Nelson Mandela
Metropolitan
University

for tomorrow

Department of Computer Science & Information Systems
Nelson Mandela Metropolitan University

Assessing Program Code through Static Structural Similarity

Kevin Alexander Naudé

January 2007

Supervisor:	Prof. Jean Greyling
Co-supervisor:	Mr. Dieter Vogts

Submitted in partial fulfilment
of the requirements for the degree
Magister Scientiae
in the Faculty of Science at the
Nelson Mandela Metropolitan University

Abstract

Learning to write software requires much practice and frequent assessment. Consequently, the use of computers to assist in the assessment of computer programs has been important in supporting large classes at universities. The main approaches to the problem are dynamic analysis (testing student programs for expected output) and static analysis (direct analysis of the program code). The former is very sensitive to all kinds of errors in student programs, while the latter has traditionally only been used to assess quality, and not correctness.

This research focusses on the application of static analysis, particularly structural similarity, to marking student programs. Existing traditional measures of similarity are limiting in that they are usually only effective on tree structures. In this regard they do not easily support dependencies in program code. Contemporary measures of structural similarity, such as similarity flooding, usually rely on an internal normalisation of scores. The effect is that the scores only have relative meaning, and cannot be interpreted in isolation, ie. they are not meaningful for assessment. The *SimRank* measure is shown to have the same problem, but not because of normalisation. The problem with the *SimRank* measure arises from the fact that its scores depend on all possible mappings between the children of vertices being compared.

The main contribution of this research is a novel graph similarity measure, the *Weighted Assignment Similarity* measure. It is related to *SimRank*, but derives propagation scores from only the locally optimal mapping between child vertices. The resulting similarity scores may be regarded as the percentage of mutual coverage between graphs. The measure is proven to converge for all directed acyclic graphs, and an efficient implementation is outlined for this case.

Attributes on graph vertices and edges are often used to capture domain specific information which is not structural in nature. It has been suggested that these should influence the similarity propagation, but no clear method for doing this has been reported. The second important contribution of this research is a general method for incorporating these local attribute similarities into the larger similarity propagation method.

An example of attributes in program graphs are identifier names. The choice of identifiers in programs is arbitrary as they are purely symbolic. A problem facing any comparison between programs is that they are unlikely to use the same set of identifiers. This problem indicates that a mapping between the identifier sets is required. The third contribution of this research is a method for applying the structural similarity measure in a two step process to find an optimal identifier mapping. This approach is both novel and valuable as it cleverly reuses the similarity measure as an existing resource.

In general, programming assignments allow a large variety of solutions. Assessing student programs through structural similarity is only feasible if the diversity in the solution space can be addressed. This study narrows program diversity through a set of semantic preserving program transformations that convert programs into a normal form.

The application of the *Weighted Assignment Similarity* measure to marking student programs is investigated, and strong correlations are found with the human marker. It is shown that the most accurate assessment requires that programs not only be compared with a set of good solutions, but rather a mixed set of programs of varying levels of correctness.

This research represents the first documented successful application of structural similarity to the marking of student programs.

Acknowledgements

I would like to express special thanks to:

Professor Jean Greyling, and Dieter Vogts for supervising my research. They provided a great deal of valuable advice and constructive criticism and this work is much better for their efforts.

Professor André Calitz who volunteered to advise me over a period during which neither Jean nor Dieter was going to be available. His advice was very valuable, both for its thoroughness and for the new perspective.

The NRF Thuthuka Research Program for financial support of this research.

All my friends and colleagues who have often shown keen interest in my work. In this regard, special thanks are extended to Doctor Charmain Cilliers, Professor Gideon de Kock, M.C. du Plessis, and Peter Dobson.

My family, who have always possessed an inexhaustible faith in my abilities, and kept many of the mundane matters of life from interfering with my work. Without their efforts and encouragement, this work would still be ongoing.

And finally, God, who showed me an enormous number of small mercies throughout my work.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

This research concerns the use of computers for the automated assessment of computer programs – particularly those developed by novice programmers in academic environments. The problem is a difficult one because any given programming assignment is expected to have a very large solution space. In most cases, there are an infinite number of solutions. However, the number of representative or *important* solutions is not necessarily so large.

This chapter serves to introduce the problems associated with the assessment of student programs. It also serves to give an overview of the current research, including research questions, the method of research, and important contributions made.

1.1 Background

Programming classes in Computer Science departments are often large. This remains the case globally in spite of some recent decline in enrolment figures that have been observed (Higgins *et al.*, 2005). For example, Lass *et al.* (2003) describe having programming classes of several hundred students each. Learning to program requires a great deal of practice, and large classes generate an enormous number of program assignments that must be marked.

To cope with the workload, Lass *et al.* (2003) describe that 1 teaching assistant is employed for every 25 students. Preston & Shackelford (1999) discuss a similar scenario with over 100 teaching assistants employed. These assistants mark over 4000 programming assignments *per week*. While it would be most beneficial if professors and lecturers could mark the assignments of their students, the shear volume makes this impossible.

At the author's own institution, the Nelson Mandela Metropolitan University (NMMU), the situation is similar. The Computer Science & Information Systems (CS&IS) department offers a traditional Computer Science curriculum, with about 140 students participating in the first undergraduate programming module each year. Over the first semester of programming, students typically complete 30 different programming tasks. This results in between 300 and 600 programs being submitted per week. Assistants are employed to check the assignments and give feedback. While the assistants are more experienced than the students, their experience does vary. This means that the feedback students receive regarding their programs is limited (and occasionally unreliable).

Dividing the assessment responsibility between a large number of assistants causes additional problems. Preston (1997) discuss some of the consistency problems that result, with similar programs receiving widely different reviews and scores from different assistants. Not being able to trust the evaluation process undermines the value of the learning activity. Thus there are at least four substantial problems in assessing student programs in large classes.

i. **Time**
   Providing a careful and meaningful assessment of student programs takes a great deal of time. Usually this is more than an educator can spare. Even if the educator can afford to take on the work, the turnaround time is expected to be substantial.

ii. **Staffing**
   The turnaround time can be reduced, and the educator's time spared, by taking on a large workforce – as is commonplace. However, this staffing can be costly, and a substantial pool of ideal candidates for teaching assistant positions may not be available. Appointing masters students into teaching assistant positions may create secondary problems, such as slowing their graduation rate.

iii. **Consistency**

Consistency is a two-fold problem. When marking work is divided between several assessors, it is almost impossible for each to apply the same standard. Unfortunately, the consistency problems can occur even if there is only one marker as self-consistency is not guaranteed for each assessor. In particular, the standard applied at the beginning of a marking session may not survive until the end.

iv. **Quality of Feedback**

As with marking consistency, quality of feedback is expected to vary between markers, and over time for the same marker. Discipline is required to write substantial and useful feedback for each submitted program.

Programs written by students are not always marked by hand. Many Computer Science departments of universities and colleges throughout the world already use some form of computer aided assessment (CAA). Carter *et al.* (2003) report on a recent international survey of academics[1], finding that 41% of respondents used a CAA system some of the time, with a further 21% using CAA extensively. The remaining 38% marked program assignments exclusively by hand.

CAA has the potential to address all of the problems associated with human marking – although some of the problems are easier to address than others. Since computer time is cheap, CAA systems address *time* and *staffing* problems very effectively. It is widely held that one of the most compelling features of CAA is the fast turnaround time of assessments. This is perhaps the most valuable feature to both students and educators – in spite of some doubt over the *quality of feedback* received (Carter *et al.*, 2003). *Quality of feedback* remains a concern in CAA systems in that the provision of good feedback is an exception, not the rule.

An important consideration is that CAA systems are very *consistent* in how they assign marks. Using CAA as an unbiased judge of student work is perhaps the second most valuable feature of CAA systems. While CAA systems are consistent, it is important to observe that they are not necessarily objective. Objectivity and subjectivity are terms that cannot be attributed to CAA systems, as they lack any ability to reason about student programs. Consider the simple example of a student program that meets all stated requirements. A CAA system should assign it a 100% score. If a small syntactic

[1] Respondents were predominantly from the UK, Finland, Australia, and the USA.

error is then introduced, the CAA system may not be able to reach any conclusion, and so assigns a 0% score. In contrast, an objective human marker would rate the program differently. Fortunately, having a basic idea of how a given CAA system performs assessment, may allow students to avoid such obvious problems. Furthermore, this kind of problem is not indicative of all CAA systems.

CAA systems can offer additional advantages. Distance education students have special needs, such as being able to do assignments when it suits their schedules. CAA systems make distance education more feasible because students can get instant feedback over the Internet, even after normal working hours. The advantage over staffing problems also becomes more important in this case.

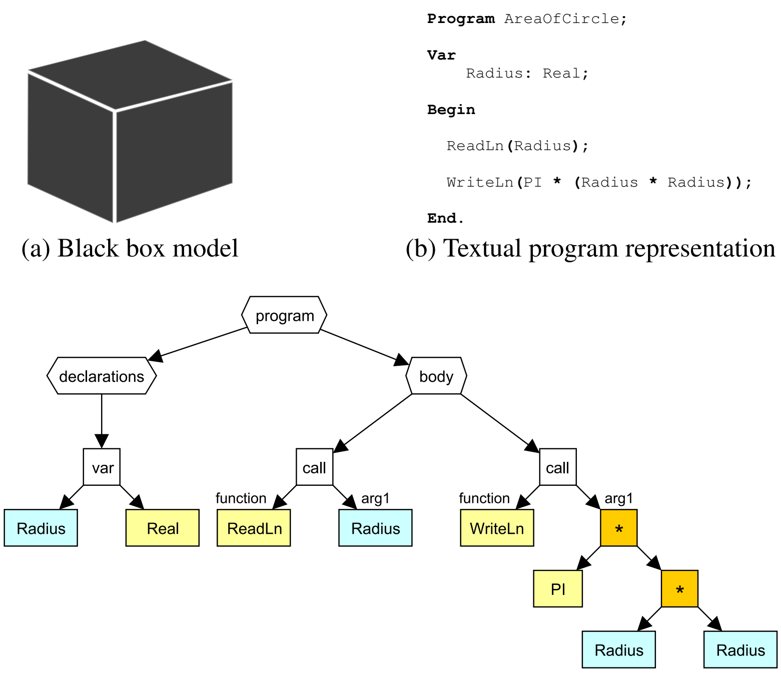
1.2 Overview of Computer Aided Assessment

A wide variety of CAA measures have been developed over the years, and this is discussed in depth in chapter 2. Nevertheless, the broad kinds of automated assessment are also described here. A distinguishing characteristic of CAA measures is how they view the student program. *Figures 1.1a*, *b* and *c* show how a program that calculates the area of a circle may be viewed by CAA systems. This program will facilitate discussion of the different CAA techniques.

With the black box model (*Figure 1.1a*) the CAA system has no understanding of how the program is constructed. In the example, it may only know that if it executes the program with the input 10.0, it should expect a number near 314.159 as output. The process of executing a program to observe its behaviour and output is called *dynamic analysis*. Other CAA measures treat programs as unstructured text (*Figure 1.1b*), or as graphs of structured data (*Figure 1.1c*). Because these techniques operate on static representation of the program, they are forms of *static analysis*.

1.2.1 Dynamic Analysis

Dynamic analysis, or program testing, is by far the most popular and prevalent of CAA techniques. The typical scenario is that the lecturer or teaching assistant will specify a variety of test cases (with input and output) that correct programs will satisfy. A batch



```
Program AreaOfCircle;

Var
    Radius: Real;

Begin

  ReadLn(Radius);

  WriteLn(PI * (Radius * Radius));

End.
```

(a) Black box model

(b) Textual program representation

(c) Graphical program representation

Figure 1.1: Program representations as seen by CAA systems

process can then execute the student programs against each test case. Students obtain a score based on the number of tests their programs pass.

The main problem with this method is that it is particularly vulnerable to program errors. The student program cannot be tested if it contains any syntax errors. In addition, the program cannot contain any contextual errors. Programs with logic errors may be testable, but if a logic error causes excessive memory allocation, or an endless loop, the testing of the program must be abandoned.

1.2.2 Static Analysis

Static analysis CAA measures have traditionally emphasised measuring program quality, rather than correctness. In this regard, measures that operate on a textual representation of a program determine important program characteristics by counting surface features. For example, a CAA measure may calculate the function density, or degree of decomposition, by finding the average number of lines of code per function.

Structural analysis is another form of static analysis which holds potential for program assessment. This technique involves the study of the structural relationships in the program, so it requires a structured graph representation. Some work has been done in this area, with recent efforts by Saikkonen *et al.* (2001) and Truong *et al.* (2004), but much more research is needed.

Structural analysis is of particular interest in this research. The main objective is to determine whether a structural similarity measure can be used effectively in assessing student programs. Given a candidate program and an ideal solution program, a structural similarity measure should find the degree of similarity between them, which may be interpreted as the awarded mark.

The advantages of structural techniques is that they can assess a more diverse set of submitted programs. In general, structural analysis can still operate even if the programs contain errors. Neither contextual errors, nor endless loops work against the assessment technique. In addition, if the program contains syntax errors, but good parser error recovery is implemented, the technique can possibly still be of use.

1.3 Objectives & Research Questions

In recent research, several promising graph similarity measures have been developed, for example the *Similarity Flooding* measure due to Melnik *et al.* (2002), and the *SimRank* measure due to Jeh & Widom (2002). The primary objective of this research is to investigate the use of graph similarity as the basis for a new structural similarity assessment measure.

Objectives may be summarised as follows:

1. To investigate the different techniques employed in computer aided assessment of programs.
2. To investigate traditional and contemporary measures of similarity between graphs.
3. To investigate the use of graph similarity as the basis for a new structural similarity assessment measure.

Clear primary and secondary research questions arise from these objectives:

1. To what extent can the structural similarity between candidate programs and ideal solutions be used in program code assessment?
    (a) What similarity measures available may be applied to program code assessment?
    (b) How may available similarity measures be adapted to better suit the problem of assessing student programs?

1.4 Research Contributions

This research identifies the problems that existing graph similarity measures have when applied to assessment. The most important of these is that the similarity scores produced are only significant relative to one another. This is critical for assessment as it disallows the interpretation of the scores as percentages. However, the *SimRank* measure (Jeh & Widom, 2002) is found to be amenable to adaption to the problem domain.

The main contribution of this research is to replace the summation procedure within the *SimRank* measure with one that considers only the locally optimal assignment between neighbour vertices. This gives rise to a novel graph similarity measure called *Weighted Assignment Similarity*. This change is valuable as it is shown to produce scores which have intrinsic meaning analogous to percentages.

Another important contribution is the first formal mechanism for including domain specific attribute similarity into the broader structural similarity measure. This allows the similarity measure to be directly influenced by domain specific relationships captured in the attributes of vertices and edges. For example, in program assessment vertices may contain operator labels and constant values that, being compared, may guide the similarity measure.

The third contribution is an effective and general method for mapping identifiers between two similar programs. A holistic similarity measure between programs is achieved by using the *Weighted Assignment Similarity* measure twice: once on derived graphs to find the similarity between identifier pairs, and a second time on the original graphs with the aid of the new information.

1.5 Method, Limitations and Delimitation

The method followed in the case study of this research is to apply the new *Weighted Assignment Similarity* measure, as well as the *SimRank* measure to the programs written by 1st-year students as a practical assignment.

The correlation between the human marker and the scores derived from the *Weighted Assignment Similarity* measure are of particular interest. In addition, the correlation between the human marker and the *SimRank* scores are determined for comparison. Furthermore, the deviations between the human marker scores and those of new measure are studied.

An important limitation of the research is that we assume the human marker marked consistently and accurately. Essentially it is assumed that the human marker assigned perfectly chosen scores to each submitted program.

An important delimitation of this study is that the existing conditions under which practical assignments are done were not altered. As students had a week of unsupervised activity to complete the assignment, the possibility of plagiarism was very real. This was addressed by eliminating duplicate programs, ignoring comments and whitespace.

A delimitation to narrow the scope of the research is to exclude from study the effects of parser recovery. Student programs may contain errors, but these are ignored as long as the parser is able to make a reasonable recovery. The significance of possibly losing information this way is not studied. In addition, the effect of different error recovery strategies on the utility of the similarity measure is not investigated.

1.6 Organisation of this Dissertation

Chapter 2 explores the literature relating to computer aided assessment. Both static and dynamic techniques are reported on, as well as more unusual strategies. The relative strengths and weaknesses of the different methods are considered, and the opportunities for further research relating to structural analysis are observed.

Chapter 3 investigates some of the many traditional measures of similarity between structured data. Following this, the contemporary similarity propagation measures are considered in detail. All of these measures are discussed in relation to the current problem domain. It is observed that the existing techniques are not directly appropriate, but may be adapted to support use in assessment.

Chapter 4 describes the *Weighted Assignment Similarity* measure, which is a novel similarity measure between graphs. It is the first published similarity propagation measure that directly supports vertex and edge attributes with domain specific similarity. A proof of convergence is presented for directed acyclic graphs, and an efficient implementation strategy is described.

Chapter 5 describes the experiment of applying the new measure in the assessment of student programs. The chapter first identifies all important sources of program diversity, which cause problems for assessment. Many of these sources of diversity are addressed by transformations which convert the programs into a normal form. The chapter continues to describe the experiments in applying the structural similarity measure to assessing the programs. The results show that strong correlations with human marks are possible.

Chapter 6 summarises the contributions made, and concludes the research.

Chapter 2

Computer Aided Assessment

Computer Aided Assessment (CAA) has seen a substantial amount of research effort. The earliest CAA system used in teaching programming is the system presented by Hollingsworth (1960), and new developments continue to be made in the field.

The three primary components of any CAA system are: submission, assessment, and feedback. Although not considered in the current research, many newer CAA systems provide more features. For example, it is not uncommon for modern CAA systems to provide plagiarism detection, assignment specification management and delivery, course and grade administration, as well as the provision of additional materials for students. In the this research, the assessment mechanisms in CAA systems are of particular interest.

Considering the wealth of literature on computer aided assessment, a reader may quickly observe that there is one very pervasive technique for program assessment – dynamic whole program testing. Each program under evaluation is executed against a batch of test cases for which correct outputs are known (once for each test case). This mainstream technique features in a very significant number of CAA systems. Since students must submit a complete program before assessment, this assessment is *summative* in nature (Ala-Mutka, 2005). It may gain an aspect of *formative assessment* if resubmissions are permitted after feedback is given. CAA systems that are strictly formative assessment tools engage the student in a learning activity during or just before the assessment.

This chapter does not consider CAA systems based on multiple choice questions. The difficulty in multiple choice assessments is not in the grading, but rather in the formulation of high quality questions. The interested reader may consider the work by Traynor *et al.* (2006), which discusses the automatic generation of good multiple choice questions for a first programming course. Also not considered here are CAA systems for assessing student essays, such as the work by Rosé *et al.* (2003).

The primary goal of this chapter is to present a review of CAA for teaching Computer Science, and to situate the current research within that context. There are a variety of aspects through which such systems may be considered, and this review is organised accordingly. First, CAA systems are considered in terms of three generations adapted from Douce *et al.* (2005). This will serve well as a base-line for subsequent discussion. There are a wide variety of techniques that can be used for assessment purposes, and characteristics of programs that may be measured. These are discussed in the section that follows, along with related pedagogic issues. Finally, some pertinent concerns are discussed in *Section 2.3: Important Issues*.

2.1 Generations of CAA

The development of CAA systems is considered by Douce *et al.* (2005) as three distinct generations. This dissertation interprets the initial work in much the same way. The first generation is largely characterised by very low-level operation, while second generation systems are typically command-line oriented and far more general. It is interesting to note that the transition from first to second generation systems seems to coincide somewhat with the rise of the Unix operating system beginning in 1969. This is significant as many of the hurdles in first generation systems fell away with rapid hardware and operating system improvements.

Douce *et al.* (2005) describe third generation systems as being web-based in nature. Under this definition, the beginning of the third generation does not actually conclude the second generation, since both lines of development continue concurrently. There are a few systems that rely on the web to provide a new kind of assessment system, such as the anonymous web-based peer evaluation (Sitthiworachart & Joy, 2004). Barring such exceptions, the application of the *third generation* moniker is largely a matter of how the CAA technology is packaged for use.

It is noted that several second generation systems can be, and have been, promoted to third generation merely by acquiring a web front-end. Other systems, like OCETJ (Tremblay & Labonté, 2003), use the web to deliver feedback but not for file submission. Also, several earlier systems such as ASSYST are distributed in that they allow email program submission (Jackson & Usher, 1997). Similary, Kassandra allows submission over the Internet via sockets (Von Matt, 1994), but is not considered third generation.

	Hollingsworth	*ASSYST*	*BOSS*	*Kassandra*	*TRY*	*CourseMaster*	*ELP*	*HoGG*	*PASS*	*Scheme-robo*
Generation	I	II	II	II	II	III	III	III	III	III
Whole Program	X	X	X	X	X	X		X		
Function								X	X	X
Statement / Expression						X	X			X

Table 2.1: Systems generations and assessment granularity

It may be argued that a more useful definition of third generation CAA systems would reflect properties of the assessment mechanism. For these reasons, an alternative is proposed. The majority of CAA systems employ *whole program* assessment – that is to say they evaluate the overall behaviour of programs. This dissertation defines third generation CAA systems as those systems capable of assessment at a finer granularity than *whole program* assessment. For example, such a system may be capable of assessing individual functions or features of a submitted program, such as Scheme-robo (Saikkonen *et al.*, 2001) which is able to assess individual Scheme functions. It is hoped that this will prove a more useful characterisation. Further, it is recognised that some systems employ uncommon assessment techniques and resist classification in this way. For interest, *Table 2.1* shows the generation and assessment granularity of several CAA systems discussed in this section.

2.1.1 First Generation CAA Systems

The era of first generation CAA systems coincides with the era of machine language programs recorded on punched cards. In the earliest known CAA system, due to Hollingsworth (1960), students prepared their programs on punched cards and handed them to their lecturer at the end of class. Every day cards were collated and processed by a computer operator. The grader program would execute the student programs against test data, and either punch a *failure* card indicating the first error case, or a *success* card. These cards would then be returned to students on the following day. The system operated on machine code programs only, and required the memory locations of variables to be specified with each program. Although the original paper does not explain how program results were determined, it was likely that the grader program simply accessed these memory locations directly to check outputs.

Forsythe & Wirth (1965) soon produced a CAA system for grading ALGOL programs. Their system consisted of a reusable library and a special grader program developed for each programming assignment. Students follow a specified convention in their programs, which allows all compilable submissions to later be integrated directly into the grader program for assessment.

The effect was much the same as that of Hollingsworth (1960) in that the student programs also became incorporated directly into the grader program. The earlier grader was vulnerable because self-modifying code could overwrite part of the grader program. The latter was vulnerable because students could introduce code into the grader program, since their code is copied into the source code of the grader.

Hollingsworth (1960) mentions that overflow and similar runtime errors in student programs could stop the entire process – necessitating human intervention. Forsythe & Wirth (1965) imply that their system is invulnerable to these problems, but do not explain how this is the case. The code necessary to trap these conditions may have been generated by their ALGOL compiler. Both systems were still vulnerable to endless loops, requiring a human being to monitor the grading process.

Hext & Winings (1969) saw the primary problem with these systems. The grader program and the student program needed to be separated. The problem to be overcome was in supplying test data to the student program, and extracting the results computed. They required operating system changes in the program loader in order to supply test data to student programs. Their solution further required the modification of compilers to ensure that they persisted the results that student programs generated. This had the desired effect and their grader was a totally independent program, but modifying both operating system and compilers required substantial work.

The same problems were effectively eliminated with the popularisation of the Unix operating system, and piped IO streams – which may be conjectured to be the missing ingredient and catalyst for the many second generation CAA systems that followed.

One should not like to choose a precise date in the transition between first and second generation CAA systems. In the 1970's, research efforts were redirected towards understanding the characteristics of good student programs. This included the development of McCabe's complexity measure (McCabe, 1976), which estimates the overall complexity of student programs. Simultaneously, the PLUM system recorded all programs submitted to the PL/1 compiler at the University of Maryland (Zelkowitz, 1976). Within the space of about a year, this system collected over 25000 student programs for careful analysis.

