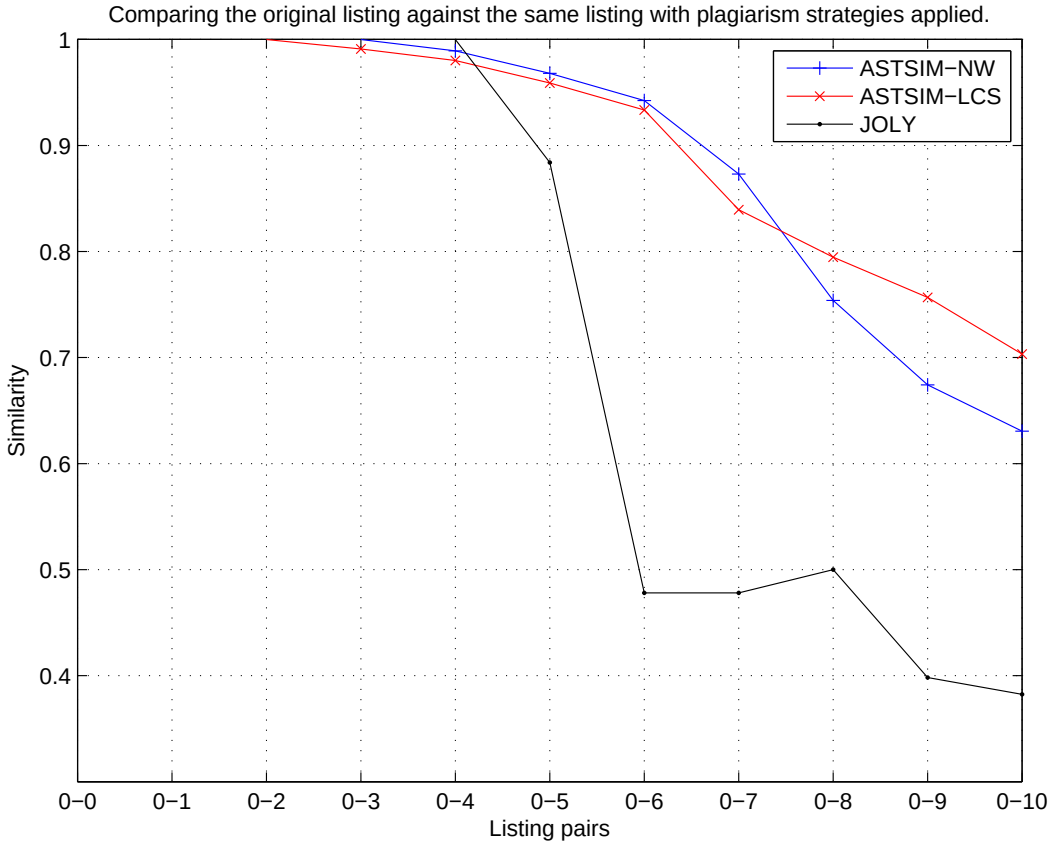


Figure 7.5: Comparing the three algorithms.

which we can see in Table 7.15. The angle is now 7.1431. For the last pair, $P_9 - P_{10}$ we can see that ASTSIM-NW and ASTSIM-LCS assign almost the same score, while Joly assigns a score of 0.8417.

7.5 Comparing the running times of the three algorithms

The algorithm in Joly is very fast since it stores the vectors of the program listings in the database. When a new listing is submitted to the system, the algorithm only needs to calculate the vector of that listing before it compares it against vectors from the database. Another thing that makes the algorithm fast is that it does not compare listings where the byte counts[1] of two listings differ with 10 % or more. Such listings are not compared, since the algorithm cannot detect that the listings are copies when the byte counts differ with 10 % or more. One example of that algorithm is very fast, is that it can handle that over 400 of the students in the course INF1000 submit their programming assignments to Joly during the last hour before deadline.

For both ASTSIM-NW and ASTSIM-LCS I have measured the running time when they compare 50 and 100 listings. For both tests I determined the average time it took to compare two listings. The results for the tests are shown in Table 7.16 for ASTSIM-NW and in Table 7.17 for

[1]The byte count is calculated after blanks, comments and strings are removed, and it is stored in the database.

Pair	ASTSIM-NW (score)	ASTSIM-LCS (score)	JOLY (score)	JOLY (angle)
0 - 1	1.0	1.0	1.0	0.0
1 - 2	1.0	1.0	1.0	0.0
2 - 3	1.0	0.9910	1.0	0.0
3 - 4	0.9892	0.9892	1.0	0.0
4 - 5	0.9790	0.9790	0.9205	0.2620
5 - 6	0.9731	0.9731	0.5060	2.9624
6 - 7	0.9197	0.9005	1.0	0.0
7 - 8	0.7513	0.9084	0.9163	0.2771
8 - 9	0.8875	0.9619	0.2981	7.1431
9 - 10	0.9241	0.9224	0.8417	0.5704

Table 7.15: The effect of applying a single cheating strategy.

ASTSIM-LCS. We can see that ASTSIM-NW uses 610 milliseconds on one comparison, while ASTSIM-LCS uses about 560 milliseconds. For both ASTSIM-NW and ASTSIM-LCS most of this time is used when comparing the ASTs, while very little time is used to build the ASTs and to do modifications on them.

No. of listings	No. of listing pairs	Total time (in seconds)	Average time for a listing pair (in seconds)
50	1225	149.38	0.0610
100	4950	601.79	0.0610

Table 7.16: The running time for ASTSIM-NW on 50 and 100 listings.

No. of listings	No. of listing pairs	Total time (in seconds)	Average time for a listing pair (in seconds)
50	1225	134.92	0.0551
100	4950	555.90	0.0562

Table 7.17: The running time for ASTSIM-LCS on 50 and 100 listings.

In equations 7.2 and 7.3 I have calculated the time that ASTSIM-NW and ASTSIM-NW would use to compare $n = 2000$ listings, when ASTSIM-NW uses $t(\text{ASTSIM-NW}) = 0.061$ and ASTSIM-LCS uses $t(\text{ASTSIM-LCS}) = 0.056$ on average to compare two listings. We can see that the total running time is high for both algorithms. ASTSIM-NW uses 121939 seconds which equals 33.87 hours, while ASTSIM-LCS uses 111944 seconds which equals 31.1 hours. For the same number of listings, the algorithm in Joly would only use a fraction of the time that ASTSIM-NW and ASTSIM-LCS uses.

