Utrecht University

Master's thesis

Plagiarism detection in Haskell programs using call graph matching

Author:
M.L. Kammer
Student number: 0307432

Supervisors:
dr. H.L. Bodlaender
dr. J. Hage

May 2011

To Shanna

"Students must have initiative; they should not be mere imitators.
They must learn to think and act for themselves - and be free."

- César Chávez, peaceful civil rights activist, 1927 - 1993

Abstract

Ongoing research in the area of plagiarism detection at Utrecht University has indicated the need for an automated tool that can help identify possible cases of plagiarism in Haskell programs. There are many modifications that can be applied to the source code of a program in order to hide or mask an attempt at plagiarism. Existing tools like Holmes and MOSS make use of token stream analysis and document fingerprinting techniques to try to see through this, but call graph matching techniques are a relatively unexplored topic in this area. We have developed a tool based on a tree search algorithm that calculates graph edit distance in order to measure call graph similarity. Although calculating graph edit distance is an NP-complete problem and the resulting search tree constructed by our algorithm is exponential in size, we use preprocessing and domain-specific optimizations to be able to use the algorithm in practice. With experiments on fabricated cases of plagiarism and on actual student submissions for a Haskell programming assignment we demonstrate the effectiveness and usefulness of our tool in finding possible cases of plagiarism, in addition to the Holmes tool. In particular, when compared to the techniques used by the Holmes tool it is shown that our tool is less sensitive to many refactoring techniques that are typically used by plagiarists.

Acknowledgements

Firstly, my thanks go out to dr. Hans Bodlaender, who not only provided me with his expert knowledge on graph theory but who also often motivated me with his positive view on things, and on life in general. In our meetings during this project, I would often afterwards walk away with renewed energy and enthusiasm. I would also like to thank dr. Jurriaan Hage, who has extensive experience in the area of software plagiarism and gave me a lot of ideas to work with.

My achievements at Utrecht University would not have been possible without the knowledge I acquired from all professors and lecturers, so I would also like to thank the staff of the Department of Computing Sciences for helping me along the way towards a Master of Science degree.

Finally, I would like to make a special note about Roel and Joke, who are always a source of inspiration and great wisdom to me. This thesis would not be complete otherwise.

Contents

Chapter 1

Introduction

A recent survey by Utrecht University [1] on plagiarism among its students showed that 78% of all students use ideas from the Internet or somewhere else as an example for their own work. Of all students, 67% sometimes express these ideas in their own words and 54% sometimes literally copy these ideas with only minor modifications. These numbers may seem shockingly high, especially when we consider the fact that the survey also showed that among students, only 19% find the copying of texts and presenting it as own work acceptable.

These results show that plagiarism is almost common practice among students, and yet they know it is unacceptable. In contrast, lecturers and reviewers have only ever spoken with 3% of all students about their suspicions of plagiarism and about 1% of all students have ever been sanctioned for it in one form or another.

1.1 What is plagiarism?

Plagiarism as defined in the Oxford English Dictionary [2] is

> "The wrongful appropriation or purloining and publication as one's own, of the ideas, or the expression of the ideas [...] of another."

This can either be copying or quoting an author's work without proper acknowledgement, writing a similar text based on another author's text, or even expressing the same ideas as another author without acknowledging where these ideas really came from.

Perhaps surprisingly, what we negatively define as plagiarism today was common among writers and artists until the 18th century, in order to "imitate the masters" as explained by Lynch [3]. In modern culture, especially in the fields of academia and journalism, plagiarism is considered to be immoral and often reason for severe sanctions. Most universities have a low-tolerance policy concerning plagiarism,

as is obvious from the following excerpt from a regulations document of Utrecht University [4] which concerns students attempting plagiarism for a second time:

> "Exclusion from all forms of examination for a period of 12 months, and the formal advice to leave the study course altogether."

1.1.1 Why plagiarism must be counteracted

The reasons why plagiarism is viewed so negatively include:

- **Monetary reasons.** If an author intends to make a profit out of his published work and someone else steals and publishes the same work without his knowledge, the original author will make less profit.
- **Loss of reputation.** An academic paper or an in-depth article by a journalist can only be significant if it shows innovation and originality. A stolen paper or article deprives the original author of a certain gain in reputation.
- **Unfairness.** If a certain result is unrightfully achieved through plagiarism (for instance, a student achieves a grade by submitting a plagiarized assignment), this would not be fair to others trying to achieve the same result with hard work.

Interestingly, in the same regulations document of Utrecht University mentioned earlier [4] a different definition of plagiarism is given than the more common definition from the Oxford English Dictionary [2]. This definition, also mentioned by Vermeer in his important research on plagiarism detection [5], is as follows:

> "Fraud and plagiarism are defined as actions, or failure to act, on the part of a student, as a result of which proper assessment of his/her knowledge, insight and skills, in full or in part, becomes impossible."

From this definition another reason why plagiarism is viewed negatively becomes apparent: if a student plagiarizes in order to pass an assignment he or she may not learn the skills that would be otherwise required to pass the assignment, which could result in a diploma being wrongly achieved. This is probably the most important reason for a university to prevent plagiarism among students and also why Utrecht University defines plagiarism as mentioned above.

All these reasons not only point out the need to counteract plagiarism but they also justify all possible means and tools in order to find cases of plagiarism. This has opened up interesting research areas in computing science, both in string analysis and graph analysis algorithms, where searching for cases of plagiarism in some tangible medium (e.g. pieces of text written in a natural language, or software source code written in a programming language) may be made easier by means of automated tools.

1.1.2 Plagiarism in software

A number of ways exist in which plagiarism can occur, the most common being plagiarism of a piece of text written in a natural language: copying some text off the Internet is easy because of the enormous abundance of available texts. In software technology, another obvious way of committing plagiarism is by copying someone else's source code for a computer program. This can be categorized into two areas:

- by a software company in order to sell a stolen software product as its own, or
- by a student in technical computer science education, in order to pass a practical assignment in which a computer program must be written.

Although it would certainly be helpful in a law suit to be able to employ some tool when determining whether or not a software company has actually stolen another company's product, it still is only a single comparison that must be made and the comparison must be checked by humans anyway in order to determine the outcome of the law suit. This reduces the need for an automated tool. Moreover, there is only a small chance that a software company will actually be formally accused of having committed plagiarism since most source code is compiled into unrecognizable machine code (usually an irreversible operation) and a different user interface can be employed to mask the inner workings of a computer program.

For these reasons, in this thesis we focus primarily on the second way of committing plagiarism - by students for a programming assignment - because it provides a more interesting research area. When comparing assignment submissions, we often have to compare a large number of programs and we can catch frauds at a much more frequent rate. For the rest of this thesis we will use the term plagiarism where we generally mean plagiarism of source code for a programming assignment, unless otherwise stated.

1.1.3 Plagiarism in natural language

As stated earlier, plagiarism of a text in a natural language is the most common form in which plagiarism occurs because of the availability of texts on the Internet. Research towards plagiarism detection in this area is also extensive, but that is outside the scope of this thesis. We will briefly point out the fundamental differences between plagiarism detection in natural language and in software source code, so that it is clear why techniques that work well on natural language are not sufficient when applied to software code.

- Software code has a much smaller **alphabet size**. A programming language typically only has a limited set of keywords, constant names and syntactic markers (in addition to identifier names), whereas any natural language has

a very large set of allowed words. This makes plagiarism detection harder in software code, because a much smaller alphabet increases the chance that a found match between words or markers is a coincidence.

- In natural language, the **ordering of words** is very important. Most languages allow for only slight variations in word ordering in a given sentence. In contrast, many aspects of computer programs are unordered. For instance, the order in which functions are defined matters little or nothing in most programming languages, including Haskell. This fundamental difference allows for a plagiarist to apply many more changes to a computer program in order to hide his attempt at plagiarism, making the detection process much harder. For instance, many matching problems in computing science have a much higher complexity class when dealing with unordered data. Error-tolerant graph matching, as we will see in the following chapter, is NP-complete (see Garey and Johnson, [6]) whereas the string edit distance problem (see Wagner and Fischer, [7]) can be solved in linear time.

- When searching for sources from where a plagiarist copied the original work, the **availability of existing works** becomes important. For texts written in a natural language the Internet can serve as a huge database of existing works to which given texts can be compared, in order to find possible cases of plagiarism. With a few exceptions such as Hoogle [8], this is much harder for software code because source code that meets very specific requirements (for instance, for a programming assignment) is not usually available online.

We can conclude that although some ideas from the area of plagiarism detection in natural language might be useful to us, the majority of techniques and tools that exist in this area will not work for plagiarism detection in software code.

1.2 Plagiarism detection and graph matching

Many tools exist for detecting possible cases of plagiarism. Most of these are specifically designed for dealing with general-purpose programming languages like C# and Java. They use a wide variety of techniques for detection, and a few of these tools have been proven to work very well. For functional programming languages like Haskell only a few successful tools exist, and they all use more or less the same general ideas from the field of string and token sequence analysis for detection.

A more advanced algorithmic technique that has been used extensively in the fields of pattern matching and object recognition is **graph matching**, examples of which are shown in Figures 1.1 and 1.2. Graph matching techniques are based on comparisons between graph representations of objects.

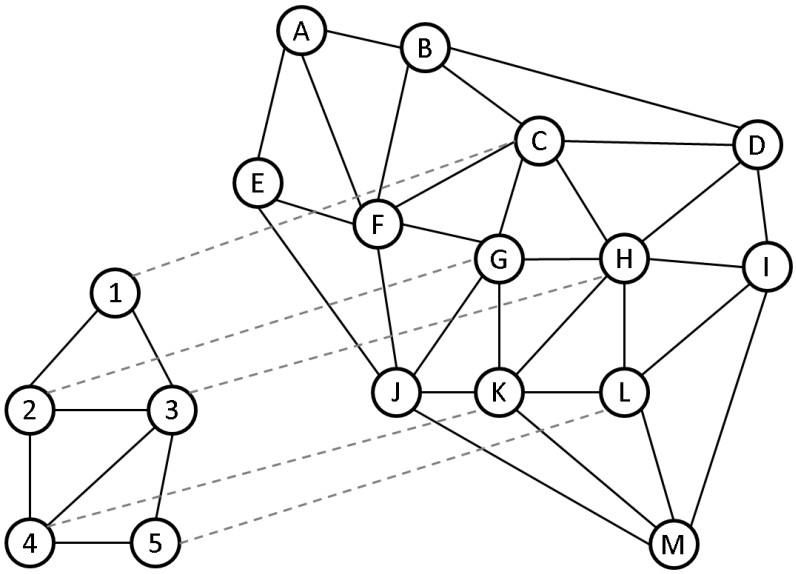

Figure 1.1: *An example of graph matching.*

For plagiarism detection on general-purpose programming languages, a few tools have successfully applied graph matching techniques in their detection process. These tools construct graphs like in Figure 1.1 to represent the content of software code and its internal dependencies. However, for functional programming languages this is still a relatively unexplored field.

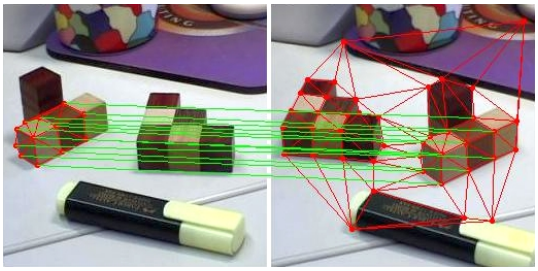

Figure 1.2: *An example of object recognition.*

Holmes is a plagiarism detection tool for the functional programming language Haskell, created by Vermeer and Hage [5] also for Utrecht University. This tool mainly uses string and token sequence analysis techniques. The work in this thesis can be viewed as an addition to the research done by Vermeer in the sense that we will now explore graph matching techniques in order to find out if they can successfully be used for plagiarism detection in Haskell programs. This would provide a valuable addition to the detection methods already applied by Holmes.

1.3 Outline

Our goal in this thesis is to build a **plagiarism detection tool** for the Haskell language that uses graph matching techniques, and to prove that it is effective in practice. Each following chapter describes a part of this process. In Chapter 2, we will begin by describing existing tools for plagiarism detection and the techniques they use in more detail. We will also describe the ways in which frauds try to hide their attempt at plagiarism, and how these tools cope with this.

Chapter 3 is entirely dedicated to graph matching problems, and we also describe several algorithmic techniques to solve them. This is followed by the implementation details for our own plagiarism detection tool in Chapter 4, which uses graph matching techniques based on some of the ideas in Chapters 2 and 3. In Chapter 5, some experiments (and their results) are described in order to see how well our tool performs on fabricated cases of plagiarism for the sake of validation, as well as experiments on real submissions for Haskell programming assignments.

A short summary of the work in this thesis is given in Chapter 6, in which we will also draw some conclusions from our experiment results given in Chapter 5. The last part of this thesis consists of a list of future work in Chapter 7, in which we describe which observations and ideas might be worth further investigating after the work in this thesis, because research should never be finished.

Chapter 2

Plagiarism of software code

In Chapter 1, we have already established a few notions about plagiarism in general and we elaborated on the possible reasons for committing it. In this chapter, we will describe a number of ways in which people try to mask their attempt at plagiarism of software code. This is followed by a description of how a few popular plagiarism detection tools work. Finally, we will describe what we require of a tool to help us search for cases of plagiarism. We use this as a basis for developing and testing such a tool in subsequent chapters.

2.1 Masking plagiarism

As can be read in a technical report by Jurriaan Hage [9], who has done a lot of research in the area of software plagiarism for Utrecht University, there are a few reasons why a student having to do a programming assignment would want to resort to plagiarism:

- not having enough time to write the required computer program.
- not having the skills to write a computer program with the required functionality.

or a combination of these.

Most students are aware that it is likely their submissions will be checked for plagiarism, either with automated tools or by randomly comparing a number of submissions by hand if a good tool is not available. Therefore it is also likely that students will try to mask their attempts at plagiarism, and they have a number of methods at their disposal with which they can accomplish this. However, as Hage also points out in [9], a few of these methods are useless because a reviewer manually looking at source code modified in such ways will immediately suspect that something is wrong. These include:

Code obfuscating which leaves the program functionally the same but greatly changes the appearance of its source code. Common code obfuscation techniques include playing around with whitespace and using strange symbols as identifiers. For Haskell programs, even more advanced methods are possible like rewriting keywords (such as `let`) as lambdas.

Redefining predefined functions with a different name and then calling the redefined function instead of the original function. For example, in a Haskell program one could copy a function from the Prelude (which is what the set of all predefined functions is called) and give it a different name.

2.1.1 Renaming and refactoring

An obvious way to hide the fact that a function was copied from someone else is to rename the function. Renaming a function is just one example of many refactoring techniques that exist. **Refactoring** is an umbrella term for a collection of *structural changes* that can be applied to a program's source code *without changing its semantics*, i.e. without changing it external functional behavior. Of course, these techniques provide an excellent way to mask plagiarism. Below is an example showing various refactoring methods being applied sequentially in order to transform a Haskell program (the `sum` and `foldr` functions are actually part of the predefined Haskell Prelude, but only used as example functions here).

1. Original Haskell program that calculates the sum of the list `[1,2,3,4]`:

```
sum []    = 0
sum (h:t) = (+) h (sum t)
main = sum [1..4]
```

2. Function `sum` renamed to `foldr`:

```
foldr []    = 0
foldr (h:t) = (+) h (foldr t)
main = foldr [1..4]
```

3. Generalized over 0 in the nil-case:

```
foldr n []    = n
foldr n (h:t) = (+) h (foldr n t)
main = foldr 0 [1..4]
```

4. Generalized function applied in the recursive case:

```
foldr f n []    = n
foldr f n (h:t) = f h (foldr f n t)
main = foldr (+) 0 [1..4]
```

5. New expression `total` introduced (which is actually the old `sum` function):

```
foldr f n []    = n
foldr f n (h:t) = f h (foldr f n t)
main = total [1..4]
  where
    total = foldr (+) 0
```

6. Function `total` moved to top-level (since it might be reusable):

```
foldr f n []    = n
foldr f n (h:t) = f h (foldr f n t)
main = total [1..4]
total = foldr (+) 0
```

Even the simple example above already results in source code that looks quite different from the original program, but it still produces exactly the same output. Many more modifications can be applied to the source code, such as:

- Change variable names
- Reorder function declarations
- Move functions only called by one function to the local scope of the calling function
- Introduce unused functions
- Remove unnecessary functions
- Introduce a function that is called by every other function (for instance, a debug function)
- Introduce a function that calls all other functions (for instance, a unit test function)

Especially the last two modifications are notorious: they look innocent to a reviewer but they can cause problems for plagiarism detection tools because they modify a large part of the source code.

2.1.2 Refactoring tools

Of all the methods for masking plagiarism, a student will typically only apply a few. Recall that a student who plagiarizes does not have enough time or skill to write the required program. The same reasons apply to the process of masking the plagiarism attempt: the student will probably not put a lot of effort into this, because otherwise he could have written the program himself with same amount of effort and he would not have bothered with resorting to plagiarism in the first place. However, the existence of automated refactoring tools such as HaRe [10] makes life easier for frauds and harder for plagiarism detection tools. These tools have the functionality to apply refactoring in a user-friendly way without requiring the user to have an extensive knowledge of the program or programming language in question. Many of these tools are freely available on the Internet and some source code editors even contain refactoring functions or a refactoring tool as a plugin. When building a tool for detecting plagiarism one must keep in mind that refactoring will often have been applied and that the detection tool must see through this.