Later Rees (1982) identified several attributes of Pascal programs that can be used for assessing programs stylistically. His measures are extended by Berry & Meekings (1985) for assessing the style of C programs. The work by these researchers mark a renewed research interest in the development of CAA systems in computer science.

2.1.2 Second Generation CAA Systems

Second generation CAA systems are characterised by their splitting of aspects of the grader into separate command-line tools, often feeding information along a tool chain through redirectable piped IO streams (Douce *et al.*, 2005). A good example of an early system from this time would be TRY (Reek, 1989). The system consists of several shell scripts which handle such tasks as interacting with the compiler, building the students' programs and testing them. Something that was really novel at the time was that the grader system is invoked by students, and not their instructor. This provided *immediate* feedback to students for the first time, but introduced new security problems. Students could create incorrect programs to probe at the testing engine to determine its test cases. This was discouraged by penalising students for each resubmission after a chosen limit. Simultaneously, this encouraged design over trial-and-error programming.

There were other security problems that had to be managed. When students invoked TRY, their code could be run under the instructor's account. The system improved security by creating a *sandbox* environment within which the student programs would execute. This sandbox also guards against problems with endless loops in student programs.

In order to determine correctness, the TRY system employed the typical character-by-character equivalence test with respect of expected outputs. The TRY system can, however, be configured to ignore whitespace to improve the quality of testing results.

Jackson & Usher (1997) describe ASSYST, a well known CAA system, and one of the earliest to have a graphical user interface. ASSYST separates submission and grading into distinct tasks. Its emphasis is in assisting tutors who grade student submissions from a computer terminal. ASSYST informs tutors of test cases which were passed or failed, but also generates a quality score based of the cyclomatic complexity measure due to McCabe (1976), and program style metrics derived from the work by Berry & Meekings (1985). In addition, testing programs with ASSYST can be made more accurate by using a formal grammar[1] for expected output – a significant improvement over character-by-character comparison.

ASSYST added pedagogic value by requiring students to submit their own test data. A final automated evaluation was used to determine the effectiveness of the student's own tests. This introduced students to the concept of code coverage – requiring them to think carefully about their code and its vulnerability in corner cases.

A system (*submit & progtst*) that is both recent and very representative of a command-line tool chain, and that executes under Linux is reported by Archer Harris *et al.* (2004). CourseMaster is an example of a system that provides a very flexible assessment system, where various kinds of assessments can be employed to obtain a composite assessment (Higgins *et al.*, 2002). Numerous other systems have been developed such as BOSS (Joy & Luck, 1998), RoboProf (Daly, 1999), and others. Extensive discussion of these is omitted as their assessment mechanisms largely share characteristics with the systems already described.

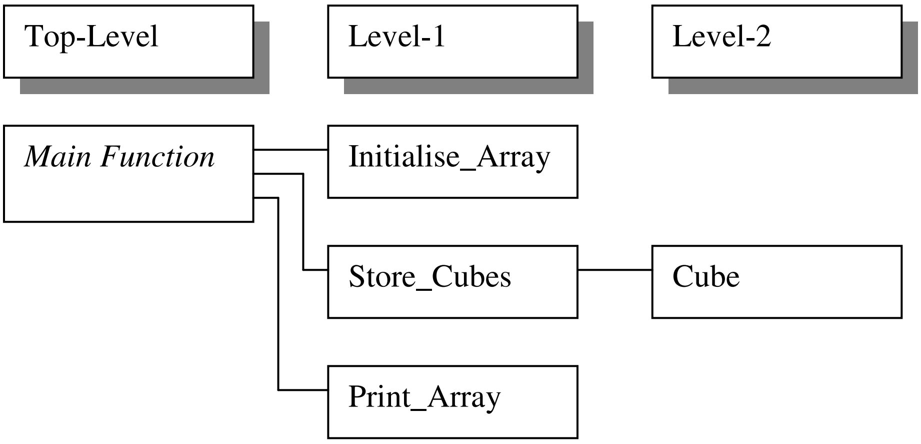
2.1.3 Third Generation CAA Systems

The current research does not follow Douce *et al.* (2005) in regarding all CAA systems with web front-ends as third generation. Rather the requirement is that the system be capable of finer grained assessment than the whole program assessment used by second generation CAA systems.

[1] The formal grammar is provided and a pair of Lex and Yacc grammars.

Thorburn & Rowe (1997) argue that a student program should not be considered perfect if it merely renders the desired behaviour. For example, a single *spaghetti code* C function constructed with internal labels and *goto* statements can produce the same output as a program with several carefully isolated functions. Instructors are likely to have a strong opinion as to which is better. In general, instructors often want students to solve a problem in certain ways – and not in others. The form of the candidate program should not be disregarded.

The solution proposed by Thorburn & Rowe (1997) is something they call a *solution plan*. This is a diagram which is essentially the *static call graph* of an ideal solution program. An example of a solution plan is shown in *Figure 2.1*. In this example, the relationship between the solution plan and the static call graph is easy to see. The top level of the solution plan is simply the main function. The next level contains all functions that can be statically determined as directly invoked by the main function. Similarly, the subsequent levels contains the functions that are invoked by the immediately preceding level, while not already being present in the graph.



Function	**Abstract Description**
Main Function	Display the cubes of integers 0 to 9
Initialise_Array	Initialise a 10 element array to zeros
Store_Cubes	Store cubes of 0 to 9 in array
Print_Array	Print the contents of the array to the screen
Cube	Return the cube of the numeric argument

Figure 2.1: An example of a solution plan used in PASS

Their system, called PASS, proceeds to test individual C functions in the student program, and maps them to the functions in the solution plan. The function outputs have to correspond exactly, or it is not considered to map into the solution plan. However, since raw function results can be compared, the system is resistant to the text formatting problems that most other systems face. The student's score is then based on the number of functions identified to correspond with functions in the solution plan.

Systems capable of finer grained assessment have an advantage over their counterparts. They may still assign marks for local correctness even if the larger program exhibits incorrect behaviour. This would go unnoticed if the assessment granularity were the whole program.

The main problem with this approach is that functions are not always independent, but rely on shared state. The example discussed in their original paper consists of four *void* functions that communicate through a global variable by means of side-effects. The effect of these functions are impossible to consider without regard for shared global variables, so it seems that this approach is more suited to the assessment of pure functional language which do not exhibit these problems.

It should be observed that what PASS measures is different from both dynamic correctness (through whole program testing) and static quality assessment (through suitable metrics). In fact, what is most significant about the work by Thorburn & Rowe is that it represents the first significant attempt at measuring static correctness, as the solution plan is a static feature.

Similar ideas have been used in Scheme-robo (Saikkonen *et al.*, 2001). This system is also capable of testing individual functions. It is observed that Scheme is a functional language, and so function-level testing makes more sense. In fact, if students are prohibited from using the *set!*, *set-car!* and *set-cdr!* functions then referential transparency is preserved – essentially, all the problems with side effects fall away.

Another property of Scheme that makes it well suited to computer aided assessment is that Scheme programs are internally represented as lists, which is also the primary data structure manipulated by Scheme programs. This makes Scheme a homoiconic language, allowing Scheme-robo to readily access and process student programs as data. Saikkonen *et al.* (2001) use this to evaluate the overall structure of a student program to determine if the supplied skeleton files have been used correctly.

If the programming language being taught does not maintain referential transparency, other means must be found to test at function-level granularity. Two systems that do this, but do it differently, are HoGG (Morris, 2003) and OCETJ[2] (Tremblay & Labonté, 2003).

In the HoGG system, student submissions constitute a Java class containing instance methods. The methods expected in the solution are first identified by method signature (rather than by name) through Java's reflection mechanism. For each method that must be tested, the system creates a subclass of the student's class which overrides every identified method with a known correct implementation – apart from the method that is to be tested. This allows each method to be tested without faults in one method causing other methods to fail. However, to support this subclassing, *private* member fields must be shunned in favour of *protected* ones – and naturally these fields must be made explicit to students. Morris recommends avoiding dependencies between methods for this reason.

Clearly, the HoGG system is very sensitive to the design of the solution. This means that particular care (more than usual) must be taken in formulating the assignment specification.

OCETJ places similar constraints on the design of the solution. It relies on JUnit, an open source Java unit testing framework, to thoroughly test each submitted class. It places greater restrictions on the form of the solution than does HoGG, since names of classes and methods must be precise. An advantage of OCETJ is that it uses an existing testing engine which is capable of constructing detailed reports. Another is that JUnit allows each unit test to construct the external state that a method will receive before it is called for testing. This must be carefully done, but reduces the problem of unforseen method side-effects. It should be noted that the problem is not eliminated if one method in an assignment specification depends on the correct operation of another – if the second method fails, both unit tests will fail.

[2] It may interest French speaking readers to know that OCETJ is an acronym for "Outil de Correction et d'Évaluation de Travaux Java". For the benefit of other readers, this can be rendered as "Tool for the marking and evaluation of Java assignments".

2.1.4 Unique Systems

It should be observed that a few systems, typically those that do not employ mainstream strategies, are not easy to categorise as above. For example, Sitthiworachart & Joy (2004) describe anonymous web-based peer assessment of programming assignments. This clearly does not follow the traditional CAA strategy for assessing programs automatically. Yet it is automatic in the sense that the whole process is managed electronically, and has similar benefits for academic resources. Peer assessment is significant because it provides a controlled environment for students to critically evaluate one another's work. Evaluation is at the highest cognitive level in Bloom's Taxonomy on Educational Objectives (Bloom, 1956), and is not as easily assessed by mainstream techniques.

An important system which takes a different approach is the *Environment for Learning to Program (ELP)* (Truong *et al.*, 2004). It performs extensive static analysis of student programs and provides immediate descriptive feedback, although it does not produce or record any scores. The distance between this and any similar system that might produce scores seems sufficiently small to still consider it a computer aided assessment system.

A similar argument might be made for the *Code Analyzer for Pascal (CAP)* (Schorsch, 1995). CAP performs extensive static analysis of student programs, and generates very accessible feedback. Unfortunately, no reasonable way to turn their analyses into scores is obvious, neither has any been reported.

2.2 CAA Measures

There are a variety of assessment measures that have been used in CAA systems. Most assessment measures may be described as either *problem-aware* or *unaware*, obtained through *static* or *dynamic* analysis, and being applied at either *coarse* (whole program) or *fine* granularity. In this regard, *Table 2.2* lists the most important categories of assessment measures, along with the properties of each. Similarly, *Table 2.3* shows some of the systems already described and the assessment measures they employ.

	Static	*Dynamic*	*Problem-Aware*
Efficiency		X	
Functionality		X	X
Test Coverage	X	X	X
Structural Features	X		X
Programming Style	X		
Complexity	X		
Software Metrics	X		

Table 2.2: Properties of assessment measures

	Hollingsworth	*ASSYST*	*BOSS*	*Kassandra*	*TRY*	*CourseMaster*	*ELP*	*HoGG*	*PASS*	*Scheme-robo*
Efficiency		X								X
Functionality	X	X	X	X	X	X		X	X	X
Test Coverage		X								
Structural Features						X	X			X
Programming Style		X								
Complexity		X					X			
Software Metrics							X			

Table 2.3: Systems and the assessment measures they employ

The following sections first consider the most important *dynamic analysis* measures, and then treat *static analysis* measures in similar detail. The issue of *problem-aware* assessment is relevant in each of these kinds of analysis, so it is addressed where appropriate in both sections.

2.2.1 Dynamic Analysis

Dynamic analysis is any analysis that requires the execution of the program being analysed. The main aspects of programs that are valuable to measure dynamically are execution efficiency, and valid functionality.

Test coverage analysis can be considered to be both static and dynamic. The reason is that determining test coverage requires the ability to track the statements reached during the execution of test cases, in addition to the executing of test cases themselves. Tracking statements requires either a high quality profiler, or instrumentation of the program code. The latter requires static analysis. However, dynamic analysis can be augmented with one of several third party tools which handle test coverage. Example for the Java language are NoUnit (2006), Cobertura (2006), and jcoverage (2006) *(sic)*. As a result it seems most reasonable to group test coverage and other test adequacy evaluation with dynamic analysis measures.

Efficiency

The measuring of the efficiency of student programs is supported by one of the earliest CAA systems (Forsythe & Wirth, 1965). However, they report that the efficiency measure was not as useful as testing student programs for reliable behaviour. Researchers seeking a more holistic assessment have not been deterred. ASSYST provides two methods of assessing efficiency (Jackson, 1996; Jackson & Usher, 1997). The first is the simple measurement of execution time. A student program is considered adequately efficient if it executes within a specified margin of the execution time of the instructor's solution.

This has had two problems. In many cases simple programs complete so quickly that the granularity of the system clock is too coarse for measuring the execution time. Many modern processors have hardware to track instruction counts, but access to this was not readily available at the time.

The second problem is that in longer running programs, the overall execution time may be dominated by IO operations performed at the start and end of execution. The primary benefit of this approach, when feasible, is that its implementation is quite simple.

The other approach offered in ASSYST is statement counting. A global variable is incremented as each high-level statement is executed. This requires program instrumentation, necessitating at least some static analysis. However, this technique is not sensitive to the problems mentioned previously, and Forsythe & Wirth (1965) report meaningful results using the same strategy.

Scheme-robo also implements a form of statement counting[3]. In their case this is easily

[3] When Saikkonen *et al.* (2001) refer to complexity they mean runtime complexity, implying time

achieved since their Scheme implementation is interpreted through a small *metacircular* evaluator (Saikkonen *et al.*, 2001). Consequently, the interpreter is well placed to count statements, and instrumenting the code is not necessary.

Regardless of the measurement technique, what is always required is a standard against which to compare. Jackson & Usher (1997) compares the performance of student programs against the performance of an ideal program, while Saikkonen *et al.* (2001) report using the dynamic statement count to estimate the order (Big-O) of computation. A limit is set under which the computation is assumed to be linear. More limits are set for other orders of computation.

It may be argued that it would be better to assess the execution time of each student program against itself, with increasingly large datasets[4]. For example, three runs of a sort program with datasets of 1000, 2000, and 3000 elements, respectively, should be sufficient to discern between linear $O(n)$ behaviour and quadratic $O(n^2)$ behaviour. More runs may be necessary to detect finer differences, such as distinguishing $O(n)$ from $O(n \log n)$, since the constant multipliers have a stronger effect on the running time.

Considering the literature, the measuring of efficiency in CAA systems has focused exclusively on the time dimension. The analysis of the dynamic memory footprint of student programs appears to have been overlooked.

The pedagogical value of measuring time efficiency is uncertain. In most assignments it is unlikely to be important at all, particularly so in first programming courses. The programs need only complete within an acceptable time limit – a necessary evil to protect against endless loops. However, in the context of data structures and algorithms course, the ability to recognise the order of execution may become more relevant. It may be especially relevant in a numerical analysis course, since the order of convergence of numerical methods is very important it that field. It is suggested that for the same courses, the analysis of the shape of the memory footprint would also be more meaningful.

efficiency.

[4] This approach is not followed by any of the CAA systems considered.

Functionality

The functionality of computer programs is probably the most visible criterion upon which they may be judged. In general, the problem of determining if a given program functions correctly is known to be undecidable.

The ability to assess functionality depends on the observer's ability to *recognise* behaviour and *discriminate* between correct and incorrect behaviour. Fortunately, it turns out that many programming assignments can be formulated in such a way that key aspects of behaviour can be recognised and assessed by computational processes. As such, evaluating functionality has become the most popular automatic means of assessing student programs.

Doing this depends critically upon two things: the means to provide test data to the running program, and the means to capture and evaluate the accuracy of the resulting outputs – both without human involvement. As noted in *Section 2.1.1*, the former requirement presented difficulty for early systems (Forsythe & Wirth, 1965; Hext & Winings, 1969), while more recent systems utilise redirected IO streams for communication with the program. The most modern CAA systems are able to use features such as Java reflection to invoke methods and access raw result values. For example, Quiver (Ellsworth *et al.*, 2004) is able to access Java method results directly. Scheme-robo is able to do the same due to the custom metacircular evaluator (Saikkonen *et al.*, 2001).

It is important that while the raw results may be boolean values, real numbers, or complex data structures, most CAA systems can only assess these if they are rendered as text on the output stream. This usually introduces unnecessary variation in formatting. In order to counter the formatting problem, reasonable solutions have been proposed such as filtering spaces and using regular expressions or formal grammars to describe program output (Jackson & Usher, 1997; Morris, 2003).

Test Coverage

Performing test coverage analysis requires that students not only submit their solution programs, but also a set of formalised test cases. This has clear pedagogical advantages. It provides an opportunity for introducing the current industry practice of applying unit tests. Perhaps more importantly, it requires students to consider the behaviour of their software under a variety of input data.

The adequacy of unit tests is commonly assessed by means of test coverage. This is usually a percentage of program statements actually reached during execution of the tests. ASSYST measures test coverage of each student's test cases with respect of their own program (Jackson & Usher, 1997).

This has two problems. Firstly, a student may get a 100% score for test coverage even if his or her test cases only cover the portion of the problem actually solved by the program. Secondly, a student with a perfect set of tests, and a solution that passes every test, may still receive less than 100% for test coverage. This would happen if his or her solution contained some unreachable code, as these statements would never be covered by the test cases[5]. For these reasons, more consistent results are expected if test coverage is measured against the instructor's solution. Related to this, Edwards (2003) describes using the instructor's solution to automatically validate student test cases.

If assessment is made possible at function level, a decision must be taken regarding unit tests associated with arbitrarily deep functional decomposition. Student functions that cannot be directly related to a function in the instructor's solution also cannot have their unit tests applied in the context of the instructor's solution. However, these functions are automatically tested in that they must serve a higher goal which is also supported by the instructor's program. In other words, they are tested by virtue of being used in a larger part of the program – which is tested. If unit tests are specified for such functions, the tests should still be invoked, but should serve primarily in feedback to the student, rather than influencing her score.

Requiring students to submit test cases has another important advantage. Feedback can be returned in terms of the student's own test cases, rather than in terms of the instructor's test cases which are used to assess functionality. This prevents the submission mechanism from being abused to extract the instructor's test cases.

A general problem with requiring students to submit thorough test cases is that they do not appreciate the important role this has in developing reliable software. Goldwasser (2002) proposes an interesting solution to motivate students to take this seriously. In his approach, students score points if their test cases expose flaws in the programs of their peers.

[5] This is true unless an additional static analysis phase is first used to remove all the unreachable code.

2.2.2 Static Analysis

Static analysis (as opposed to dynamic analysis) is the discovery of the properties of a computer program without actually running it. In the context of computer aided assessment, static analysis methods are seldom problem-aware. Assessments that are not problem-aware are valuable because they are immediately applicable – they do not even require a problem specification. However, sometimes a grading specification is used to qualify the broad static characteristics that are reasonable for a given problem.

The only notable forms of problem-aware static analysis in CAA involve structural similarity and feature discovery. This sub-section begins with a discussion of structural analysis, and continues to consider the use of program style and software metrics in assessment.

Structural Analysis

Limited work is reported to have used structural features in computer aided assessment of student programs – the most significant of which is probably Saikkonen *et al.* (2001) and Truong *et al.* (2004). It seems that common terminology between existing work has not yet been established. In this text, *structural analysis* is considered to be any analysis that operates on a structured representation of the student program – with the notable examples being here called *stuctural similarity* and *feature discovery*. Structural similarity is a holistic study of whether the structure of a given program conforms to what is expected in a correct solution program. Feature discovery is considered to be the search for a structural feature pattern within a subset of the structure of a program. This is similar to cliché recognition described by Rich & Wills (1990). As a natural result of these definitions, any analysis of program structure requires a non-linear representation of the computer program – the typical choices being parse trees, abstract syntax trees (ASTs), or similar data structures.

Chanon (1966) presents the earliest application of structural information in CAA. The author describes the intention to compare programs structurally, but doing so directly was found to be too computationally intensive for the equipment available. Chanon also wanted to use structural similarity to link plagiarised programs, necessitating $O(N^2)$ comparisons, without which the original idea may perhaps have been feasible. The compromise found was to compute a *structural coefficient* for each program's syntax tree. This coefficient is computed through a carefully designed hash function over the program's structural representation.

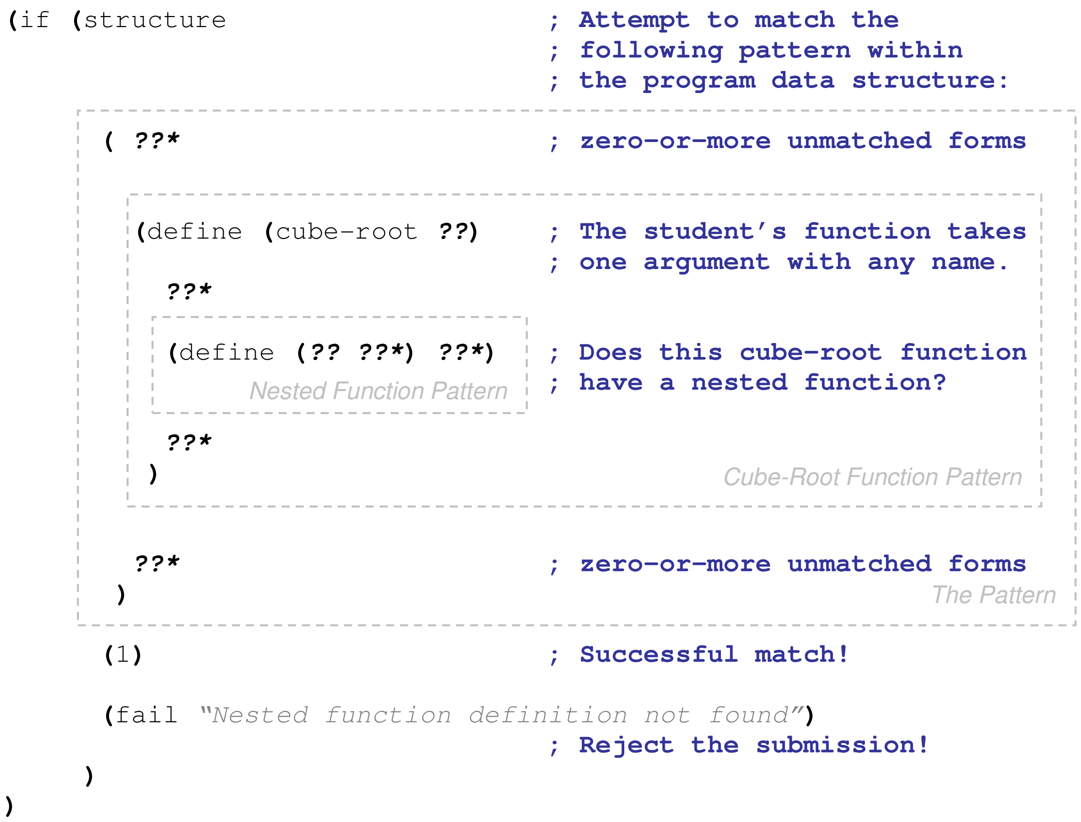
Although program structure has been used by some CAA systems, it is typically incorporated into feedback instead of being used in deriving a submission score. The Feature Tool found in CourseMaster is almost an exception. It searches programs for features specific to the given exercise (Higgins *et al.*, 2002). The example offered is the detection of a *while* loop where a *for* loop is required by the assignment specification. This tool does contribute marks to a final score, but it accomplishes its search by means of regular expressions rather than by considering structural information. For this reason it is not truly a structural analysis technique.

A feature discovery technique which does operate structurally is offered by Saikkonen *et al.* (2001). Their solution uses a pattern language created in Scheme. These patterns provide a declarative specification for the problem specific structural relationships required in students programs. *Figure 2.2* shows an example (adapted from the Saikkonen *et al.* paper) of a declarative feature discovery pattern that searches for a *cube-root* function, which itself must have a nested function.