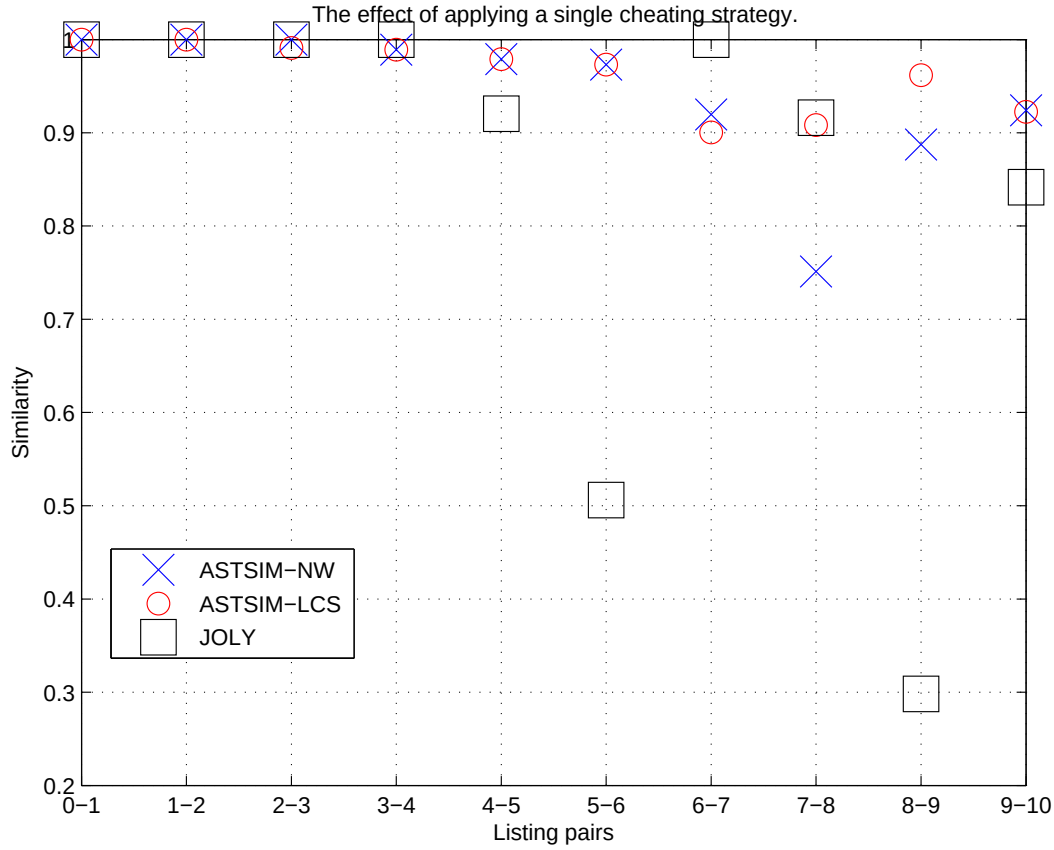


Figure 7.6: The effect of applying a single cheating strategy.

$$\begin{aligned} \text{Total running time} &= \frac{n \cdot (n-1)}{2} \cdot t(\text{ASTSIM-NW}) \\ &= \frac{2000 \cdot 1999}{2} \cdot 0.061 \\ &= 121939 \text{ sec} \end{aligned} \tag{7.2}$$

$$\begin{aligned} \text{Total running time} &= \frac{n \cdot (n-1)}{2} \cdot t(\text{ASTSIM-LCS}) \\ &= \frac{2000 \cdot 1999}{2} \cdot 0.056 \\ &= 111944 \text{ sec} \end{aligned} \tag{7.3}$$

Chapter 8

Discussion

In this chapter I discuss possible shortcommings of the new algorithms, alternative representations to AST of the code, ASTSIM-NW's advantages over ASTSIM-LCS and vice versa, and issues regarding the implementation of the new algorithms in Joly.

8.1 Possible shortcommings of the new algorithms

8.1.1 Removal of redundant information from the AST

During the testing of the new algorithms it was discovered that some trees of non-similar code, became similar after the removal of redundant nodes. For example, the statements `i++;` and `return a;` have similar trees after the removal of redundant nodes, as seen in Figure 8.1. I do not consider this to be a serious problem. This situation will only occur for some statements and/or expressions, and since the trees of these structures normally are small the mismatch will also be small. In this example there would only be a mismatch of 1, when matching *Name (i)* and *Name (a)*.

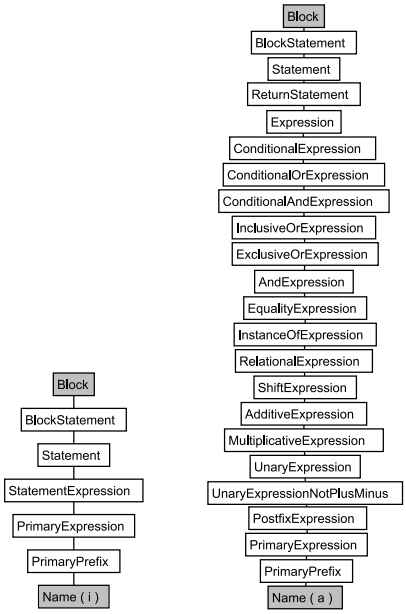
8.1.2 Unmodified AST representation

The algorithms have not been tested on unmodified ASTs. To do such a test the algorithms would need to be somewhat modified, since they are now optimized for the modified trees. It is difficult to predict the similarity scores such a test would produce, but an unmodified AST will be much larger than the modified AST one and thus cause substantial increase of the running time.

8.1.3 The use of unordered nodes

ASTSIM-NW treats all the nodes in the tree as unordered nodes, with exception of the nodes that are roots in subtrees of statements and local variable declarations within method bodies. During the testing, few unwanted rotations of nodes were detected. Most of the unwanted rotations were related to operands in expressions, where the order of the operands was important. Especially for the additive expression there were some unwanted rotations, when it was used as an argument to methods that print to screen. Below are two method calls to `println` in `System.out`, where both calls use an additive expression as argument. We can see that the order of the operands is important in both expressions.

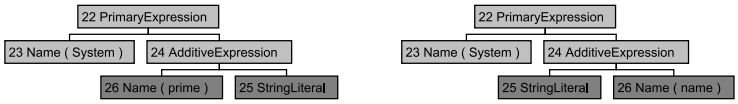
1. `System.out.println(prime + "is a prime number.");`
2. `System.out.println("My name is "+ name);`



(a) AST of *i++;* (b) AST of *return a;*

Figure 8.1: Two different statements which become similar after the removal of redundant nodes. The nodes in white are removed, while the nodes in gray are still in the tree.