2.2 Detecting plagiarism

First, it should be noted that completely automated detection of plagiarism as such is impossible. Automated tools can be used to search for similarity among objects, but it is up to the user to decide what degree of similarity constitutes a match and what doesn't. Even when a match is found, it is still only an indication that it may be a case of plagiarism. The decision whether or not it is actually a case of plagiarism not only depends on local regulations and guidelines but also always has an element of subjectivity, because like any question of guilt in a law system it must have a human answer. This is so because it can never be guaranteed that an automated tool does not generate any *false positives*: indications of plagiarism where there is in fact no plagiarism at all.

This subtle distinction between a **similarity search** and **plagiarism detection** is not always so clear in the literature. In fact, even in this thesis we often use the term *plagiarism* where actually a *suspected case of plagiarism* is meant, for the sake of conciseness. Suffice it to say that when developing a tool for detecting plagiarism we must always keep in mind that it is merely an aid in finding possible cases of plagiarism and that a human decision must always follow after running the tool.

2.2.1 Requirements of an automated tool

This brings us to the most important requirements of a plagiarism detection tool. A person must always check a suspected case of plagiarism by hand after the tool has been run, but it would be infeasible to manually cross-compare all submissions

because it would simply take too much time. This is where an automated tool comes in: the tool should eliminate all cases that are definitely not plagiarism and give an overview of which suspected cases remain, all in a reasonable amount of time. Also, an important side effect of creating such a tool and spreading knowledge of its existence among students is that students are less inclined to resort to plagiarism if they are not able to complete an assignment otherwise, because they are more fearful their attempted fraud will be discovered.

Scalability

More formally speaking, the program must be **scalable**. It must be able to compare a large number of submissions not only with each other but also cross-compare them with submissions from previous years (if the same assignment was given). The number of comparisons that must be made is $n * (n - 1)/2$, where n is the total number of submissions.

Fortunately, the time constraint on this is not too strict because grading the submissions and running the tool on them can be done concurrently: each submission has to be reviewed by hand anyway in order to grade it. This usually takes at least a few days so it doesn't really make an important difference whether the tool completes its work in ten minutes or in a day. However, we do require that the tool takes a most one day, to avoid any scheduling problems that might arise among the reviewers of the assignment.

False negatives and false positives

Another obvious requirement is that the program must at least mark all actual cases of plagiarism as suspects. The situation where a **false negative** (a case of plagiarism that is not marked by our tool as such) that slips through our fingers, so to speak, must be prevented at all costs because this seriously threatens the validity of grades handed out for an assignment. The tool must also minimize the number of **false positives**: cases that are marked by our tool as suspects but are not really plagiarism at all. These situations are a lot less serious errors because the worst that can happen is a reviewer having to check more suspected cases of plagiarism by hand, which only costs more time. If there happen to be too many false positives, the settings of the tool might also need some adjustment.

Often, a balance between minimizing the number of false negatives and minimizing the number of false positives must be sought when configuring an automated tool. For instance, a lower *threshold* (i.e., a required degree of similarity) for identifying similar programs as suspects may lead to more frauds being caught (less false negatives) but also to more legitimate work being wrongly marked as suspicious

(more false positives). Ideally, the tool must work in such a way that it can be configured to have a low enough threshold to eliminate all false negatives but still have a minimum number of false positives.

See through refactoring

One of the most important properties of a plagiarism detection tool is how well it deals with refactored code, or code in which an attempt at plagiarism has been otherwise masked. Typically, the success rate of a tool will be different for different types of plagiarism masking that are applied. Ideally, the tool should be successful in a general sense (with no weak vulnerabilities for specific techniques of masking plagiarism) and even a combination of methods for masking plagiarism should not have too great an impact on the success rate.

Code templates

Often when a programming assignment is handed out, a piece of existing source code is handed out with it. This is called **template code** and students must expand the given source code in order to complete the assignment. Functions predefined in template code are called **template functions**. It is important to realize that these functions should be disregarded by a plagiarism detection tool, because all submissions share these functions. Otherwise, the tool would detect far too many suspected cases of plagiarism, while it is perfectly legal and often obligated for students to have identical versions of these template functions. The same notion applies to functions defined in the Haskell prelude, because these are also functions that will be identical among submissions.

Other useful properties of plagiarism detection tools include a user-friendly way of feeding submissions to it and showing the similarity results in an ordered, comprehensive way. Also, it should be easily configurable in order to get optimal results given the size and complexity of a programming assignment.

2.2.2 MOSS and other existing tools

Many tools for detecting plagiarism in software code have been developed over the years. A widely used tool is MOSS [11], already developed in 1994 at Berkeley. A great advantage of MOSS over other plagiarism detection tools is that MOSS also appears to handle functional languages like Haskell well, in addition to only object-oriented languages like Java and C#. Other plagiarism detection tools are either not freely available, rarely used or not suited to Haskell programs at all.

MOSS stands for Measure Of Software Similarity and has been freely available for use via the Internet since 1997. The tool uses *winnowing* [12], a local document fingerprinting technique. This technique uses the notion of k-grams: a k-gram is a substring of length k taken from a text. A sequence of hashes is calculated for all possible k-grams in a text, then a subset of these hashes is taken as the document's fingerprints. Document fingerprinting is perhaps best explained visually with the example below.

1. The original text from a document.

   ```
   The quick brown fox jumps over the lazy dog.
   ```

2. The same text but with irrelevant features like spaces removed.

   ```
   thequickbrownfoxjumpsoverthelazydog
   ```

3. The sequence of all 5-grams derived from the text.

   ```
   thequ hequi equic quick uickb ickbr ckbro kbrow brown
   rownf ownfo wnfox nfoxj foxju oxjum xjump jumps umpso
   mpsov psove sover overt verth erthe rthel thela helaz
   elazy lazyd azydo zydog
   ```

4. Hypothetical sequence of hashes of the 5-grams.

   ```
   43 56 88 97 23 15 75 42 40 69 77
   17 50 11 39 45 31 87 83 46 53 72
   84 57 40 98 51 62 47 55 38
   ```

5. A subset of hashes, by accepting a hash number h if $h \ mod \ 4 = 0$.

   ```
   56 88 40 72 84 40
   ```

Of course, the last step in which certain hashes are selected to form the fingerprints determines the quality of the comparison between documents. Vital information may not be included in the comparison because the corresponding hashes were not selected. This is where winnowing comes in (described in more detail by Schleimer et al. in [12]). The basic idea is that a minimum word length for detection, say t, is chosen (for instance, $t = 8$ because we want to be certain that words of length at least 8 are detected). Then, we make sure that we choose a hash value from each *window* of hashes of length $w = t - k + 1$, by selecting the minimum hash value in that window and only selecting a particular hash once. The last step in the scheme above then becomes:

5. Windows of hashes (window size $w = 4$).

```
(43 56 88 97) (56 88 97 23) (88 97 23 15) (97 23 15 75)
(23 15 75 42) (15 75 42 40) (75 42 40 69) (42 40 69 77)
(40 69 77 17) (69 77 17 50) (77 17 50 11) (17 50 11 39)
(50 11 39 45) (11 39 45 31) (39 45 31 87) (45 31 87 83)
(31 87 83 46) (87 83 46 53) (83 46 53 72) (46 53 72 84)
(53 72 84 57) (72 84 57 40) (84 57 40 98) (57 40 98 51)
(40 98 51 62) (98 51 62 47) (51 62 47 55) (62 47 55 38)
```

6. Selected fingerprints.

```
43 23 15 40 17 11 31 46 53 40 47 38
```

This procedure still selects only a reasonably sized subset of hash values but guarantees that they are uniformly distributed in a way such that words of length $> t$ will always be found when matching document fingerprints. The authors of [12] (and creators of MOSS), Schleimer et al., show in greater detail why this procedure works well in practice.

Marble

As stated earlier, in recent years Jurriaan Hage has done a lot of work towards plagiarism detection at Utrecht University. A technical report appeared in 2006 about a tool he created, called Marble [9]. Marble was designed for plagiarism detection in Java programs and a few other object-oriented languages as well. The technique being used is extensive normalization followed by lexical analysis with the Unix `diff`. Following experiments in which Hage applied this technique to Haskell programs were not successful.

Domain-specific knowledge

Studying the techniques used in MOSS, it is clear why the tool is successful for a broad range of programming languages because it does not rely on any domain knowledge about specific languages. One might argue that adding domain-specific knowledge improves a tool's success because the tool then has the knowledge which specific constructs in the source code are important to compare and what is less important. In other words, the tool "knows what to look for".

In contrast, when dealing with plagiarists that also know which language constructs are important the success rate of a tool might suffer from domain-specific knowledge instead of having a higher success rate. Consider for example the Haskell

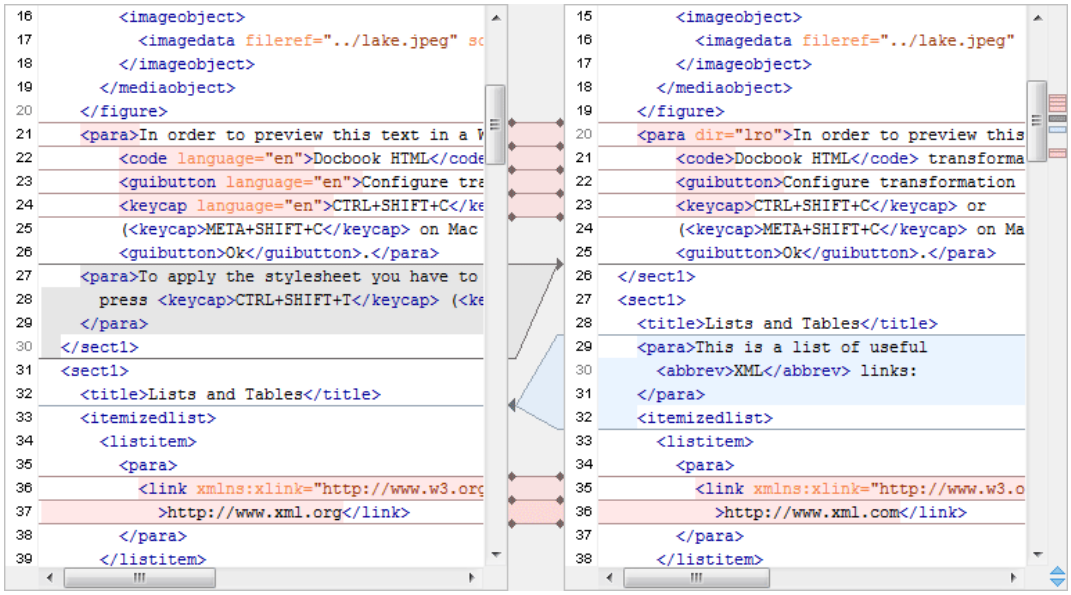

Figure 2.1: *Example of a diff viewer that points out added, removed or modified parts of a text. A similar technique is used in Marble, and in many plagiarism detection tools for texts in a natural language (see also Section 1.1.3).*

language and a hypothetical tool that deems type information of functions very important in its similarity search. A student who plagiarizes may cause problems for this tool when, for instance, he introduces new parameters in function calls (like in the third refactoring step in the example of Section 2.1.1).

What we can conclude from this is that it is not always justified to assume that domain-specific knowledge increases the success rate of a plagiarism detection tool. In fact, it would be more appropriate to say that a tool that uses domain-specific knowledge is more vulnerable to plagiarism attempts from frauds with tool-specific knowledge!

2.2.3 Holmes

Intrigued by both Marble's lack of success when coping with functional languages like Haskell and by the open question whether or not a Haskell-specific tool would be more successful than a general-purpose tool like MOSS, further research into plagiarism detection in Haskell programs was done by Vermeer and Hage in 2010 [5]. This resulted in a tool called Holmes. Holmes is named after the famous fictional detective Sherlock Holmes [13] and consists of two parts: a pre-processor (Sherlock) and a comparison part (Holmes).

The first part, **Sherlock**, is built on top of the Helium parser. Helium [14] is both a (large) subset of the Haskell language and a compiler specifically designed for teaching Haskell, developed by Utrecht University. Much more useful and specific error messages can be given by Helium so that students learn more quickly. Sherlock uses the Helium parser to extract what the authors of [5] call *abstractions* of program submissions: normalized versions of the source code with irrelevant features removed such as comments, unused functions (dead code) and template functions. These features are filtered out to reduce problem complexity and to improve the similarity search in the second phase.

The second part, **Holmes**, takes all information stored by the pre-processor and performs different types of comparisons between submissions. Four types of analysis that are carried out can be distinguished:

- **Literal string analysis** in the forms of approximate string matching on all literal strings that occur in the compared programs, and on all comment sections in the compared programs.
- **Token stream analysis** where the source code is transformed into a normalized token stream in which whitespace is removed, literal strings and identifier names are replaced by symbols, and language specific symbols like brackets are preserved. Both token streams in which the function definitions were first ordered and streams in which the functions were not ordered are compared among submissions using the Haskell diff library, similar to the Unix `diff`.
- **Call graph metrics**: static call graphs are constructed for only the top-level functions, where each graph node represents a function and each edge represents one function calling another function. For each type of node degree (in-degree, out-degree and total degree) the minimum, maximum and average is calculated and compared among submissions. Also the *diameter* of each graph is calculated and compared, which is the length of the longest path among all shortest paths between nodes.
- **Document fingerprint matching** and winnowing are implemented in very much the same way as in MOSS [12], using 25-grams for hashing and a window size of 5.

The authors of [5] implemented the first three methods in order to find out what works well and what does not. The methods are very different as to make sure that each type of analysis (literal, structural and semantic analysis) is properly represented. The fourth method, fingerprint matching, was implemented in order to see how well the first three methods would compare to an existing tool such as MOSS. Research and experimentation showed that of all techniques employed by Holmes,

only token stream analysis and fingerprint matching are reasonably successful when applied to test data (a Haskell program 'plagiarized' manually in different ways) and real-life data (submissions for an actual programming assignment at Utrecht University).

2.2.4 Call graph matching

With the abstract data gathered by Sherlock, even different types of comparisons than the ones mentioned in the previous section can be implemented. From an academic viewpoint, for instance, it was very interesting to find out how well approximate graph matching on the derived call graphs is suited to finding suspected cases of plagiarism. But also for the sake of practicality and completeness it would be useful to have graph matching in addition to the checks performed by Holmes, hopefully to find suspected cases of plagiarism that are not detected even by the more successful techniques of token stream analysis and fingerprint matching. Whereas token stream analysis and fingerprint matching might be vulnerable to certain specific ways of masking an attempt at plagiarism (or a combination of them), graph matching might be better able to deal with all sorts of masking techniques.

Implementing a plagiarism detection tool based on graph matching that could be used in addition to Holmes or that might even perform better than Holmes in a general sense was the main motivation for the work in this thesis. In this sense, our work can be seen as a continuation of the research done by Vermeer and Hage [5]. In the next chapter, we will go further into the theory of graph matching and how we can employ certain techniques when detecting plagiarism.

Chapter 3

The graph matching problem

The *graph matching problem* is the problem of, given two *graphs*, computing their similarity and deciding whether or not the degree of similarity constitutes a *match*. In this chapter we give a few examples of applications in which this problem arises and lay out some basic concepts and foundations. We will also give an overview of methods and algorithmic techniques that can be used in order to compute graph similarity and solve the graph matching problem.

3.1 Graphs

As a short introduction into graph matching theory, we will first briefly recall some basic graph theory for the less experienced reader. Those familiar with graph theory may skip this section altogether.

A **graph** is an abstract model in which objects of some kind are linked together with one-way or two-way connections. The objects are represented by *vertices* (sometimes called *nodes*) and the links between them are called *edges* (or sometimes *arcs*). More formally speaking,

> A **graph** G is a pair $G = (V, E)$ consisting of a set of vertices V and a set of edges E, and each edge $(v, w) \in E$ consists of a pair of vertices $v \in V$ and $w \in V$.

A graph can be *directed* which implies that all edges are one-way links from one vertex to another (each edge $(v, w) \in E$ is an ordered pair), or *undirected* which only implies that each edge connects two vertices without any particular direction (each edge $(v, w) \in E$ is an unordered pair). A graph that contains a cycle (a path from a vertex to itself) or multiple cycles is *cyclic*, a graph without cycles is *acyclic*. Connected acyclic graphs in which each vertex has at most one incoming edge are *trees*, and a collection of trees is aptly named a *forest*.

Many extensions of this basic definition exist, including graphs in which vertices and/or edges can have various attributes such as weights or labels. Vertices can be used to represent all kinds of objects and their properties, although in many applications all vertices of a graph are usually objects of the same type.

The area of graph theory is vast and contains many complex mathematical problems and families of algorithms to solve them. The most well-known graph problem is that of computing a shortest path between two vertices. Dijkstra's algorithm from [15] is widely known for solving the shortest path problem efficiently and it is still used today in many applications such as route planning in car navigation systems, albeit with extensive modifications and advanced heuristics.

Because of the versatile nature of graphs and the large amount of algorithms that have been developed to deal with graphs, graphs are used in many applications so that practical problems can be solved using existing algorithms that have been proven to be correct and efficient.