In the pattern language, the **`??`** operator matches any single atomic value or parenthesised combination. Similarly, the **`??*`** operator matches any sequences of values. All other symbols in the pattern must match exactly. In this way, the function definition keyword **`define`** and the identifier **`cube-root`** are treated as literals – they must be present in the student program in exactly the same form as the pattern. Further, the pattern **`(define (?? ??*) ??*)`** matches any function definition, regardless of name, parameters, or function body. The important criterion of the *Figure 2.2* is that such a function must occur nested within a larger function, named **`cube-root`**. Scenarios where instructors require students to solve a problem using a specified strategy are common in teaching but is impossible to verify with standard functionality testing.

The Scheme-robo system is perhaps the most well placed system for structural analysis, because the student program is already available in a structured form. However, Scheme-robo currently only uses Feature Discovery, and then only to reject submissions which do not follow the overall strategy required in the assignment specification.

The only known CAA system that employs structured similarity is ELP (Truong *et al.*, 2004). It compares the structural organisation of the statements in a student program



```
(if (structure                        ; Attempt to match the
                                      ; following pattern within
                                      ; the program data structure:
     ( ??*                            ; zero-or-more unmatched forms

       (define (cube-root ??)         ; The student's function takes
                                      ; one argument with any name.
         ??*

         (define (?? ??*) ??*)        ; Does this cube-root function
                                      ; have a nested function?
         ??*
        )

       ??*                            ; zero-or-more unmatched forms
      )
     (1)                              ; Successful match!

     (fail "Nested function definition not found")
                                      ; Reject the submission!
    )
)
```

Figure 2.2: Scheme-robo feature discovery pattern

with that of a model answer. For example, it may report that a loop and a function call were found, where two loops and an *if* statement were expected. In this way, ELP offers a formative evaluation which could lead the student towards discovering the correct solution. The structural similarity does not contribute to the student's score. In terms of their algorithm, the authors do not disclose any detail, but they report that their technique is only effective for small introductory-level assignments.

Both feature discovery and structural similarity show promise for further work in computer aided assessment. In particular, a means to score programs based on structural similarity has not yet been developed.

Style & Software Metrics

Style and software metrics are typically surface features of a program's source code that can be easily measured. The foundational work on assessing programs stylistically is due to Rees (1982), who identified ten simple metrics through which the quality of Pascal programs could be judged. The particular metrics Rees measured are described in the list below. The first five metrics govern program formatting, while the last five are concerned with identifiers and the construction of the program.

i. **Line Length**
The average number of characters per line, after removing leading and trailing whitespace. Long lines reduce readability. The same is true for short lines, but only in the extreme case.

ii. **Comment Density**
The proportion of program lines containing comment text, also counting lines with partial comments. Programs with comments are considered to be more maintainable and easier to read than those without, but too many comments makes following the flow between statements more difficult.

iii. **Indentation**
The ratio of the number of leading whitespace characters to the total number of characters. Indentation is frequently used in programming to communicate the relationships between nested constructs. This metric estimates the quality of indentation, but is crude in that it only considers identation directly, and disregards relative indentation. This is significant as changes in indentation communicates better than indentation itself.

iv. **Blank Lines**
The percentage of lines that contain nothing more than whitespace. Blank lines should be used to separate different parts of the program and serve to draw the eye to discrete starting points.

v. **Embedded Space**
The average amount of whitespace per line, excluding leading and trailing whitespace. Internal whitespace serves the same role between expressions as blank lines between groups of related program statements.

vi. **Program Decomposition**
The average number of program lines per function or procedure in the program. Lower scores are generally associated with better decomposition.

vii. **Reserved Words**
The total number of distinct keywords used. A larger score may indicate that the programming language is being used more effectively, as more constructs from the language are being employed.

viii. Identifier Length

The average length of defined identifiers. Longer variable names imply that the names are carefully chosen to convey meaning.

ix. Variety of Identifiers

The total number of different identifiers occurring in the program. Rees (1982) argues that a program with a small number of distinct identifiers is easier to read than a similar program with many identifiers.

x. Labels and *Gotos*

The sum of the number of labels and the number of *goto* statements in the program. Unstructured control flow is widely believed to be hard to read, understand and maintain, so this score should be low in most well constructed programs.

Most of these metrics are ratios for which there is an ideal value. Having done so, each metric score must be converted into a quality score. The method suggested by Rees (1982) (see *Figure 2.3*) involves using two score intervals which surround the ideal metric score – inner and outer regions. Programs within the inner region receive 100%, while those outside the outer region receive a 0% quality score for the metric considered. The intervals (u_0, u_1) and (v_0, v_1) are the interesting parts of the domain. Here quality scores are obtain from the linear growth and decay functions, $\frac{x-u_0}{u_1-u_0}$ and $1 - \frac{x-v_0}{v_1-v_0}$, respectively.

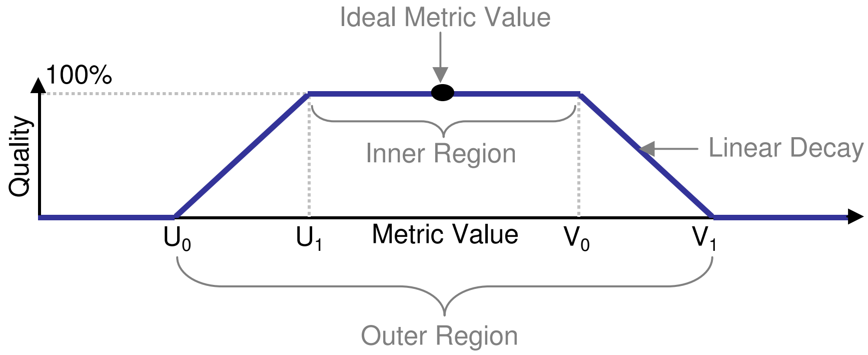


Figure 2.3: Converting metric scores into quality scores

The style metrics proposed by Rees are useful across a wide variety of programming languages – although the ideal metric scores vary with each language. For example, Berry & Meekings (1985) adapt the Rees metrics for C programs, with modest changes.

Software metrics may also be useful in assessment, but few have found their way into CAA sofware. Ala-Mutka (2005) notes that software metrics are relevant in the educational setting. However, software metrics themselves are only numeric measures. What is lacking is the means to interpret them valuably in a teaching context. One software metric that has proved useful in teaching environments is the Cyclomatic Complexity measure (McCabe, 1976), which estimates the complexity of a program control flow. This measure is employed in both ASSYST (Jackson & Usher, 1997) and ELP (Truong *et al.*, 2004). ELP also finds use for other software metrics, such as reporting *switch-case* statements that are not terminated with a *break* statement, and the detection of shadowed variables.

2.2.3 Uncommon Techniques

In this sub-section, several uncommon techniques that may be employed in CAA systems are considered. For example, with the rise in popularity of the Java programming language, many programming courses teach the creation of graphical user interfaces (GUIs) in Java. These programs are particularly difficult to assess via traditional CAA techniques. To remedy this, English (2004) describes JEWL – a library for creating GUIs that support automatic evaluation. JEWL is simpler than the Java AWT, but not sufficiently dissimilar to discourage its use in an early Java course. The system allows events to be introduced into the event queue by a test harness program. The test harness can also access and update the controls in the user interface. In this way, it is able to perform functionality testing in a similar way to functionality testing based on command-line IO.

The study of computer aided assessment of computer programs has not exlusively focussed on assessing program code. Some researchers have considered the possibility of grading design diagrams. In this regard, CourseMaster offers a tool to assess students' flowcharts (Higgins *et al.*, 2002), while Thomas *et al.* (2005) describe their initial work in automatic grading of ER diagrams.

All of these kinds of deliverables can be assessed through guided peer assessment. While this approach has problems of its own, Sitthiworachart & Joy (2004) motivate that their anonymous peer assessment system has the benefits of the automated assessment offered by many CAA systems, without much of the complexity. They also observe new pedagogical advantages in that students can learn by evaluating other student's code.

Finally, in a sense the compiler itself is a CAA system in that it may aid a human evaluator in identifying both *syntactic* and *contextual* errors in student programs. Compilers vary in terms of the quality of feedback they provide, and usually do not look for the kinds of logic errors typically made by novice programmers.

A system (CAP) that analyses Pascal programs for logic errors, in addition to syntactic and contextual errors, is described by Schorsch (1995). His system is pedagogically important because the feedback explains each error in a way that is appropriate for a first programming course.

2.3 Important Issues

Having carefully surveyed the literature for the computer aided assessment of student programs, some important issues remain to be addressed. One such issue, which has received some debate in literature, is the precision with which assignment tasks should be specified. This is addressed first.

The second issue concerns the effect of program errors on the viability of using particular CAA measures, as different techniques are sensitive to program errors in different ways. The third issue is a related problem and deals with the fact that functionality testing – the most popular CAA measure – can occassionally yield meaningless scores which go undetected as such.

2.3.1 Precision of Assignment Specifications

Jackson (2000) considers the question of whether an assignment specification should be so precisely stated as to leave no room for interpretation. If the objective is to calculate π, should the grader accept "PI = 3.14159", but still reject "PI: 3.14159" due to formatting? Jackson argues that if the assignment is too tightly specified, it leaves the student no opportunity for innovation.

While innovation is important, the present author considers that the pedagogical value lies in having students think about their programs. Much of this opportunity for thought is removed in a truly unambiguous assignment specification. Beaty (2001) presents a similar argument saying that at least some parts of the assignment should only be specified informally to require students to invest thought into their solutions.

In contrast, Archer Harris *et al.* (2004) argue that the assignment specification should be absolutely unambiguous, down to the individual characters expected in output – especially for a first programming course. They make a compelling argument that any latitude offered by the specification will enable students to substitute code they are able to produce for code they *should* be able to produce. They further argue that since computers are literal and exact in their operation, assignment specifications should mirror this – ensuring that students become familiar with the notion that computers do not think and require precise commands.

It seems most of the benefits for a particularly tight assignment specification are achieved in the first few weeks of assignments. Longer use of this strategy will result in diminishing returns. As students gain experience in programming, having them think more about the behaviour of their programs becomes increasingly valuable, pedagogically. It would also be better if the students are encouraged to think about progressively deeper semantic detail, rather than the shallower aspects of output formatting. When students' reach this level, unnecessarily strict output formatting may just be an annoyance.

2.3.2 Sensitivity to Errors

Programming errors are broadly considered to be either *syntactic*, *contextual*, or *semantic*. For example, if functionality testing is applied to the whole program, the assessment is necessarily critically sensitive to syntactic and contextual errors, as programs containing these cannot be run. The purpose of functionality testing is to expose semantic errors, but here also it is over-sensitive. The effects of a small semantic error are not localised by the assessment process, but can cause every test case to fail. For this reason, functionality testing at program granularity is highly sensitive to all error kinds.

Consider, however, that if testing is done at function granularity, the effects of semantic errors may be localised to the specific function, as seen in HoGG (Morris, 2003). So sensitivity to semantic errors is reduced. The degree to which it may be reduced depends upon the technique's ability to manage functional dependencies, because these determine how semantic errors may propagate through a program.

When testing at function granularity, it is possible to localise the effect of contextual errors as well. This is because testing an individual function requires that dependent functions are replaced with correct versions anyway. The same is true for any syntax errors that may be localised to a given function. This will improve the quality of assessment as it minimises the interdependency of measurable artefacts. As Ala-Mutka (2005) observes: "although many approaches emphasize the possibility for iteration, they still expect the students to submit a complete version of the program already on the first submission". This is unfortunate, since clearly this is not an inherent requirement of all assessment techniques.

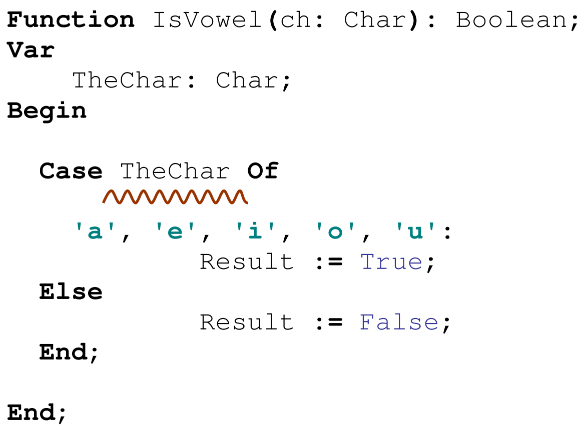
It should be noted that structural analysis is largely immune to semantic errors in the submitted programs, since it does not perform assessment by executing the program. In the same way, it is almost entirely immune to contextual errors[6]. With a parsing error recovery strategy that localises the effect of syntax errors, structural analysis can be made to work on any program submission, regardless of the degree of completion. It is the only problem specific CAA method with this property. As a result, for educational environments the technique may support iterative programming in a more effective way than dynamic analysis.

Lastly, many style metrics are also immune to all error kinds because they only count surface features of the program. However, existing CAA systems that make extensive use of style metrics, such as ASSYST, still reject program submissions which do not compile.

2.3.3 Meaningful Assessment

As observed in *Section 2.3.2*, testing of functionality is sensitive to semantic errors. Occasionally, a small error may cause dynamic analysis to give a program a meaningless score. As an example, consider the program extract shown in *Figure 2.4*.

[6] A possible exception would be abstract interpretation, which cannot operate in the presence of contextual errors. However, abstract interpretation has not received attention in literature relating to assessment.



```
Function IsVowel(ch: Char): Boolean;
Var
    TheChar: Char;
Begin

  Case TheChar Of

    'a', 'e', 'i', 'o', 'u':
            Result := True;
  Else
            Result := False;
  End;

End;
```

Figure 2.4: Identifying vowel characters

The function in *Figure 2.4* accepts a character parameter, and is supposed to return *true* only when the character is a vowel. The error in the program is that the decision operates on a local variable, rather than the incoming parameter. This means that the function will return the incorrect result much of the time. The result depends on what happens to be at the memory location held by the uninitialised local variable[7]. If the function consistently returns false, it will pass 21 of 26 test cases (81%). If the function consistently returns true, the balance is awarded (19%) – neither of which are meaningful. In contrast, structural similarity should fair well in this case as the overall structure of the program is largely intact.

2.4 Summary

This chapter discussed the origins of modern CAA systems for computer science education. With respect of the classification of CAA systems (Douce *et al.*, 2005), third generation systems were redefined – bringing to fore the pedagogical advantages of fine grained assessment strategies.

In addition, a large variety of assessment techniques were discussed. While many aspects to automated assessment of computer programs have stagnated, structural similarity shows potential for further work as it has not yet been used to assign scores to student programs – the focus of this research.

[7] The result also depends on compiler behaviour. When compiled with the Delphi 7 compiler, this program exhibits different behaviour depending on whether optimizations are enabled or not.

A main difficulty in doing this is in obtaining a numeric relationship for program structure between the student submissions, and programs known to be correct. A pragmatic difficulty also exists. How do we extract a valid approximation to the program structure in the presence of syntactic errors? Existing parsing error recovery methods may not be appropriate since they tend to focus on producing valid error messages without spurious reports, and may skip over an unnecessarily large portions of the student program. The following chapter will deal with these subjects in greater depth.

Chapter 3

Structural Similarity

In the problem domain of this research, candidate student programs and ideal solutions are both represented as graphs. The intention is to use a structural similarity measure to obtain a similarity score between candidate and ideal programs. An important question as outlined in *Section 1.2.2* is whether this similarity score can be used as a reasonably accurate percentage mark for the submitted work. If this is the case, it is feasible that a structural similarity measure could replace a human marking strategy. Similarity measures which accumulate scores at each structural sub-component have further utility. They would allow drilling down into the program's subgraphs, which in contrast with simply receiving a global score, may be useful in understanding where the student's marks were accumulated. Since student programs can be related to the structure of known solutions, the same procedure may also be useful in understanding which strategies are being employed by students for particular problems.

A fundamental problem exists in measuring the similarity of any pair of things. One might ask the question: *What is similarity?* Similarity is a much harder concept to define than equivalence. However, a definition is required if similarity is to be measured. A practical view may be that similarity is defined by the algorithm that measures it – a view which is unfortunately self-referential.

This definition introduces a range of philosophical problems. For example, as two distinct similarity measures cannot provide independent definitions of similarity, *how should these measures be compared?* If many similarity measures have been investigated for a given problem domain, one may have gained sufficient academic favour to be considered the baseline against which other similarity measures should be compared. In this way, tree edit distance similarity has become the basis for a practical definition of similarity over hierarchical tree-structured data (Yang *et al.*, 2005), particularly so for ordered trees.

Philosophical matters aside, it is evident that measuring similarity of structured data requires:

1. a representation which captures important structural information.
2. an algorithm that produces useful similarity scores.

For similarity scores to be useful they must be symmetric – so it is required that $sim(\text{Data}_A, \text{Data}_B) = sim(\text{Data}_B, \text{Data}_A)$. For assessment there is the additional requirement that score be in the interval $[0, 1]$.

Several representations of structured data exist, including XML and LISP-style symbolic expressions. *Graphs* are also a good tool for representing structured data, because the graph is a mathematical abstraction designed for describing and reasoning about the formal relationships between connected entities. Fortunately, it suffices to consider only graph representations, as other representations can easily be mapped to graphs.

The algorithms available for measuring structural similarity can be divided into those that depend upon the exact equivalence of subgraphs[1], and those that introduce a fuzzy notion of matched subgraphs. The issue of using exact equivalence is important because graph equivalence and isomorphism is costly to compute (Garey & Johnson, 1990), whereas fuzzy heuristics are cheap. However, when representations are restricted to ordered trees, the associated algorithms frequently become considerably simpler. Consequently, several researchers have focused on tree representations with strict ordering relationships between sibling vertices.

The contemporary alternative approach is more flexible but may sacrifice some precision. The main idea is that in relaxing the notion of equivalence, a more effective measure of similarity may be obtained, since similarity is not exact anyway. The common approach is to have similarity scores propagate through a graph, updating the scores held at each vertex, until the scores stabilise. This kind of algorithm is described as *Similarity Flooding* by Melnik *et al.* (2002). The current research will

[1] The subgraphs to which equivalence is applied may be of a specific kind, such as in all path oriented techniques.

use the term *Similarity Propagation* to encompass both *Similarity Flooding* and several related algorithms[2].

In the discussion of this chapter, some matrix operators feature which are not usually common. Before continuing, some of the most important notation is first introduced. The remainder of the chapter consists of three main sections. The first presents the most important of the traditional structural similarity measures, all of which are among the algorithms based on the exact equality of subgraphs. The second section focusses on the more recent similarity measures based on similarity propagation. Finally some conclusions are drawn with regard to applying the similarity measures in assessment.

Important Notation

G_A and G_B	the graphs being compared
T_A and T_B	the graphs being compared if known to be trees
A and B	the adjacency matrices of G_A and G_B, respectively
n_A and n_B	the number of vertices in G_A and G_B, respectively
h_A and h_B	the height of T_A and T_B, respectively
$V(G)$	the set of vertices in graph G
M_i	the ith row of the matrix M
M_{ij}	the jth entry of the ith row of matrix M
$A \circ B$	the Hadamard matrix product (element-wise product)
$A \otimes B$	the Kronecker matrix product (matrix direct product)
$\|\|M\|\|_p$	the *entrywise* $p-$norm of matrix M

[2] The author does not wish to create confusion by overloading the term *Similarity Flooding*, already described by Melnik *et al.* (2002).

The matrix norms that are important in this chapter bare special mention as the definition of the *entrywise* $p-$norm differs from the common $p-$norm. The form used here is given in *Equation 3.1*.

$$\|M\|_p = \left(\sum_i \sum_j {M_{ij}}^p \right)^{\frac{1}{p}} \tag{3.1}$$

Among the most important cases are $\|M\|_1$ which computes the sum of the matrix entries, and $\|M\|_\infty$ which determines the largest entry. Another important case is $\|M\|_2$ which, as defined here, is exactly equivalent to the Frobenius matrix norm, and computes the square root of the sum of the squares of the entries. This is similar to the Euclidean vector norm.

The Hadamard and Kronecker products are defined in *Equations 3.2* and *3.3*, respectively. The Kronecker matrix is shown as a block matrix, because it is otherwise very large.

$$\textit{Let } K \textit{ and } L \textit{ be } m \times n \textit{ matrices}$$
$$\textit{Let } M \textit{ be a } p \times q \textit{ matrix}$$

$$(K \circ L)_{ij} = K_{ij} \times L_{ij} \tag{3.2}$$

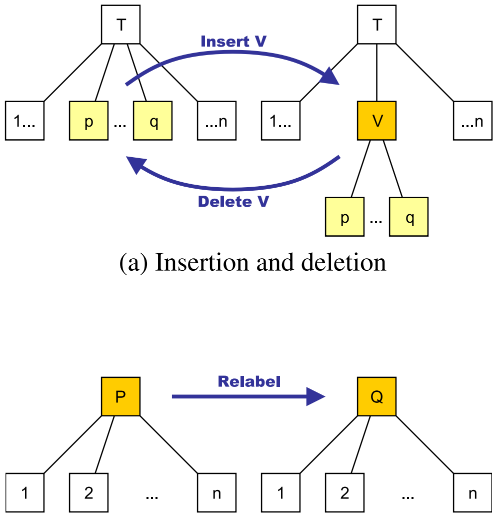
$$(K \otimes M)_{ij} = \begin{bmatrix} K_{11}M & K_{12} & \cdots & K_{1n}M \\ K_{21}M & K_{22} & \cdots & K_{2n}M \\ \vdots & \vdots & \ddots & \vdots \\ K_{m1}M & K_{m2} & \cdots & K_{mn}M \end{bmatrix} \tag{3.3}$$

3.1 Traditional Similarity Measures

This section reviews structured similarity measures based on the exact equivalence of subgraphs. These include tree edit distance similarity, maximum common isomorphic subgraph methods, tag-oriented similarity, and path-oriented methods.

3.1.1 Edit Distance Similarity

Given the two trees, T_A and T_B, there are in general many sequences of edit operations that could transform T_A into T_B. If each kind of edit operation has an associated cost, then the cost of each edit sequence is the cumulative cost of each edit operation it contains. Further, the edit distant between T_A and T_B is the lowest total cost among all edit sequences that transform T_A into T_B. The basic edit operations described by Tai (1979) are *Insert Vertex* and *Delete Vertex* (*Figure 3.1a*), and *Relabel Vertex* (*Figure 3.1b*). These usually have the same unit cost.



(a) Insertion and deletion

(b) Relabelling

Figure 3.1: The primary edit operations used in finding a tree edit distance

All of the important algorithms for determining the minimum tree edit distance are bottom-up algorithms based on dynamic programming. This usually involves a post-order traversal of the tree during which the problem is first solved for the smallest subtrees. These results are stored in a memo table to eliminate repeating work when considering progressively larger subtrees.

In one of the earliest algorithms, only leaf insertions and deletions are permitted, with $O(n_A n_B)$ performance (Selkow, 1977). This restriction was lifted by Tai (1979) by allowing insertions and deletions to take place anywhere within the tree. These unrestricted edit operations are those demonstrated in *Figure 3.1a*, and have become the accepted minimum set of edit operations.

The difficulty in allowing single-vertex deletion in the middle of a tree can be seen by considering what should be done with the target node's children. As the figure shows, when a vertex with children is deleted, the parent vertex must adopt the children of the deleted node. The only condition is that vertex deletions cannot take place at the root unless the root contains only one child, in which case, that child becomes the new root. Insertions are handled in a similar way.

The performance cost of these algorithms can be seen intuitively. Consider that if a vertex of an unordered tree has n children, there are $2^n - 1$ subsets of the children which could be adopted by a newly inserted vertex. The case for ordered trees is not as dire, since there are only $O(n^2)$ viable subsets of the children. In consequence, most of the published work is restricted to considering ordered trees, since the same problem for general trees is NP-Hard[3] (Zhang *et al.*, 1992).

Tai's algorithm is limited to ordered trees. It's main problem is still its poor performance as the algorithm completes in $O\left(n_A n_B h_A^2 h_B^2\right)$ time. Working to better this, Shasha & Zhang (1989) present an algorithm which exhibits $O\left(n_A n_B \cdot min(h_A, h_B)\right)$ behaviour.