In Figure 8.2 are the ASTs of the two statements, where nodes in dark gray have been matched by rotations. We can see in this example that few nodes are affected by this unwanted rotation. In general, unwanted rotations of operands in expressions is not a big problem, since only small subtrees are rotated. These rotations will have little effect on the result when comparing the ASTs of two program listings.



(a) AST of statement 1 (b) AST of statement 2

Figure 8.2: Matching between statements 1 and 2. The nodes are numbered to show which node they are matched to. Dark gray is used to indicate that a node is matched to a rotated node.

8.1.4 Possible counter stategies

ASTSIM-NW and ASTSIM-LCS are vulnerable to techniques that make large changes to the structure of the AST. Such techniques includes: Replacing method calls with large method bodies (1), splitting a large method into two or more methods (2), insertion of a very large number of redundant statements and local variable declarations (3), insertion of large redundant methods

or classes (4), and so on. All these techniques can make the teaching assistent suspicious when he grades the assignment, but if we do not incorporate this human factor it will become much harder to uncover cheating.

The problem with (1) and (2) above is that the algorithms can only match a method against one other method. Consider this problem: A Java listing p_1 has only one method m that is of a large size. We split this method into the three methods m_1, m_2 and m_3 of some what equal size, where m_1 has a method call to m_2 and m_2 has a method call to m_3, and we get the Java listing p_2. This operation can involve insertion of new code to make p_2 work. When we then measure the similarity between the p_1 and p_2, m is matched against one of the three methods, large parts of m will be unmatched, and two of the methods in p_2 will also be unmatched. One possible solution could be to match the nodes in m against nodes in more than one method in p_2, but this approach can lead to problems when using it on other Java listings as we then can get a too high score between two listings.

For (3) and (4) the problem is that we will get many unmatched nodes in one of the trees. This will again give a lower similarity score between the two trees, since the score is affected by the number of unmatched nodes in the two trees. For statements it can be hard for the plagiarism program to determine if a statement is redundant or not. It may be easier to determine if local variable declarations, methods and classes are redundant. One possible solution, is that the program can check in the code to see if the variable, method or class is used in the program.

8.2 Alternative representations of the code

A Java listing can also have another representation than an AST. Two other possible representations are tokens and Java bytecode. In this section I will discusess pros and cons of using these representations instead of ASTs for plagiarism detection.

8.2.1 Tokens

The main advantage of using tokens instead of ASTs is that we do not need syntactically correct code. Joly has this advantage, even though it does not use tokens. A possible problem with tokens is that it can be hard to identify which parts of a token stream that corresponds to a class declaration, method and so on. For our problem we need to measure the similarity of each class declaration with all the other class declarations in order to find the best match between the different class declarations. In order to identify the class declarations or method declarations in a token stream we need to use regular expressions, while in an AST we can identify these structures by finding subtrees that are rooted at a node with a specific label, which is much easier.

8.2.2 Java bytecode

One advantage with Java bytecode is that it is easier to detect the similarities between different loops and between different selection statements. For instance, a for-loop and a while-loop will have the same representation in the bytecode. The main problem with bytecode is that the bytecodes of two Java listings can be more similar than the listings are. This is due to the fact that listings that have a high semantically similarity also can have very similar bytecodes, even if the two listings do not have a high structural similarity.

8.3 ASTSIM-NW vs. ASTSIM-LCS

In Section 7.4 we looked at the effect of the different cheating strategies on the similarity scores assigned by the different algorithms. For some of the strategies we saw that ASTSIM assigned better scores than ASTSIM-LCS, and vice versa. In this section I will elaborate on this in some more detail.

Needleman-Wunsch vs. Longest Common Subsequence

When we change the order of independent statements and local variable declarations, then ASTSIM-NW always finds the best alignment between those statements and local variable declarations, while ASTSIM-LCS does not necessarily find the best alignment. The reason that ASTSIM-NW always finds the best alignment is that the Needleman-Wunsch algorithm tests all possible alignments of the subtrees of the statements and local variable declarations in the two blocks. On the other hand, ASTSIM-LCS first finds the alignment between the two traversals of the two blocks. In this alignment we can have aligned nodes that will not be part of the final alignment. In Figure 8.3 we can see that ASTSIM-NW finds a better alignment than ASTSIM-LCS when comparing two blocks of statements and local variable declarations of the program listings P_6 and P_7 from Section 7.4. We can see that there is one local variable declaration that is not matched by ASTSIM-LCS, but that is matched by ASTSIM-NW.

Finding similarities between different structures

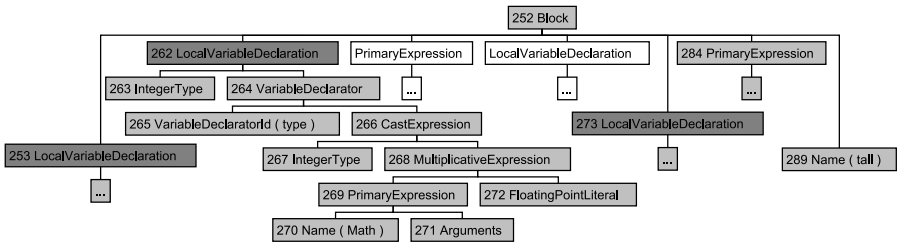
ASTSIM-NW cannot find similarities between subtrees where the labels of the root nodes do not correspond. If, for instance, a for-loop is replaced with a while-loop, we have subtrees where the root nodes do not correspond. On the other hand, ASTSIM-LCS can find similarities between such subtrees. In Section 7.4 we had an example where 3 for-loops were replaced by while-loops. In Figure 8.4 is an example of the similarity that ASTSIM-LCS found between one of the for-loops and one of the while-loops. For the same example ASTSIM-NW would find no similarity.

8.4 Practical implementation of the new algorithms in Joly

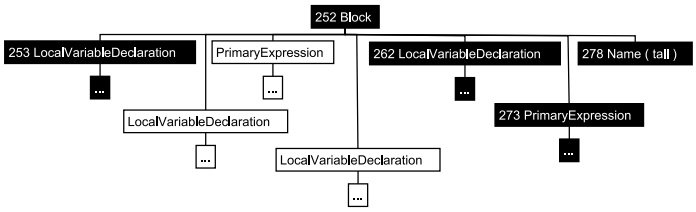
In this section I outline some practical implementation issues, regarding the implementation of the new algorithms in Joly.