3.1.1 Haskell call graphs

Before going further into giving an overview of graph matching methods, it is important to know more about call graphs that represent the semantics of source code, especially for Haskell programs. In this section, we will describe the characteristics of this type of graphs.

A call graph can be constructed to represent the static structure of a Haskell program. This is a *directed, labeled, attributed graph* that can be cyclic. Each edge represents one function calling another function. The graph is labeled because all functions have a name and a module to which they belong. Functions can also have various other attributes such as the number of arguments and their types.

The call graph is cyclic if the Haskell program contains self-recursive functions (such as the `maxL` function in the example Figure 3.1), mutually recursive functions (two functions that call each other) or call cycles of more than two functions. From experience, we know that the first category of cycles (self-recursion and mutual recursion) is very common in Haskell programs but that bigger cycles are quite rare. This makes Haskell call graphs look a lot like directed acyclic graphs, which is fortunate because many efficient algorithms exist for computations on a directed acyclic graph.

When Prelude functions are left out of the graph, the call graphs of some smaller Haskell programs (like in Figure 3.1) become trees or tree-like, reducing the complexity of certain calculations even further. However, this is usually not the case

```
module Main where

main = total [1..4]

total    = foldr (+) 0
productL = foldr (*) 1

maxL [x]    = x
maxL (x:xs) = max x (maxL xs)
```

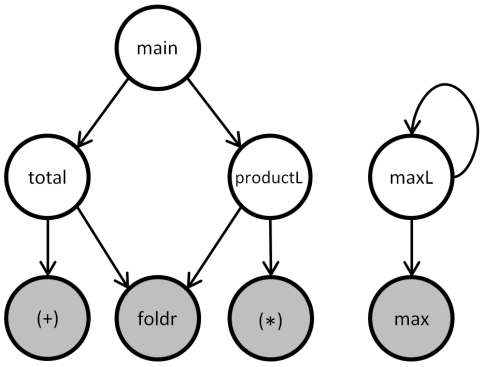

Figure 3.1: *The example Haskell program from Section 2.1.1 but expanded a little (left) and its corresponding call graph (right). Prelude functions are shaded in gray.*

in a standard Haskell program because it almost always contains functions that are called from more than one other function.

3.2 Applications

As mentioned at the beginning of this chapter, the *graph matching problem* is the problem of, given two graphs, computing their similarity and deciding whether or not the degree of similarity constitutes a *match*. There are various subfields in science and engineering in which the graph matching problem is of significant importance. Because graphs are such a powerful tool for modeling structured objects, the problem of computing graph similarity arises often when measuring object similarity. Applications that require measuring object similarity include:

Chemical structure analysis One of the earliest real-world applications of graph matching is described in detail by Rouvray and Balaban in [16]: finding similar molecules.

Biometric identification An application that is especially important in the law enforcement sector is the problem of finding fingerprints that are similar to a fingerprint that was found at a crime scene. Neuhaus and Bunke [18] describe a way to represent fingerprints as attributed graphs and they describe procedures for matching these attributed graphs. Other forms of biometric identification can be carried out in similar ways.

Computer vision Examples of three-dimensional object recognition in an environment that requires robot or computer vision are given in [19] and [20], both papers by Wong.

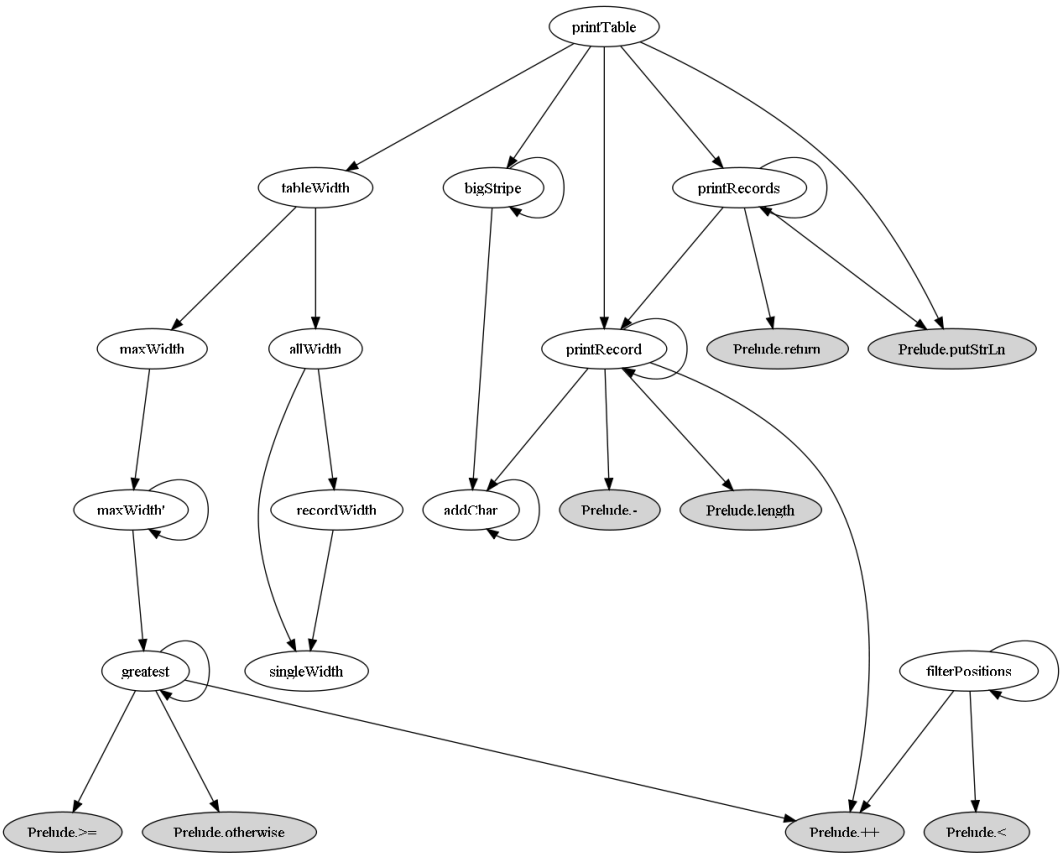

Figure 3.2: *Call graph of a real Haskell program as outputted by Sherlock. Note the common occurrence of self-recursion, and the tree-like appearance when Prelude functions are left out of the graph.*

In addition to the listed examples, many more applications of graph matching have been explored in the last three decades. Bunke [21] gives a brief overview of many applications described in the literature, and Conte et al. [22] give an extensive overview of various applications in pattern recognition as well as a more technical overview and classification of graph matching algorithms that can be applied to them.

3.2.1 Plagiarism detection

Plagiarism detection in software code might seem different from the aforementioned applications in the fact that the distortions to the source code are intentionally applied by humans (or a transformation tool), whereas distortions to structures in other applications are the result of some random or natural process. However, the resulting problem remains the same: finding similar structures while taking into account *any* type of distortion. Furthermore, research has shown that the

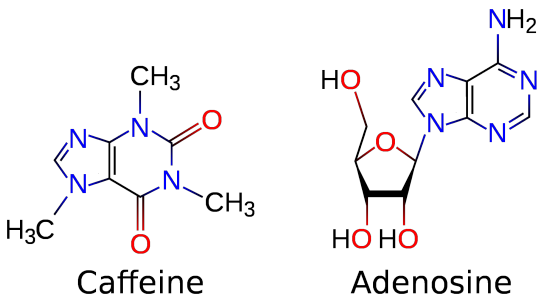

Figure 3.3: *Caffeine counteracts the effects of adenosine by binding itself to adenosine receptors on brain cells, which is possible because of the structural similarity between the molecules (from [17]). This temporarily reduces the sensation of being tired.*

same general techniques and algorithms for graph matching on graphs with natural distortions are equally effective when applied to graphs with artificial distortions. For instance, Bruschi et al. [23] and Krügel et al. [24] successfully applied graph matching techniques in order to find malware that tries to escape detection by mutating its own source code. Also, plagiarism detection in Java programs by graph matching has been researched by Liu et al. [25] resulting in a practical tool (GPlag) to find cases of plagiarism.

3.3 Basic concepts and theory

There are two main groups of graph matching methods: *exact matching* and *inexact (or error-tolerant) matching*. Exact matching methods require a strict correspondence between certain properties of the two graphs being compared, while inexact matching methods allow a degree of error or distortion.

3.3.1 Graph isomorphism

An important concept in exact graph matching is *graph isomorphism*: two graphs are *isomorphic* if two nodes in the first graph that are linked by an edge are also linked by an edge in the second graph, and the other way around. More formally speaking, two graphs G and H are isomorphic if there exists a bijective mapping f from the vertices of G to the vertices of H such that for any two vertices v and w in G there exists an edge $v \to w$ if and only if there exists an edge $f(v) \to f(w)$ in H. *Subgraph isomorphism* is the problem of finding out whether or not a subgraph of graph G exists that is isomorphic to graph H.

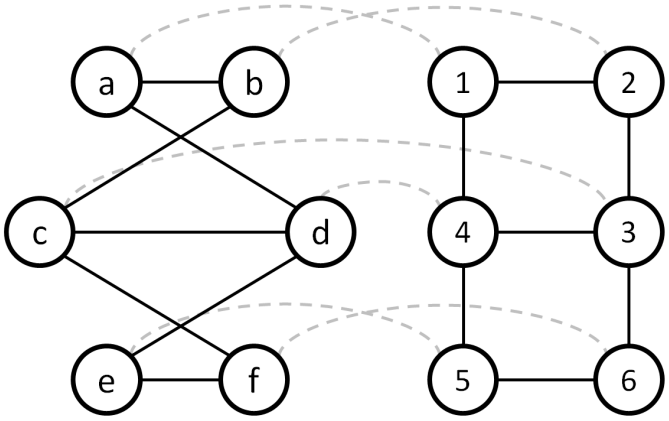

Figure 3.4: *Two undirected graphs that are isomorphic if vertex a is mapped to vertex 1, vertex b to 2 and so on.*

Complexity of graph isomorphism

It was already proven in 1971 by Cook [26] that subgraph isomorphism is NP-complete by reducing to 3-satisfiability, so no known polynomial-time algorithm exists. Graph isomorphism is a special case of subgraph isomorphism and, interestingly, the complexity of graph isomorphism remains unknown. It has been shown, however, that for certain special types of graphs efficient algorithms (that run in polynomial time) do exist. For instance, Bodlaender [27] proved that a polynomial-time algorithm for graph isomorphism exists for graphs of bounded treewidth $\leq k$. For trees, an even more efficient algorithm that runs in linear time was already given in 1974 by Aho et al. [28].

Isomorphism in Haskell call graphs

As we have seen in Section 3.1.1, Haskell call graphs tend to look a lot like directed acyclic graphs. Further, with a few simple transformations (see following chapters) the call graphs can be modified so that they look even more like trees. This makes it possible to search for isomorphic subgraphs efficiently. Unfortunately, due to the exact nature of graph isomorphism which does not tolerate any errors, graph isomorphism is not well suited to perform plagiarism detection in source code that has undergone significant changes such as adding or removing functions which correspond to inserted or deleted vertices in the call graph. However, an isomorphism check could still be usable (as only a part of our tool) to detect plagiarism when, for instance, only function names have been changed.

3.3.2 Maximum common subgraph

Two other important concepts in graph matching that are closely related to each other are *maximum common subgraph* and *minimum common supergraph*. A maximum common subgraph of two graphs G and H is a graph K that is a subgraph of both G and H and has the maximum possible number of vertices. Minimum common supergraph is a relatively new concept that denotes a graph L of which both G and H are subgraphs and that has the minimum possible number of vertices. The concept is described in [29], in which the authors Bunke et al. also show that the minimum common supergraph can be computed using a computed maximum common subgraph. The maximum common subgraph problem is widely studied and known to be NP-complete, as can be read in the overview of NP-complete problems given by Garey and Johnson in [6].

3.3.3 Graph edit distance

When dealing with inexact graph matching and no large common subgraph can easily be determined, an intuitive way to measure graph similarity (or rather, dissimilarity) is to compute the *graph edit distance*, denoted by $d(G, H)$, which measures the cost of transforming one graph G into another graph H using a series of edit operations. Basic edit operations that are common in the literature include the substitution, insertion or deletion of vertices, or insertion or deletion of edges. An ordered sequence of edit operations that transform one graph into another graph is called an *edit path* (see Figure 3.5 for an example). A *cost function* defines an assignment of costs to the individual edit operations, and the cost of an edit path is the sum of the costs of all edit operations in the path. The graph edit distance is simply the cost of the minimum cost edit path, also called the *optimal edit path*.

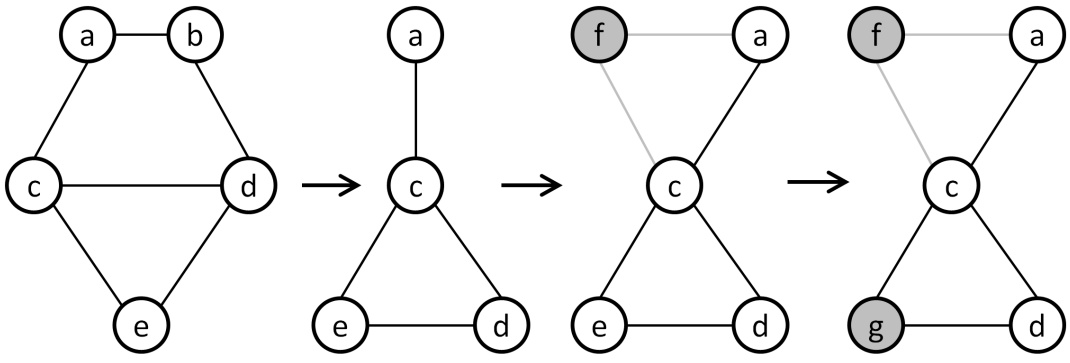

Figure 3.5: *An example of an edit path: deletion of vertex b, insertion of vertex f, substitution of vertex e by g. Newly introduced vertices and edges are shaded in gray.*

Although many algorithms designed for dealing with the graph edit distance problem compute a lowest-cost edit path between two input graphs, when reasoning about problem complexity the graph edit distance problem is often formulated as its decision variant:

> *Given two input graphs G and H, is there an edit path with cost at most k to transform G into H?*

Interestingly, it was shown by Bunke [30] that under a special class of cost functions, the graph edit distance problem and the maximum common subgraph problem are equivalent. These cost functions have the restriction that substituting a vertex v by vertex w directly is never cheaper than deleting v and then inserting w. A consequence of this is that the optimal edit path will only consist of deletions and insertions. This means that graph edit distance can be viewed as a generalization of maximum common subgraph and thus inherently even harder to solve from a computational point of view.

Graph edit distance is one of the harder NP-complete problems to solve. In fact, Zhang [31] proved that even if the two graphs being compared are binary trees and the labels of the tree nodes have an alphabet of size 2, the problem remains hard to solve. In practice, this means that no efficient algorithm (that has a running time polynomial in the size of the input graphs) exists to compute the edit distance optimally.

3.4 Graph matching algorithms

Various methods have been employed over the years to match graphs and to compute graph edit distance. These can be divided into optimal methods and suboptimal methods or approximation methods. **Optimal methods** are guaranteed to find the optimal solution (that is, the lowest-cost mapping between to two input graphs or the lowest-cost edit path between them) but have a running time and/or memory usage that is exponential in the size of the input graphs, due to the hardness of the problem. **Suboptimal or approximation methods** often have a running time polynomial in the size of the input graphs which is much faster, but are not guaranteed to find the optimal solution. Instead, for most approximation methods an upper bound is given for the solutions they find, which is proven to be within a certain error margin from the optimal solution. An example of this is the well known vertex cover problem, discussed by Rivest and Leiserson who give a simple approximation algorithm in [32] for which they prove that the solutions it finds contain at most two times the optimal number of vertices.

In this section, we will look into both classes of algorithms (optimal and suboptimal) and elaborate on which algorithms are suitable for our special case of Haskell call graph matching.

First, we must note that there are subtle differences among what algorithms actually compute but that this does not matter for what we are trying to achieve. Many algorithms designed for graph matching compute a lowest-cost *edit path* between two input graphs (see Neuhaus et al. [33]), but some algorithms compute a lowest-cost *mapping* between the vertices of graph G and the vertices of graph H (see Riesen et al. [34]). This is equivalent with computing an edit path because a mapping can be viewed as simply a series of substitutions. The corresponding edit path only has additional deletions for vertices that are in G but not in H, and insertions for vertices that are in H but not in G. There is no need here to make a clear distinction between algorithms that compute the edit distance and algorithms that actually compute a lowest-cost mapping or edit path, because the latter can also be used to calculate the edit distance. The edit distance is simply the total cost of the found mapping or edit path, which is what we are interested in.

3.4.1 Tree search based methods

In their most general form, **tree search algorithms** are algorithms that take a starting solution for a given problem and construct a **search tree** with the starting solution as its root node. In this search tree, each tree node represents a partial solution and each edge from a parent node v to one of its child nodes w represents a transition from partial solution v to partial solution w. Leaf nodes represent complete solutions. In most tree search algorithms, the transitions between solutions (edges in the tree) are simple modifications. The problem description defines which modifications are allowed on which partial solutions, and which solutions are complete solutions. Together, these determine the width and depth of the search tree.