Chawathe *et al.* (1996) focus on constructing a "minimum-cost edit script". This script allows a more meaningful study of the changes between two trees, as other algorithms compute only the edit distance and not the edit sequence. Their algorithm operates on unordered trees, but to counter the NP-Hardness, may yield sub-optimal edit scripts. Further advancements are made by allowing the additional edit operations: *Subtree Move*, *Subtree Prune* and *Subtree Graft* (Chawathe & Garcia-Molina, 1997).

A more recent contribution is due to Nierman & Jagadish (2002). They allow the three standard edit operations in addition to *Delete Subtree* and *Insert Subtree*, which are similar to the prune and graft operations of Chawathe & Garcia-Molina (1997). Once again, they consider only ordered trees. In spite of the additional edit operations, they obtain performance similar to that of Shasha & Zhang (1989). Further, if a constant limit is placed on the number of children any given node may have, then their algorithm exhibits $O(n_A n_B)$ behaviour.

Application to Program Similarity

Regardless of how the edit distance is computed, it is still a distance metric and not a similarity score. Fortunately, there is a simple way to obtain a similarity score from the edit distance. Considering that in the worst case of converting T_A into T_B, every vertex

[3] NP-Hard problems are at least NP-Complete, but possibly harder.

of T_A would have to be replaced with a vertex from T_B, with the surplus either deleted or inserted, depending on which tree is larger. This means that at most $max(n_A, n_B)$ edit operations are required. This yields the standard expression for similarity based on tree edit distance, given in *Equation 3.4* (Buttler, 2004).

$$Sim_{\text{TreeEdit}}(T_A, T_B) = 1 - \frac{dist_{\text{TreeEdit}}(T_A, T_B)}{max(n_A, n_B)} \tag{3.4}$$

An important observation can be made if this measure is to be applied for the automated assessment of student programs: the contribution of an edit operation to the final similarity score is dependent on the sizes of *both* trees. Suppose two students submit independent candidate solution programs of different size. Suppose also that when these are respectively compared with an ideal solution, the same edit operation is required. Although the same change is applied to both submissions, the impact is larger for the smaller program. This can be ameliorated by using assignments with large ideal solutions, as the effect diminishes as the size of the ideal solution increases.

Allali & Sagot (2005), who report on using tree edit distances in RNA[4] comparison, describe a problem they call the "scattering effect". The problem occurs when two structurally similar subtrees are incorrectly associated. They observe cases where (through information innaccesible to the edit distance algorithm) it is known that the two trees are not actually related. To illustrate this problem in the context of comparing programs, consider *Figure 3.2*. If for the moment the choice of identifier names is ignored, the two fragments shown are very similar – requiring only two edit operations to reduce one into the other. It is unlikely that a human marker would associate the two fragments because the choice of variable names carries significance. The human can use this to tell that the functions do not serve the same intention.

The problem is avoided if identifier names are required to be strictly equivalent. However, this approach implies assignment specifications that are so precise as to leave little room for problem solving. Otherwise this requirement is unreasonable as it is very unlikely that student submissions and the ideal solution would agree on variable names. The tree edit distance measure does not make this an easy issue. What is needed is a bijection between the names used in the two programs to evaluate their similarity. This applies to all user defined names, including variable and function names. Unfortunately,

[4] Ribonucleic acid (RNA) is a complex polymer synthesised from DNA by emzymes. RNA is important for the construction of proteins in the body.

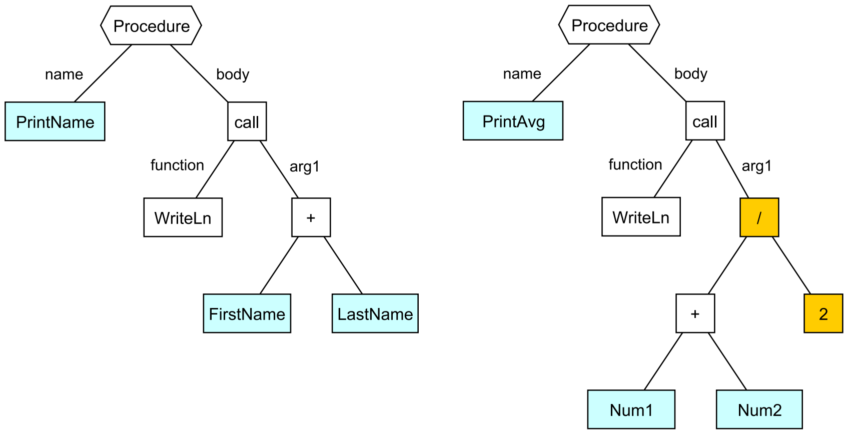


Figure 3.2: Coincidental structural similarity between program fragments

a similarity measure is precisely what is needed to evaluate the viability of a chosen name mapping.

To address the problem of name mapping, Sager *et al.* (2006) simply avoid comparing names at all. The same authors suggest the possibility of using a text-based similarity measure between names to decide if they can be mapped. Such a strategy may be feasible in domains with strong naming conventions.

3.1.2 Maximum Common Isomorphic Subgraph

A pair of graphs are considered isomorphic if they are equivalent, subject to some relabelling of vertices if the labels are not themselves significant. If two graphs have large subgraphs which are isomorphic, they could be considered similar. An example of maximum common isomorphic subgraphs between programs is shown in *Figure 3.3*.

The problem of finding the largest common isomorphic subgraph is a well known NP-Complete problem (Garey & Johnson, 1990). This usually necessitates the use of greedy algorithms to find approximate solutions. To obtain a numeric measure of similarity based on this technique, suppose that the graphs G_A and G_B are being compared. Let the size of the maximum common isomorphic subgraph be n_{common}. The similarity between G_A and G_B is then given in *Equation 3.5* (Sager *et al.*, 2006), which determines the average proportion common to either graph. Since $n_{\text{common}} \leq min(n_A, n_B)$, this score is in the $[0, 1]$ interval, as required.

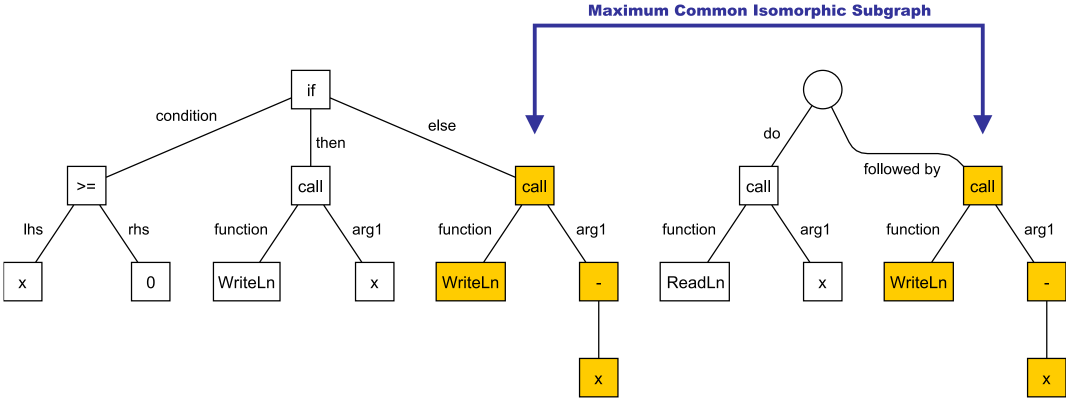
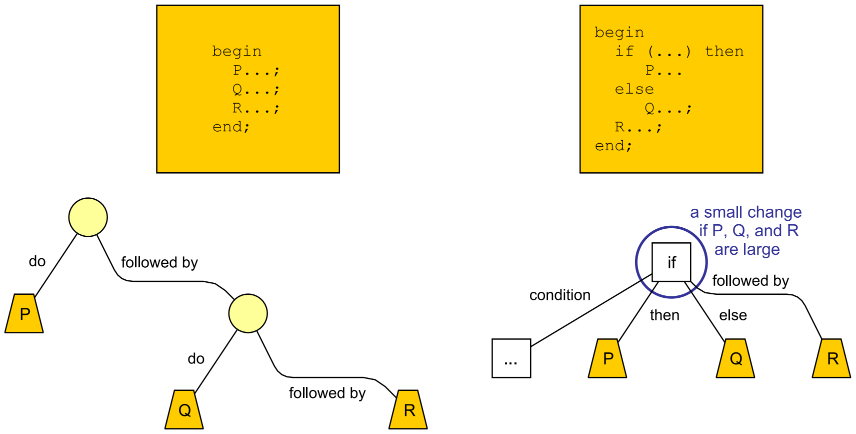


Figure 3.3: The maximum common isomorphic subgraph between two ASTs

$$Sim_{\text{MaxCommon}}(G_A, G_B) = 2\frac{n_{\text{common}}}{n_A + n_B} \tag{3.5}$$

This expression makes a certain amount of sense, but it does not lead to a reliable measure. Let us suppose that for G_A and G_B the expression yields an accurate similarity score. In order to be reliable, a change in G_A should result in a proportional change in the similarity score – and, by extension, a proportional change in the common subgraph. This is not a reasonable expectation.



```
begin
  P...;
  Q...;
  R...;
end;
```

```
begin
  if (...) then
     P...
  else
     Q...;
  R...;
end;
```

(a) The maximum common isomorphic subgraph of two program graphs, G_A and G_B

(b) The effect of a small change in the program text

Figure 3.4: The unreliability of the maximum isomorphic subgraph measure

The technique is vulnerable to even small changes in the graphs. Firstly, a change may occur in an unrelated part of G_A and not affect the similarity score very much at all. Secondly, a small change at a critical location of the common region, say the root of a tree, can split the common subgraph into multiple pieces. This scenario is demonstrated in *Figure 3.4*. The maximum common isomorphic subgraph of the example is shown in *Figure 3.4a*, containing three large subgraphs, P, Q and R. Suppose that one of the original graphs is edited to change the way P, Q and R are combined to form the larger graph. In *Figure 3.4b* this is done by introducing an *if* statement. When this happens, the connection between the three subgraphs is no longer subsumed by the maximum common isomorph. The new maximum common isomorphic subgraph becomes the largest of P, Q or R – all three of which are large. Thus the new isomorphic subgraph is expected to be substantially smaller than the original, even though the change is itself small. The significant reduction of the maximum common isomorphic subgraph leads to a significant change in similarity score, making the technique unreliable. In evidence of this argument, Sager *et al.* (2006) have found the tree edit distance similarity to be substantially more robust than methods based on common isomorphic subgraphs.

One interesting aspect of this technique is that it is not biased towards particular regions of the graph. In the case of trees, for example, it is not biased towards leaf vertices, since the maximum common subgraph could contain any connected subset of the available vertices. There may be a large common region near the roots of the trees, indicating that the overall design of the two programs is similar, but the fine detail near the leaves is different. This is in contrast to the edit distance measure in which no vertex can be considered in isolation from its children.

3.1.3 Tag and Path-Oriented Similarity

This sub-section discusses a sample of the large variety similarity measures developed for comparing XML documents. The measures discussed here are based on shared tags, or similar nesting of tags. However, the same methods are applicable in a more general context, since XML tags in XML documents are largely equatable with labels on the vertices of tree graphs.

Tag Similarity

Tag similarity is remarkably simple to compute and is based on finding the set of tags (labels) that are shared between two documents (trees). Suppose the technique is applied to two trees T_A and T_B, with label sets L_A and L_B, respectively. There are several ways to obtain a similarity score based on the label sets. A suggested method is the *Jaccard Coefficient* (Ganesan *et al.*, 2003), which yields the following equation:

$$Sim_{\text{Tags}}(T_A, T_B) = \frac{|L_A \cap L_B|}{|L_A \cup L_B|} \tag{3.6}$$

Tag similarity can be very effective in structured domains with an unbounded set of possible tags. Conversely, it is ineffective if a relatively small set of tags are available, as most documents will utilise all of them as a matter of course. In the context of program comparison, the labels on vertices are likely to relate to the constructs and operators supplied by the programming language. Since there is a modest finite set of these, the measure is expected to perform poorly for program comparison.

Weighted Tag Similarity

The weighted tag similarity method, as described by Buttler (2004), is an adaption of the previous measure to counter its stated problem. This is achieved by counting the number of occurrences of each tag, and then dividing the total number of shared tags by the average number of tags occurring in the documents.

In terms of the trees of the previous example, suppose that n distinct labels occur over the vertices of T_A and T_B, and that $w_{A,i}$ and $w_{B,i}$ represent the weight (number of occurrences) in each tree, respectively, of the ith label. Then the weighted tag similarity should be computed by:

$$Sim_{\text{WeightedTags}}(T_A, T_B) = 2\frac{\sum_{k=1}^{n} min(w_{A,k}, w_{B,k})}{\sum_{k=1}^{n} (w_{A,k} + w_{B,k})} \tag{3.7}$$

The weighted tag similarity is still a simple heuristic, but can be somewhat effective despite being cheap to compute. Buttler (2004) suggests that the measure is most effective in comparing documents governed by a strict schema that limits structural variation. Consider that under this measure two programs containing the same number of assignments, loops and decision constructs, but organised in totally unrelated ways, would still obtain the highest possible similarity score. For this reason, it is expected that the method would yield inflated scores if used for automated assessment.

Path Similarity

The weakness of tag-oriented similarity methods is that they consider tags in isolation, and disregard the relationships and relative positioning of tags. Path similarity measures can be thought of as a generalisation of tag similarity. Instead of comparing the set of tags occurring in each graph, path similarity is concerned with finding the set of common paths. The method is a little harder to formulate since the set of paths occurring in a graph may be infinite if the graph contains cycles. Even if the graph does not contain cycles, enumerating all paths is usually not feasible.

One way of thinking about path similarity is that it should estimate the probability of any randomly selected path of length k from the first graph, being found represented in the second. Determining this probability is potentially expensive, but a variety of approximations are possible.

An interesting approach is the application of document shingles to the set of paths present in each graph (Buttler, 2004). What this entails is selecting a fixed size random subset of the available paths and computing a hash value for each of these paths. This results in two sets of integers, one set for each graph, say H_A and H_B. The similarity between graphs G_A and G_B can now be approximated by the set resemblance between H_A and H_B, given by:

$$Sim_{\text{PathShingles}}(G_A, G_B) = \frac{|H_A \cap H_B|}{|H_A \cup H_B|} \tag{3.8}$$

The similarity score is a reasonable approximation for the probability of a randomly selected path being common to both graphs, but only if the paths from which they are derived are randomly chosen, and the sets H_A and H_B are large enough. Care must be taken in selecting random paths. It is convenient to develop random paths from the root of each graph, but this over-values vertices near the root, and tends to de-emphasise leaves.

3.2 Contemporary Similarity Measures

Several similarity measures, based on the propagation of scores through a propagation graph, have recently been developed – with important contributions by Melnik *et al.*

(2002), Jeh & Widom (2002), and Blondel & Van Dooren (2004). These measures are important because they can be applied to more general graphs than simply ordered trees. Thus these measures are described to measure the similarity between graphs G_A and G_B, with adjacency matrices A and B, respectively. Algorithms concerned with self-similarity specifically, can be treated as comparing G_A with itself.

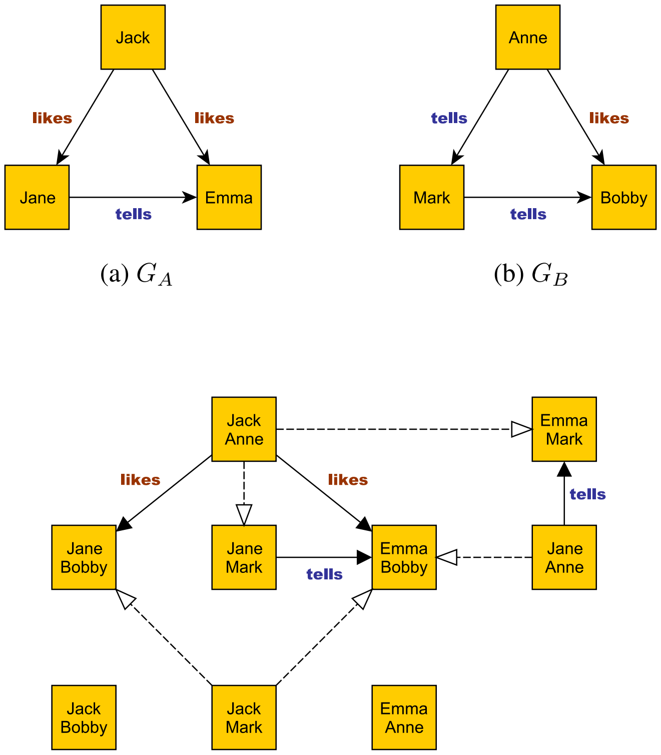
The measures presented here share some general characteristics. They each construct either a matrix or a vector describing pairwise vertex similarity scores. The issue of obtaining a composite graph similarity score from the individual vertex similarity scores has not received significant attention in literature. In this regard, Melnik *et al.* (2002) use the Euclidean length of the score vector as the overall similarity.

Similarity propagation algorithms are similar in many respects, and the product graph ($G_A \times G_B$) is an important concept relevant to each algorithm – so it is introduced in *Section 3.2.1* before the individual algorithms are discussed in detail. *Section 3.2.2* discusses *Similarity Flooding* in detail, paving the way for further discussion of similar algorithms, which are explained in relation to *Similarity Flooding*. The *SimRank* measure could have been chosen as the measure against which to relate the other algorithms, since it was developed independently at the same time as *Similarity Flooding*. However, the *Similarity Flooding* is the better choice for this purpose because it allows bidirectional propagation of similarity information. As such it is more useful in comparison with a larger variety of algorithms.

3.2.1 The Product Graph

In similarity propagation, the intent is usually that a pair of vertices are considered similar if their neighbours have been found to be respectively similar. This is made explicit through the product graph ($G_A \times G_B$). In the product graph every node represents a pair of vertices, one from each of G_A and G_B. The initial similarity scores of each vertex-pair propagate through the product graph, making updates at each step until convergence.

The construction of the product graph is relatively simple. *Figure 3.5* presents an example used in this section. The figure is based on the very abstract example offered in Melnik *et al.* (2002), but has been adapted to a practical real life scenario which models two social networks between small children (*Figures 3.5a* and *3.5b*).



(a) G_A

(b) G_B

(c) Product Graph of interactions between children

Figure 3.5: Example of the application of the product graph

In *Figure 3.5c* it can be seen that $G_A \times G_B$ has a vertex for every vertex-pair from the source graphs. The most important aspect of the figure that requires explanation is the placement of edges. Supposing that $p, q \in V(G_A)$, and $r, s \in V(G_B)$, formally if an $p \longrightarrow q$ edge exists in G_A, and an $r \longrightarrow s$ edge exists in G_B, then an $\langle p, r\rangle \longrightarrow \langle q, s\rangle$ edge is placed in the product graph. In the example, since *Jack likes Jane* and *Anne likes Bobby*, the product graph must have a directed edge from ⟨*Jack, Anne*⟩ to ⟨*Jane, Bobby*⟩. The edge labels are normally not significant in the product graph, but have been retained in *Figure 3.5c* to emphasise the relationship to the original graphs.

Looking again at the figure, it is seen that *Jack likes Jane* and *Anne tells Mark*, so the product graph has the directed edge ⟨*Jack, Anne*⟩ $\longrightarrow$ ⟨*Jane, Mark*⟩. This case is interesting because the original edge labels do not match. To maintain this distinction, all such product graph edges have been rendered as dashed lines in *Figure 3.5c*.

The use of product graphs in measuring similarity is important as reasoning in terms of a product graph helps understanding. However, it is not necessary that implementations construct the graph in order to operate. Mathematically, the computations used in similarity propagation can be formulated with matrix operations, whether in terms of the product graph or the original graphs. Some authors prefer to use the product graph as a thinking tool rather than an implementation tool. For example, the *SimRank* measure operates directly on the original graphs (Jeh & Widom, 2002). Other academics prefer working with the matrix formulation as it facilitates reasoning about convergence characteristics (Blondel *et al.*, 2004 and 2005).

Useful definitions

When expressed in terms of the original graphs, the solution is a matrix S_k with entry $[S_k]_{ij}$ being the similarity between vertex u_i of G_A and v_j of the compared graph. Alternatively, when using a product graph matrix formulation, the solution vector $\mathbb{S}_k$ is a column vector of similarity scores for each of the vertices in the product graph. While different symbols are used for S_k and $\mathbb{S}_k$, this reflects little more than variation in the approaches to the problem. The number of elements in these matrices is the same, and $\mathbb{S}_k$ can be obtained from S_k by stringing together its columns to form a single vector.

3.2.2 Similarity Flooding

The similarity measure introduced by Melnik *et al.* (2002) is called *Similarity Flooding*. They use a special variety of the product graph, which they call the *induced propagation graph*. This graph is obtained from the product graph after three additional steps. Firstly, each directed edge in the product graph for which the underlying edges have mismatched labels is removed from the graph. In the example of *Figure 3.5*, this means that all dashed-line edges are removed. The effect is to filter out improper vertex mappings, and their propagation paths. This action also removes any possibility of considering the similarity between edge labels, should they differ.

The second step is to update the propagation graph as follows: for every directed edge, an edge in the reverse direction is added. This allows similarity information to propagate in both directions. Melnik *et al.* (2002) are not specific about what happens if the reverse directed edge already exists. If the edge is duplicated, the graph is changed into a multi-graph. This is problematic for a variety of reasons – the most important is that not all current matrix formulations of similarity propagation work for multi-graphs.

The third and final step applied is to attach a weight to every directed edge. These weights are called *propagation coefficients*, and give the proportion of contribution between adjacent vertices. These coefficients may be determined in a variety of ways. One suggested method is: for every vertex m of the product graph, all the directed edges leaving m receive the same weight, which is given by $\frac{1}{Degree_{\text{out}}(m)}$. This approach, called the *inverse-product* formula, divides the similarity contribution evenly between all of the vertices immediately reachable from v. It is called the inverse-product because $Degree_{\text{out}}(m)$ is equal to $Degree_{\text{out}}(u) \times Degree_{\text{out}}(v)$, where m corresponds to $\langle u, v \rangle$, $u \in V(G_A), v \in V(G_B)$, and the graphs are not edge-labelled.

A similar procedure can be applied to edge-labelled graphs, where instead the similarity contributions are divided equally on a per label basis. This is the case in the example being discussed, and the cumulative effect of the three steps is shown in *Figure 3.6*. Here vertex 4 has two reversed edges leaving it, both with weight 1.0. This is because these reversed edges have different labels in the original graphs, *likes* from vertex 1 and *tells* from vertex 3, so the propagation contribution is not shared between them.

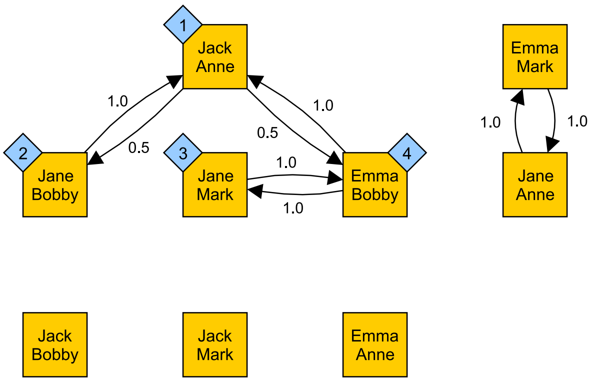


Figure 3.6: The induced propagation graph for the running example

Melnik *et al.* (2002) do not merely treat the propagation graph as an abstraction but use it directly in the computation. In particular they do not offer a detailed mathematical expression for the whole update operation across the entire graph. However, an expression for the propagation at each vertex is offered, though temporarily ignoring normalisation. *Equation 3.9* expresses the propagation update at the $\langle u, v \rangle$ vertex-pair after k iterations. It is observed that the first double summation corresponds to backward propagation of scores, while the second can be attributed to forward score propagation.

$$
\begin{aligned}
Propagate_{uv}\left(S_k\right) = \quad & \sum_{v \rightarrow q \,\in\, G_B} \; \sum_{u \rightarrow p \,\in\, G_A} w_{\langle u,v \rangle \rightarrow \langle p,q \rangle} \left[S_k\right]_{pq} \\
+ & \sum_{s \rightarrow v \,\in\, G_B} \; \sum_{r \rightarrow u \,\in\, G_A} w_{\langle r,s \rangle \rightarrow \langle u,v \rangle} \left[S_k\right]_{rs}
\end{aligned}
\tag{3.9}
$$

$$
Propagate_m\left(\mathcal{S}_k\right) = \sum_{m \rightarrow n \,\in\, G_{\text{prop}}} w_{m \rightarrow n}\left[\mathcal{S}_k\right]_n \; = \; \left[\mathcal{W}_{edges} \circ P\right]_m \mathcal{S}_k \tag{3.10}
$$

A corresponding expression is possible in terms of the mth vertex of the propagation graph (see *Equation 3.10*). In the equation W_{edges} is the matrix of edge weights. Similar algorithms often have the adjacency matrix (P) of the propagation graph in the position of W_{edges}, so it is retained for comparison.