8.4.1 Methods for reducing the running time

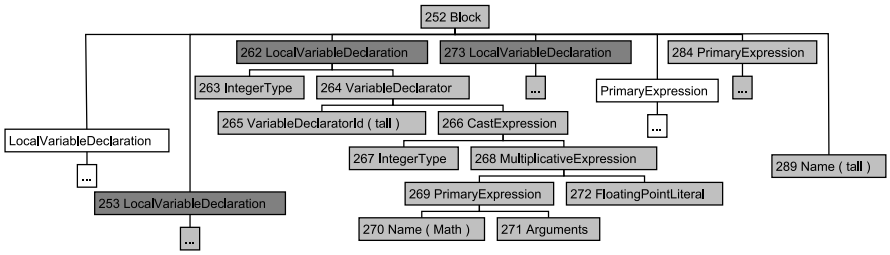
When a new assignment is submitted to Joly by a student, the assignment is compared against all the other assignments, where the difference between the byte counts is less than 10%. In Section 7.5 we saw that both algorithms used a very long time to compare a large set of Java listings. Due to this, the algorithms cannot be used to compare assignments at run time. There are three ways to address this problem: (1) to not compare the listings when they are received, but compare them later when the students have finished submitting to the system, (2) reduce the running time of the new algorithms, (3) use a faster algorithm at run time to select listing pairs that the two algorithms can compare later, when the students have finished submitting to the system. The last option implies that we use a fast filtering algorithm with the purpose of removing listing pairs that cannot be copies of each other. It appears that it may be easier to apply this strategy compared to dramatically improve the running time of the algorithms.



(a) ASTSIM-NW's alignment of some statements and local variable declarations from the listing P_6.



(b) ASTSIM-LCS's alignment of some statements and local variable declarations from the listing P_6.



(c) ASTSIM-NW's alignment of the same statements and local variable declarations from the listing P_7.

Figure 8.3: Alignment by ASTSIM-NW and ASTSIM-LCS. The nodes in white are unmatched nodes, while the others are matched.

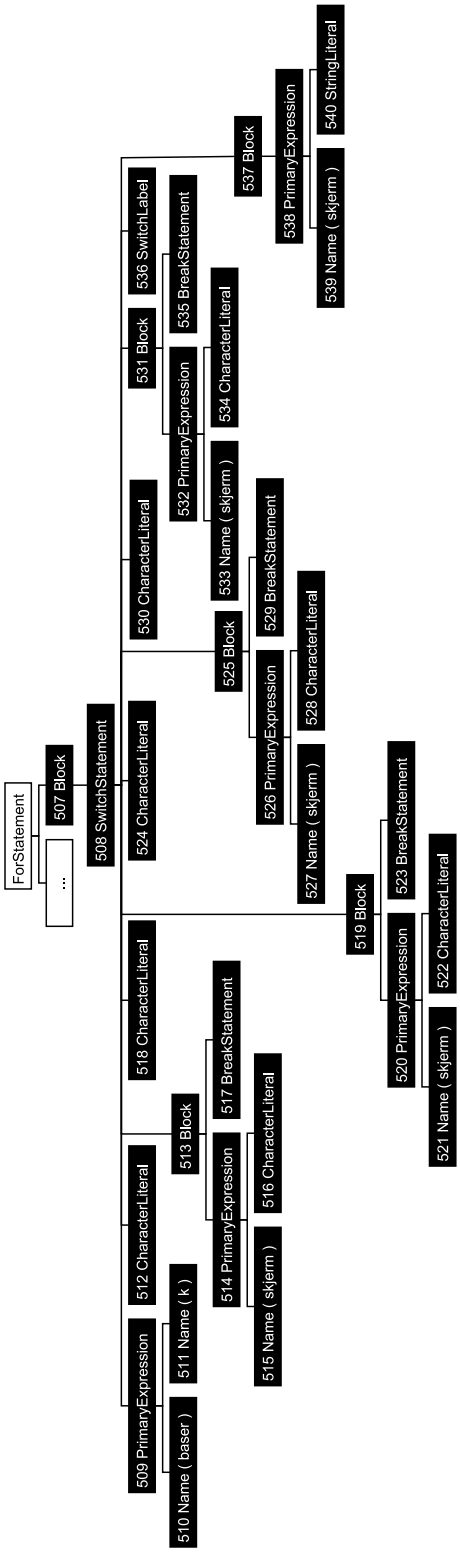


Figure 8.4: The match that ASTSIM-LCS finds between the body of a for-loop and the body of a while-loop, here shown in the for-loop. The nodes in black are the nodes that are matched.

8.4.2 Selecting threshold values

ASTSIM-NW and ASTSIM-LCS reports a similarity score s, in the range $[0, 1]$, when measuring the similarity between two program listings. To decide if a listing pair is a possible copy the score s is compared with a threshold t. When a plagiarism detection program is used on a set of programs listings P, we need a t which classifies the majority of the actual copies in P as copies. Moreover, t needs to control the proportion of false positives among all listing pairs classified as copies.

A threshold used for one set of program listings, does not neccesarily need to be a good threshold for another set of listings. The average size of a program listing can be very different in the two sets, or the listings in one of the sets can use a predefined structure, or the listings in one of the sets can use predefined code, and so on. For example, in Section 7.2.3 most of the students used a predefined structure, given by the course administration, to solve oblig 4. This is possibly one of the reasons that many of the none-copy listing pairs were assigned scores equal to or above 0.5 by ASTSIM-NW and ASTSIM-LCS. In Sections 7.2.1 and 7.2.2 none of the none-copy listing pairs were assigned scores above 0.5. Even though the three listing sets are small, there are idications of a pattern.

One way to choose a threshold value is through estimation. In this case the program listings used need to be of the same kind or have a similar structure to the set of listings which the threshold value is going to be applied on. In the following I present different methods to this end.

Simple methods for estimating threshold values

When we estimate a threshold value, we have to decide how large the proportion of false positives among all positives can be. If we only allow few or none false positives, then the threshold value will normally be high. The consequence of this is that many of the listings pairs that are actual copies, can be classified as not copies. On the other hand, if we allow many false positives, then the threshold value can become to low and we can get many false positives among the classified copies. The consequence of this is extra work for the person(s) which manually classifies the listing pairs.