The way in which the tree is expanded determines whether a found (complete) solution is optimal, because a yet unexplored area of the search tree may contain a solution that is even better than the one that was found first. Also, the way in which the tree is expanded has a great impact on the efficiency and memory usage of the algorithm because the search tree grows exponentially if each parent node has at least two child nodes.

When measuring graph edit distance, the tree search is meant to find the lowest-cost edit path from a graph G to graph H. In this particular application, the root of the tree corresponds to the original graph G and each non-leaf node corresponds to a modified version G' of G (graph G partially transformed to graph H). Each

edge between tree node t and n stands for a single edit operation transforming the graph in node t to the graph in node n, and each leaf node represents a graph that is completely transformed into H.

The general techniques for tree search described in this subsection are also described in Rivest and Leiserson [32], and in more detail in Russel and Norvig [35]. Especially the latter book provides an excellent reference for finding applications of tree search other than graph edit distance.

Breadth-first search

Two basic ways exist to expand a search tree: breadth-first search and depth-first search. **Breadth-first search** makes sure that all nodes at a certain depth of the search tree are examined before expanding to the next depth. This way, it is guaranteed that any complete solution that is found is on the smallest possible depth from the root node (starting solution). If this depth corresponds to the quality of the solution, then the breadth-first search algorithm always finds an optimal solution. But of course, the requirement that each depth must be expanded fully before moving on to the next depth results in exponential running time and memory usage.

Depth-first search

Depth-first search, on the other hand, always chooses a deepest unexpanded node for expansion. This is so that a complete solution is reached as fast as possible and with much less needed memory (fully expanded branches may be removed from the search tree), but the found solution may not be optimal if the depth corresponds to the quality of the solution: the still unexplored part of the search tree may contain a better solution at a lesser depth.

Because of their simplicity and general applicability, breadth-first search and depth-first search are often used as a substep in many algorithms, albeit in slightly modified form. However, they are *uninformed* algorithms that do not have specific knowledge about the quality of solutions and they both have serious disadvantages when it comes to optimality and performance.

Best-first search

A basic *informed* algorithm that does have specific knowledge about the quality of solutions is **best-first search**. A best-first search algorithm has a function $g(v)$ which measures the quality of the solution S_v corresponding to tree node v. The tree node with the best value for $g(v)$ is always expanded. When measuring graph edit distance, the value $g(v)$ is the cost of the edit path from the original graph G_0

to tree node G_v and is equal to the value $g(p)$ of v's parent node p plus the cost of the edit operation from G_p to G_v. Note that best-first search is equivalent with breadth-first search when the cost of every edit operation is equal to 1.

A* search

Because the worst-case running time of best-first search is still exponential, heuristics are often used to guide the search in a certain direction in order to find a solution much faster. A **heuristic best-first search** algorithm has a heuristic function $h(v)$ that estimates the cost from tree node v to a complete solution (the estimated remaining cost), and the tree node v with the lowest value for $f(v) = g(v) + h(v)$ is always expanded. A heuristic function is said to be *admissible* if it is "optimistic", i.e. it never overestimates the remaining cost. An algorithm that uses such an admissible heuristic function is itself also called an admissible heuristic search algorithm or an **A* search algorithm**. Once the A* search algorithm finds a solution v with estimated remaining costs $h(v) = 0$, that solution is returned. The A* search algorithm (see also Figure 3.6) was first described in 1968 by Hart et al. [36]. The authors also gave proof that the A* search algorithm is optimal and always returns the optimal solution.

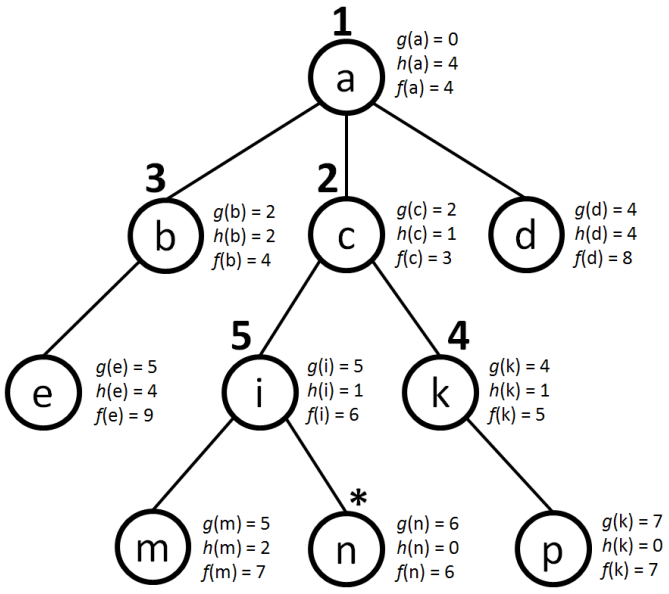

Figure 3.6: *An example of a search tree that is constructed by the A* algorithm. The numbers indicate in which order the nodes were expanded: the node with the lowest estimated cost f among all currently known nodes is always picked first. Eventually node n is reached and recognized as the optimal solution because the estimated remaining costs are 0 and no unexpanded node has a lower estimated total cost.*

The main idea of the proof of optimality for A* search is this: after its execution, the algorithm has found a solution with an actual cost g that is lower than (or equal to) the estimated cost f of any tree node that has not yet been expanded. But since those estimates are always optimistic, there cannot be another solution with lower cost through one of those unexpanded nodes. Therefore, the found solution is the optimal solution.

The running time of the A* search algorithm depends greatly on the quality and speed of the used heuristic function h. If the simple heuristic function $h(v) = 0$ is used, the A* search algorithm is equal to best-first search and will take a lot of time. In contrast, if a complex heuristic function is used that very accurately estimates the remaining cost, the search algorithm itself may be fast but computing the heuristic function may become problematic. Often a tradeoff is sought so that the heuristic function can be computed efficiently but still adds enough information for guiding the search algorithm so that a solution is found quickly.

A* search for graph edit distance

A* search algorithms are widely used because of their optimality and expected speed. In fact, almost all optimal methods for computing graph edit distance are variations of the A* search algorithm, as Bunke points out in an overview of techniques for computing graph edit distance given in [21]. Heuristic functions include calculating the number of remaining possible node substitutions, estimating the number of remaining edge insertions and deletions or even approximation algorithms.

3.4.2 Decision tree approach

Few other optimal methods for computing graph edit distance have been employed in practice because of the problem's complexity. One approach is the construction of a **decision tree** in order to classify an input graph among a set of model graphs, described in a 1999 article by Messmer and Bunke [37]. Their original algorithm has an intensive preprocessing step in which all possible permutations of the vertices of the model graphs are calculated and transformed into a decision tree. In the second phase of the algorithm, the unknown input graph can be matched with one of the model graphs in a running time quadratic in the number of vertices of the input graph. However, the decision tree that is contructed is exponential in the number of vertices of the model graphs so the preprocessing step costs a lot of time and memory.

In another paper by Messmer and Bunke [38], a way is described to extend the decision tree method from exact subgraph isomorphism to error-tolerant matching. A **maximal admissible error** is introduced, and can either be dealt with in the

preprocessing step (by incorporating distorted copies of the model graphs into the decision tree) or at run time (by matching distorted copies of the input graph with the decision tree as well). However, in the first approach the size of the decision tree increases even further depending on the maximal admissible error and in the second approach the running time becomes also dependent on the maximal admissible error (albeit still polynomial). Messmer and Bunke show in [38] that both approaches are only feasible for very small graphs in practice.

Decision trees for graph edit distance

The decision tree approach has a number of serious disadvantages when applied practically to compute graph edit distance. For instance, we do not want to match a single input graph against a set of model graphs, but we want to cross-compare a large set of input graphs. With the decision tree approach we would have to do an intensive preprocessing step (resulting in a decision tree of exponential size) for every input graph we want to cross-compare against all other submissions which would be inefficient to say the least. More importantly, the maximal admissible error must be kept low in order to keep the running time and memory usage feasible but this would prevent us from finding certain important suspected cases of plagiarism. Consider the following simple example: in Haskell program A (with a size of 100 functions) a set of 10 functions was copied from program B (also with a size of 100 functions). Any maximal admissible error below the difference of 90 functions would prevent program A from being matched to program B.

Both A* tree search based methods and decision tree based methods have exponential running time in the worst case. Perhaps the most important reason why we prefer A* search based methods over decision tree based methods for plagiarism detection is that the efficiency of A* search based methods can be greatly improved by using a simple but effective heuristic function. Moreover, as we shall see in the next section, A* search based methods can be easily modified so that they run even (a lot) faster while still approximating the optimal solution with small differences.

3.4.3 Suboptimal A* search variants

A few simple modifications can be made to the basic A* search algorithm to speed up graph edit distance computation. For instance, in a 2006 paper [33] Neuhaus et al. describe two A* search variants that are suboptimal but a lot faster than the basic A* search algorithm.

A*-Beamsearch

In its most basic implemented form, A* search stores all solutions (search tree nodes) and the transitions between them in memory. However, most implementations store

only the root solution and the **frontier** of solutions: the list of all unexpanded nodes, which are leaf nodes until they are expanded. In all internal nodes of the search tree (that already have been expanded) only transitions to parent and children nodes are kept in memory. Much less memory is used in this manner if solutions themselves take a lot of space, while all solutions in unexpanded nodes (which is the only relevant information in most applications) are still available. Still, no information is lost in this manner: the specific solution in an internal tree node can still be computed recursively by computing the solution in its parent node and then applying the transition from its parent to itself. Also, complete edit paths are still available by simply walking from a leaf node back up the tree to the root node.

With the modification described above, much less memory is used but optimality is still guaranteed because the algorithm stays functionally the same. However, we can reduce memory usage even further by storing only the most promising portion of the frontier solutions. A predefined setting can indicate, for instance, that only the best 100 (at most) frontier nodes are kept in memory. This is called **A*-beamsearch** and it is a commonly used technique. The number of solutions that is kept in memory is called the **beam width**.

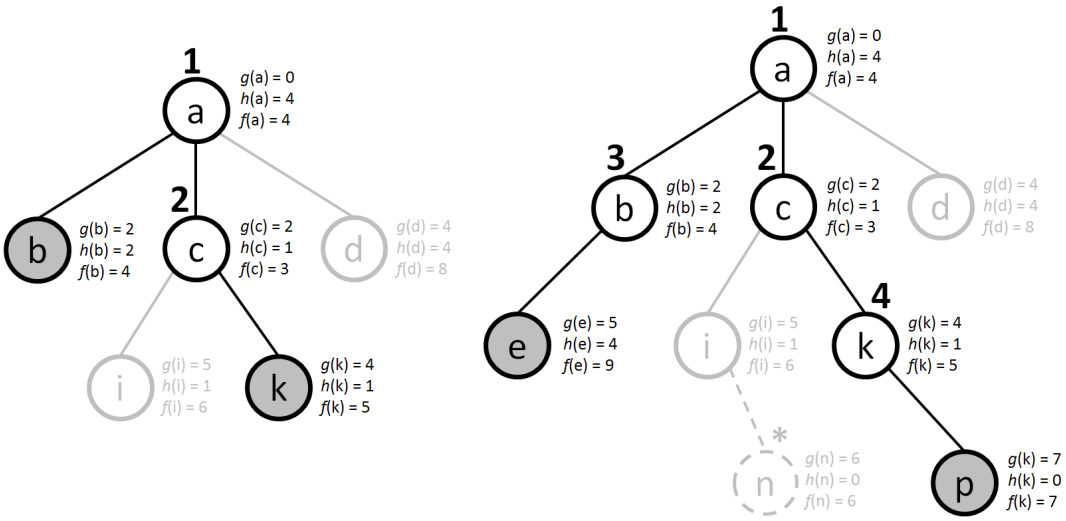

Figure 3.7: *The same example search tree as in Figure 3.6, but now constructed by A*-beamsearch with beam width 2. On the left, the situation after the second node expansion is sketched, with only the two shaded nodes kept in memory and the gray branches removed from the search tree. The situation on the right is after the fourth node expansion.*

A*-beamsearch can also greatly speed up the algorithm because only a part of the search tree is expanded. Unfortunately, this also immediately provides proof for

its suboptimality: because only part of the search tree is investigated, the algorithm might find a solution that is not optimal. Consider, for instance, the example of A*-beamsearch given in Figure 3.7. This is the same search tree as in Figure 3.6 but now constructed using A*-beamsearch with beam width 2. In the original search tree from Figure 3.6, a child node of node i was found as the optimal solution. In this example, node i is never expanded because it is removed from memory after expansion of node c, since at that point there are two nodes (b and k) that have better estimated total costs and therefore look more promising. Eventually, the suboptimal solution in node p is returned.

A*-Pathlength

Neuhaus et al. [33] observed that when running an A* search algorithm on two similar graphs, in the first stages of execution there is often a long branch being expanded in the search tree along a single direction. This long branch is a logical result of a series of low-cost substitutions since the two graphs are similar, and this partial edit path is probably the optimal one. However, once the algorithm stumbles on an expensive edit operation along this path, the search algorithm starts to explore other parts of the search tree. This dramatically increases running time and may be unnecessary because the optimal solution might be just a few edit operations away from the currently deepest node in the search tree.

Therefore, Neuhaus et al. proposed another variation on the standard A* search algorithm for graph edit distance in [33], in which long partial edit paths are favored over shorter ones. This variation is called **A*-Pathlength**. In A*-pathlength, instead of using $f(v) = g(v) + h(v)$ to calculate the estimated total costs of node v, the following formula is used:

$$f(v) = \frac{g(v) + h(v)}{t^{depth_v}}$$

in which the depth of node v is equal to the number of edit operations in the partial edit path from the tree root to v. The higher the value of parameter t (which is usually a little above 1), the lower the cost of longer partial edit paths.

Efficiency of A*-Beamsearch and A*-Pathlength

A*-beamsearch and A*-pathlength are very efficient when applied to two similar graphs. This is based on the fact that between similar graphs, there is often one edit path with costs that are significantly lower than all other edit paths because many substitutions between vertices of similar graphs are cheap. Both adapted algorithms are quite effective in eliminating other, uninteresting edit paths from the search tree. Neuhaus et al. test the effectiveness of A*-beamsearch and A*-pathlength with experiments in [33], which show that the speedup in those cases

is indeed substantial (over ten times faster) but that the edit distance accuracy remains nearly unaffected (less than one percent difference from the optimal edit distance).

Is is important to note that a considerable speed improvement of A*-beamsearch and A*-pathlength over regular A* search is often a much stronger justification for their use than the decrease in memory usage. This is so because even more radical ways to spare memory exist. One example is to not store *any* solution in memory other than the original root solution and the current solution, while still keeping all estimated costs and transitions between solutions. This still preserves all relevant information and the algorithm stays functionally the same as long as it is walking down a path in the search tree, but it needs to calculate a new current solution every time it jumps to a different part of the search tree.

Concludingly, A*-beamsearch and A*-pathlength are variations of the standard A* search that are both very useful when it comes to graph edit distance computation (and possibly plagiarism detection in Haskell call graphs), or they provide at least useful insights needed for constructing a plagiarism detection algorithm. Both have been shown to greatly improve the expected running time of A* search while still maintaining high accuracy, and as a sort of bonus they use much less memory.

3.4.4 Approximate bipartite matching

The graph edit distance problem can be approximated by computing an optimal **bipartite matching** between nodes and their local structure in the two graphs. Important research in this area has been done by Riesen et al. in [34] and [39]. In these papers, the authors use the **Hungarian method** for constructing an optimal bipartite matching between the graph nodes. The Hungarian method was first described already in 1955 by Harold Kuhn [40] and later proved to run in strongly polynomial time by [41]. The algorithm is nowadays usually referred to as **Munkres' algorithm** for assignment problems.

The bipartite matching version of the algorithm for approximating graph edit distance, described by Riesen et al., computes an optimal match between nodes of the two input graphs with respect to local structure (the edges of the nodes to their direct neighbors). Although the algorithm is fast (polynomial time) it is suboptimal because it does not take implicit edge edit operations into account. In the found matching, a match between nodes v and w (which implies edge edit operations needed to make both nodes connect to the same neighbors) may have an impact on a different match between two other nodes so that the matching may not be optimal globally with respect to the optimal edit path between the whole graphs.

On a side note, important related work for graphs that are trees has been done by Zhang in [42], who showed that the edit distance between unordered labeled trees can also be approximated accurately in polynomial time. The algorithm he describes puts certain constraints on the mapping between nodes of the two input trees, but those constraints follow logically anyway from the ancestor-descendant relationships that occur in trees. The algorithm then reduces the bipartite matching problem to a minimum-cost flow problem.

Other approximation methods

Many other methods for approximating graph edit distance have been described in literature. Probabilistic relaxation schemes [43], genetic local search [44] and neural networks have been proposed in literature and applied practically in pattern recognition and object recognition problems. Most of these techniques and algorithms run in polynomial time. All these methods are suboptimal in the same manner as the assignment algorithm described above, because they only consider local structure for similarity. A linear programming approach described in [45] also runs in polynomial time, and computes a lower and upper bound from which an approximation can be derived.