Continuing by dropping the m subscript in *Equation 3.10* a formula is obtained that determines the propagation scores for each vertex simultaneously (*Equation 3.11*). Since this new equation covers all vertices, Melnik *et al.* (2002) introduce normalisation to ensure that scores remain in the $[0, 1]$ interval (regardless of how the weight matrix is determined), giving *Equation 3.12*. Now the fixed-point solution of this equation represents the final $\mathcal{S}$ score vector after the progagation of scores stabilises.

$$
Propagate\left(\mathcal{S}_k\right) = (\mathcal{W}_{edges} \circ P)\mathcal{S}_k \tag{3.11}
$$

$$
\mathcal{S} = \frac{(\mathcal{W}_{edges} \circ P)\mathcal{S}}{\left\|(\mathcal{W}_{edges} \circ P)\mathcal{S}\right\|_{\infty}} \tag{3.12}
$$

The natural choice to find the fixed point of *Equation 3.12* is to use the iterative formula $\mathcal{S}_{k+1} = \frac{(\mathcal{W}_{edges} \circ P)\mathcal{S}_k}{\left\|(\mathcal{W}_{edges} \circ P)\mathcal{S}_k\right\|_{\infty}}$. However, according to Blondel & Van Dooren (2004), this strategy may not converge. An important contribution by Melnik *et al.* (2002) is the empirical study of the convergence behaviour for several alternative forms of iteration formulae. The formulae studied are shown in *Table 3.1*, and were measured for a variety of graphs.

It is now known that if the initial similarity scores are positive ($\mathcal{S}_0 > 0$), convergence can be ensured for certain iteration formulae (Blondel *et al.*, 2005). This was applied in obtaining *Table 3.1* by using a minimum initial similarity score of 0.001, and stopping the iteration after $||\mathcal{S}_k - \mathcal{S}_{k-1}||_2 \leq 0.05$. The table summarises the average number of iterations required for convergence with each of the formulae.

	Formula	Avg. Iterations
basic	$\mathcal{S}_{k+1} = \frac{\mathcal{S}_k + Propagate(\mathcal{S}_k)}{\|\|\mathcal{S}_k + Propagate(\mathcal{S}_k)\|\|_\infty}$	*not reported*
variation A	$\mathcal{S}_{k+1} = \frac{\mathcal{S}_0 + Propagate(\mathcal{S}_k)}{\|\|\mathcal{S}_0 + Propagate(\mathcal{S}_k)\|\|_\infty}$	206
variation B	$\mathcal{S}_{k+1} = \frac{Propagate(\mathcal{S}_0 + \mathcal{S}_k)}{\|\|Propagate(\mathcal{S}_0 + \mathcal{S}_k)\|\|_\infty}$	49
variation C	$\mathcal{S}_{k+1} = \frac{\mathcal{S}_0 + \mathcal{S}_k + Propagate(\mathcal{S}_0 + \mathcal{S}_k)}{\|\|\mathcal{S}_0 + \mathcal{S}_k + Propagate(\mathcal{S}_0 + \mathcal{S}_k)\|\|_\infty}$	8

Table 3.1: Forms of iteration formulae studied by Melnik *et al.* (2002)

In the formulae presented in *Table 3.1*, a $\mathcal{S}_k$ term has a dampening effect on the update and is valuable in encouraging convergence. A $\mathcal{S}_0$ term may be used for the same purpose. An important observation is that adding a $\mathcal{S}_0$ term can change the fixed point value that is found on convergence. However, if each entry in $\mathcal{S}_0$ is small, the effect is likely to be negligible. Either way, adding a $\mathcal{S}_k$ term is the safer choice as this has no effect on the fixed point found.

Each of these formulae are easy to apply iteratively to find a fixed point. For example, *Table 3.2* shows the first six iterations of propagation through the largest connected component of *Figure 3.6*, using the *basic* iteration formula. From these results it can be seen that *Jack* and *Anne* are most similar, closely followed by *Emma* and *Bobby*. Looking at the original graphs, these relationships are not surprising, although intuitively one might have expected *Emma* and *Bobby* to be most similar since they are both *liked* and *told*, whereas *Jack* and *Anne* only have one common relationship – both *liking* others.

The question of why *Emma* and *Bobby* are not the most similar should be addressed. The reason is that the similarity score is not really between the vertices, but rather between their reachable subgraphs. In the example, the *Jack* and *Anne* vertices are roots of their respective graphs, making them strong candidates for the highest overall similarity score. If G_A and G_B are rooted acyclic graphs, this suggests taking the similarity between the roots as the overall graph similarity score.

	Similarity Scores				
	(Jack-Anne)	(Jane-Bobby)	(Jane-Mark)	(Emma-Bobby)	
k	Vertex 1	Vertex 2	Vertex 3	Vertex 4	$\|\mathcal{S}_k - \mathcal{S}_{k-1}\|_2$
0	0.50	0.50	0.50	0.50	
1	1.00	0.50	0.67	0.83	0.624
2	1.00	0.43	0.64	0.86	0.079
3	1.00	0.41	0.66	0.88	0.032
4	1.00	0.40	0.67	0.89	0.023
5	1.00	0.39	0.68	0.90	0.016
6	1.00	0.39	0.69	0.91	0.011

Table 3.2: First six iterations of the *basic* iteration formula (Melnik *et al.*, 2002)

3.2.3 Related Similarity Propagation Measures

Two related similarity propagation measures are now considered. The first is the *SimRank* measure (Jeh & Widom, 2002) which is significant as it is the only published alternative that does not require normalisation. The second is the measure studies in Blondel & Van Dooren (2004). This last measure is interesting in that Blondel & Van Dooren forgo the possibility of edge and vertex weights to better reason about the methods convergence properties.

The Jeh & Widom (2002) Measure: *SimRank*

Initially the *SimRank* measure introduced by Jeh & Widom (2002) seems to have several differences to *Similarity Flooding*. It is described for finding self-similarity, rather than similarity between a pair of graphs. In this way, the method can be thought to operate on the product graph. However, nothing in the method prevents finding similarity between disconnected subgraphs on G_A, so by extension the method could be adapted to compare distinct graphs.

Jeh & Widom (2002) use the unmodified product graph, rather than the induced propagation graph of Melnik *et al.* (2002). The effect is that propagation proceeds in only one direction between any pair of adjacent vertices. This usually means that the algorithm allows only forward propagation, but backward propagation can just as easily be achieved.

In terms of the relationships between linked web sites, backward propagation can be thought of as computing hub scores, while forward propagation computes authorities (Blondel & Van Dooren, 2004). Clearly, the most appropriate choice between *forward*, *backward* or *bi-directional* propagation depends on the data representation and the problem domain.

The propagation update for backward *SimRank* propagation is presented in *Equations 3.13* and *3.14*. In these equations, C is a free parameter chosen to encourage convergence. A typical choice might be $C = 0.8$. Since the factor C is accumulated through each propagation step, contributions across multiple edges decay with the sequence $\{0.8, 0.64, 0.51.2, 0.4096, \ldots\}$, which correspond to increasing powers of C. The effect is that the similarity contributions from distant vertex-pairs in the product graph are damped out. The parameter C can therefore be tuned to guarantee convergence, but reducing the value conteracts the main idea of propagating of scores.

$$Propagate_{uv}\left(S_k\right) = \frac{C}{Degree_{\text{out}}(u) \times Degree_{\text{out}}(v)} \sum_{v \to q \,\in\, G_A} \; \sum_{u \to p \,\in\, G_A} [S_k]_{pq} \qquad (3.13)$$

$$or\ldots$$

$$\begin{aligned} Propagate_{m}\left(\mathbb{S}_k\right) &= \frac{C}{Degree_{\text{out}}(m)} \sum_{m \to n \,\in\, G_A \times G_A} [\mathbb{S}_k]_n \\ &= \frac{C}{Degree_{\text{out}}(m)} P_m \mathbb{S}_k \qquad (3.14) \end{aligned}$$

In the same way as before, an equation for the propagation over all vertices is easily obtained (*Equation 3.15*). Here the weight matrix $W_{vertices}$ is carefully chosen to ensure that similarity scores remain in the $[0, 1]$ interval. Since for vertex m, at most a score of 1.0 could be propagated from each adjacent vertex in the product graph, the cumulative propagation it receives cannot exceed $Degree_{\text{out}}(m)$. The weight matrix is arranged to divide each cumulative score by the relevant vertex's *out-degree*, ensuring that the required score interval is maintained. This is important because normalisation is no longer needed.

$$Propagate\left(\mathcal{S}_{k}\right) = \mathcal{W}_{vertices} \circ P\mathcal{S}_{k} \tag{3.15}$$

$$where...$$

$$\mathcal{W}_{vertices} = \begin{bmatrix} \vdots \\ \frac{C}{Degree_{\text{out}}(m)} \\ \vdots \end{bmatrix} \tag{3.16}$$

The Blondel & Van Dooren (2004) Measure

The similarity measure by Blondel & Van Dooren (2004) is developed along similar lines to both *Similarity Flooding* and *SimRank*. The update equation they devise (*Equation 3.17*) is expressed in terms of the original graphs G_A and G_B, not the product graph.

$$S_{k+1} = \frac{B^T S_k A + B S_k A^T}{\|B^T S_k A + B S_k A^T\|_2} \tag{3.17}$$

An important contribution is that they formalise the relationship between this expression, and a similar one in terms of the product graph. Suppose that $vec(M)$ is an operator which constructs a single vector from the columns of the matrix M. Then it is known that using the Kronecker product, the operator supports the property that $vec(CSD) = (D^T \otimes C)vec(S)$. Allowing that $\mathcal{S}_k = vec(S_k)$, this enables the transformation of *Equation 3.17* into *Equation 3.19* (see *Equations 3.18a-e*).

$$\text{let } M_k = B^T S_k A + B S_k A^T \tag{3.18a}$$

$$\text{then } vec(M_k) = vec(B^T S_k A) + vec(B S_k A^T) \tag{3.18b}$$

$$= (A^T \otimes B^T)vec(S_k) + (A \otimes B)vec(S_k) \tag{3.18c}$$

$$= (A^T \otimes B^T + A \otimes B)vec(S_k) \tag{3.18d}$$

$$= (A^T \otimes B^T + A \otimes B)\mathcal{S}_k \tag{3.18e}$$

$$\mathcal{S}_{k+1} = \frac{M_k \mathcal{S}_k}{\|M_k \mathcal{S}_k\|_2} = \frac{\left(A^T \otimes B^T + A \otimes B\right) \mathcal{S}_k}{\|(A^T \otimes B^T + A \otimes B) \mathcal{S}_k\|_2} \tag{3.19}$$

Having obtained *Equation 3.19*, the connection to *Similarity Flooding* is clear. The term $A \otimes B$ gives the adjacency matrix of the product graph $G_A \times G_B$. Similarly $A^T \otimes B^T$ is the adjacency matrix for the same graph, but with the direction of all edges reversed. If G_A and G_B have no edge-labels, the sum of these matrices is exactly the adjacency matrix of the induced propagation graph used by Melnik *et al.* (2002). The Blondel & Van Dooren method is still unique in that it applies no edge or vertex weights.

The relationship with the *SimRank* measure can be seen in a similar way. If only the backward propagation part of *Equation 3.17* is considered, and the $vec(CSD)$ property is applied, then an expression much like the *SimRank* propagation function *Equation 3.15* is obtained. The difference once again being the absence of the weight matrix. In consequence, the most distinguishing characteristic between the similarity propagation algorithms considered is in how each assign vertex and edge weights in the propagation graph.

Further work focused on the conditions for convergence. The initial result offered is that while *Equation 3.19* is unlikely to converge, the sequences of odd and even terms do converge independently. The suggestion is that the vector to which the even term sequence converges be taken as the final similarity vector. Subsequent work has revealed other ways to achieve convergence (Blondel *et al.*, 2005).

3.3 Conclusion

This chapter first introduced the philosophical and practical problem in defining structural similarity, or comparing two structural similarity measures. This is a problem with no simple solutions. In practice, any similarity measure must be carefully evaluated in a given domain to determine if its results are useful.

Such problems notwithstanding, a variety of traditional structural similarity measures were considered. While the tree edit distance provides a respected similarity measure, practically speaking it is limited to ordered trees. Unfortunately, due to the nature of program source code diversity, it is unlikely that ordered trees will be an effective data structure to support the comparing of programs. This is because an ordered tree representation is over-specified as ordering is not essential in every program construct.

Techniques based on subgraph isomorphism as well as tag-oriented measures were also considered, but these are argued to have problems in producing reliable scores – at least in the current domain. The problems with tag-oriented methods can be overcome by considering tag nesting, so path-oriented similarity measures may be a viable solution. Nevertheless, path-oriented algorithms reveal nothing about the locus of similarity within the programs compared – an important requirement if the score must be understood in the context of the programs.

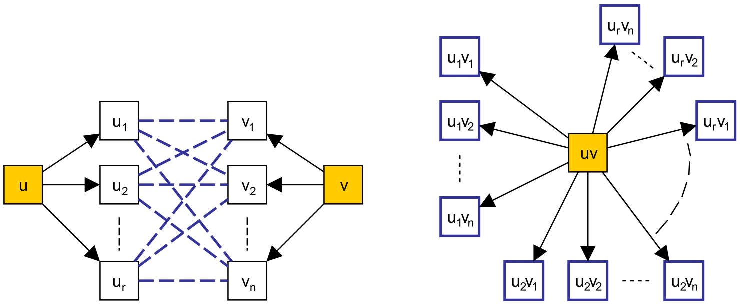
The prospect of determining pairwise similarity between constructs in candidate and ideal programs is appealing. As constructs will find representation in vertices, the similarity propagation algorithms make this possible. However, in assessment it is required that the resulting scores have intrinsic meaning[5]. This presents difficulty as Blondel & Van Dooren (2004), for example, require only that scores can be compared for an ordering relationship. The presence of a normalisation step in the iterative computation makes it impossible for scores to have independent meaning.

Consider, for example, that the $||\cdot||_\infty$ norm used by Melnik *et al.* (2002) guarantees that at least one pair of vertices will be considered completely similar (1.0). This will be true even if the vertices do not serve exactly equivalent roles in the respective graphs. The Jack and Anne vertices of *Figure 3.5* make a good example.

The Frobenius norm ($||\cdot||_2$) used by Blondel & Van Dooren (2004) exhibits a different problem. It ensures that the length of the score vector is always exactly 1.0. This means that at most one pair of vertices are completely similar, but even this case is exceedingly unlikely because it requires *all* other vertex pairs to be completely dissimilar (0.0). A distribution of low scores should rather be expected, even if the graphs being compared are identical. This is a significant problem if the final similarity is to be interpreted as a percentage mark.

The *SimRank* measure is promising in that it does not require the use of normalisation. It is not without problems, although they are not as severe. Consider the pair of vertices u and v, with children $u_1, u_2, \ldots u_r$ and $v_1, v_2, \ldots v_n$ respectively (*Figure 3.7a* and related *Figure 3.7b*). The *SimRank* measure sums the similarity contribution from every possible pairing of the vertex children. This is then divided by the number of pairings. In

[5] Ideally the score could be interpreted as a percentage.



(a) Portions of Graphs G_A and G_B

(b) Related portion of Product Graph $G_A \times G_B$

Figure 3.7: SimRank averages the similarity between all possible pairs of children

other words, the similarity between u and v is related to the average similarity between all possible assignments between their respective vertices. In domains where an optimal mapping between the children of u and v is expected, the averaging (which does not respect the ideal assignment between children) will yield pessimistic scores. None of the existing similarity propagation measures address this problem.

To illustrate the same issue in a more concrete way, consider what would happen if a program were compared with itself. At every vertex-pair there is a perfect mapping between their children (since they are identical), but the *SimRank* measure will still average this perfect mapping with the scores from all poorer mappings. The effect is that a program is never considered exactly similar to itself.

In the following chapter, this particular problem is addressed. A novel adaption of the *SimRank* measure is introduced to yield a new similarity measure called *Weighted Assignment Similarity*. This new algorithm is expected to be more appropriate for assessing program code by similarity, because the scores it produces may reasonably be regarded as percentages.

Chapter 4

A Novel Program Similarity Measure

In the previous chapter, it was observed that the *SimRank* measure averages the propagation scores of all contributing neighbour pairs. The result has a strong relationship to averaging the cumulative propagation score over all possible assignments between the neighbour sets. This is innappropriate as the similarity between a pair of vertices should be governed by a mapping between their parts, rather than all possible such mappings.

This chapter introduces a novel similarity measure called the *Weighted Assignment Similarity* measure, which does not exhibit this problem. Based on a single local mapping between the sets of child vertices, it supports a more natural idea of similarity and its score may be regarded as a percentage of mutual coverage.

Before the new measure is introduced the exact argument regarding the *SimRank* measure should be formalised. To facilitate this, *Definition 4.1* defines the important concept of a neighbour assignment.

Definition 4.1 (Neighbour Assignment)**.**
Given two vertices, u and v, with reachable neighbours $u_1, u_2, \dots u_r$ and $v_1, v_2, \dots v_n$, respectively; a neighbour assignment between u and v is a set H of vertex pairs, $\langle u_i, v_j \rangle$, with $i \in [1, r]$ and $j \in [1, n]$, such that each u_i and v_j occur in H at most once. Furthermore, it is required that the cardinality of H be $\min(r, n)$, such that H cannot be any larger. The similarity score of a neighbour assignment is considered to be the average similarity of its pairs.

The argument regarding the *SimRank* measure can now be formalised in *Lemma 4.1*.

Lemma 4.1 (Reinterpreting *SimRank* scores)**.**
Given two vertices, u and v, with reachable neighbours $u_1, u_2, \ldots u_r$ and $v_1, v_2, \ldots v_n$, respectively; the *SimRank* similarity between u and v is exactly equal to the average similarity score derived from all possible neighbour assignments between u and v.

Proof. Without loss of generality, it is assumed that $r \leq n$.

Thus the total number of neighbour assignments between u and v is given by

$$T = {}^{n}\mathrm{C}_{r} r! = {}^{n}\mathrm{P}_{r}$$

since r of vertex v's n neighbours must be selected for pairing with the neighbours of u, and the pairing can be done in $r!$ distinct ways.

Let H_i be the ith neighbour assignment between u and v.
Let $N = \{\langle u_i, v_j \rangle \mid i \in [1, r],\ j \in [1, n]\}$.

Then the average *SimRank* score propagated across all neighbour assignments is

$$\frac{\displaystyle\sum_{i=1}^{T} \frac{C}{r} \sum_{\langle p,q \rangle\, \in\, H_i} [S_k]_{pq}}{T}$$

Since each vertex pair occurs in the same number of assignments, in particular, $\frac{T}{n}$ of them, the expression can be rewritten as

$$\frac{T}{n} \times \frac{\displaystyle\frac{C}{r} \sum_{\langle p,q \rangle\, \in\, N} [S_k]_{pq}}{T}$$

$$= \frac{C}{rn} \sum_{i=1}^{r} \sum_{j=1}^{n} [S_k]_{ij}$$

which corresponds with *SimRank* propagation (*Equation 3.13*). ■

The significance of this proof is best seen in an example. *Figure 4.1a-f* shows all possible neighbour assignments when comparing structured representations of the expressions $9 + x^2$ and $x^2 + y + 9$. There are six such assignments, but *Figure 4.1b* is clearly the appropriate mapping. The remaining five assignments only reduce the effectiveness of any similarity measure that considers them all. Since the number of neighbour assignments is combinatoric, the difficulty is in selecting the ideal assignment. This leads to a new similarity measure introduced in the next section.

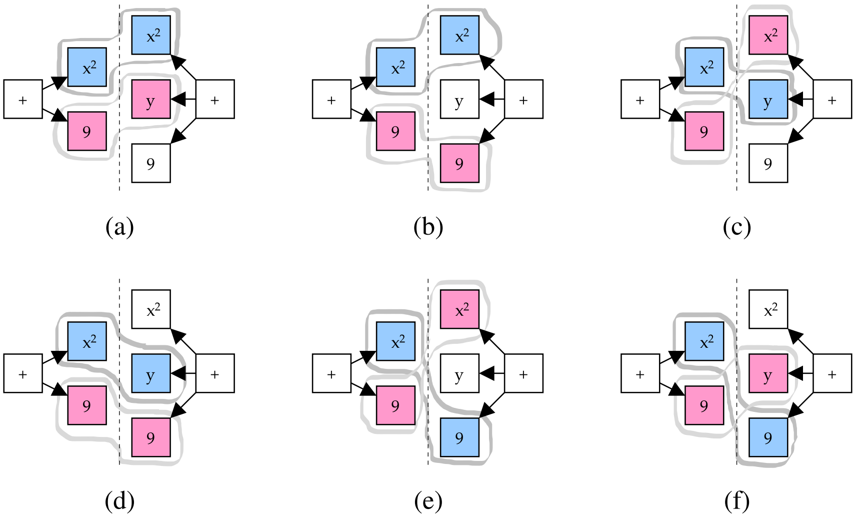


Figure 4.1: Neighbour assignments for comparing $9 + x^2$ **with** $x^2 + y + 9$

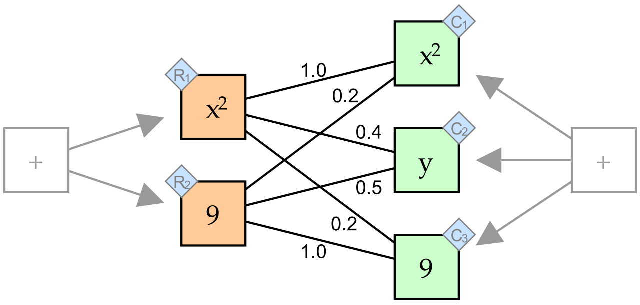
4.1 Weighted Assignment Similarity

The *Weighted Assignment Similarity* measure is based on finding the ideal neighbour assignment between each pair of vertices considered. The ideal assignment is considered to be the neighbour assignment with the largest average pairwise similarity. This assignment can only be found if the similarity between vertices is already known. Here this is not the case, as finding the similarity is itself the broader objective.

The approach taken in the *Weighted Assignment Similarity* measure is to approximate the ideal neighbour assignments using the vertex similarity scores resulting from the preceding iteration. The premise is that as similarity scores become more accurate with successive iterations, the chosen neighbour assignments begin to reflect the set of ideal assignments.

Finding an optimal neighbour assignment is not difficult, given reasonable approximations for similarity scores. If the degree of the vertices is known to be small, then a *brute-force* search over all assignments is easy to implement and will probably be fast enough. If the graph representations of the problem domain allow each vertex to have an arbitrarily large number of neighbours, then the combinatoric explosion necessitates a better algorithm. Fortunately this is a well studied problem.

Consider the previous example. *Figure 4.2a* shows that the similarity information between neighbours at any given iteration can be represented as a bipartite graph. Using vertex pair similarity scores as edge weights, the problem of finding an optimal neighbour assignment is now equivalent to that of finding a *maximum score maximal matching* in the bipartite graph. Saip & Lucchesi (1993) offer a review of several good algorithms for this problem.