A reasonable proportion of false positives can for instance be 5 %. We can use a set of 1000 listing pairs, where none of the listing pairs are copies, to estimate a threshold t. The similarity scores for these pairs are then sorted in increasing order, as shown in equation 8.1. Then we set $t = s_{950}$, which is the lowest score of the 5 % highest similarity scores. By doing so, we get a high threshold value that will ensure that the proportion of false positives will be reasonable.

$$s_1, \ldots, \boldsymbol{s_{950}}, \ldots, s_{1000} \tag{8.1}$$

Statistical methods for estimating threshold values

Hypothesis testing can be used for estimating a threshold value. When doing hypothesis testing on our problem, we want to find out if a set of program listings P contains program listings that are copies of each other. Program listings that are copies of each other can be called "associated". For each listing pair $(p_i, p_j) \in P \times P$ we can then test the null hypothesis $H_0(i, j)$: Program listing i and j are not associated. The alternative hypothesis for the same pair is $H_1(i, j)$: Program listing i and j are associated.

For a set of program listing P with size $n = |P|$ the null hypotheses are

$$\begin{aligned} H_0(1,2) &: \text{Program listing 1 and 2 are not associated.} \\ H_0(1,3) &: \text{Program listing 1 and 3 are not associated.} \\ &\vdots \\ H_0(n-1,n) &: \text{Program listing n-1 and n are not associated.} \end{aligned}$$

This is a multiple hypotheses test since we test several hypothesis at the same time. For each listing pair $(p_i, p_j) \in P \times P$, where $1 \leq i \leq n-1$ and $i < j \leq n$, we compare the similarity score s of the listing pair with a threshold t. We then have four possible outcomes for each null hypothesis test which are shown in Table 8.1.

$H_0(i,j)$	Comparing s and t	Actual classification of (p_i, p_j)
true	$s < t$	True negative
true	$s \geq t$	False positive (Type I error)
false	$s < t$	False negative (Type II error)
false	$s \geq t$	True positive

Table 8.1: The four possible outcomes for a null hypothesis test.

By running the plagiarism detection program with a threshold t we get the results in Table 8.2. In this table m is the number of null hypotheses tests, m_0 is the number of true null hypotheses, m_1 is the number of false null hypotheses, U is the number of true negatives, V is the number of false positives, T is the number of false negatives, and S is the number of true positives. U, V, T and S are unobservable random variables, while R is an observable random variable. R is observable since this is the number program pairs that the detection program classified as copies. V and S become observable variables after we have manually inspected the R listing pairs.

	non-significant	**significant**	**Total**
true null hypotheses	U	V	m_0
false null hypotheses	T	S	$m_1 = m - m_0$
Total	$m - R$	R	m

Table 8.2: Some random variables associated with the multiple hypotheses test.

After we have manually inspected the R listing pairs, we can calculate the proportion of false positives (Type I errors) among all program pairs classified as copies, as shown below.

$$\mathrm{E}\left[\frac{V}{V+S}\right] = \mathrm{E}\left[\frac{V}{R}\right] \tag{8.2}$$

We want to find a t such that $\mathrm{E}\left[\frac{V}{R}\right] < \alpha$, where α is the significance level for all the the null hypothesis tests. The significance-level is the probability of making a Type I error. If $\mathrm{E}\left[\frac{V}{R}\right] > \alpha$, then we reject the null hypothesis. We then need to try a new threshold t', where $t' > t$. Only an increase of the threshold value can reduce the proportion $\mathrm{E}\left[\frac{V}{R}\right]$.

When we select the significance-level for the null hypotheses, we need to select a level with a good compromise between the numbers of false positives (Type I errors) and false negatives (Type II errors). The selection of a very small significance-level will give a higher threshold value and the test will become more significant. The problem is that we then can get many false negatives, and so have a test with less statistical power.

Two methods that can be used to select the significance level α are false discovery rate (FDR) and familywise error rate (FWER). FDR is used to control the expected proportion of Type I errors in a multiple hypothesis test. It is given by equation 8.2, and one wants to keep it below the significance level α. If we set α equal to 0.05, we accept less than 5 % false positives among all the listing pairs that were classified as copies. FWER is the probability of one or more Type I errors in a multiple hypothesis test. If we set FWER equal to 0.05, then α can be calculated as shown below.

$$\begin{aligned} 1-(1-\alpha)^m &= 0.05 \\ (1-\alpha)^m &= 0.95 \\ \alpha &= 1-\sqrt[m]{0.95} \end{aligned} \tag{8.3}$$

In this equation 0.05 is the probability of not making any Type I error in any of the m null hypothesis tests. If $m = 1000$, then α is 0.0005. We can see that FWER is much more conservative than FDR with respect to the proportion of Type I errors in a multiple hypothesis test. With FWER we would normally estimate a higher threshold than when we estimate the threshold with FDR, but this depends of course on the significance level that we use.

A FWER equal to 0.05 would result in a high significance for the test, but the test would have less statistical power. In our case it would be better to use FDR with a significance-level of 0.05 or higher. Then we would get a good compromise between the significance and the statistical power of the test. There are different methods that can be used for estimateing a threshold t for a set of program listings P with FDR. One simple method that can be used is to select lower and lower threshold values, until $\mathrm{E}\left[\frac{V}{R}\right]$ is as close to α as possible. For each new threshold t that we select, we only need to manually classify the new listing pairs that were added to the set of possible copies by t. If $\mathrm{E}\left[\frac{V}{R}\right] > \alpha$ we would need to increase the threshold value.

Chapter 9

Conclusion and further work

In this thesis I set out to develop new similarity measures for Joly focused on quantifying the structural similarity between ASTs of Java listings. In order to achieve this I have modified the standard AST representation to ease the comparison between trees, assessed the impact of common cheating strategies on ASTs, and implemented two new algorithms, ASTSIM-NW and ASTSIM-LCS, with respect to the cheating strategies which potentially have the greatest impact on the trees. I have documented that ASTSIM-NW and ASTSIM-LCS on small test sets performed better than the current algorithm in Joly in terms of false positives and false negatives. Even though more extensive testing needs to be done before any firm conclusion can be drawn, the results so far are nevertheless quite promising. However, I have also documented that the computing time demands of ASTSIM-NW and ASTSIM-LCS are unacceptable when comparing a large number of program listings. One way to resolve this is to preprocess the listing set with a faster algorithm first to remove listing pairs that cannot be copies of each other, and then test the remaining pairs with ASTSIM-NW and ASTSIM-LCS. Such a faster algorithm can, for instance, remove those listing pairs where there is some predefined difference between the node counts of the ASTs. Another concern, which was given little attention during the development of the new measures, is that my approach is dependent upon code that can be parsed in order to build the ASTs. We often experience that students submit syntactically incorrect code to Joly, and a simple solution to this problem is to make specific use of algorithms that are insensitive to syntax errors in these cases.