Besides their suboptimality, many of these approximation methods are designed to deal with edit distance for a specific class of graphs and as such their effectiveness in a more general sense can be disputed. Tree search based methods like A* search variants are much more flexible to apply. Another downside of some of these approximation methods (like local search based techniques) is that they are nondeterministic (or rather, probabilistic) which makes it hard to repeat experiments in order to show efficiency and accuracy.

3.4.5 Summary of algorithms and their applicability

Summarizingly, the mentioned approximation techniques in the previous sections can approximate graph edit distance accurately when one of the two input graphs is similar to a subgraph of the other, or when a subgraph of an input graph is similar to a graph in a set of predefined model graphs. However, this level of accuracy has not been shown when only a (small) subgraph of both graphs is similar, which is typically the case in a plagiarism detection tool. A tree search based method like A* search is both optimal and flexible, and it can be modified so that it already recognizes which (possibly small) parts of the two input graphs are similar in an early stage, by finding an edit path that is still only partial but has relatively low cost. Another optimal method for computing graph edit distance, by means of constructing a decision tree, requires so much time and memory that it is infeasible for Haskell call graphs which typically have more than a handful of vertices.

Although the approximation methods from the previous sections do not appear very suitable for the graph edit distance problem on Haskell call graphs, perhaps they can be used as a heuristic function in an A* search algorithm. An idea described by Riesen et al. in [46] is to use the bipartite matching procedure as a substep of the A* search algorithm, in order to estimate the cost of the remaining part of the edit path constructed so far. Of course, a requirement for this to work is that the bipartite matching procedure must be modified so that it provides an *admissible* heuristic function, i.e. it returns an optimistic estimation of the remaining costs.

Chapter 4

A tool for plagiarism detection

Based on ideas from graph matching techniques described in the previous chapter, we have implemented a tool for detecting suspected cases of plagiarism. In this chapter, we will describe the basic design behind this tool and we will go into some implementation details of the computer program. We also elaborate on some of the settings we can use and what kind of information the program outputs.

4.1 General approach and design

We can distinguish five phases in a single run of our program, which we will describe in greater detail in subsequent sections:

1. **Reading input graphs**. The input in this case is a collection of files that contain call graph data, which represent Haskell programs. These are parsed into graph objects in the memory of our program.

2. **Preprocessing**. Some modifications are performed on the parsed graph structures which are primarily meant to reduce graph size and to speed up the algorithm in the next step, but we argue that they can also improve plagiarism detection accuracy.

3. **Edit distance algorithm**. The main algorithm in this step is based on a modified version of A* search for computing graph edit distance, which gives a similarity score for each pair of compared graphs.

4. **Subgraph isomorphism algorithm**. As an additional check, an exact subtree isomorphism algorithm is executed as well for all graph pairs.

5. **Outputting suspected plagiarism information**. With the information gathered during execution of the algorithms in the previous step, a sorted list of suspected cases of plagiarism is outputted with the most suspicious case at the top.

4.1.1 Implementation in Java

The Holmes program [5] was written in Haskell, and our tool may be seen as an addition to the checks performed by Holmes. Although we understand that implementing our program in Haskell as well would have facilitated the integration of the two programs into a single tool, we wanted to use a general-purpose programming language like C# or Java instead. This is mainly because for us, algorithms written in pseudocode translate more easily to code in these languages. We finally decided to implement our program with Java because of our own experience with this language in previous projects, which is far more extensive compared to our experience with Haskell, C# and other languages similar to C. Not only does this choice speed up implementation time, but it also prevents a lot of bugs.

Our program can be run independently from Holmes because both programs use the same input graph files outputted by Sherlock. Holmes can even execute our program directly in one of its steps; this should not be difficult to configure. However, the integration of our program and Holmes into one tool may be worth investigating once we have demonstrated the effectiveness of our program. For instance, maintainability might start playing a role if the used graph format changes. For this reason, we will list this task in Chapter 7 on future work.

4.2 Reading input graph files

The first part of our program parses input graph files and constructs graph objects that are stored in memory. Functions from a Haskell program become vertices in a graph object, and function calls are implicitly stored: each vertex contains a list of references to "parent" vertices (functions that call this function) and a list of references to "child" vertices (functions that this function calls). This means that each edge in the graph is always a double object reference in the memory of our program. Double object references are always a possible source of bugs in computer programs, especially in this case when edit operations are applied to the vertices of graphs. However, this approach is necessary if in our algorithm we want to have access to all functions calling a given function in constant time. We are confident that after much testing the correctness of our program is proven by the results it gives, as shown in the next chapter.

In Section 2.2.3, we briefly mentioned the call graphs that Sherlock creates in order for Holmes to calculate a few metrics on them. A visual example of such a call graph was shown in Figure 3.2. The standard DOT file format that our program reads is described in Appendix A. We use the same graphs as input for our own tool, for two reasons. The first reason is that by using exactly the same input graph type we can better compare performance and success rate of our own tool against that

of Holmes. Another reason is that by first using Sherlock to extract call graphs, we use an existing piece of software that has been proven to work well. Otherwise, we would either have to write our own Haskell parser (which can be a tedious and buggy task) or use an external library that may not produce the right type of call graphs. Also, this makes a possible future integration of our tool and Holmes easier (see future work in Chapter 7).

4.2.1 Reading output graphs of Sherlock

It is important to point out what additional properties these graphs have on top of the general properties of Haskell call graphs described in Section 3.1.1. The call graphs extracted by Sherlock are static call graphs of only top-level functions. This means that any local function definitions (such as in a `where` or `let...in` construction) are excluded. Type information of functions is not extracted either, so the only attributes that vertices will have are label (function name) and module name. Furthermore, Sherlock is able to exclude template functions from the graph if they are marked as such first, but Prelude functions are still included in the graph. Finally, Sherlock already filters out some dead code: functions that are not statically referenced from any other function. Note that this cannot remove all dead code: a function that is called by another function on a certain condition that will never be satisfied during runtime is also dead code (dynamically speaking), but will not be removed by Sherlock.

Although type information and local function definitions are important in any Haskell program, we argue that we do not need this information when searching for plagiarism with the graph matching algorithm in our tool. One reason for this is that we claim our tool already has enough information to successfully identify suspected cases of plagiarism. More importantly, however, we think that this additional information may in fact even hinder our tool when students have masked their attempt at plagiarism by modifying exactly this information in their submission, as explained earlier in Section 2.2.2.

In the next chapter we will show that the claim about the success of our tool without the need for additional information is justified. However, the claim that including type information and local functions does not improve the success of our tool is based on our own logical assumptions rather than empirical evidence. This makes it an interesting research question to find out how this additional information will exactly influence the success rate of our tool. For this reason, we have included this topic when discussing possible future work in Chapter 7. Nevertheless, in the rest of our program we will limit ourselves to the graphs outputted by Sherlock: with only top-level functions and without type information.

4.3 Preprocessing

A few modifications are made to the parsed graph objects in this step of the program. These modifications are meant to reduce graph size and speed up edit distance computation, but also to increase the accuracy of the algorithm. We will now describe the different types of modifications in more detail, and explain the reasons for applying them.

4.3.1 Adding a dummy root vertex

The last step in preprocessing adds a **dummy root vertex** to each graph. Since this is the only step that adds something to the graph, we will describe this step first. Edges are added from this dummy vertex to each other vertex in the graph that has no incoming edges, which is always at least one vertex. This is to make sure that the graph is connected, i.e. every vertex can be reached from the dummy root vertex.

In the algorithm that follows that computes graph edit distance, we make sure that this dummy vertex does not have any influence on the matching process: we set the cost of substituting this dummy vertex for the dummy vertex of another graph to zero, and we set the cost of substituting it for any other vertex to infinity (a very high number in programming code). This way, a dummy root vertex in the first graph will always be matched to the dummy root vertex in the second graph.

4.3.2 Removing Prelude functions

Functions that are part of the Haskell Prelude are parsed just like any other function by Sherlock, and they are included in the outputted graph file. In the example graph of Figure 3.2, Prelude functions are shaded in gray. Luckily, Sherlock does not include edges from Prelude functions to other Prelude functions. However, the Prelude functions that remain in the graph hinder the matching process because they expand the search tree of the algorithm that computes edit distance: as far as the algorithm is concerned, substituting a Prelude function for a different Prelude function, or even for a non-Prelude function, may be expensive but still valid.

We could perform the same method as with the dummy root vertex, by assigning zero costs to substitution of a certain Prelude function for itself and infinite costs to substitution for any other function. This would make sure that the algorithm always substitutes a Prelude function for itself, but all other possible substitutions for this function would still be included in the search tree.

Instead, we simply remove Prelude functions from the graph. In all vertices representing functions that call Prelude functions, we remove the object references to those Prelude function vertices and keep a separate list of references to Prelude functions. This way, no information is lost in a vertex object concerning calls to Prelude functions while the search tree of the edit distance algorithm is reduced in size, so that the algorithm runs faster.

4.3.3 Removing recursion

Self-recursion and mutual recursion are common among functions in any given Haskell program. Although other cycles of functions with a size of 3 functions or more are possible in Haskell, these are quite uncommon. That means that virtually all Haskell call graphs are cyclic (because of recursion) but that only very few remain cyclic when recursion is removed from the call graph. Removing cycles is beneficial for the complexity and running time of many graph algorithms, because many efficient algorithms for all sorts of purposes exist for directed acyclic graphs (which the call graphs then become).

We remove self-recursion from the graph by removing all object references from vertices to themselves and then adding a boolean flag as an attribute to all vertices indicating whether or not that vertex is self-recursive, so that no information about recursiveness is lost. Mutual recursion is also easily defined in Haskell, consider the following example code that can determine whether an integer is even or odd:

```
even 0 = True
even (n+1) = odd n
odd 0 = False
odd (n+1) = even n
```

Mutual recursion is a bit trickier to resolve than self-recursion because there are in fact four object references in our program: an outgoing edge from the `even` function to the `odd` function, an outgoing edge from `odd` to `even`, and two incoming edges as well. Still, we remove all these object references from the normal lists of incoming and outgoing edges and replace them by a single attribute in both vertices that indicates the other vertex with which this vertex is mutually recursive.

This somewhat reduces the graph size (in terms of edges) and speeds up the edit distance algorithm, but only a little since of course no vertices are removed. As said earlier, these modifications also facilitate the use of certain efficient algorithms that assume that no cycles exist in the graph if we ignore the possibility of other cycles (of size 3 or larger) for now.

If such a larger cycle does exist in a call graph, an option would be to always find the edge (in that cycle) that is closest to the dummy root vertex and remove that edge from the cycle. The information that there was in fact an edge removed between two vertices could be stored in an extra attribute in those vertices. However, we included a check for the existence of cycles in our algorithm and we found no remaining cycles in any of the call graphs we ran our experiments on, after having removed recursion from the graphs. For this reason, we decided not to include this modification in the preprocessing phase. Because this does leave our tool vulnerable to certain errors that might occur when a large cycle is indeed found, we listed this modification under future work in Chapter 7.

4.3.4 Other possible modifications

More complex modifications than the ones already described are possible. For instance, one could think of modifications that transform a call graph into a tree. Many algorithms exist for calculations on trees that are of an even lower complexity than algorithms that work with directed acyclic graphs, so that we could improve the speed of our tool even further. Most Haskell call graphs are already very tree-like after the modifications described above, as can be seen in the example call graph in Figure 3.2.

Transformation into a tree

To transform a call graph into a tree, a possible modification would be to remove all **back edges** from the graph (edges from a certain vertex v back to a vertex w on a higher level) and store that information in other attributes (of vertex v). Another option would be to remove the edge from v to w, copy vertex w and then add an edge from v to the copy of w. But then the problem arises of deciding whether to match a vertex in a second graph with w or its copy, and we would make the tree unnecessarily large because there are possibly many duplicates. Moreover, in both approaches any information regarding the children of vertex w or even the whole subgraph reachable from w would be lost in the simple copy of w.

Constructing a reachability tree

Another way of obtaining a tree representation of a call graph is to construct its **reachability tree**, in which the shortest way of reaching a vertex from the dummy root vertex is represented by the path in the reachability tree from the root to that vertex. A reachability tree can be constructed in linear time with breadth-first search (it is simply the breadth-first search tree). However, in a reachability tree the information regarding cycles and back edges is lost and as such it is not a completely accurate representation of the original call graph. Therefore, any optimal edit path

that is computed from this reachability tree to another graph's reachability tree may not be optimal when looking at both original graphs.

Both approaches to obtaining a tree described above have the disadvantage that certain information regarding graph structure is lost. Any algorithm that is applied afterwards to calculate similarity cannot take this information into consideration and therefore is not optimal with respect to the similarity of the original graphs. For this reason, we chose not to implement these modifications so that we can achieve a higher accuracy. However, the question remains how these modifications influence accuracy exactly. We have added this topic of research to Chapter 7, the list of future work.

4.4 Tree search based algorithm

In this section, we will give the details of our algorithm. The basic version is listed in Algorithm 4.1.

4.4.1 Construction of the search tree

As in standard A* search described earlier in Section 3.4.1, the algorithm constructs a search tree initially consisting of only the root node (which contains the original untransformed graph G). Then, in each step the most promising node n is expanded by adding its children nodes to the search tree. This means that for every possible edit operation on the partially transformed graph G' in n, a child node is constructed in which that edit operation has actually been carried out on a copy of graph G'. Possible edit operations on graph G' are:

- **Substitution** of a vertex v that was in the original graph G, for a vertex w that was in graph H and that is not yet present in G'.
- **Deletion** of a vertex v that was in the original graph G.
- **Insertion** of a vertex w that was in graph H and is not yet present in G'.

Edge insertions and deletions are implied by the vertex edit operations listed above. Consider, for instance, a search tree node n which stores partially transformed graph G'. When a vertex v in G' (that was also in the original graph G) is substituted for a vertex w that was originally in graph H, a check is performed on all existing edges from and to the replaced vertex v. Edges to all vertices with which v was connected but w isn't are removed, and edges to all vertices with which w is connected but v wasn't are inserted.

Algorithm 4.1 Main algorithm for calculating graph similarity.

Input: Two preprocessed input graphs G and H
Output: Similarity score between G and H

```
1: PriorityQueue frontier ← new empty priority queue
2: TreeNode root ← new search tree node with G as source and H as target
3: Add root to frontier
4: while frontier is not empty do
5:     TreeNode n ← remove best node from frontier
6:     if no edit operations possible anymore in n then
7:         return similarity score in n
8:     if vertices of original graph G still exist in current graph in n then
9:         Vertex v ← a still unmatched vertex in the current graph
10:        for all still unmatched vertex w from graph H do
11:            TreeNode c ← new child node of n with substitution of v for w
12:            Add c to frontier
13:        TreeNode c ← new child node of n with deletion of v
14:        Add c to frontier
15:    else
16:        for all still unmatched vertex w from graph H do
17:            TreeNode c ← new child node of n with insertion of w
18:            Add c to frontier
```

4.4.2 Heuristic function

Recall from Section 3.4.1 that in an A* search algorithm, the estimated total cost $f(n)$ of an edit path through search tree node n is given by $f(n) = g(n) + h(n)$, where $g(n)$ is the actual cost of the partial edit path up to node n and $h(n)$ is an optimistic estimation of the remaining cost. In the first line of the while loop in Algorithm 4.1, we choose the "best" node from the frontier: that is the node n with the lowest value for $f(n)$. The heuristic function $h(n)$ must be quick to calculate and somewhere between 0 and the real remaining cost (preferably as close to the real cost as possible). We use the following simple heuristic function for a tree node n containing a partially transformed graph G':

$$h(n) = c_{sub} \cdot \min(|R_G|, |R_H|) + c_{ins,del} \cdot \text{abs}(|R_G| - |R_H|)$$

where c_{sub} and $c_{ins,del}$ are settings that indicate the standard cost of substituting and inserting/deleting a vertex, respectively. R_G is the set of remaining vertices of the original graph G that are still in the partially transformed graph G', and R_H is the set of remaining vertices of the original graph H that have not yet been inserted into G'.

In simple terms, the heuristic function optimistically assumes that all vertices from the original graph G that are still remaining in G' will be substituted by equal vertices from graph H. After that a few more remaining vertices might need to be deleted from G', or a few more remaining vertices from H might need to be inserted into G'.

Estimating edge operations

The number of remaining edge edit operations is not approximated by this function since it assumes that the remaining substitutions are for equal vertices that do not imply any edge insertions or deletions. If that is indeed the case during the rest of the transformation, any approximation of the remaining number of edge edit operations would give a pessimistic view on things. Moreover, this would mean that the heuristic function is no longer **admissible** and that the algorithm can no longer guarantee that it gives the optimal edit path in the end (see section 3.4.1).

Some heuristic functions are described in literature (see Bunke [21] for an overview) that do try to take the remaining number of edge edit operations into account in such a way that the heuristic function remains admissible, but these heuristic functions are much more complex and take more time to compute. The most important example of this is bipartite graph matching (see Riesen et al., [34] and [46]). In Chapter 5 we will show with experimental results that the simple heuristic function we propose is efficient and gives good results.

4.4.3 Similarity score

Once a tree node n is found that contains a graph G' on which no edit operations are possible anymore, the graph has been fully transformed into H and we can return the similarity score of the corresponding edit path. This edit path is complete because no edit operations are possible anymore, and it is guaranteed to be the optimal one as explained earlier in Section 3.4.1. To indicate a **similarity score** $s(n)$ for the found tree node, we use a number between zero (completely dissimilar graphs) and 100 (equal graphs). To calculate this score using the cost of the edit path from the root node to this node, we need to know a lower and upper bound on what the total costs might be for transforming a graph of the same size as G into a graph of the same size as H.