(a) Neighbour similarity as a complete bipartite graph

$$\begin{bmatrix} \mathbf{1.0} & 0.4 & 0.2 \\ 0.2 & 0.5 & \mathbf{1.0} \end{bmatrix}$$

(b) Similarity Score Matrix

$$\begin{bmatrix} \mathbf{0.0} & 0.6 & 0.8 \\ 0.8 & 0.5 & \mathbf{0.0} \end{bmatrix}$$

(c) Dissimilarity Matrix

$$\begin{bmatrix} \mathbf{1} & 0 & 0 \\ 0 & 0 & \mathbf{1} \end{bmatrix}$$

(d) Munkres Solution

Figure 4.2: The example of comparing $9 + x^2$ with $x^2 + y + 9$ continued

In this research the well known algorithm due to Munkres (1957) has been selected[1], as it is reasonably efficient and is not as complex as some alternatives. The MUNKRES algorithm operates on the weight matrix, as in *Figure 4.2b*. The rows and columns of the matrix correspond to the vertices of *Figure 4.2a* (see the diamond tags). It is noted however, that the MUNKRES algorithm can only directly be used to find a *minimum cost* maximal matching. By using $cost_{u,v} = 1.0 - sim_{u,v}$, the dissimilarity matrix is obtained (*Figure 4.2c*), for which the minimisation problem is equivalent to the original maximisation problem. The MUNKRES algorithm proceeds to construct a mask matrix

[1] The MUNKRES algorithm is a reformulation of the pen-and-paper Hungarian Method, for electronic implementation.

(*Figure 4.2d*) which represents the chosen optimal assignment – optimal subject to the accuracy of the current similarity scores.

4.1.1 Basic Mathematical Form

The means to establish the locally optimal neighbour assignments are necessarily algorithmic, and do not have an analytic form. To incorporate them into a mathematical expression for the *Weighted Assignment Similarity* measure, some definitions are first required.

$Assign(\mathcal{S}_k, \langle u, v\rangle)$ a function that determines the optimal neighbour assignment set between vertices u and v, subject to similarity scores $\mathcal{S}_k$

$$a_{k,\langle u,v\rangle\rightarrow\langle p,q\rangle} = \begin{cases} 1 & \text{if } \langle p,q\rangle \in Assign(\mathcal{S}_k, \langle u,v\rangle) \\ 0 & \text{otherwise} \end{cases}$$

$$w_{\langle u,v\rangle\rightarrow\langle p,q\rangle} = \text{weight of the } \langle u,v\rangle \rightarrow \langle p,q\rangle \text{ product graph edge}$$

$$\mathcal{A}_{k,\langle u,v\rangle} = \text{the row vector with columns } a_{k,\langle u,v\rangle\rightarrow\langle p,q\rangle}\ \forall\ \langle p,q\rangle$$

$$\mathcal{A}_k = \text{the square matrix with rows } \mathcal{A}_{k,\langle u,v\rangle}\ \forall\ \langle u,v\rangle$$

$$\mathcal{W} = \text{weight matrix with } w_{\langle u,v\rangle\rightarrow\langle p,q\rangle}\ \forall\ \langle u,v\rangle,\ \forall\ \langle p,q\rangle$$

Now the new *Weighted Assignment Similarity* measure can be defined by the propagation function shown in *Equation 4.1*. (The matrix form is given by *Equation 4.2*). This is the most basic form of the measure's propagation function. Incidentally, the name *Weighted Assignment Similarity* is taken from the three factors that occur in this function. It should be noted that this equation concerns non-leaf vertices only. Leaf vertices receive no propagation scores in backward propagation similarity. In the absence of a measure of similarity between the labels of leaves, all leaf vertices must be considered indistinguishable. A later section addresses support for vertex and edge label similarity, obtaining a unified formulation.

$$Propagate_{uv}(\mathcal{S}_k) = \sum_{u \to p \,\in\, G_A} \sum_{v \to q \,\in\, G_B} w_{\langle u,v\rangle \to \langle p,q\rangle} \; a_{k,\langle u,v\rangle \to \langle p,q\rangle} \; [\mathcal{S}_k]_{\langle p,q\rangle} \tag{4.1}$$

$$Propagate(\mathcal{S}_k) = (\mathcal{W} \circ \mathcal{A}_k)\mathcal{S}_k \tag{4.2}$$

4.1.2 Choosing Weights

There are two important criteria that govern how edge weights are chosen:

1. The score propagated to each vertex pair must be in the interval $[0, 1]$.
2. The weights must be chosen to promote meaningful convergence.

To address these issues the edge weight matrix is composed of two distinct column matrices by the Kronecker product. These additional matrices represent two different kinds of vertex weights, and are called the *source* and *sink* weights. The means of combining them is expressed in *Equation 4.3*. The choice of these weights is further governed by a vertex significance function *(Sig)*, as shown in *Equations 4.4* and *4.5*.

$$\mathcal{W} = \mathcal{W}_{\text{source}} \otimes \mathcal{W}_{\text{sink}}{}^{T} \tag{4.3}$$

$$[\mathcal{W}_{sinks}]_{\langle u,v\rangle} = Sig(u) + Sig(v) \tag{4.4}$$

$$[\mathcal{W}_{sources}]_{\langle u,v\rangle} = \frac{1}{\mathcal{D}_{u,v}} \tag{4.5}$$

$$with$$

$$\mathcal{D}_{u,v} = \sum_{u \to p \,\in\, G_A} Sig(p) + \sum_{v \to q \,\in\, G_B} Sig(q) \tag{4.6}$$

It is useful to consider how this affects the form of the propagation function. By substitution into either *Equation 4.1* or *Equation 4.2* the new form in *Equation 4.7* can be obtained.

$$Propagate_{uv}(\mathcal{S}_k) = [\mathcal{W}_{\text{source}}]_{\langle u,v\rangle} \sum_{u \to p \,\in\, G_A} \sum_{v \to q \,\in\, G_B} [\mathcal{W}_{\text{sink}}]_{\langle p,q\rangle} \, a_{k,\langle u,v\rangle \to \langle p,q\rangle} \, [\mathcal{S}_k]_{\langle p,q\rangle} \tag{4.7}$$

$$\begin{aligned} & Propagate_{uv}(\mathcal{S}_k) \\ & \qquad \leq \sum_{\langle p,q\rangle \,\in\, Assign(\mathcal{S}_k,\langle u,v\rangle)} \left[\mathcal{W}_{\text{sink}}\right]_{\langle p,q\rangle} && (4.8) \\ & \qquad \leq \sum_{u \to p \,\in\, G_A} Sig(p) + \sum_{v \to q \,\in\, G_B} Sig(q) = \mathcal{D}_{u,v} && (4.9) \end{aligned}$$

Two useful properties are observed in *Relations 4.8* and *4.9.* Firstly, the relations guarantee that the divisor in each *source* weight ($\mathcal{D}_{u,v}$) is at least as large as the accumulation of *sink* weights for any given assignment. This means that every propagation score is certain to be in the interval $[0, 1]$.

The second observation is just as important. If a perfect neighbour assignment exists with each pair being perfectly similar, then the accumulation of *source* and *sink* weights neutralises one another. Thus equivalent vertices are guaranteed a propagation score of exactly 1. This may seem self-evident, but it is important because it is a property not guaranteed by any previous similarity propagation measure.

The remaining influence on the weights is the choice of the significance function. Three alternatives are considered:

1. Uniform: $Sig_{\text{uniform}}(\nu) = 1$
2. Reachability-assigned: $Sig_{\text{reach}}(\nu) = \left|Subgraph_{\text{reachable}}(\nu)\right|$
3. Height-assigned: $Sig_{\text{depth}}(\nu) = \left|Paths_{\text{longest-acyclic}}\left(Subgraph_{\text{reachable}}(\nu)\right)\right|$

Uniformly chosen vertex significance simply takes advantage of the properties of *source* and *sink* weights that have already been established. The first alternative is based on the size of the subgraph that is reachable from the vertex considered. Consider the following simple example. Suppose a neighbour assignment (H) of two vertex pairs is chosen, with $H = \{\langle p, q\rangle, \langle r, s\rangle\}$. Suppose also that the corresponding similarity scores are 0.0 and 0.5, but that p and q are leaf vertices, while r and s are the roots of sizeable subgraphs. It makes sense that r and s should have a larger influence over the similarity than the leaf vertices. The reachability-assigned significance function attempts to capture this logic by biasing the propagation summation to the neighbours that gather similarity information from a larger portion of each graph.

Height-assigned significance is similar, but is less susceptible to domain specific normalisation procedures. For example, converting logic expression graphs into disjunctive normal before comparison may change the size of the graphs substantially. In such cases, height is more stable than size, so it makes a good measure of similarity significance.

4.1.3 Convergence over DAGs

The convergence behaviour of the *Weighted Assignment Similarity* measure is in general hard to determine. In particular, the finding of neighbour assignments cannot be easily reasoned about mathematically. Whether the measure converges for all general graphs is not known. However, it is still valuable to consider the convergence behaviour over restricted graphs. Directed acyclic graphs (DAGs) are a useful class of graphs. While they do not allow cycles, they are still considerably more general than ordered trees, and are especially useful for describing program structure. This sub-section offers a proof in *Theorem 4.3* that the *Weighted Assignment Similarity* measure converges over DAGs. However, a small lemma is first introduced here to simplify the main proof.

Lemma 4.2 (A lemma to assist in the proof of convergence)**.**
For a fixed value M and an initial value $\mathbb{S}_0$, the iteration formula $\mathbb{S}_{k+1} = \frac{\mathbb{S}_k + M}{2}$ converges to M.

Proof. The iteration produces the sequence

$$\mathbb{S}_0,\ \frac{1}{2}\mathbb{S}_0 + \frac{1}{2}M,\ \frac{1}{4}\mathbb{S}_0 + \frac{3}{4}M,\ \frac{1}{8}\mathbb{S}_0 + \frac{7}{8}M,\ \ldots$$

So the nth term in the sequence is

$$\mathcal{S}_n = \frac{1}{2^n}\mathcal{S}_0 + \frac{2^n - 1}{2^n}M \;=\; 2^{-n}\mathcal{S}_0 + (1 - 2^{-n})M$$

$$\begin{aligned}
\mathcal{S}_\infty &= \lim_{n\to\infty}\left(2^{-n}\mathcal{S}_0 + \left(1 - 2^{-n}\right)M\right) \\
&= \lim_{n\to\infty} 2^{-n}\mathcal{S}_0 + \lim_{n\to\infty}\left(1 - 2^{-n}\right)M \\
&= \left(\lim_{n\to\infty} 2^{-n}\right)\mathcal{S}_0 + \left(\lim_{n\to\infty}\left(1 - 2^{-n}\right)\right)M \\
&= 0 \times \mathcal{S}_0 + 1 \times M \;=\; M
\end{aligned}$$

■

Theorem 4.3 (DAG convergence of *Weighted Assignment Similarity*)**.**
Given a pair of graphs, G_A and G_B, such that the product graph $G_A \times G_B$ is a directed acyclic graph; the *Weighted Assignment Similarity* measure over G_A and G_B, with iteration formula $\mathcal{S}_{k+1} = \frac{\mathcal{S}_k + Propagate(\mathcal{S}_k)}{2}$, will converge to a solution.

Proof. The proof offered is a proof by mathematical induction for backward propagation. The base case of which is assured: convergence is certain for any DAG with only one vertex.

Let G_j be a DAG with j vertices for which the similarity measure converges.

Suppose a new DAG, G_{j+1}, is constructed by adding a vertex r to G_j. Let G_{j+1} contain $n \leq j$ directed edges of the form $r \to v_i, v_i \in V(G_j)$, such that r is a root vertex of G_{j+1}.

It is observed that scores do not propagate from r to any other vertex under *backward* propagation. Thus the propagation of scores for the remaining vertices must occur within $G_{j+1} - r$ which is simply G_j.

It is already accepted that the similarity measure converges over G_j, so in applying the iterative procedure over G_{j+1} all vertices that also occur in G_j will converge to the same values as they did in G_j.

Let k be the number of iterations after which this convergence occurs.

Now the propagation score at r may be determined as the measure requires:

$$Propagate_r(\mathcal{S}_k) = [\mathcal{W}_{\text{source}}]_r \sum_{r \to v \,\in\, G_A \times G_B} \left[\mathcal{W}_{\text{sink}}\right]_v \; a_{k,r \to v} \; [\mathcal{S}_k]_v \qquad (4.10)$$

Since the calculation of the score at r has been determined from stable scores, none of which depend upon r, *Lemma 4.2* ensures that the iteration formula also converges for r.

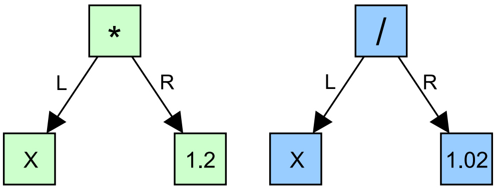
Thus by mathematical induction, and the fact that the construction of G_{j+1} for sufficiently large j subsumes the construction of any given directed acyclic graph, the similarity measure converges for all directed acyclic graphs, including $G_A \times G_B$. ■

An interesting feature of this proof is that it suggests a very efficient implementation strategy, if the measure is only going to be applied to DAGs. The measure may be applied in a bottom-up procedure from leaves to roots. This ordering of vertices is essentially a reversed topological order. Each step is concerned with only one new vertex. The measure would have already converged for the connected subgraph it derives. In addition, the final score for each new vertex can be determined in a single step.

4.1.4 Supporting Label Similarity

Almost all practical applications of graphs require the meaningful use of vertex and edge labels. When representing programs as graphs, vertex labels describe which constructs are being represented. Edge labels may be used to create distinction between sibling vertices. This makes it particularly important to have a strategy for dealing with these domain specific vertex and edge attributes.

In this regard, little has been reported in prior work. Jeh & Widom (2002) suggest that domain specific information could be incorporated with their *SimRank* measure, but do not describe a mechanism by which this may be achieved. The measure introduced by Melnik *et al.* (2002) takes advantage of any domain specific information captured on edge labels. This occurs as a by product of their extended product graph construction process, and only supports exact equivalence of edge labels. For example, they do not



(a) G_A: $X \times 1.2$ (b) G_B: $X \div 1.02$

Figure 4.3: Expression graphs with labelled vertices and edges

support using a text similarity measure on similar edge labels. It is significant that it is not sufficient to simply apply a label similarity measure to initialise the $\mathcal{S}_0$ matrix. For example, Blondel *et al.* (2005) have shown that the similarity propagation formulation that they have studied converges to a unique solution. As a result, when it converges, it converges to the same result regardless of which positive values populate $\mathcal{S}_0$.

In this section, the *Weighted Assignment Similarity* measure is further extended to support local domain specific similarity measures between arbitrary vertex and edge attributes. This work represents the first formally specified method of integrating local domain specific similarity measures into a similarity propagation measure.

The strategy followed here is to adapt the product graph for dealing with attribute similarities. In this way, the existing similarity propagation mechanism can support propagating the attributes' similarities in the same way as structural similarities. This is best discussed in the context of a small example. *Figure 4.3* shows two small expression graphs. The operators are vertex labels with significance in the problem domain, and should influence the overall similarity. In addition, numbers that are close to one another should be considered more similar than others. Edge labels are also significant. In this case, these labels provide a local ordering relationship for operands. While not important for multiplication, it is critical for division since the operator is not commutative.

A solution to the problem of vertex labels is presented in *Figure 4.4*. After constructing the product graph, every vertex pair for which a domain specific label similarity measure is available, is given an additional *pseudo* vertex (the purple diamonds). These additional vertices are all leaves, and serve only to calculate the label similarity and propagate that score like any other vertex pair in the neighbour assignment.

The same technique can be extended to support edge label similarity. First each labelled edge is split into two edges with the label being attributed to an intermediate *pseudo*

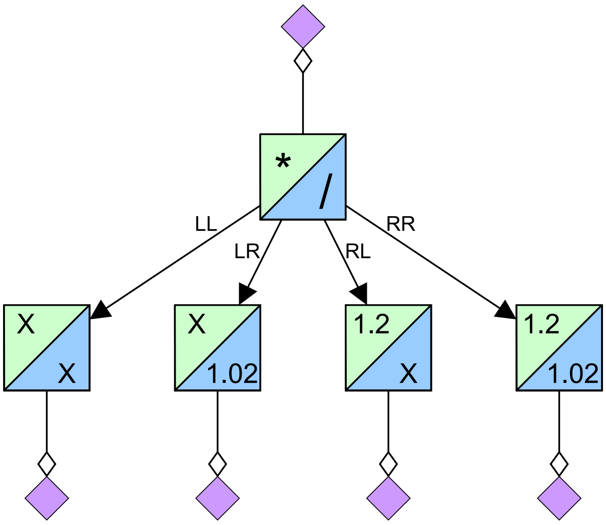


Figure 4.4: Augmented product graph of G_A and G_B

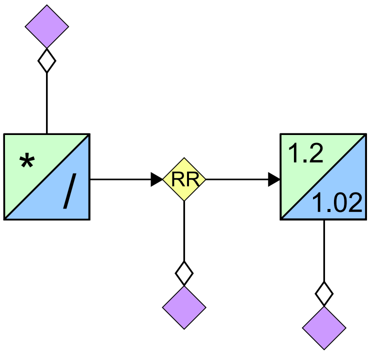


Figure 4.5: Vertex inserted to support edge label comparison

vertex. This is demonstrated in *Figure 4.5* for the *RR* edge of the example. The new vertex receives the exact same treatment as normal labelled vertices described above. In so doing, the comparison of edge labels is indirectly supported.

In the discussion of these techniques, the newly introduced vertices were called *pseudo* vertices. The reason for this is that these effects can be achieved without actually constructing the product graph or modifying it. Once again, this requires the introduction of additional notation.

$V_{\text{sim},\langle u,v\rangle}$	the domain specific local similarity between the labelled attributes of vertices u and v
$E_{\text{sim},\langle u,v\rangle\rightarrow\langle p,q\rangle}$	the domain specific local similarity between the labelled attributes of edges $u \rightarrow p$ and $v \rightarrow q$
$V_{\text{sig},\langle u,v\rangle}$	the significance of any local vertex similarity found between u and v (usually 0 or 1)
$E_{\text{sig},\langle u,v\rangle\rightarrow\langle p,q\rangle}$	the significance of any local edge similarity found between edges $u \rightarrow p$ and $v \rightarrow q$ (usually 0 or 1)

Consider the edge-vertex-edge combination that results when a labelled edge is split. It is useful to construct an equation for the two step propagation across both new edges, since the propagation over the original edge occurs in a single step. The expression for this very special case is offered in *Equation 4.11.* An important observation should be made about this equation. If the relevant E_{sig} value is zero, the equation yields the score that would normally be propagated over the original labelled edge.

$$\xi_{\langle u,v\rangle\rightarrow\langle p,q\rangle}(\mathcal{S}_k) = \frac{[\mathcal{W}_{\text{sink}}]_{\langle p,q\rangle}\,\mathcal{S}_{k,\langle p,q\rangle} + E_{\text{sig},\langle u,v\rangle\rightarrow\langle p,q\rangle}\,E_{\text{sim},\langle u,v\rangle\rightarrow\langle p,q\rangle}}{[\mathcal{W}_{\text{sink}}]_{\langle p,q\rangle} + E_{\text{sig},\langle u,v\rangle\rightarrow\langle p,q\rangle}} \tag{4.11}$$

To obtain the final and most general form of the propagation function, the vertex labelled similarity needs to be addressed. Recall that vertex label similarity is treated as though there were an additional neighbour in the neighbour assignment. The simplest solution is to incorporate this new neighbour directly into the propagation function, yielding *Equation 4.12.* This equation is admittedly complex in appearance. It is nevertheless understandable when one observes that it reduces to the earlier form when V_{sig} and E_{sig} values are zero.

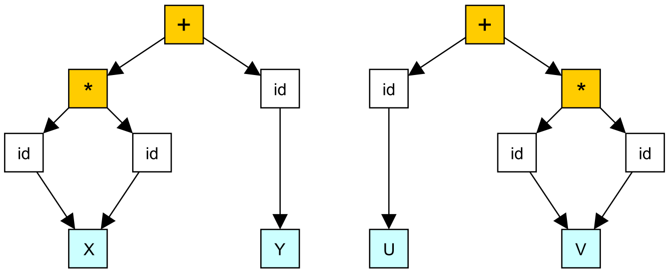
$$Propagate_{u,v}(\mathcal{S}_k) = \frac{\displaystyle\sum_{u \to p \,\in\, G_A} \sum_{v \to q \,\in\, G_B} [\mathcal{W}_{\text{sink}}]_{\langle p,q \rangle}\, a_{k,\langle u,v \rangle \to \langle p,q \rangle}\, \xi_{\langle u,v \rangle \to \langle p,q \rangle}(\mathcal{S}_k) + V_{\text{sig},\langle u,v \rangle} V_{\text{sim},\langle u,v \rangle}}{\mathcal{D}_{u,v} + V_{\text{sig},\langle u,v \rangle}} \tag{4.12}$$

4.1.5 Mapping Identifiers

An important problem in the domain of the current research is that different programs, even if similar, use different sets of variable names. If programs are to be compared for similarity, this problem must be addressed. As yet, good generally applicable solutions have not seen any significant representation in literature. The nature of the problem is such that establishing a mapping between identifiers is easy, but there are many possible mappings, and obtaining a good selection is hard. A good variable mapping should improve the accuracy of the overall measured similarity between the two programs. This suggests that there is a co-dependence between finding a good variable name mapping, and measuring the similarity of programs. A good mapping is required to accurately measure similarity, and a good similarity measure is required to assess the quality of a chosen mapping.

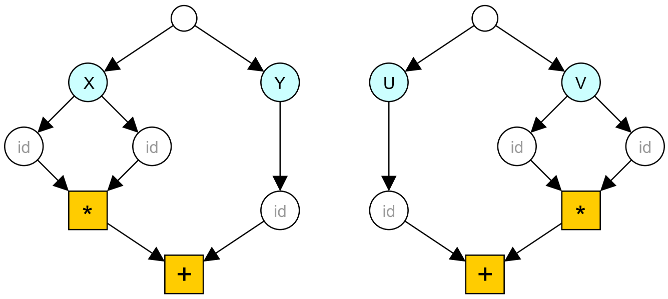
While this situation does seem bleak, it also suggests a solution. Since identifiers typically occur as leaves in program graphs, they are characterised by the paths that must be travelled through the graph to reach them. Stated another way, two variables should be considered similar if they occur in similar structural context of use. Furthermore, the paths that lead to identifiers are unlikely to contain identifiers themselves, as internal vertices consist mostly of operators and constructs. This suggests that a characteristic graph can be constructed which describes the use of the identifiers in the program, without itself relying on arbitrary identifiers. The *Weighted Assignment Similarity* measure may then be applied between the characteristic graphs to find the optimal mapping between the identifier sets, as well as their relative similarity.

The idea is illustrated in the following example. Consider the two graphs for program fragments, *Figures 4.6a* and *4.6b*. These fragments each have two variables. The characteristic graphs describing the use of their variables is offered in *Figures 4.7a* and *4.7b*. In these graphs, the original *id* vertices are retained for ease of construction, but hold only structural significance.



(a) G_A (b) G_B

Figure 4.6: Graphs for a pair of similar program fragments



(a) Characteristic graph for G_A (b) Characteristic graph for G_B

Figure 4.7: Characteristic graph for the use of identifiers in G_A and G_B

Applying the *Weighted Assignment Similarity* measure to the characteristic graphs yields a matrix of similarity scores comparing each vertex pair between these two graphs. Usually the similarity between root vertices would be of most interest. In this case, it is the similarity between the immediate neighbours of the roots that hold the similarity between identifiers. The result is that vertices X and V are strongly similar, and so are Y and U. This information can be used to construct a variable name mapping. Alternatively, the resulting pairwise similarity scores may be used exactly as they are, and utilised as a domain specific label similarity function. Either way, the *Weighted Assignment Similarity* measure is reapplied between the original graphs with an effective general way to relate the identifier sets.

4.2 Conclusion

This chapter introduced a novel similarity propagation measure. It is notionally similar to prior work, in particular the *SimRank* measure, but introduces a sufficiently large number of innovations to be considered a new measure in its own right.