Even though my suggested algorithms may not qualify to supplement the existing one in Joly in the end, I have still succeeded in showing how abstract syntax trees, which is a concept from the field of computer science, can be combined with statistical methods to quantify structural similarity. I also succeeded in showing that it is possible to consider the ASTs as unordered trees, when assessing the similarity between them. In Sager et al. (2006) similarity measures for ordered trees were used, while I have shown that the Top-Down Unordered Maximum Common Subtree Isomorphism algorithm can be used in combination with either the Needleman-Wunsch or the Longest Common Subsequence algorithm for ASTs. The advantage of this approach is that my measures are not so vulnerable against reordering of independent code as the measures in Sager et al. (2006) are.

9.1 Further work

Both ASTSIM-NW and ASTSIM-LCS can get unordered matching of statements and local variable declarations within constructor bodies. The grammar for the constructor body is a bit different from the grammar of the method body, and this makes it harder to compare two

constructor bodies than two method bodies. A suggestion here is to change the grammar for the parser that generates the ASTs for my algorithms, so that the grammar for the constructor body becomes more similar to the grammar of the method body. That would ease the comparison of constructor bodies.

Both algorithms are quite demanding concerning computing time and the possibility of reducing the tree size further without throwing out significant information should be investigated.

In software projects we try to minimize the amount of duplicated code. My algorithms could perhaps, after some modifications, be used to search for code which is similar to the code that the programmar is writting. In this way the programmar can be told that he is writting duplicated code. The algorithms can also might be used to find differences between different versions of a program. Then we can see how the program evolve.

Bibliography

[Butterfield,1991] F. Butterfield. *Scandal over cheating at MIT stirs debate on limits on teamwork*, The New York Times, May 22 1991

[Cormen et al.,2001] T. Cormen, C. Leiserson, R. Rivest and C. Stein. *Introduction to Algorithms*, The MIT Press, 2^{nd} edition, 2001

[Evensen,2007] M. Evensen. *"Joly" fanger eksamensfuskerne*, Aftenposten, June 17 2007

[Fawcett,2004] T. Fawcett. *ROC Graphs: Notes and Practical Considerations for Researchers*, March 16 2004

[Gitchell and Tran,1999] D. Gitchell and N. Tran. *Sim: A utility for detecting similarity in computer programs*, 1999

[Kielland,2006] C. Kielland. *Metoder for likhetsvurdering av innleverte obligatoriske oppgaver i Java*, Master thesis, University of Oslo, 2006

[Kohavi and Provost,1998] R. Kohavi and F. Provost. *Glossary of terms*, Machine Learning, 30 1998

[Louden,1997] K. Louden. *Compiler Construction: Principles and Practice*, Course Technology, 1997

[Mehlhorn and Näher,1999] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*, Cambridge University Press, 1999

[Needleman and Wunsch,1970] S. Needleman and C. Wunsch. *A general method applicable to the search for similarities in the amino acid sequence of two proteins*, Journal of Molecular Biology, 48:3 1970

[Roberts,2002] E. Roberts. *Strategies for promoting academic integrity in CS courses*, 32^{nd} ASEE/IEEE Frontiers in Education Conference, November 6-9 2002

[Sager et al.,2006] T. Sager, A. Bernstein, M. Pinzger and C. Kiefer. *Detecting Similar Java Classes Using Tree Algorithms*, 2006

[Steensen and Vibekk,2006] T. Steensen and H. Vibekk. *DHIS and Joly: two distributed systems under development: design and technology*, Master thesis, University of Oslo, 2006

[Stephens,2001] S. Stephens. *Using metrics to detect plagiarism*, March 2001

[Valiente,2002] G. Valiente. *Algorithms on Trees and Graphs*, Springer, 2002

[Verco and Wise,1996] K. Vecro and M. Wise. *Plagiarism à la Mode: A Comparison of Automated Systems for Detecting Suspected Plagiarism*, The Computer Journal, 39:9 1996

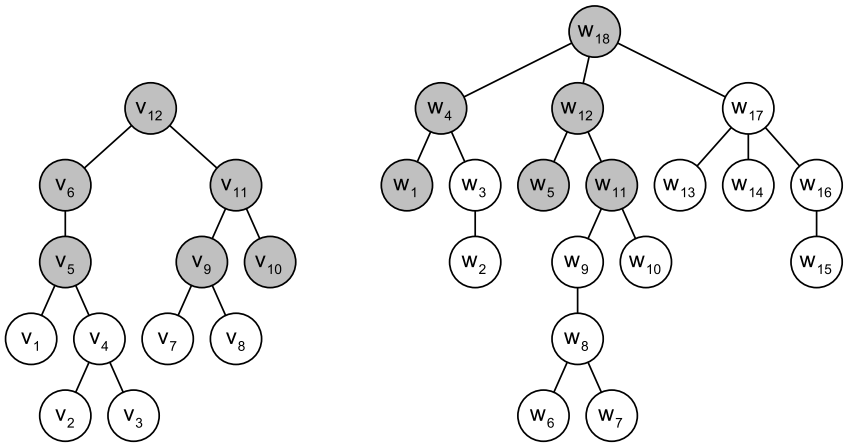
[Whale,1990] G. Whale. *Identification of program similarity in large populations*, The Computer Journal, 33:2 1990

Appendix A

Examples of different maximum common subtrees

This appendix contains examples of the for different maximum common subtree isomorphisms. All the examples are from Valiente (2002), and all the trees are unlabeled.

A.1 Top-down ordered maximum common subtree

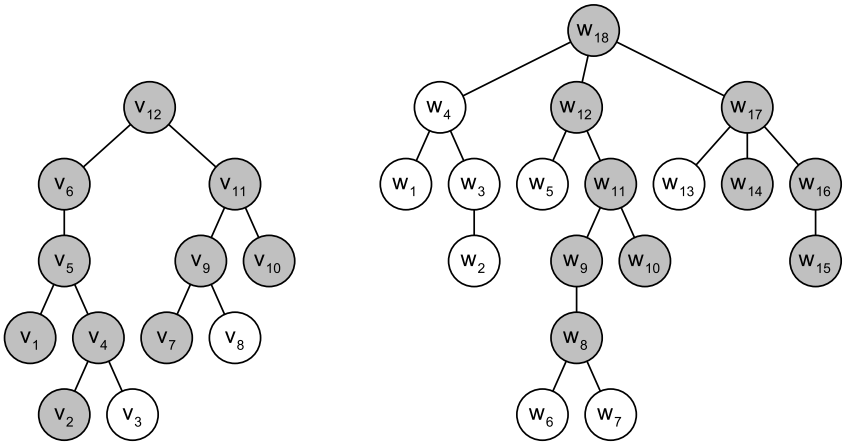


(a) Ordered tree T_1.