For the **upper bound**, we assume that G and H are completely dissimilar (i.e. they have no vertices or edges in common) and we take the costs of deleting all vertices and edges in G and then inserting all vertices and edges from H:

$$maxCosts(G, H) = c_{ins,del} \cdot (|V_G| + |V_H|) + c_{edge} \cdot (|E_G| + |E_H|)$$

where $|V_G|$ is the number of vertices in graph G, $|E_G|$ is the number of edges in graph G, and c_{edge} is a setting that indicates the standard cost of inserting or deleting an edge.

As a **lower bound**, we assume that G and H are equal (or that one is a subgraph of the other) and we take the maximum possible number of substitutions plus the remaining insertions or deletions:

$$minCosts(G, H) = c_{sub} \cdot \min(|V_G|, |V_H|) + c_{ins,del} \cdot \text{abs}(|V_G| - |V_H|)$$

Note that the number of edges is irrelevant here because no edges may be inserted or deleted at all if both graphs are equal.

We can then use the following simple formula to calculate the similarity score $s(n)$ for graphs G and H, with the cost $g(n)$ of the optimal edit path ending in search tree node n:

$$s(n) = 100 \cdot \frac{maxCosts(G, H) - g(n)}{maxCosts(G, H) - minCosts(G, H)}$$

The closer this score is to 100, the more similar graphs G and H are and the more likely it is that this be a case of plagiarism.

4.4.4 Modifications for speedup

Algorithm 4.1 finds the complete and optimal edit path from graph G to H, and outputs the resulting similarity score for these graphs. This is not the algorithm

we actually use in our program, but only shown for the sake of simplicity so that a few implicit substeps of it could be described in detail more easily. Because of its high complexity (it constructs a search tree of exponential size) it would be infeasible to run on real Haskell programs with, for instance, 50 functions or more. A few modifications are essential for speeding up our main algorithm and they are required to be able to run our program on real data. These modifications result in Algorithm 4.2 which is the actual algorithm we use in our tool. Parts that are unchanged (from Algorithm 4.1) are in gray; modified or introduced parts are in black.

Breaking off expensive matches

In practical cases of plagiarism, a common observation also mentioned by Hage in [9] is that only a (small) part of a program is plagiarized in the form of a group of functions that call each other. An ideal plagiarism detection tool would immediately recognize this subgraph G' to be similar to a subgraph H' from another program before looking further into the rest of the graphs. Our A* search algorithm is already designed to match more similar parts of the graphs first because edit operations with lower costs are applied first. Once the more similar parts of the graphs have been matched by substituting similar vertices for each other, edit operations will start to get relatively expensive: a vertex from G' for which there is no similar vertex from H must either be deleted or replaced in an expensive substitution.

If we can recognize this moment during execution when the matching process starts to get expensive, there is no need to process any further because the most interesting part of the graphs (the part that is similar) has already been found. This captures the essence of a plagiarism detection tool perfectly: the possibly plagiarized part of the program has been found and the rest is not really interesting. Towards this end, we have added two settings to our search algorithm that influence the moment when the algorithm terminates prematurely and returns a result:

MaxEditCost A number that represents the costs of an edit operation that we deem too expensive. When our search algorithm applies an edit operation that is at least as expensive as this setting, it terminates and returns the similar subgraph found so far.

MinSimilarity A number between zero and 100 that is a threshold for the similarity score. If the similarity score of the most promising search tree node drops below this threshold during execution, the algorithm also terminates and returns the current result.

Breaking off the matching process in the ways described above has two advantages. The first is simply that it makes the algorithm run faster. There is no need for our

algorithm to spend execution time on uninteresting and dissimilar parts of graphs that are being compared. If there are parts of the graphs that are too dissimilar to be of any interest, we do not care how dissimilar they are precisely if this saves us time.

The biggest advantage of breaking off the matching process in these ways is that the information about which parts of the graphs are interestingly similar is available to us immediately. Although this could be calculated from a complete edit path (after complete execution of the algorithm), the partial edit path that is returned gives exactly this information.

Subgraph and total similarity score

We can now make the distinction between the subgraph similarity score and the total similarity score. The **subgraph similarity score** indicates the similarity of the subgraph that was implicitly matched by the found partial edit path, and is an adaptation of the similarity score mentioned earlier. For a partial edit path ending in node n containing partially transformed graph G' it is calculated as follows:

$$s(n) = 100 \cdot \frac{maxCosts(G, G') - g(n)}{maxCosts(G, G') - minCosts(G, G')}$$

The **total similarity score** t takes the size of this similar subgraph into account with respect to the total size of the compared graphs:

$$t(n) = s(n) \cdot \frac{|P_n|}{max(|V_G|, |V_H|)}$$

Here, $|P_n|$ is the length of the partial edit path from the root node to node n, or simply the number of edit operations needed to transform graph G into current graph G'. We output both scores in our results. In essence, the subgraph similarity score indicates that a matching subgraph between G and H was found with the indicated similarity, and the total score indicates how similar the complete graphs G and H are.

A*-Beamsearch and A*-Pathlength

As can be seen in Algorithm 4.2, **A*-Beamsearch** is easily implemented by letting the priority queue (that stores the frontier) keep track of the worst tree node currently in the frontier. When a new child node of an expanded node is constructed, it is only actually added to the frontier if the frontier has not yet reached its maximum size (BEAMSIZE) or if it contains a worse node. Of course, in the latter case the worst node is removed from the frontier.

Algorithm 4.2 Modified algorithm for calculating estimated graph similarity.

Input: Two preprocessed input graphs G and H

Output: Similarity score between G and H, or **false** if the match is uninteresting

```
 1: PriorityQueue frontier ← new empty priority queue
 2: TreeNode root ← new search tree node with G as source and H as target
 3: Add root to frontier
 4: int expansions ← 0
 5: int deepest ← 0
 6: float similarityAtDeepest ← 0
 7: while frontier is not empty do
 8:     TreeNode n ← remove best node from frontier
 9:     int depth ← depth of n
10:     float similarity ← estimated similarity score in n
11:     if no edit operations possible anymore in n then
12:         return similarity
13:     if depth > deepest then
14:         deepest ← depth
15:         similarityAtDeepest ← similarity
16:         e ← edit operation performed in n
17:         if similarity < MinSimilarity or cost of e > MaxEditCost then
18:             return similarity
19:         if depth ≥ MaxTreeDepth then
20:             return similarity, noting that the max depth was reached
21:     if expansions > MaxNumExpansions then
22:         return similarityAtDeepest, noting that max expansions was reached
23:     worst ← the highest-cost node currently in frontier
24:     if vertices of original graph G still exist in current graph in n then
25:         Vertex v ← a still unmatched vertex in the current graph
26:         for all still unmatched vertex w from graph H do
27:             TreeNode c ← new child node of n with substitution of v for w
28:             if |frontier| < beamsize or cost of c < worst then
29:                 Add c to frontier
30:         TreeNode c ← new child node of n with deletion of v
31:         if |frontier| < beamsize or cost of c < worst then
32:             Add c to frontier
33:     else
34:         for all still unmatched vertex w from graph H do
35:             TreeNode c ← new child node of n with insertion of w
36:             if |frontier| < beamsize or cost of c < worst then
37:                 Add c to frontier
38:     expansions ← expansions + 1
```

A*-Pathlength is also easily implemented by modifying the estimated total cost function f which is not explicitly shown in the pseudocode of Algorithm 4.2. The new function then depends on a setting $p > 1$ that indicates a degree of reward for longer edit paths, and becomes:

$$f(n) = \frac{g(n) + h(n)}{p^d}$$

where d is the depth of tree node n and equal to the length of the partial edit path to node n. The value for p must be only a little above 1, otherwise long paths would be rewarded too heavily and the algorithm would start to look too much like depth-first search. If the value is 1, longer paths are not rewarded and the algorithm works the same as before.

As explained earlier in Section 3.4.3, A*-Beamsearch and A*-Pathlength are especially efficient and accurate when dealing with similar graphs. This means that letting the tree search algorithm only investigate subgraphs which are more similar than the whole graphs, increases the usefulness of A*-Beamsearch and A*-Pathlength as well as a sort of bonus. Both A* search variants can be turned on or off with settings, and we will investigate the effectiveness of these variants in Chapter 5 with experiments.

Memory limitations

Even with the optimizations described in the previous paragraphs, the search tree that is constructed by our algorithm remains very large in theory and could cause performance problems or too much memory use. For this reason, we added two more settings to our program that can control the maximum search depth and maximum number of node expansions (MaxTreeDepth and MaxNumExpansions, respectively). If during execution of our algorithm the search tree becomes too deep or too large, these settings make sure that the algorithm terminates and returns the current subgraph similarity score. However, since we do no longer have an idea of what the similarity would be of the remaining unmatched graph parts, this returned similarity score can only be an estimation.

Ideally, we would rather not resort to cutting off the search algorithm in this rather harsh manner because then the result no longer accurately indicates what the most similar parts of the compared graphs are. Luckily, during our experiments in Chapter 5 we saw that due to the optimizations described earlier the algorithm tends to reach a solution quickly anyway without having to construct an unnecessarily large search tree. With proper parameter tuning for these optimizations, the limitations for tree depth and node expansions are almost never reached. In the few cases where the algorithm does reach graph parts it does not know how to match, these limitations make sure that the algorithm does not get stuck or cause a crash.

4.5 Subgraph isomorphism checks

In this step of the program, a collection of all (largest possible) subtrees in the set of compared graphs is first created. Then, the exact isomorphism algorithm from Aho et al. [28] is executed for every pair of subtrees to determine isomorphism. This step of the program serves as an additional check to the main error-tolerant graph edit distance algorithm given in the previous section. Although that algorithm gives good results as we will see in the next chapter, the exact subtree isomorphism algorithm might find some possible cases of plagiarism that are not found by the main algorithm.

We have implemented the exact isomorphism algorithm from [28], sometimes named the **AHU Algorithm** after its authors Aho, Hopcroft and Ullman, to run in linear time. We will not describe the algorithm in detail here because it is well-known and many descriptions of it have been given in literature. The algorithm is based on the following idea: working up from the leaves, tuples are assigned to each vertex indicating degree information of its children. Every time the algorithm goes up one level, these tuples are sorted and used to calculate new tuples for the vertices on the higher level. The sorted sequence of tuples must be equal for the two compared trees for them to be isomorphic. If the root nodes of the compared trees turn out to have the same tuple, then the trees are isomorphic. A setting for our program can indicate the minimum number of vertices that isomorphic subtrees must have to be marked as possible cases of plagiarism.

4.6 Output of results

The similarity scores of all graph comparisons made by Algorithm 4.2 are kept in memory during a run of the program, as well as the isomorphic subtrees that are found by the isomorphism algorithm in the previous section. These results are sorted and printed to the user as a list, with the most interesting and suspicious cases at the top. A reviewer for the given programming assignment can then look at the source code of these submissions manually to determine if there are any real cases of plagiarism. The following is an example of the output that our program gives:

```
12:19:55 Parsed graphs in folder 'C:\...\toolinput':
12:19:55 graph1_original (37 vertices, 95 edges)
12:19:55 graph3_bogus (3 vertices, 8 edges)
12:19:55 graph6_trace (38 vertices, 131 edges)
12:19:55 graph8_unit (38 vertices, 124 edges)
12:19:55 graph9_merge (34 vertices, 80 edges)
12:19:55
12:19:55 Memory limitation settings: maxdepth 80, maxexpansions 100
12:19:55 A* search settings: beamsize 20, pathlength 1.05
12:19:55 Termination criteria: minsimilarity 60.0, maxeditcost 5.0
12:19:55 Graph comparison:
12:19:56 graph1_original, graph1_original: Similarity score 100.0% (38 edits)
```

```
12:19:56 graph3_bogus, graph1_original: Subgraph similarity 70.0% (3 edits), total score 3.68%
12:19:58 graph1_original, graph6_trace: Subgraph similarity 93.28% (40 edits), total score 94.19%
12:20:00 graph1_original, graph8_unit: Subgraph similarity 94.62% (39 edits), total score 94.62%
12:20:03 graph9_merge, graph1_original: Subgraph similarity 80.09% (36 edits), total score 75.95%
12:20:03 graph3_bogus, graph6_trace: Subgraph similarity 70.0% (3 edits), total score 3.58%
12:20:03 graph3_bogus, graph8_unit: Subgraph similarity 70.0% (3 edits), total score 3.58%
12:20:04 graph3_bogus, graph9_merge: Subgraph similarity 70.0% (3 edits), total score 4.57%
12:20:06 graph6_trace, graph8_unit: Subgraph similarity 83.11% (40 edits), total score 85.3%
12:20:09 graph9_merge, graph6_trace: Subgraph similarity 80.09% (36 edits), total score 73.95%
12:20:13 graph9_merge, graph8_unit: Subgraph similarity 77.26% (36 edits), total score 71.3%
12:20:13 Total running time: 18.191 seconds
12:20:13
12:20:13 Results sorted by total score, highest first:
12:20:13 100.0%: graph1_original, graph1_original. Matched functions: ...
12:20:13 94.62%: graph1_original, graph8_unit. Matched functions: ...
12:20:13 94.19%: graph1_original, graph6_trace. Matched functions: ...
12:20:13 85.30%: graph6_trace, graph8_unit. Matched functions: ...
12:20:13 75.95%: graph9_merge, graph1_original. Matched functions: ...
12:20:13 73.95%: graph9_merge, graph6_trace. Matched functions: ...
12:20:13 71.30%: graph9_merge, graph8_unit. Matched functions: ...
12:20:13 4.57%: graph3_bogus, graph9_merge.
12:20:13 3.68%: graph3_bogus, graph1_original.
12:20:13 3.58%: graph3_bogus, graph6_trace.
12:20:13 3.58%: graph3_bogus, graph8_unit.
```

In the example output above, a graph (`graph1_original`) is being compared to refactored copies of itself so that most comparisons yield a high similarity score. All graphs are also compared to a fabricated graph (`graph3_bogus`) that is totally different, but the algorithm recognizes this graph to be very dissimilar to the other graphs.

Chapter 5

Experiments and results

In this chapter, we will demonstrate the effectiveness of our tool when applied to test data. Towards this end, we have set up two main experiments. The first experiment is performed on a single Haskell program that has been copied and modified using refactoring tools, resulting in unequal but similar call graphs. The aim of this experiment is to verify that our tool indeed recognizes these refactored copies as being similar to the original graph. We will also compare our results in this verification process against those of Holmes, described in detail in Vermeer [5].

A second experiment is then carried out in order to see how well our tool performs on actual student submissions for a real programming assignment. In this experiment we are mainly interested in which submissions our tool deems the most similar or interesting and whether or not these submissions are actually cases of plagiarism. The results of this experiment are also compared with those in [5].

In all our experiments, our tree search algorithm uses the following edit operation costs: 1.0 for a vertex substitution of two functions with the same name, 1.1 for a substitution of two isomorphic vertices, 1.2 for other substitutions, 2.0 for a vertex insertion or deletion, and 0.5 for an added or removed edge. These costs make sense because we prefer substitutions over insertions and deletions, and we find edge edit operations less important than vertex edit operations.

5.1 Experiments on refactorings

The call graphs in this experiment are all derived from a single submission for a programming assignment given in the Functional Programming course at Utrecht University in 2005. The submission has been copied and the copies have been refactored using different techniques. The refactorings that have been applied are all typical modifications that a plagiarist could also make. Most of these modifications

were described in Section 2.1.1, but for completeness we will also briefly describe them here:

NameChange - changing the names of functions.

Trace - introducing a `trace` function that is called by all other functions.

TranslateComments - translating code comments to a different language.

Relocation - changing the order of function declarations.

Rewrite - a few simple transformations between certain language-specific constructs, such like converting a `let...in` clause to a `where` clause.

Compact - moving functions that are only called by one other function to the local scope of that function.

Unit - introducing a `unit` function that calls all other functions.

Combinations of these refactoring methods are also applied. For reference scores, the original graph is compared to itself and to a completely dissimilar graph `bogus`. The total similarity scores for comparisons between all these call graphs and the original call graph are listed in Table 5.1. The results for two methods from [5] that Vermeer concludes to be the most generally usable are also included in Table 5.1, to see how they compare to the results of our own tool. Both the token stream method and the fingerprinting method are implemented in the Holmes tool, and the fingerprinting method is implemented in MOSS [11] as well.

Used settings

The settings for our tool during this first experiment were as follows: MAXDEPTH = 80, MAXEXPANSIONS = 100, BEAMSIZE = 100, PATHLENGTH = 1.05, MINSIMILARITY = 60, MAXEDITCOST = 4.0.

5.1.1 Observations

A few observations can be made from the results in table 5.1. For instance, all three compared tools give a 100% score when comparing the original graph to itself, as can be expected. Also, they all give a very low score when comparing the original graph to the totally dissimilar graph `bogus`.