Several critical problems in comparing programs for similarity are directly and indirectly addressed by the *Weighted Assignment Similarity* measure. The first is that it is imperative that similarity scores are both independently meaningful and can be interpreted as a percentage score. For the reasons described in chapter 3, none of the previous similarity propagation measures have this property, even though they produce scores in the $[0, 1]$ interval. The novel feature of the new measure is that similarity only propagates across the locally optimal neighbour assignment, rather than across all possible neighbour assignments. The distinction may seem small, but has not been considered in any prior work. Furthermore, it is a critical feature in allowing similarity scores to be thought of as percentages.

The second important feature of the similarity measure is that it directly supports the use of attributes on vertices and edges, as well as domain specific similarity measures over these attributes. Prior work has suggested that this is possible, but the method in the *Weighted Assignment Similarity* measure is the first formally published general technique by which this may be done.

The third major contribution is that the similarity measure can be applied in a two step process to address the problem of distinct variable names in compared programs. This approach is elegant and makes efficient use of the similarity measure as an existing resource.

Finally it is observed that convergence over directed acyclic graphs not only guaranteed, but very efficient implementations thereof are possible. In this case, careful implementations need to work proportional to the number of vertices in the product graph without necessarily constructing the graph, although finding optimal neighbour assignments is still governed by the degree of the vertices.

These properties make the *Weighted Assignment Similarity* measure a strong candidate for measuring the similarity between programs. The following chapter considers its application in the case study of assessing student programs developed in Object Pascal.

Chapter 5

Case Study: Object Pascal Assessment

The previous chapter described the novel *Weighted Assignment Similarity* measure, which is similar to the *SimRank* measure. *SimRank* scores are taken as the average propagation from *all* neighbour assignments (both good and bad) and there are generally more poorly matched neighbour assignments than well-matched ones. In contrast, the *Weighted Assignment Similarity* measure uses a locally optimal neighbour assignment. Although the same approach has not been taken in other published works, it is proposed that this is a critical requirement for a graph similarity measure to be used in assessment.

This chapter is concerned with evaluating the use of the new measure by way of a case study assessing novice Object Pascal programs. Object Pascal is currently still a popular programming language, although not as popular as Java or C#. Nevertheless, Object Pascal remains a reasonable choice for this case study as similar results are expected over all programming languages that use a largely *imperative* programming style within sub-routines, such as Java and C#.

Section 5.1 is concerned with the diversity of the solution space for any given programming assignment. Program diversity is an important problem as there are usually an unbounded number of possible solutions to each assignment. The section enumerates the most important sources of program diversity and explains how these are addressed in this study. The second section discusses some finer points regarding the accumulation of similarity scores and their conversion to marks on the scale set for the assignment. The final section discusses the experiments that were conducted to evaluate the application of the similarity measure in this case study.

5.1 Program Diversity

The amount of variation in solutions that any programming assignment admits is inextricably linked to the tightness of the assignment specification. Program diversity in turn negatively affects all forms of automated assessment. The simplest way to address this problem is to make the assignment specification more precise, however, this is a point of contention among academics (see *Section 2.3.1*).

The diversity that may be present in the solution space is manifested in two forms: *essential* and *spurious diversity*. These are analogous to *essential* and *spurious ambiguity* in languages, as described by Grune & Jacobs (1990). *Essential diversity* occurs when program phrases are expressed differently and have different semantics. For example, *while* and *repeat-until* loops are expressed differently and have subtly different semantics. Unless a concession is made, occurrences of the two constructs must be treated as *essential diversity*. *Spurious diversity* covers those cases where phrases are expressed differently, but are semantically equivalent. For example, incrementing a variable through assignment is semantically equivalent to an appropriate call to the `Inc` procedure. *Spurious diversity* complicates assessment as it gives rise to a multitude of variations that differ only in semantically insignificant local detail.

Little can be done with regard to *essential diversity* besides enlarging the set of standard solutions against which the programs are assessed. On the other hand, *spurious diversity* should be identified and addressed wherever possible, as this makes the assessment strategy more effective without burdening the educator. This section first describes several sources of diversity in program source code, and then discusses the method employed in this case study to limit its effect.

5.1.1 Sources of Diversity

Most programming languages have the property that any single objective can be achieved through a variety of different programs. The case study conducted is essentially concerned with programs written in an imperative style, which is certainly among the programming paradigms that permit a large amount of variation in the solution space. The most important sources of this diversity are described here.

i. **Algorithm Selection**
   When multiple algorithms to solve a problem are available, they necessarily represent *essential diversity*.

ii. **Choice of Identifiers**
   The particular names chosen for variables, constants, functions and user-defined data types is open to the programmer. The probability of two independently written programs having the same selection of names is very low for all but the smallest programs.

iii. **Output Presentation**
   String constants used in the presentation of output will vary between programs. The amount of variation depends on the wording of the assignment specification. If students are given an example of the desired output format, differences may largely be in the form of capitalisation and spacing. A related (and perhaps more significant) problem is that the string constants may be broken into portions, with the rendering of these portions and computed values finely interspersed.

iv. **Arbitrary Ordering**
   Independent programming statements and definitions can usually be supplied in any order. Graph similarity measures ignore any ordering in their children, so as long as the independence of phrases can be easily established, their relative order need not be a problem. Statements are more problematic because inter-dependence between a sequence of statements must be assumed unless it can be proven that the statements are independent. Such proofs are in general difficult as they require complex global analysis to detect statements that have hidden side-effects. Another problem with ordering is that user-defined sub-routines introduce an arbitrary parameter ordering that must be respected throughout the program.

v. **Program Decomposition**
   All forms of program decomposition introduce problems for structural similarity analyses as they directly concern restructuring the program. The three important forms of decomposition are: *functional decomposition*, *statement decomposition* and *expression decomposition*. Students of a first programming course do not often apply *functional decomposition*. *Statement decomposition* does occur

occassionally. For example, a pair of `Write` calls are equivalent to a single `Write` call with two parameters.

In novice programs, the most important form of decomposition is probably *expression decomposition*, as sub-expressions may easily be extracted out into temporary variables. This practice is commonplace when expressions are otherwise large.

vi. Operator Identities

As mathematical expressions can take different (but equivalent) forms, the same is true when they are implemented in computer programs. Without considering associativity and commutativity laws, two mathematically equivalent expressions would be regarded as distinct in computer programs (typical in compilers). An exception here is floating-point arithmetic for which these laws do not always apply. However, novice programmers are unlikely to write programs in which this fact is significant, so this case should also be treated as *spurious diversity*.

vii. Language Specific Idioms and Synonyms

Most programming languages offer phrases which are idiomatic synonyms for other phrases. Examples from C++ include the pre- and post-increment operators, as well as the *for* loop which is semantically equivalent to an initialised *while* loop. Object Pascal offers the `Inc` procedure which is equivalent to an incrementing assignment. Such synonyms are strictly *spurious*.

viii. Loop Variation

C++ offers three loop constructs (*for*, *while* and *do-while*), however, the *for* loop is not semantically different from an initialised *while* loop. Object Pascal has four looping constructs: the *repeat-until* loop, the *while* loop, an *incrementing for* loop and a *decrementing for* loop. The *incrementing for* loop is semantically similar to an initialised *while* loop. Unlike C++ it is not strictly equivalent as the Object Pascal *for* loop is internally organised to prevent an endless loop in the event of the loop variable overflowing. This semantic detail is rather subtle, so it may be worthwhile making a concession in treating this as *spurious* rather than *essential diversity*.

ix. Decision Variation

Object Pascal's *case*-statement[1] is interesting in that it does not introduce much diversity. However, the *case*-statement is semantically equivalent to an appropriately constructed *if*-ladder, which is a significant source of *spurious diversity*. Consider, for example, an *if*-ladder which selects one of n conclusions. There are $n!$ ways to order the available conclusions. For each of these orderings, the conditions required to discriminate between the conclusions can be arranged to construct a new *if*-ladder with the same semantics as the original, in spite of structural changes in the selection predicates.

For example, the main loop of a binary search contains a decision construct that selects one of three conclusions, as in *Figure 5.1a*. This important *if*-ladder can be arranged in $3! = 6$ different ways. For comparison, if the order of the conclusions is reversed, the new *if*-ladder is arranged as shown in *Figure 5.1b*.

```
begin
  Mid := (First + Last) div 2;

  if (Data[Mid] < Goal) then
    First := Mid + 1
  else
  if (Data[Mid] > Goal) then
    Last := Mid - 1
  else
    Found := True;
end
```

(a)

```
begin
  Mid := (First + Last) div 2;

  if (Data[Mid] = Goal) then
    Found := True
  else
  if (Data[Mid] > Goal) then
    Last := Mid - 1
  else
    First := Mid + 1;
end
```

(b)

Figure 5.1: Equivalent decision constructs for a binary search

The following sub-section discusses how these sources of diversity may be managed.

5.1.2 Normalising Programs

This case study follows the strategy of bringing programs into a normal form before comparison, to manage program diversity. This eliminates much of the *spurious diversity* available in the solution space. The approach is a two-step process. First the program's parse tree is converted into a LISP-*like* symbolic expression[2] for easier manipulation. The symbolic expression is then brought into normal form through a tree

[1] In C++ and several other languages, this statement is called a *switch*-statement.

[2] LISP is a language similar to Scheme. In fact, Scheme is a stricter form of Lisp.

rewriting strategy, yielding another symbolic expression. The final symbolic expression is later converted into the program graphs used for similarity comparison.

The tree rewriting strategy is aided by a simple LISP-*like* pattern matching mini-language. Saikkonen *et al.* (2001) use a similar support language in Scheme-robo (see *Section 2.2.2*), although they only use their pattern matching to exclude programs that do not follow a specified program strategy.

Instead of rejecting programs that follow diverse strategies, a large collection of rewrite rules (specified in this mini-language) are used for transforming the programs into a normal form. If the rewrite rules are carefully chosen and sufficiently plentiful, the solution space can be drastically reduced. As an example, the programming task described in *Section 5.3* has hundreds of solutions with small variations between them. By transforming programs into normal form, only 18 distinct solutions were required to adequately cover 96 student programs[3].

To further illustrate the utility of this approach, consider that incrementing integer variables occurs frequently in programming. It has already been observed that this can be achieved through either assignment, or the `Inc` procedure – two local alternatives. Consider a programming task for which an ideal solution contains n such variable updates. It is immediately clear that without conversion to normal form, there are 2^n variations of the ideal solution – each associated with a different combination of assignments and `Inc` calls. The simple rewrite rule in *Figure 5.2* solves the problem. In this case, the first pattern matches calls to the `Inc` procedure, while the second pattern indicates that the matched expression should be rewritten as an assignment.

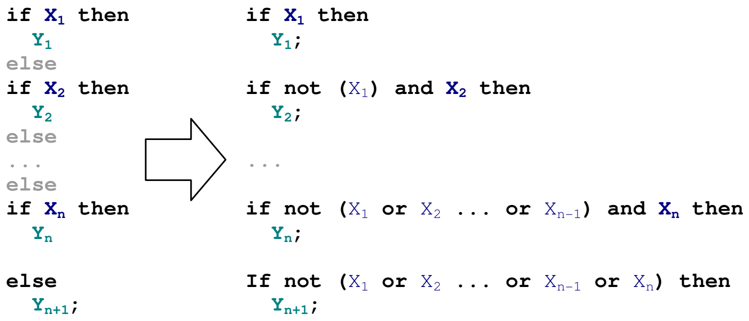
```
(:
    (call Inc (args x?))       ← The Pattern
::
    (assign x? (+ x? 1))       ← The Rewrite
:)
```

Figure 5.2: An example rewrite rule for obtaining canonical programs

Most of the rewrite rules operate in concert to achieve a collective effect. By using the tree pattern rewriting approach, new groups of rules can easily be introduced. The method was found to be efficient enough for most of the program normalisation processes.

[3] While manageable, this is still a large number of solutions for a relatively simple case, and more research is required to further address the problem.

While developing the program normalisation, one important transformation that was found to perform poorly was the special normalisation of decision constructs – so this transformation was carefully written as a recursive tree manipulating function, rather than as a set of rewrite rules. The first phase of the general form of this transformation is shown in *Figure 5.3*. Each conclusion is made disjoint of all related decisions, creating conditions that are more complex. The second phase applies algebraic simplification to each *if*-statement, and almost always reduces them to a simple readable form.



```
if X1 then
  Y1
else
if X2 then
  Y2
else
...
else
if Xn then
  Yn

else
  Yn+1;
```

```
if X1 then
  Y1;

if not (X1) and X2 then
  Y2;

...

if not (X1 or X2 ... or Xn-1) and Xn then
  Yn;

If not (X1 or X2 ... or Xn-1 or Xn) then
  Yn+1;
```

Figure 5.3: First phase of transforming *if*-statement decision ladders

The transformation of decisions may be best described through a more detailed concrete example. The main decision of a binary search is used again in *Figure 5.4*. This figure shows how the initial *if*-ladder is easily broken into three disjoint *if*-statements. The resulting conditions are quite complex compared with the original. Fortunately, each of these simplifies to the simple recognisable cases that are expected in any solution.

Figure 5.4 shows the detail at three points during this simplification process. That is to say, after logical negation is applied, relations are combined into simpler forms, and relations are transformed into their mirrored normal form. This latter transformation is simply the conversion of $a > b$ to $b < a$, further reducing the number of distinct operators that occur in normal form. It proved less useful to transform $a \leq b$ into $(a < b) \vee (a = b)$ as this replaces one operator with three, and the amount of work done by the simplifier is related to the number of operators it must consider.

What is important about the transformation is that all six variations of the decision construct have the same normal form, subject to ordering the three disjoint *if*-statements that result. Such ordering is not significant as the three *if*-statements are known to be independent (assuming no side-effects occur within the conditions[4]). Consequently, the *if*-statements are arranged in the program graph to take advantage of its disregarding of

[4] If the conditions contain side-effects, this transformation cannot be safely applied.

order between children. In the general case where an *if*-ladder has $n!$ variations, a very good logic simplifier may still manage to reduce all variations to a single normal form.

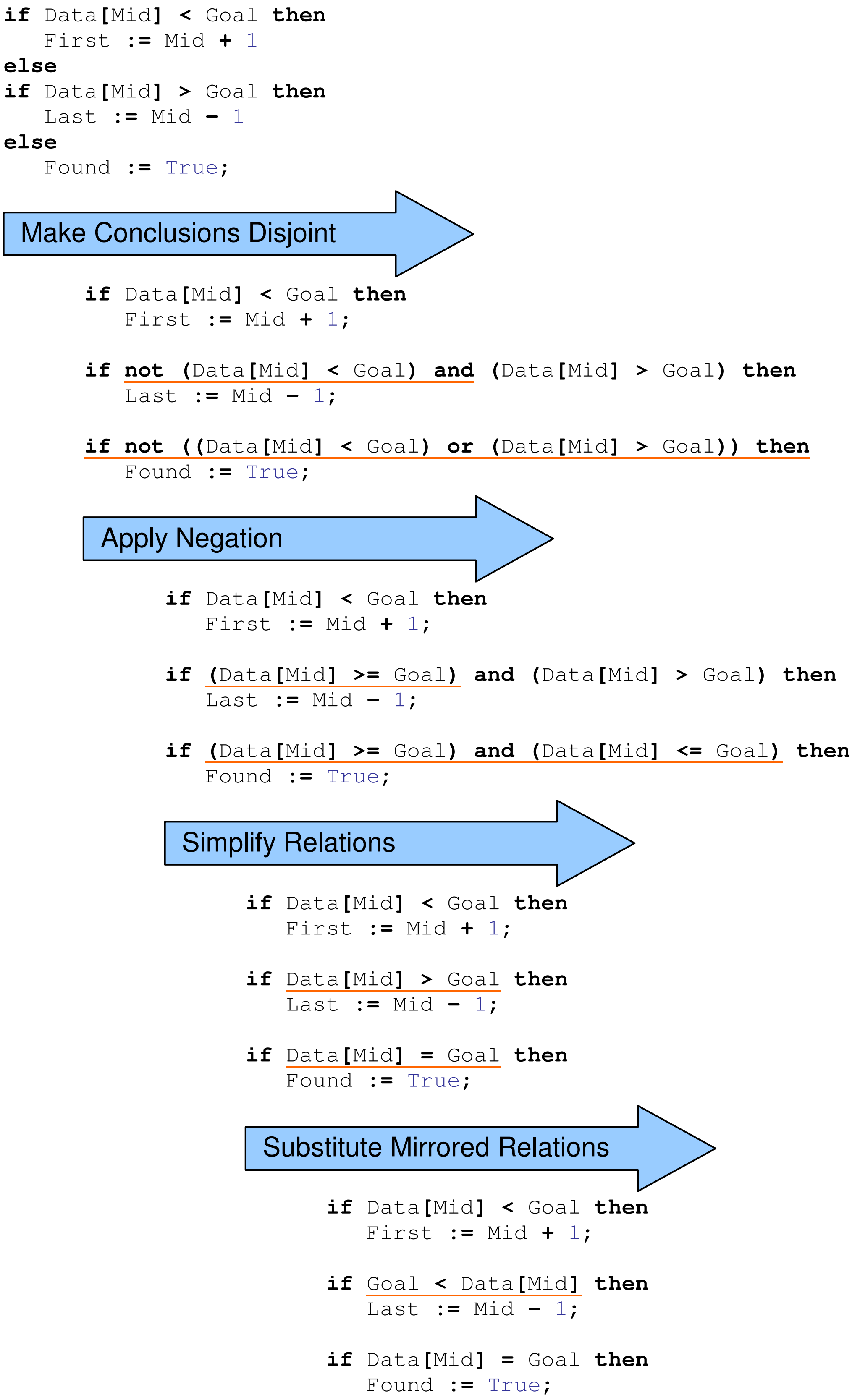
A number of transformations are applied in addition to those already described. Most of these concern language idioms, or deal with less important details such as treating all varieties of floating point data types, as `Real`. Similar treatment is provided for integer data types. The transformations applied in this study are briefly summarised in the list that follows.

i. **Choice of Identifiers**
   Identifiers may be arbitrarily chosen. The means for supporting this is the two step identifier mapping process described in *Section 4.1.5*.

ii. **Output Presentation**
   Several small transformations simplify the use of `Write` and `WriteLn` procedures. If these contain multiple arguments, they are first decomposed. The next step is the removal of any empty `Write` or `WriteLn` calls that may be found, leaving no *line-feeds* as these are not considered important in the study. The writing of string constants may also be eliminated at this point. Following this, the remaining adjacent calls to `Write` are recombined. The cumulative effect is that only pertinent output rendering is retained in each program.

iii. **Operator Identities**
   Most of the common mathematical identities are addressed in the rewrite rules. Particular care is taken with addition and multiplication so that a variety of similar arithmetic expressions have the same normal form. For example, *Figure 5.5* shows two different ways to express the calculation of a sphere's volume, $\left(\frac{4\pi r^3}{3}\right)$. While the assignments are organised in different ways, after transformation they have the same normal form.

iv. **Language Specific Idioms and Synonyms**
   Calls to the `Inc` and `Dec` procedures are replaced with equivalent assignments. In addition, data types such as `Single`, `Real48` and `Extended` are simply treated as `Real`.



```
if Data[Mid] < Goal then
   First := Mid + 1
else
if Data[Mid] > Goal then
   Last := Mid - 1
else
   Found := True;
```

```
      if Data[Mid] < Goal then
         First := Mid + 1;

      if not (Data[Mid] < Goal) and (Data[Mid] > Goal) then
         Last := Mid - 1;

      if not ((Data[Mid] < Goal) or (Data[Mid] > Goal)) then
         Found := True;
```

```
            if Data[Mid] < Goal then
               First := Mid + 1;

            if (Data[Mid] >= Goal) and (Data[Mid] > Goal) then
               Last := Mid - 1;

            if (Data[Mid] >= Goal) and (Data[Mid] <= Goal) then
               Found := True;
```

```
                  if Data[Mid] < Goal then
                     First := Mid + 1;

                  if Data[Mid] > Goal then
                     Last := Mid - 1;

                  if Data[Mid] = Goal then
                     Found := True;
```

```
                        if Data[Mid] < Goal then
                           First := Mid + 1;

                        if Goal < Data[Mid] then
                           Last := Mid - 1;

                        if Data[Mid] = Goal then
                           Found := True;
```

Figure 5.4: Detailed transformation of binary search decision

```
Volume := 4.0 * PI * r * r * r / 3.0;
Volume := (4.0 / 3.0) * PI * (r * r * r);
```

Figure 5.5: Two ways to calculate a sphere's volume $\left(\frac{4\pi r^3}{3}\right)$

v. Loop Variation

Although in Object Pascal *for* loops encompass subtly different semantics to *while* loops, the difference is very seldom evident. So in the interests of reducing diversity, both *incrementing for* loops and *decrementing for* loops are transformed into initialised *while* loops. *Repeat-until* loops are left intact.

vi. Decision Variation

The transformations of *if*-ladders has been carefully described above. *Case*-statements are simply transformed into an equivalent *if*-ladder and leverage the transformations already designed for them.

5.1.3 Program Transformation Example

This section briefly discusses the application of the described processes to a whole program. The program discussed must obtain a number from the console and display its square-root. If the number is negative, however, it displays the message “undefined”. Since this string literal may be directed to the output, `Write` calls with string constants as arguments are not eliminated for this program[5]. A program that solves this problem is shown in *Figure 5.6*.

```
program PrintSqrt;
uses
  SysUtils, Math;

var
  x: Integer;

begin
  ReadLn(x);
  if (x < 0) then
    WriteLn('undefined')
  else
    WriteLn( Sqrt(x) );
end.
```

Figure 5.6: Print the square-root of a number

Processing this program requires the construction of the symbolic expression that captures its meaning. This symbolic expression is then converted to normal form.

[5] This particular transformation is simply not applied for such assignments.

Figure 5.7a shows the initial symbolic expression, while *Figure 5.7b* shows the modifications made through normalisation. Because the program is small, only a small number of the rewrite rules are applicable. The most obvious change is the transformation of the decision, as described before. However, careful inspection also reveals that the *line-feed* of the `Write` call has been discarded.

```
(program
    (use sysutils math)
    (decls
        (var x integer)
    )
    (body
        (call
            readln
            (args x)
        )
        (if
            (< x 0)
            (call
                writeln
                (args "undefined")
            )
            (call
                writeln
                (args
                    (call
                        sqrt
                        (args x)
                    )
                )
            )
        )
    )
)
```

(a) Before transforms

```
(if-group
    (if
        (< x 0)
        (call
            write
            (args "undefined")
        )
    )
    (if
        (<= 0 x)
        (call
            write
            (args
                (call
                    sqrt
                    (args x)
                )
            )
        )
    )
)
```

(b) After transforms

Figure 5.7: Symbolic expressions for *Figure 5.6*

The final step before the graph similarity measure may be applied is the conversion of the normalised symbolic expression to a graph representation. This graph is shown rendered in *Figure 5.8*. In the graph, statement ordering is implicit through dependency edges (rendered grey). The two disjoint *if*-statements introduced during normalisation do not have a dependency edge between them. This is important in that it means there is intentionally no inherent ordering of these two decisions. The two *if*-statements have a mutual dependency on the `ReadLn` call by means of an intermediate *if-group* vertex. The green dependency edges are not essential, but serve to retain more detail in the characteristic graph used in the identifier mapping.