(b) Ordered tree T_2.

Figure A.1: An example of a top-down maximum common subtree of the two ordered trees T_1 and T_2. Nodes are numbered according to the order in which they are visited during a postorder traversal. The common subtree of T_1 and T_2 is in gray in both trees.

A.2 Top-down unordered maximum common subtree

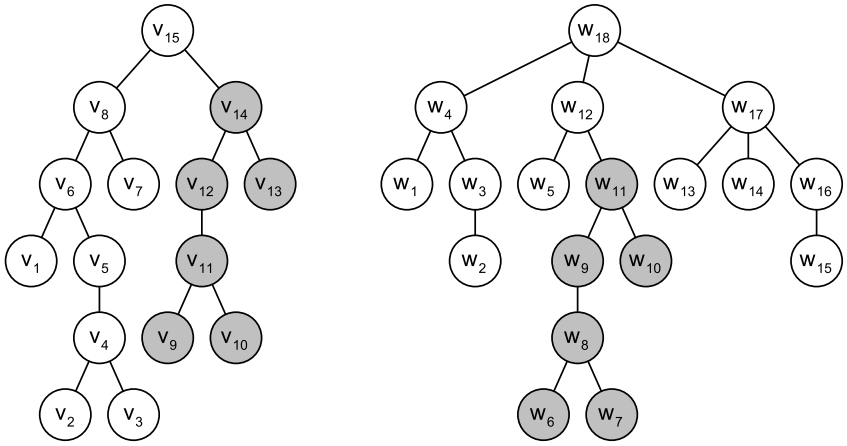


(a) Unordered tree T_1.

(b) Unordered tree T_2.

Figure A.2: An example of a top-down maximum common subtree of the two unordered trees T_1 and T_2. Nodes are numbered according to the order in which they are visited during a postorder traversal. The common subtree of T_1 and T_2 is in gray in both trees.

A.3 Bottom-up ordered maximum common subtree

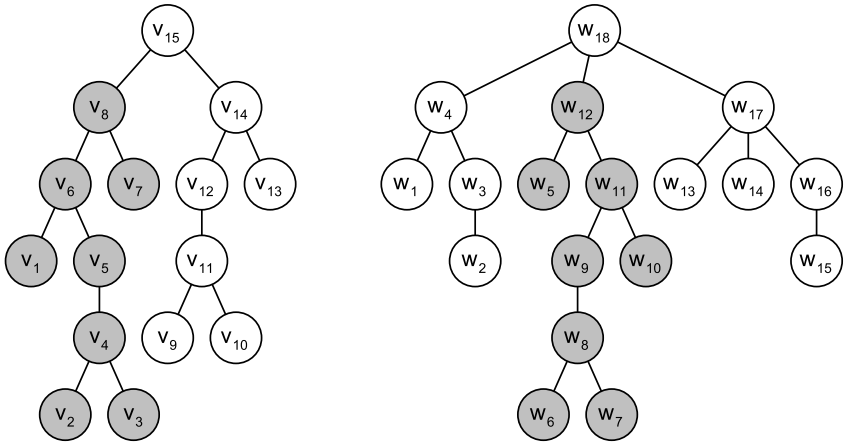


(a) Ordered tree T_1.

(b) Ordered tree T_2.

Figure A.3: An example of a bottom-up maximum common subtree of the two ordered trees T_1 and T_2. Nodes are numbered according to the order in which they are visited during a postorder traversal. The common subtree of T_1 and T_2 is in gray in both trees.

A.4 Bottom-up unordered maximum common subtree



(a) Unordered tree T_1.

(b) Unordered tree T_2.

Figure A.4: An example of a bottom-up maximum common subtree of the two unordered trees T_1 and T_2. Nodes are numbered according to the order in which they are visited during a postorder traversal. The common subtree of T_1 and T_2 is in gray in both trees.

Appendix B

Source code listings

This appendix contains source code listings of P_0 and P_{10} from Section 7.4. All the program listings (P_0 and $P_3 - P_{10}$) from this section can be found at *http://olavsli.at.ifi.uio.no/thesis*.

B.1 The program listing P_0

```
import easyIO.*;

class Oblig {
    In tast = new In();
    Out skjerm = new Out();

    public static void main(String[] args) {
        Oblig oblig = new Oblig();
        oblig.meny();
    }

    void meny(){

      int valg = 0;
        while (valg != 6) {
            skjerm.outln("");

            skjerm.outln("Velg et tall:  ");
            skjerm.outln("");

            skjerm.outln("1:Fakultet ");
            skjerm.outln("2:Adder til oppgitt sum");
            skjerm.outln("3:Trekk et kort    ");
            skjerm.outln("4:BlackJack   ");
            skjerm.outln("5:DNA   ");
            skjerm.outln("6:Avslutt ");

            valg = tast.inInt();

            switch (valg) {
                case 1:
                    fakultet();
                    break;
                case 2:
                    adderTilOppgittSum();
                    break;
                case 3:
                    trekkEtKort();
```

```
                break;
            case 4:
                blackJAck();
                break;
            case 5:
                dNA();
                break;
            case 6:
                break;
            default:
                skjerm.outln("Du har tastet feil. Prøv igjen");
        }

    }
}

void fakultet(){
    skjerm.outln("Tast inn et heltall");
    int n = tast.inInt();

    int fak = 1;

    for (int m = 1; m <= n; m++) {
        fak *= m;
    }

    skjerm.outln(n + "!=" + fak);
}

void adderTilOppgittSum() {
    skjerm.outln("Oppgi et heltall som maksimalverdi:");

    int n = tast.inInt();
    int k;
    int sum = 0;

    for (k = 1; sum <= n; k++) {
        sum += k;
    }

    skjerm.outln("Summen av 1,2,3,...," + (k - 1) + " er " + sum + " ↩
        som er større eller lik maksverdien:" + n);
}

int trekkEtKort() {
    String[] kortfarge = { "Hjerter", "Ruter", "Kløver", "Spar" };
    int type = (int) (4.0 * Math.random());
    skjerm.out(kortfarge[type] + " ");

    String[] kortnr = { "2", "3", "4", "5", "6", "7", "8", "9", "10", ↩
        "Knekt", "Dronning", "Konge", "A" };
    int tall = (int) (13.0 * Math.random());
    skjerm.outln(kortnr[tall]);
    return tall;
}

void blackJAck() {
    int g = trekkEtKort();
    int h = trekkEtKort();
    int verdi1, verdi2, verdi3;
    int sum = 0;
```

```
        if (g == 9 || g == 10 || g == 11 || g == 12) {
            verdi1 = 10;
        }

        else {
            verdi1 = g + 2;
        }

        int poeng = verdi1;

        if (h == 9 || h == 10 || h == 11 || h == 12) {
            verdi2 = 10;
        }

        else {
            verdi2 = h + 2;
        }

        int p = verdi2;

        int b = p + poeng;

        skjerm.outln("Summen av de to første kortene er " + b);

        do {
            skjerm.outln("Vil du trekke enda et kort? ");
            String svar = tast.inWord();

            if (svar.equals("ja")) {
                int o = trekkEtKort();

                if (o == 9 || o == 10 || o == 11 || o == 12) {
                    verdi3 = 10;
                }

                else {
                    verdi3 = o + 2;
                }

                int s = verdi3;
                sum = s + b;

                skjerm.outln("Summen av de kortene er " + sum);

                if (sum > 21) {
                    skjerm.outln("Game over!!!!!!!!!!");
                }
            }

            else {
                skjerm.outln(" Nå avsluttes spillet. Din sum var " + b);
                break;
            }

        } while (sum <= 21);

    }

    void dNA() {
        skjerm.outln("Tast inn DNA sekvensen :");
```

```
        String senseStreng = tast.inLine();

        char[] baser = senseStreng.toCharArray();

        for (int k = baser.length - 1; k >= 0; k--) {
            switch (baser[k]) {
                case 'A':
                    skjerm.out('T');
                    break;
                case 'T':
                    skjerm.out('A');
                    break;
                case 'G':
                    skjerm.out('C');
                    break;
                case 'C':
                    skjerm.out('G');
                    break;
                default:
                    skjerm.outln("Vennligst tast inn kun bokstavene A,T,G ↩
                        og C ");
            }
        }
    }

}
```