Comparison with	TreeSearch	Tokens	Fingerpr.
original	100	100	100
bogus	4	3	0
nameChange (nc)	85	100	68
trace (trc)	95	85	68
translateComments (tc)	100	100	100
relocation (rl)	100	100	91
rewrite (rw)	95	87	78
compact (cp)	95	86	99
unit (un)	95	91	86
nc+rw	80	87	53
nc+rw+tc	80	87	53
nc+rw+tc+cp	76	77	53
nc+rw+tc+cp+trc	67	74	42
nc+rw+tc+cp+trc+un	60	68	37
nc+rw+tc+cp+trc+un+rl	60	68	36

Table 5.1: *Test results on refactored copies. The total similarity scores that our algorithm gives are in the TreeSearch column, the scores obtained with two methods (Tokens and Fingerprinting) from [5] and [11] are in the last two columns.*

Our tool is insensitive to the *TranslateComments* and *Relocation* modifications, as these modifications give a 100% similarity score when compared to the original graph. This only follows logically from the definition of the call graphs that we use as input: comment sections are not retained in call graphs and the functions contained in a call graph are by definition unordered, so that a different ordering does not have any influence on a used graph algorithm. Note that although the Tokens method is insensitive to these modifications too, the Fingerprinting method does have some trouble with the *Relocation* modification.

Since it becomes harder to compare a graph to its original version with each modification that is applied, all three tools give a lower similarity score for combined refactorings (the lower part of Table 5.1) than for single refactorings. Generally speaking however, our tool seems the least sensitive so the individual refactoring methods *Trace*, *Rewrite*, *Compact* and *Unit*, while the Tokens method seems the least sensitive to changing function names (*nameChange*). The Tokens method also gives better scores on combined refactorings, but this is probably also due to the fact that the other two methods are more sensitive to changed function names.

A possible explanation for the fact that our tool is more sensitive to changed function names than the Tokens method lies in the edge edit operations that are implicitly performed by our algorithm. When our tool applies an edit operation to a vertex, it checks whether or not the edges connected to it need to be modified as well. It does so by simply looking at the function names of the connected vertices (called and calling functions). If these names are different the algorithm assumes that they are different functions and needs to perform more edge edit operations, resulting in a lower similarity score.

A possible way to reduce sensitivity to changed function names would be to let the cost of needed edge edit operations depend on the similarity of the function names, instead of taking a standard cost for all edge edit operations. However, we do not know how this change would affect the general accuracy of our tool. For this reason, we have included this research in the list of future work in Chapter 7.

5.2 Experiments on real data

We performed a second experiment on actual student submissions from a programming assignment called `fp-cal`, given in the 2005 Functional Programming course at Utrecht University. We chose this particular programming assignment because it is also used by Vermeer in [5], again providing us with an excellent opportunity to compare our results against those of Holmes.

The call graphs of 59 student submissions, anonymized and (not uniformly) numbered from `Gr01` to `Gr68`, were processed by our tool resulting in $58 * 59/2 = 1711$ call graph comparisons. The total similarity scores for these comparisons were sorted and outputted in result files.

Table 5.2 indicates the 10 pairs of call graphs that our tool finds the most similar among all compared submissions. The results that Holmes gives for these comparisons are not known to us except for the pair with the highest similarity score: Gr22 vs. Gr34 (see also Table 5.2).

Table 5.2 indicates the pairs of submissions that Holmes finds the most similar among all compared submissions, showing the top 5 results for both the Tokens and Fingerprinting methods. The table also shows the results of our own tool in the column labeled TreeSearch, for the same comparisons. Note that the pairs Gr22 vs. Gr34 and Gr05 vs. Gr40 are the only two pairs that appear in both the top 5 results for Tokens and the top 5 results for Fingerprinting, and that the pair Gr22 vs. Gr34 is the *only* pair that appears in the top results of all three methods.

Comparison	Score
Gr22 vs. Gr34	96
Gr12 vs. Gr37	90
Gr12 vs. Gr60	88
Gr37 vs. Gr60	88
Gr20 vs. Gr58	86
Gr01 vs. Gr20	83
Gr01 vs. Gr58	83
Gr01 vs. Gr53	82
Gr18 vs. Gr39	82
Gr06 vs. Gr53	82

Table 5.2: *Top 10 total similarity scores of all comparisons done by our tool.*

Comparison	TreeSearch	Tokens	Fingerpr.
Gr22 vs. Gr34	96	94	42
Gr05 vs. Gr40	72	74	34
Gr33 vs. Gr40	63	71	30
Gr04 vs. Gr40	64	70	28
Gr05 vs. Gr42	62	69	26
Gr53 vs. Gr59	51	67	32
Gr18 vs. Gr66	46	60	32
Gr12 vs. Gr66	41	54	32

Table 5.3: *The similarity scores given by our tool and by Holmes, containing only the comparisons that appear in the top 5 results for the Tokens and Fingerprinting methods.*

Used settings

The settings for our tool during this second experiment on real data were as follows: MAXDEPTH = 100, MAXEXPANSIONS = 100, BEAMSIZE = 20, PATHLENGTH = 1.1, MINSIMILARITY = 60, MAXEDITCOST = 3.5. These settings are meant to make our algorithm terminate faster than with the settings used in Section 5.1, while making only a minor sacrifice in accuracy as explained in Section 3.4.3. This was necessary in order to keep the total running time feasible (see also Section 5.3 about performance), since most submission call graphs contained more than 50 or even 100 vertices.

5.2.1 Observations

From Tables 5.2 and 5.2, a few interesting observations can be made. Most importantly, all three methods (our own tool, Tokens, and Fingerprinting) have a different set of highest-scoring pairs of submissions. The only comparison that has a high similarity score for all methods is between submissions Gr22 and Gr34. A possible explanation for these different lists of top results is that all three methods show varying sensitivity to all possible types of modifications that can be made to plagiarized programs, as was explained in Section 5.1. In other words, each method has its own strengths and weaknesses.

Based on the fact that submissions Gr22 and Gr34 receive a high similarity score from all methods, we inspected the source code of these two submissions manually and found that they indeed look suspiciously similar. This was also observed by Vermeer in [5]. Although even this cannot provide conclusive evidence that this is in fact a case of plagiarism, our results strongly suggest so.

5.2.2 Distribution of similarity scores

Figure 5.1 shows the similarity scores of all comparisons made by our tool, sorted from high similarity (left) to low similarity (right). Figure 5.2 zooms in on the most similar submissions.

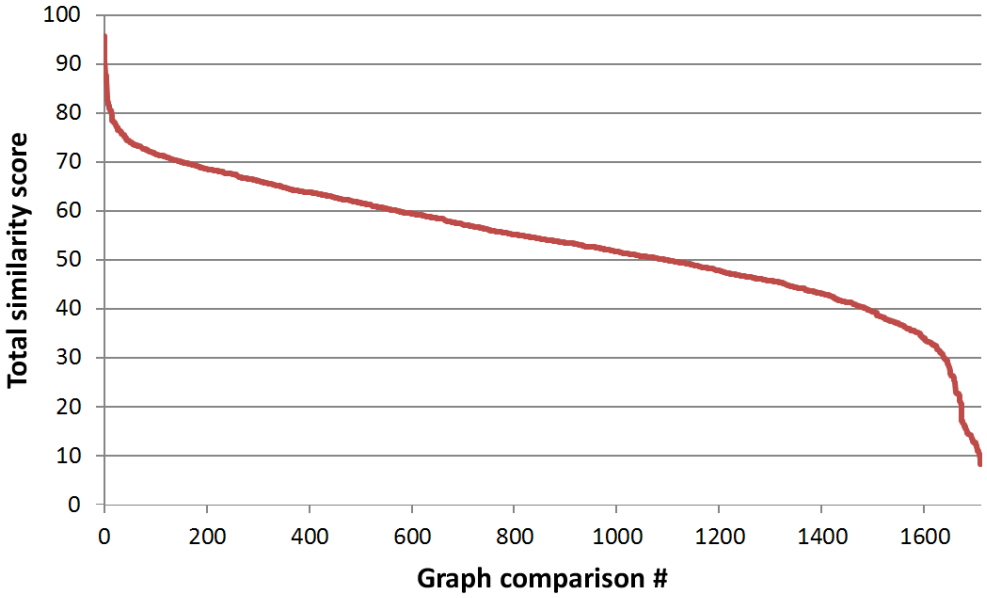

Figure 5.1: *Similarity scores for all 1711 submission comparisons, sorted from high (left) to low similarity (right).*

When looking at the graph in Figure 5.1 it is clear that it shows a linear distribution of similarity scores among submissions, with only two major deviations from this: an upward spike among the most similar submissions and a downward spike among the least similar submissions. What we would expect to see for a programming assignment where each student puts in a serious effort to do the assignment (and on his own), is a simple straight line instead of the graph in Figure 5.1. Some submissions may be more similar than others (resulting in a non-horizontal line), but if there are no pairs of submissions that are accidentally very similar, there should be no large deviations from the general trend.

An explanation for the fact that the similarity score is unexpectedly low for some comparisons is that these submissions may be from students who have either not made a serious effort to do the programming assignment properly. If they approach the programming assignment in the wrong way or if they simply lack the skills to do it right, it is only logical that these submissions will probably be quite dissimilar from other submissions.

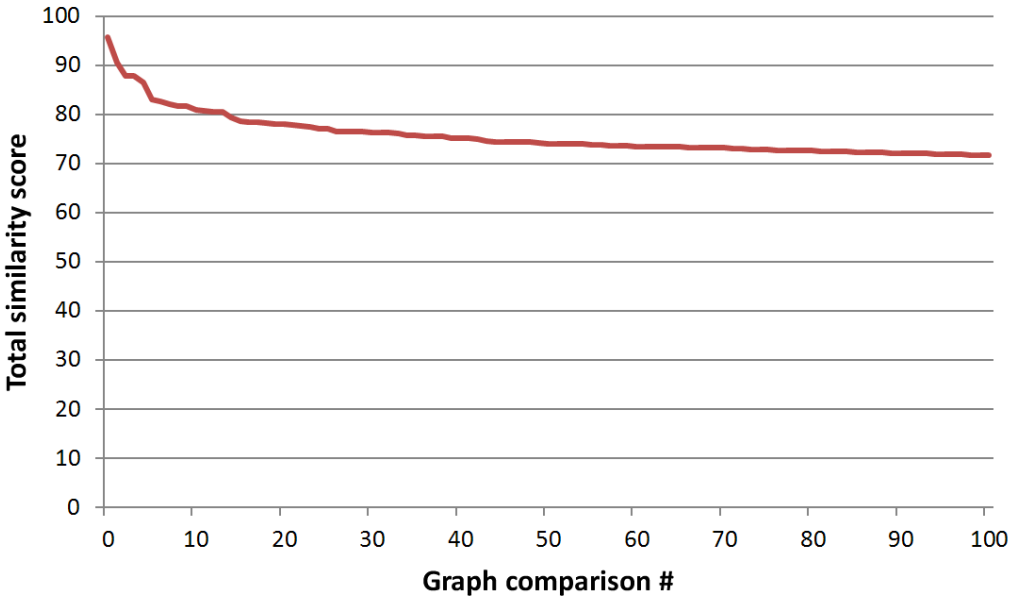

Figure 5.2: *The 100 highest similarity scores among all comparisons.*

It should be noted that regardless of the shape of the graph in Figure 5.1, all similarity scores are relatively high indicating a certain level of similarity among *all* compared graphs. This is due to the fact that in the original programming assignment `fp-cal`, a large set of Haskell code was already given out to students as a starting point. This template code is equal among all submissions. Although in

future assignments a teacher experienced with Sherlock could mark template code as such so that it is filtered out in the parsed call graphs, this was not done before the `fp-cal` assignment was handed out (in 2005). Filtering out template code could help the matching process: our tool only gives us useful information if it indicates a level of similarity between two pieces of code which we didn't already know to be similar. However, trying to figure out which parts of a program were template code after it has already been expanded into a full submission would present a tedious and error prone job.

In Figure 5.2, the leftmost part of the graph in Figure 5.1 is enlarged so that we can have a better view of the highest similarity scores among all submissions. The fact that at some point towards the left of the graph the scores start to rise disproportionately (at about 80% similarity) can logically mean only two things: these submissions are very similar by chance, or they are possible cases of plagiarism. In any case, the results suggest that these pairs of submissions are worth investigating by hand to see if they really are cases of plagiarism.

For the pairs with the highest similarity, also shown in Table 5.2, manual inspection of the source code revealed only one additional case of possible plagiarism besides submissions Gr22 and Gr34: submissions Gr20 and Gr58. This is interesting because these two submissions look suspiciously similar although the Tokens and Fingerprinting methods used in Holmes failed to see this. However, as with the pair Gr22 and Gr34, we cannot give a definitive answer to the question whether or not the submissions Gr20 and Gr58 are in fact plagiarized.

Manual inspection of the other pairs of submissions in Table 5.2 revealed no other suspicious cases. Nonetheless, the graphs in Figures 5.1 and 5.2 clearly illustrate why our program can be a helpful tool in deciding which pairs of submissions deserve further (manual) investigation. Furthermore, Tables 5.2 and 5.2 show that our tool points out different submissions worth investigating than the Tokens and Fingerprinting methods used in Holmes and Moss, so that the methods could be described as complementing each other.

5.2.3 Isomorphic and equal subtrees

A check for isomorphic and/or equal subtrees is also part of our program, in addition to the tree search algorithm for graph edit distance that is executed. On the `fp-cal` submissions used in the previous section, however, only one isomorphic (and equal) subtree was consistently recognized among almost all submissions. This subtree has the `Fql.voorInterval` function as its root node and consists of 7 functions and 4 called Prelude functions. The reason that this same subtree was found in almost all submissions is that these functions were already part of the given assignment.

Although no other isomorphic or equal subtrees were found among compared submissions, this isomorphism check remains a useful addition to the main graph edit distance algorithm because more isomorphic subtrees might be found in submissions for other programming assignments.

5.3 Performance

The experiments described in the previous sections were performed on a low-end desktop PC with a 2 GHz processor with 1 GB of RAM. On this computer, comparing two actual submissions from the used programming assignment took about one minute on average. Comparing all 59 submissions took about a day. One minute per comparison is generally slower than the methods used by Holmes and MOSS, but this is to be expected when executing a tree search algorithm on call graphs of size 50 or more.

For a plagiarism detection tool, performance is not really an issue as explained earlier in Section 2.2 because reviewers that have to grade the programming assignment can do their work concurrently with the execution of our tool. However, if the number of submissions is higher or if the submissions also have to be compared to submissions for the same assignment from previous years, the performance of our tool might become somewhat problematic. For this reason, we added an option for distributed calculation to our program. *Slave threads* can be started in other CPU cores or even on other machines, while the *master* thread delegates comparison calculations to available slave threads. On a quad-core machine this already cuts down total running time to about one third, and even more speed can be gained by using additional machines.

Chapter 6

Summary and conclusions

After a short introduction about software plagiarism, we have looked at the reasons why this form of plagiarism is viewed so negatively in Chapter 2. Although these reasons may seem obvious, software plagiarism occurs quite commonly and as such the need arises for tools that assist us in finding suspected cases of plagiarism. In the same chapter, we have also described how a plagiarist may mask an attempt at software plagiarism and what we require of a tool to help us see through this.

In Chapter 3 we have described how graph matching techniques can be employed in order to detect possible cases of plagiarism. One technique that is particularly well suited to computing graph edit distance on Haskell call graphs is tree search with heuristics. Using these ideas, we have constructed a tool that executes an algorithm based on such tree search techniques and outputs a list of suspected cases of plagiarism. We laid out our approach and design for this program in Chapter 4. In this chapter we also described a few preprocessing modifications to the input graphs that reduce the search space and speed up the algorithm, and we gave some implementation and optimization details regarding the program.

We have run a series of experiments on both deliberately constructed cases of plagiarism and real-life data in the form of actual submissions for a Haskell programming assignment. The results of these experiments have been discussed in Chapter 5 and they show that the tool we have developed is a helpful aid in finding possible cases of plagiarism in addition to the methods used by Holmes. Whereas our tool gives better results seeing through most refactoring techniques, the token stream method used by Holmes is less sensitive to changed function names.

We can conclude that the combined use of both Holmes and our graph edit distance algorithm gives the most accurate results when trying to find possible cases of plagiarism among Haskell programs. However, as with all automated tools that help us find possible cases of plagiarism, the suspicious cases that they find still have

to be investigated by hand in order to determine whether or not they are actually cases of plagiarism. Our tool is now ready to be used for other Haskell programming assignments at Utrecht University so that we can further investigate its effectiveness in practice.

A few questions remain open for future research, most of which we will discuss in the next chapter. The most important research questions are how we can improve the accuracy and performance of our algorithm even further, and how we can modify our tool so that it becomes less sensitive to changed function names.

Chapter 7

Future work

In the list below we briefly describe some of the ideas we have for possible future research, based on our work in this thesis.