Listing B.1: The program listing P_0

B.2 The program listing P_{10}

```
import easyIO.*;

class Oblig10 {
    In tast = new In();
    Out skjerm = new Out();

    public static void main(String[] args) {
        Oblig10 oblig = new Oblig10();
        oblig.meny();
    }

    void meny(){
        int valg = 0;
        skjerm.outln("Velkommen til programmet");

        while (valg != 6) {
            skjerm.outln("");

            skjerm.outln("Velg et tall:  ");
            skjerm.outln("");

            skjerm.outln("1:Fakultet ");
            skjerm.outln("2:Adder til oppgitt sum");
            skjerm.outln("3:Trekk et kort    ");
            skjerm.outln("4:BlackJack   ");
            skjerm.outln("5:DNA   ");
            skjerm.outln("6:Avslutt ");

            valg = tast.inInt();

            if(valg == 1){
                skjerm.outln("Fakultet");
                int b;

                double fak = 1.0;
                skjerm.outln("Tast inn et heltall");
                int n = tast.inInt();

                int m = 1;

                while(m <= n){
                    fak = fak * m;
                    m++;
                }

                skjerm.outln(n + "!=" + fak);
            }
            else if(valg == 2){
                skjerm.outln("Adder til oppgitt sum");
                adderTilOppgittSum();
            }
            else if(valg == 3){
                skjerm.outln("Trekk et kort");
                trekkEtKort();
            }
            else if(valg == 4){
                skjerm.outln("Black Jack");
                blackJAck();
```

```
            }
            else if(valg == 5){
                skjerm.outln("DNA");
                dNA();
            }
            else if(valg == 6){
                skjerm.outln("Programmet avsluttes");
            }
            else {
                skjerm.outln("Du har tastet feil. Prøv igjen");
            }

        }
    }

    void adderTilOppgittSum() {
        int f;

        int k = 1;
        skjerm.outln("Oppgi et heltall som maksimalverdi:");

        int n = tast.inInt();
        double sum = 0.0;

        while(sum <= n){
            sum = sum + k;
            k++;
        }

        skjerm.outln("Summen av 1,2,3,...," + (k - 1) + " er " + sum + " ↩
            som er større eller lik maksverdien:" + n);
    }

    int trekkEtKort() {
        String[] kortnr = { "2", "3", "4", "5", "6", "7", "8", "9", "10", ↩
            "Knekt", "Dronning", "Konge", "A" };

        String[] kortfarge = { "Hjerter", "Ruter", "Kløver", "Spar" };

        int tall = (int) (Math.random() * 13.0);

        int type = (int) (Math.random() * 4.0);

        skjerm.out(kortfarge[type] + " ");

        skjerm.outln(kortnr[tall]);
        return tall;
    }

    void blackJAck() {
        int x;
        int verdi1, verdi2, verdi3;
        int g = trekkEtKort();
        int h = trekkEtKort();

        if (g == 9 || g == 10 || g == 11 || g == 12) {
            verdi1 = 10;
        }

        else {
            verdi1 = 2 + g;
```

```
        }

        if (h == 9 || h == 10 || h == 11 || h == 12) {
            verdi2 = 10;
        }

        else {
            verdi2 = 2 + h;
        }

        int b = verdi1 + verdi2;

        double sum = 0.0;

        skjerm.outln("Summen av de to første kortene er " + b);

        do {
            skjerm.outln("Vil du trekke enda et kort? ");
            String svar = tast.inWord();

            if (svar.equals("ja")) {
                int o = trekkEtKort();

                if (o == 9 || o == 10 || o == 11 || o == 12) {
                    verdi3 = 10;
                }

                else {
                    verdi3 = 2 + o;
                }

                sum = verdi3 + b;

                skjerm.outln("Summen av de kortene er " + sum);

                if (sum > 21) {
                    skjerm.outln("Game over!!!!!!!!!!");
                }
            }

            else {
                skjerm.outln(" Nå avsluttes spillet. Din sum var " + b);
                break;
            }

        } while (sum <= 21);

    }

    void dNA() {
        skjerm.outln("Tast inn DNA sekvensen :");
        String senseStreng = tast.inLine();

        char[] baser = senseStreng.toCharArray();

        int k = baser.length - 1;

        while(k >= 0) {
            switch (baser[k]) {
                case 'A':
                    skjerm.out('T');
```

```
                    break;
                case 'T':
                    skjerm.out('A');
                    break;
                case 'G':
                    skjerm.out('C');
                    break;
                case 'C':
                    skjerm.out('G');
                    break;
                default:
                    skjerm.outln("Vennligst tast inn kun bokstavene A,T,G ↩
                        og C ");
            }

            k--;
        }
    }

}
```

Listing B.2: The program listing P_{10}