- Write our tool in Haskell or integrate it with Holmes in another way. This way, both applications would not have to be run separately. However, converting the Java source code of our program to Haskell code may be problematic.
- Investigate how the addition of type information and local function definitions in parsed call graphs would influence the matching process exactly.
- Instead of performing our algorithm on call graphs, construct tree representations of the call graphs and execute a more efficient algorithm on these, and see how this affects accuracy and how much faster this is.
- Remove cycles of size 3 or bigger from the call graphs in the preprocessing phase. This would facilitate the use of algorithms that take directed acyclic graphs as input, which are generally of a lower complexity class than algorithms that work on more general classes of graphs.
- Implement more advanced heuristics for the A* search procedure, see how this influences the algorithm. A possible heuristic could be the bipartite matching algorithm from Riesen et al. [34], but other algorithms that work on directed acyclic graphs may be used as well.
- Let the cost of implicitly needed edge edit operations depend on the degree of similarity of function names instead of using a standard cost for all edge edit operations. It should be carefully investigated how this influences the general accuracy of the graph edit distance algorithm, and how this influences the sensitivity to changed function names.

Bibliography

[1] Stichting Onderwijs Evaluatie Rapport. A study on plagiarism among students of Utrecht University (een onderzoek naar plagiaat onder studenten aan de Universiteit Utrecht). Technical report, Utrecht University, June 2005. www2.hum.uu.nl/solis/ict-centrum/Archief/OERrapport-plagiaat.pdf.
[cited on page 9]

[2] John Simpson and Edmund Weiner. *The Oxford English Dictionary*. Oxford University Press, 1989.
[cited on pages 9 and 10]

[3] Jack Lynch. The perfectly acceptable practice of literary theft: Plagiarism, copyright, and the eighteenth century. *Colonial Williamsburg: The Journal of the Colonial Williamsburg Foundation*, 24(4):51–54, 2002.
[cited on page 9]

[4] Education and examination regulations 2009-2010, Utrecht Graduation School of Natural Sciences. http://www.uu.nl/faculty/science/en/education/masterphd/gsons/organisation/Documents/EER2009-2010.pdf.
[cited on page 10]

[5] Brian Vermeer. Holmes: Hunting for Haskell frauds. Master's thesis, Utrecht University, 2010.
[cited on pages 10, 13, 23, 24, 25, 46, 61, 62, 63, 64, and 66]

[6] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman & Co Ltd, January 1979.
[cited on pages 12 and 33]

[7] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the Association for Computing Machinery*, 21:168–173, January 1974.
[cited on page 12]

[8] Hoogle, the Haskell API search engine. http://www.haskell.org/hoogle.
[cited on page 12]

[9] Jurriaan Hage. Plagiarism detection in programming assignments with Marble (programmeerplagiaatdetectie met Marble). Technical report, Department of Information and Computing Sciences, Utrecht University, 2006.
[cited on pages 15, 22, and 55]

[10] Huiqing Li. *Refactoring Haskell programs*. PhD thesis, University of Kent, 2006.
[cited on page 18]

[11] MOSS: A system for detecting software plagiarism, 1994. http://theory.stanford.edu/~aiken/moss/.
[cited on pages 20, 62, and 63]

[12] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 76–85. ACM, 2003.
[cited on pages 21, 22, and 24]

[13] Wikipedia page on Sherlock Holmes. http://en.wikipedia.org/wiki/Sherlock_Holmes.
[cited on page 23]

[14] Wiki page of Utrecht University on Helium. http://www.cs.uu.nl/wiki/Helium.
[cited on page 24]

[15] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, June 1959.
[cited on page 28]

[16] Dennis H. Rouvray and Alexandru T. Balaban. *Applications of Graph Theory*, chapter Chemical applications of graph theory, pages 177–221. Academic Press, 1979.
[cited on page 29]

[17] Wikipedia page on caffeine. http://en.wikipedia.org/wiki/Caffeine.
[cited on page 31]

[18] Michel Neuhaus and Horst Bunke. A graph matching based approach to fingerprint classification using directional variance. In Takeo Kanade, Anil Jain, and Nalini Ratha, editors, *Audio- and Video-Based Biometric Person Authentication*, volume 3546 of *Lecture Notes in Computer Science*, pages 191–200. Springer Berlin / Heidelberg, 2005.
[cited on page 29]

[19] Edward K. Wong. *Syntactic and Structural Pattern Recognition-Theory and Applications*, chapter Three-dimensional object recognition by attributed graphs, pages 381–414. World Scientific, 1990.
[cited on page 29]

[20] Edward K. Wong. Model matching in robot vision by subgraph isomorphism. *Pattern Recognition*, 25:287–304, 1992.
[cited on page 29]

[21] Horst Bunke. Graph matching: Theoretical foundations, algorithms, and applications. *International Conference on Vision Interface*, pages 82–88, 2000.
[cited on pages 30, 38, and 53]

[22] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(3):265–298, 2004.
[cited on page 30]

[23] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Using code normalization for fighting self-mutating malware. In *Proceedings of International Symposium on Secure Software Engineering*, Washington, DC, USA, March 2006. IEEE.
[cited on page 31]

[24] Christopher Krügel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In Alfonso Valdes and Diego Zamboni, editors, *Recent Advances in Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*, pages 207–226. Springer Berlin / Heidelberg, 2006.
[cited on page 31]

[25] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881, New York, NY, USA, 2006. ACM.
[cited on page 31]

[26] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.
[cited on page 32]

[27] Hans L. Bodlaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. In *SWAT 88: 1st Scandinavian workshop on*

algorithm theory, number 318, pages 223–232, London, UK, 1988. Springer-Verlag.
[cited on page 32]

[28] Alfred. V. Aho, John. E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Boston, MA, USA, 1974.
[cited on pages 32 and 59]

[29] Horst Bunke, Xiaoyi Jiang, and Abraham Kandel. On the minimum common supergraph of two graphs. *Computing*, 65:13–25, 2000.
[cited on page 33]

[30] Horst Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, 18(9):689–694, 1997.
[cited on page 34]

[31] Kaizhong Zhang. *The editing distance between trees: algorithms and applications*. PhD thesis, New York University, 1989.
[cited on page 34]

[32] Ronald L. Rivest and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, Inc., New York, NY, USA, 1990.
[cited on pages 34 and 36]

[33] Michel Neuhaus, Kaspar Riesen, and Horst Bunke. Fast suboptimal algorithms for the computation of graph edit distance. In *Lecture Notes in Computer Science*, volume 4109, pages 163–172. Springer Berlin / Heidelberg, 2006.
[cited on pages 35, 39, and 41]

[34] Kaspar Riesen, Michel Neuhaus, and Horst Bunke. Bipartite graph matching for computing the edit distance of graphs. In *Proceedings of the 6th IAPR-TC-15 international conference on Graph-based representations in pattern recognition*, pages 1–12, Berlin, Heidelberg, 2007. Springer-Verlag.
[cited on pages 35, 42, 53, and 73]

[35] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.
[cited on page 36]

[36] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, February 1968.
[cited on page 37]

[37] Bruno T. Messmer and Horst Bunke. A decision tree approach to graph and subgraph isomorphism detection. *Pattern Recognition*, 32(12):1979–1998, 1999.
[cited on page 38]

[38] Bruno T. Messmer and Horst Bunke. Error-correcting graph isomorphism using decision trees. *International Journal of Pattern Recognition and Artificial Intelligence (IJPRAI)*, 12(6):721–742, September 1998.
[cited on pages 38 and 39]

[39] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision Computing*, 27(7):950–959, 2009. 7th IAPR-TC15 Workshop on Graph-based Representations (GbR 2007).
[cited on page 42]

[40] Harold W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(2):83–97, 1955.
[cited on page 42]

[41] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, March 1957.
[cited on page 42]

[42] Kaizhong Zhang. A constrained edit distance between unordered labeled trees. *Algorithmica*, 15:205–222, 1996.
[cited on page 43]

[43] W.J. Christmas, J. Kittler, and M. Petrou. Structural matching in computer vision using probabilistic relaxation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17(8):749–764, August 1995.
[cited on page 43]

[44] Andrew D.J. Cross, Richard C. Wilson, and Edwin R. Hancock. Inexact graph matching using genetic search. *Pattern Recognition*, 30(6):953–970, 1997.
[cited on page 43]

[45] Derek Justice and Alfred Hero. A binary linear programming formulation of the graph edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28:1214, 2006.
[cited on page 43]

[46] Kaspar Riesen, Stefan Fankhauser, and Horst Bunke. Speeding up graph edit distance computation with a bipartite heuristic. In *Mining and Learning with Graphs*, August 2007.
[cited on pages 44 and 53]

[47] DOT language for graph files. http://en.wikipedia.org/wiki/DOT_language.
[cited on page 83]

List of Figures

Appendix A

DOT graph files

The input graph files we use are standard DOT files [47]. An example of such a graph file is listed below, with all vertices defined first and all edges (the lines with -> in them) following after.

```
digraph graph1_original
{
  "Fql.maxWidth'" [label="Fql.maxWidth'"]
  "Fql.greatest" [label="Fql.greatest"]
  "Prelude.++" [label="Prelude.++"]
  "Fql.recordWidth" [label="Fql.recordWidth"]
  "Fql.singleWidth" [label="Fql.singleWidth"]
  "Prelude.length" [label="Prelude.length"]
  "Prelude.>=" [label="Prelude.>="]
  "Prelude.otherwise" [label="Prelude.otherwise"]
  "Prelude.return" [label="Prelude.return"]
  "Prelude.putStrLn" [label="Prelude.putStrLn"]
  "Fql.printRecords" [label="Fql.printRecords"]
  "Fql.printRecord" [label="Fql.printRecord"]
  "Fql.addChar" [label="Fql.addChar"]
  "Prelude.-" [label="Prelude.-"]
  "Fql.bigStripe" [label="Fql.bigStripe"]
  "Fql.maxWidth" [label="Fql.maxWidth"]
  "Fql.allWidth" [label="Fql.allWidth"]
  "Fql.project'" [label="Fql.project'"]
  "Fql.addColumn" [label="Fql.addColumn"]
  "Fql.getColumn" [label="Fql.getColumn"]
  "Fql.getColumn'" [label="Fql.getColumn'"]
  "Fql.getPosition" [label="Fql.getPosition"]
  "Prelude.==" [label="Prelude.=="]
  "Prelude.+" [label="Prelude.+"]
  "Fql.getPositions" [label="Fql.getPositions"]
  "Fql.emptyLists" [label="Fql.emptyLists"]
  "Prelude.<" [label="Prelude.<"]
  "Fql.filterPositions" [label="Fql.filterPositions"]
  "Fql.boolRecords" [label="Fql.boolRecords"]
  "Fql.boolRecord" [label="Fql.boolRecord"]
  "Fql.boolSingle" [label="Fql.boolSingle"]
  "Fql.leaveOut" [label="Fql.leaveOut"]
  "Fql.joinRecordTable" [label="Fql.joinRecordTable"]
  "Fql.joinRecord" [label="Fql.joinRecord"]
  "Fql.getValue" [label="Fql.getValue"]
  "Fql.joinTableTable" [label="Fql.joinTableTable"]
```

```
"Fql.compareTables" [label="Fql.compareTables"]
"Fql.tableWidth" [label="Fql.tableWidth"]
"Fql.count" [label="Fql.count"]
"Prelude.error" [label="Prelude.error"]
"Fql.snd2" [label="Fql.snd2"]
"Fql.fst2" [label="Fql.fst2"]
"Fql.top10" [label="Fql.top10"]
"Fql.genre" [label="Fql.genre"]
"Fql.printTable" [label="Fql.printTable"]
"Fql.project" [label="Fql.project"]
"Fql.select" [label="Fql.select"]
"Fql.join" [label="Fql.join"]

"Fql.maxWidth'"->"Fql.maxWidth'"
"Fql.maxWidth'"->"Fql.greatest"
"Fql.recordWidth"->"Prelude.++"
"Fql.recordWidth"->"Fql.singleWidth"
"Fql.recordWidth"->"Fql.recordWidth"
"Fql.singleWidth"->"Prelude.++"
"Fql.singleWidth"->"Prelude.length"
"Fql.singleWidth"->"Fql.singleWidth"
"Fql.greatest"->"Prelude.>="
"Fql.greatest"->"Prelude.++"
"Fql.greatest"->"Fql.greatest"
"Fql.greatest"->"Prelude.otherwise"
"Fql.printRecords"->"Prelude.return"
"Fql.printRecords"->"Prelude.putStrLn"
"Fql.printRecords"->"Fql.printRecords"
"Fql.printRecords"->"Fql.printRecord"
"Fql.printRecord"->"Prelude.++"
"Fql.printRecord"->"Fql.addChar"
"Fql.printRecord"->"Prelude.-"
"Fql.printRecord"->"Prelude.length"
"Fql.printRecord"->"Fql.printRecord"
"Fql.addChar"->"Fql.addChar"
"Fql.addChar"->"Prelude.-"
"Fql.bigStripe"->"Prelude.++"
"Fql.bigStripe"->"Fql.addChar"
"Fql.bigStripe"->"Fql.bigStripe"
"Fql.maxWidth"->"Fql.maxWidth'"
"Fql.allWidth"->"Prelude.++"
"Fql.allWidth"->"Fql.recordWidth"
"Fql.allWidth"->"Fql.singleWidth"
"Fql.project'"->"Fql.project'"
"Fql.project'"->"Fql.addColumn"
"Fql.project'"->"Fql.getColumn"
"Fql.addColumn"->"Prelude.++"
"Fql.addColumn"->"Fql.addColumn"
"Fql.getColumn"->"Prelude.++"
"Fql.getColumn"->"Fql.getColumn'"
"Fql.getColumn"->"Fql.getColumn"
"Fql.getPosition"->"Prelude.=="
"Fql.getPosition"->"Prelude.otherwise"
"Fql.getPosition"->"Prelude.+"
"Fql.getPosition"->"Fql.getPosition"
"Fql.getPositions"->"Prelude.++"
"Fql.getPositions"->"Fql.getPosition"
"Fql.getPositions"->"Fql.getPositions"
"Fql.emptyLists"->"Prelude.++"
"Fql.emptyLists"->"Fql.emptyLists"
"Fql.emptyLists"->"Prelude.-"
"Fql.getColumn'"->"Prelude.=="
"Fql.getColumn'"->"Prelude.otherwise"
"Fql.getColumn'"->"Fql.getColumn'"
```

```
  "Fql.getColumn'"->"Prelude.-"
  "Fql.filterPositions"->"Prelude.<"
  "Fql.filterPositions"->"Prelude.++"
  "Fql.filterPositions"->"Fql.filterPositions"
  "Fql.filterPositions"->"Prelude.otherwise"
  "Fql.boolRecords"->"Prelude.++"
  "Fql.boolRecords"->"Fql.boolRecords"
  "Fql.boolRecords"->"Prelude.otherwise"
  "Fql.boolRecords"->"Fql.boolRecord"
  "Fql.boolRecord"->"Fql.boolSingle"
  "Fql.boolRecord"->"Fql.boolRecord"
  "Fql.boolRecord"->"Prelude.-"
  "Fql.leaveOut"->"Prelude.=="
  "Fql.leaveOut"->"Fql.leaveOut"
  "Fql.leaveOut"->"Prelude.-"
  "Fql.leaveOut"->"Prelude.otherwise"
  "Fql.leaveOut"->"Prelude.++"
  "Fql.joinRecordTable"->"Prelude.=="
  "Fql.joinRecordTable"->"Prelude.++"
  "Fql.joinRecordTable"->"Fql.joinRecord"
  "Fql.joinRecordTable"->"Fql.joinRecordTable"
  "Fql.joinRecordTable"->"Prelude.otherwise"
  "Fql.joinRecordTable"->"Fql.getValue"
  "Fql.joinTableTable"->"Prelude.++"
  "Fql.joinTableTable"->"Fql.joinRecordTable"
  "Fql.joinTableTable"->"Fql.joinTableTable"
  "Fql.compareTables"->"Prelude.<"
  "Fql.compareTables"->"Prelude.otherwise"
  "Fql.compareTables"->"Fql.compareTables"
  "Fql.compareTables"->"Prelude.+"
  "Fql.compareTables"->"Fql.getPosition"
  "Fql.compareTables"->"Prelude.length"
  "Fql.joinRecord"->"Prelude.++"
  "Fql.joinRecord"->"Fql.leaveOut"
  "Fql.getValue"->"Fql.getValue"
  "Fql.getValue"->"Prelude.-"
  "Fql.tableWidth"->"Fql.maxWidth"
  "Fql.tableWidth"->"Fql.allWidth"
  "Fql.count"->"Prelude.length"
  "Fql.printTable"->"Prelude.putStrLn"
  "Fql.printTable"->"Prelude.++"
  "Fql.printTable"->"Fql.printRecords"
  "Fql.printTable"->"Fql.printRecord"
  "Fql.printTable"->"Fql.bigStripe"
  "Fql.printTable"->"Fql.tableWidth"
  "Fql.project"->"Fql.project'"
  "Fql.project"->"Fql.getPositions"
  "Fql.project"->"Fql.emptyLists"
  "Fql.project"->"Fql.count"
  "Fql.select"->"Fql.boolRecords"
  "Fql.select"->"Fql.getPosition"
  "Fql.join"->"Prelude.=="
  "Fql.join"->"Prelude.error"
  "Fql.join"->"Prelude.otherwise"
  "Fql.join"->"Prelude.++"
  "Fql.join"->"Fql.joinRecord"
  "Fql.join"->"Fql.joinTableTable"
  "Fql.join"->"Fql.compareTables"
  "Fql.join"->"Fql.snd2"
  "Fql.join"->"Fql.fst2"
}
```