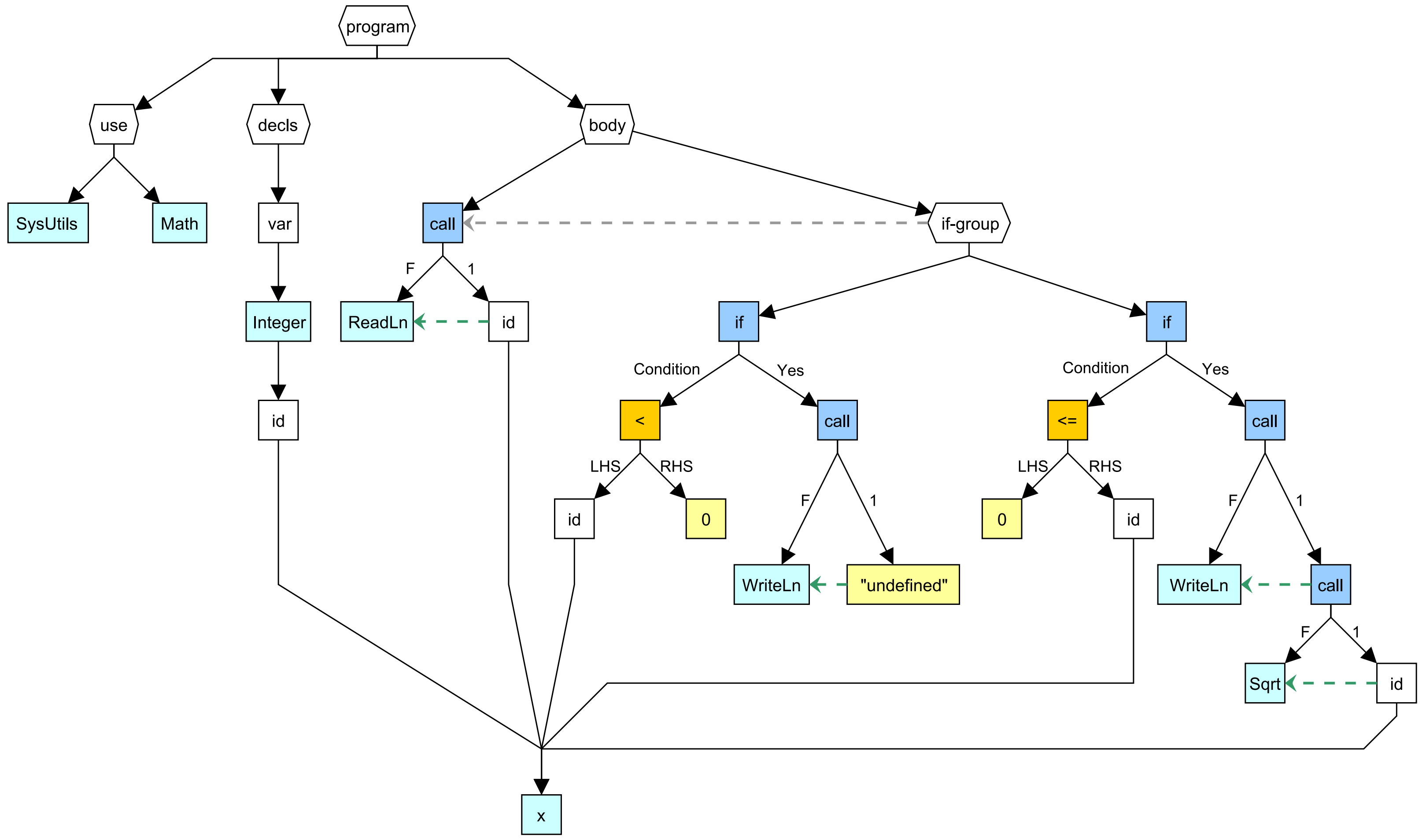


Figure 5.8: Graph for *Figure 5.6*

5.2 Scoring of Programs

The *Weighted Assignment Similarity* measure derives similarity scores that are primarily obtained from structural information. The measure has no innate way to understand the content of two structures. Most significantly, two graphs may have the same structural form, but different meaning. Such a case is precisely the cause of the “scattering effect” observed by Allali & Sagot (2005) with the tree-edit distance. The *Weighted Assignment Similarity* measure may, however, be influenced by non-structural detail by means of local graph attribute similarity, allowing the larger similarity measure to yield more meaningful scores. This section discusses how attributes on program graphs can influence the similarity, and also explores the conversion of similarity scores into marks.

5.2.1 Using Graph Attributes

Attributes in the program graph can occur on either vertices or edges. Vertices may contain labels denoting structural groupings, such as a section of declarations. Other possibilities include labels denoting operators, identifiers, and literal values. Edge labels may serve to create a distinction between sibling vertices. For example, the *condition*, *then* and *else* parts of an *if*-statement may use edge labels for exactly this purpose, as there is no other ordering relationship between siblings. In another example, arguments to a function call could be distinguished by numerically labelled edges.

The *Weighted Assignment Similarity* measure allows for the comparison of attributes to influence the overall similarity. It has already been shown that the same measure may be used in a two step process to determine a similarity mapping between variables. What has not been addressed is the handling of undefined identifiers. In this regard it seems most prudent, when trying to map variables, to treat undefined variables as though they are defined at the current scope level – global for the main program and local for functions. It is believed that this approach is likely to match the intent of the programmer frequently, since the use of global variables are discouraged when local variables may be used. After the similarity measure is applied to the characteristic graphs (showing identifier use), the direct variable similarity scores are available and may be immediately re-employed as a local attribute similarity measure.

Literal values may also occur in attributes. The view taken in this study is that literals of different data types are not comparable and receive a 0 local similarity score.

Furthermore, literals of the same type should receive a non-zero local similarity even if different, as being of the same type is enough for two values to be considered somewhat similar. As an example, consider the local similarity between two integer literals, x and y. It turns out that defining the similarity between two arbitrary integers, is surprisingly difficult – each measure chosen is tuned to a particular distribution of values. It seems best that such a local similarity measure have some free parameters so that it may easily be modified for the task at hand. The model of integer similarity used in this study is the half-life exponential decay function given by *Equation 5.1*. In the equation, λ is a free parameter called the half-life and indicates the distance between x and y over which their similarity is halved.

$$Sim_{\text{integers}}(x, y) = 2^{-\frac{|x-y|}{\lambda}} \tag{5.1}$$

This model has the desirable property that the similarity score is 1, if x and y are equal. Otherwise the score decays according to the distance between the numbers, but never reaches 0. *Figure 5.9* shows the shape of this similarity measure for various half-life values.

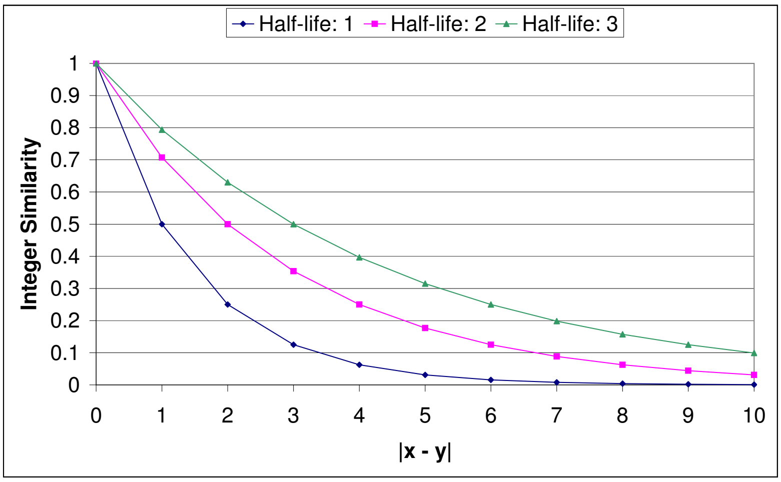


Figure 5.9: Integer similarity: $2^{-\frac{|x-y|}{\lambda}}$ for different values of λ

The difficulty that remains is in choosing a suitable half-life. It is observed that student programs frequently deal with small constants, such as the numbers of beverages purchased. For this reason, the integer similarity measure should be reasonably effective at discriminating small integer differences. The implication is that the half-life should be small. A half-life of 2 is suggested, but in this study it was found that any small integer worked just as well. In practice, many assignments may not be sensitive to the choice of λ, since they actually only require a measure of relative similarity between pairs of integers.

The problem of determining local similarity measures for other data types is related. As with integers, the relative similarity of pairs is more important than the local measure itself. This is largely because given alternative neighbour assignments, the relative difference in the local similarity measures may become the deciding factor. This is especially true in the first few iterations through the iterative formula[6]. Since structural information has not yet propagated deeply throughout the graph, local similarity information is more useful (and reliable) at that point.

5.2.2 Converting Scores to Marks

In general, the set of known programs against which the student programs are to be compared form the standard of assessment. For this reason, it is referred to as the *standard set*. Each program in the standard set must have an associated human assigned mark. Comparing a student program with the programs of the *standard set* yields a set of matches. Each of these are tuples containing first the similarity score and second the human mark of the corresponding matched solution.

These tuples are arranged in decreasing (or at least non-increasing) order to form the sequence $< S_1, H_1 >, < S_2, H_2 >, < S_3, H_3 >, \ldots$, called the solution sequence. If the *standard set* includes only perfectly ideal solution, each of the human marks will be the upper limit (H_{max}) for the assignment. In general, however, the *standard set* may contain solutions of mixed quality levels.

The conversion of the similarity scores into a mark must take into account a subsequence of the similarity scores and the corresponding human assigned marks. If only the highest matching solution is used, then these sequences are trivially restricted to S_1 and H_1. As S_1 represents the fraction of the student program that matches the solution, the mark awarded is given by $M_1 = S_1 H_1$. The mark depends on the simplifying assumption that the human assigned marks in the solution are evenly distributed throughout the marked program. The residual is that portion of the student program not covered by the matched program, and is given by $R_1 = 1 - S_1$.

[6] This assumes that the efficient implementation strategy described for DAGs in *Section 4.1.3* is not being used.

It is quite possible that unforeseen solutions could be constructed by mixing parts of the code in two or more distinct solutions. This is analogous to the crossover operation in genetic algorithms. What this means practically is that the residual may contain part of a solution strategy used in a different solution.

Suppose that the second closest matching solution is used to make the mark more accurate. Its contribution over the residual portion of the program is given by $(1 - S_1)S_2H_2$, yielding the mark $M_2 = S_1H_1 + (1 - S_1)S_2H_2$. The simplifying assumption here is that the second most similar solution best covers the residual portion of the program. This may not be the case, but no better selection is easy to obtain. The new residual portion of the program is given by $R_2 = 1 - S_1 - (1 - S_1)S_2 = R_1 - R_1S_2$.

This approach may be extended to the general case of deriving a mark from the n closest matching solutions. This will be called an *n-Solution Conversion*, and requires an expression for the residual resulting from the $n - 1$ preceding solutions. This is given as follows:

$$\begin{aligned} R_0 &= 1 \\ R_{j+1} &= R_j - R_jS_{j+1} \\ &= R_j(1 - S_{j+1}) = \prod_{i=1}^{j+1}(1 - S_i) \end{aligned} \tag{5.2}$$

The *n-Solution Conversion*, gives the n^{th} order mark by:

$$M_n = \sum_{j=1}^{n} R_{j-1}S_jH_j \tag{5.3}$$

This is a very general model for obtaining marks from similarity scores. Two important observations can be made. The first is simple: the limit of the residual tends to zero, as expected. The second is more difficult to see, but since each successive term only contributes a mark from the residual portion of the program (for which no mark has yet been attributed), the assigned mark cannot exceed H_{max}.

As presented here, an *n-Solution Conversion* uses only the *n closest* matching solutions. These n solutions may, of course, be selected in other ways. For example, if the top two closest matches are themselves very similar, then little is gained in using the second match. Better results may possibly be achieved by ignoring programs in the solution sequence if they are highly similar to their predecessors. However, performing this check is both a practical complication, and requires applying the similarity measure more often.

A simpler solution may be to consider (for example) only every odd numbered solution in the sequence of matching solutions. The idea is made explicit as follows: C_s shall denote an $|s|$-*Solution Conversion*, where s is a set of indices into the set of matching solutions. For example, $C_{\{1,3,5\}}$ will denote the *3-Solution Conversion* based on the 1st, 3rd and 5th closest matching solutions.

5.2.3 Effect of Poor Programming Practices

While the experiments in *Section 5.3* directly test the feasibility of using a similarity measure in assessing student programs, it is also valuable to consider briefly the effect of poor programming practices. For example, it is not expected that any ideal program solution would use global variables where local variables are expected, but a student program might. If the *standard set* consists exclusively of good quality programs, poorer quality programs may not be effectively assessed.

To better understand how bad programming decisions influence the effectiveness of using similarity in assessment, several variations of the programming example in *Figure 5.10* are considered. This program repeatedly obtains integers from the console, and accumulates their absolute values in a sum until either zero is read, or ten numbers have been obtained from the console. Variations of this program are shown in the highlighted portions of *Figures 5.11a-c*. Each of these programs exhibit a different example of poor practice, and their respective similarity to the ideal solution is summarised in *Table 5.1*.

The first and second program variations of (*Figures 5.11a* and *b*) concern the exclusive use of global variables, and some duplication of code. *Table 5.1* shows that they suffer modest penalties to the overall similarity. The case with code duplication receives the heavier of the two penalties because code is not only duplicated but structured to allow the duplication.

The third case (*Figure 5.11c*) is interesting in that it includes a formal test of a precondition (which should always be true). The main problem is not the testing of the condition (unit tests are actually good practice), but that the test causes a structural change in the main body of code (which unit tests should never do). This third change

```
  procedure SumAbsolutes(Limit: Integer);
  // Precondition: Limit > 0
  var
    Num: Integer;
    Sum: Integer;

  begin
    Sum := 0;

    repeat
      Write('Enter num: ');
      ReadLn(Num);

      if (Num < 0) then Num := - Num;
      Sum := Sum + Num;

      Dec(Limit);
    until (Num = 0) or (Limit = 0);

    WriteLn('Sum is ', Sum);
  end;

begin
  SumAbsolute(10);
end.
```

Figure 5.10: Sum absolutes of numbers entered until limit reached or zero entered

	All	Code	Poorly Structured
	Variables Global	Duplication	Pre-Condition Testing
Similarity to Solution	0.879	0.806	0.580

Table 5.1: Similarity of poor programs to ideal solution

is the most benign, but has the largest overall effect (0.580) of the three modifications considered. Since the structural change occurs nearer to the root of the program graph, it suggests that the similarity measure is more sensitive to structural changes near the root than those near leaves. This is interpreted as a side-effect of using vertex significance based of the size of the subgraph reachable from the given vertex. Further research may be needed to develop vertex significance functions which do not exaggerate this problem.

5.3 Experiments

This section discusses the experiments to gauge the feasibility of using the *Weighted Assignment Similarity* measure in assessing student programs. It is divided into three sub-sections. The first describes how the data was obtained and the precautions taken against possible sources of bias. The second sub-section discusses the experimental

```
var
  Num: Integer;
  Sum: Integer;

  procedure SumAbsolutes(Limit: Integer);
  // Precondition: Limit > 0

  begin
    Sum := 0;

    repeat
      Write('Enter num: ');
      ReadLn(Num);

      if (Num < 0) then Num := - Num;
      Sum := Sum + Num;

      Dec(Limit);
    until (Num = 0) or (Limit = 0);

    WriteLn('Sum is ', Sum);
  end;

begin
  SumAbsolute(10);
end.
```

(a) All variables global

```
procedure SumAbsolutes(Limit: Integer);
// Precondition: Limit > 0
var
  Num: Integer;
  Sum: Integer;

begin
  Sum := 0;

  repeat
    Write('Enter num: ');
    ReadLn(Num);

    if (Num < 0) then
      begin
        Num := - Num;
        Sum := Sum + Num;
      end
    else
      begin
        Sum := Sum + Num;
      end;

    Dec(Limit);
  until (Num = 0) or (Limit = 0);

  WriteLn('Sum is ', Sum);
end;

begin
  SumAbsolute(10);
end.
```

(b) Code duplication

```
procedure SumAbsolutes(Limit: Integer);
// Precondition: Limit > 0
var
  Num: Integer;
  Sum: Integer;

begin
  Sum := 0;

  if (Limit > 0) then
    begin
      repeat
        Write('Enter num: ');
        ReadLn(Num);

        if (Num < 0) then Num := - Num;
        Sum := Sum + Num;

        Dec(Limit);
      until (Num = 0) or (Limit = 0);
    end;

  WriteLn('Sum is ', Sum);
end;

begin
  SumAbsolute(10);
end.
```

(c) Poorly structured pre-condition testing

Figure 5.11: Poorer variations of *Figure 5.10* (changes highlighted)

method, while the results are discussed in the third sub-section.

5.3.1 Data Preparation

A large pool of Object Pascal program submissions were collected in the CS&IS department of NMMU. Each submission was marked by a human marker on a five point scale. It was determined that this scale of human marking was too coarse for a meaningful comparison of assessment techniques. For this reason a particular programming assignment was singled out as being the task with largest diversity in human scores, and requiring a variety of programming constructs in its solution.

```
var
  NumJudges: Integer;
  Count: Integer;
  Sum: Integer;
  Score: Integer;
  Low, High: Integer;
  FinalScore: Real;

begin
  Write('Number of Judges: ');
  ReadLn(NumJudges);

  Write('Enter Score: ');
  ReadLn(Sum);

  Low := Sum;
  High := Sum;

  for Count := 2 to NumJudges do
    begin
      Write('Enter Score: ');
      ReadLn(Score);

      Sum := Sum + Score;

      if (Score > High) then High := Score;
      if (Score < Low) then Low := Score;
    end;

  FinalScore := (Sum - High - Low) / (NumJudges - 2);
  WriteLn('Final score: ', FinalScore:0:2);
end.
```

Figure 5.12: Program to calculate an Olympic ice-skater's score

The programming task was to write a program that requested and obtained the number of judges of an Olympic ice-skating event. The average score of the judges had to be reported, after the highest and lowest scores were discarded. An ideal solution to this problem is shown in *Figure 5.12*.

This selected assignment was remarked on a finer assessment scale. To minimise the possibility of any bias resulting from the human marker, the marking proceeded

Criteria	✓
Variables Defined: `NumJudges`, `Count`	
Variables Defined: `Sum`, `Score`	
Variables Defined: `Low`, `High`	
Console IO: Read `NumJudges`	
Appropriate Initialisation: `Sum`	
Appropriate Initialisation: `Low`, `High`	
Quality: Initialisation Is Clean (no magic numbers)	
Loop: Initialised	
Loop: Valid Condition	
Loop: Update Loop Variable	
Console IO: Read `Score`	
Update: `Sum`	
Update: `Low`	
Update: `High`	
Quality: Update Is Clean (not intertwined in complex logic)	
Arithmetic: `Sum` - `Low` - `High`	
Arithmetic: `NumJudges` - 2	
Console IO: Write Final Score	

Table 5.2: The 18-point rubric used for marking the assignment

according to an eighteen point rubric (*Table 5.2*) covering the artefacts that should be present in a solution. This reduced the number of judgement decisions that had to be made on a per submission basis.

The assignments were not completed in a controlled environment so the possibility of plagiarism was very real. Unfortunately, there is no foolproof way to identify which programs were plagiarised and which were not. To reduce the problem with plagiarism, the programs were preprocessed to remove comments and to collapse whitespace. Duplicates were then eliminated. This process discarded about 15% of the student submissions which were clearly adapted from other students' work, leaving 96 unique submissions.

5.3.2 Method

Using the *Weighted Assignment Similarity* measure in assessment requires the construction of a *standard set* of solutions. This set may be obtained either through the synthesis of the artefacts known to be required in a solution, or by gathering a set of historic data for the same problem. Both approaches are investigated in the study.

i. Synthetic Standard Set

A synthetic *standard set* is a set of solutions that have been carefully constructed by hand in advance of seeing any student submissions. Two synthetic standard sets were developed for these experiments. The first contains 40 unique program solutions, all of which are considered good solutions. This reflects the strategy of assessing against ideal or near-ideal solutions only. The second synthetic *standard set* contains all 40 of the first set, but is salted with a further 2 examples of particularly bad programs. These poor quality programs lack the main loop that any true solution is expected to have, and have received correspondingly low human marks. The first set was called the *synthetic good solutions* set, while the second was called the *synthetic mixed solutions* set.

The construction of the first synthetic *standard set* was very labour intensive. Beginning with the ideal solution presented in *Figure 5.12*, alternatives were carefully synthesised and collected into the *standard set*. The first main variation of the ideal solution involves initialising the highest and lowest judge scores to fixed constants. Additional variations arise from the loop bounds, as these can be formulated in several ways. The solution in *Figure 5.12* counts from `2` to `NumJudges`, but it is just as valid to count from `0` to `NumJudges - 2`, for example. More variation can be attributed to the updating of the highest and lowest scores. Here the decisions can be formulated in several ways – some less than obvious – or replaced entirely with calls to the `Min` and `Max` functions.

The problem is that almost none of these local variations are mutually exclusive, giving rise to many combinations. It was not clear how many variations would be necessary to obtain reasonable results. After 40 different variations had been found, the author was reluctant to consider more, as allowing another local variation was likely to yield a standard set almost as large as the number of submissions. Clearly many variations are possible, creating a high degree of similarity between solutions in the standard set. Many of the variations developed were not actually helpful as only 18 found strong representation amongst student programs, but this could not be determined in advance.

ii. Historic Standard Set

If a sample of student submissions have been marked by a human being, they may be regarded as a *historic standard set*. For example, if the same assignments are used in successive or alternating years of offering, a rich source of example programs is readily available. This is historic data in the truest sense. However,

> if the class is large enough the same effect can be achieved by taking a sample of programs from the current year of offering and marking these by hand.
>
> To evaluate the effectiveness of this idea, the 96 student programs were sorted on human mark, and then split into two samples (A and B) with the same distribution of scores. Either set could then be treated as the *standard set*, while the other became the set of student submissions requiring automated marking. In other words, half of the available data could be treated as being accurately marked historic data.

The synthetic standard sets were each used to assess all 96 student submissions. Historic set A was used to assess the 48 submissions in historic set B, and vice versa. In each case, several similarity score conversion functions, as described in *Section 5.2.2*, were used and the marks recorded.

5.3.3 Results

After performing the experiments, the correlation[7] of the similarity derived marks to the human assigned marks was calculated. These are presented in *Table 5.3*. All correlations were significant at the $p < 0.01$ level.

The correlations in *Table 5.3* are all strong. To highlight which conversion functions performed best, each entry is in **bold** typeface if it is larger than the average for its row. The conversion functions $C_{\{1,2\}}$, $C_{\{1,3\}}$ and $C_{\{1,3,5\}}$ consistently yielded above average correlations. These all proved to be good candidates for converting similarity scores into marks. Of these three, $C_{\{1,3,5\}}$ achieved the highest average correlation.

The scatter plots that correspond to these correlations, using $C_{\{1,3,5\}}$, are presented in *Figures 5.13* and *5.14*. *Table 5.4* summarises the results of the regression analysis. From both the figure and the table it is clear that each regression line has a gradient of almost 1, except the *synthetic good solutions* case. A gradient of 1 is desired because it indicates that similarity derived marks may be substituted for human marks (they do not merely predict human marks). The *standard set* containing only good solutions does not have this property, as its similarity scores appear positively biased. This indicates that if the standard set includes only good solutions, poor programs with low human marks receive very inaccurate similarity derived marks.

[7] In all cases, the non-parametric Spearman's correlation was used, as the data was not normally distributed.

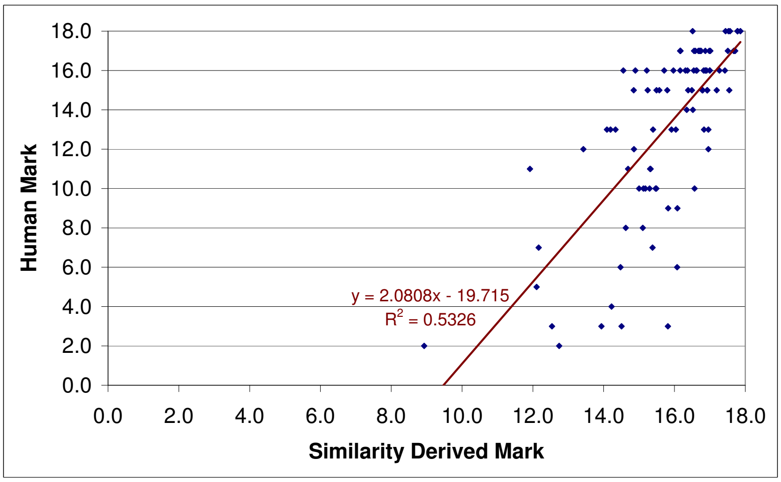
	$C_{\{1\}}$	$C_{\{1,2\}}$	$C_{\{1,2,3\}}$	$C_{\{1,2,3,4\}}$	$C_{\{1,2,3,4,5\}}$	$C_{\{1,3\}}$	$C_{\{1,3,5\}}$	$C_{\{1,5\}}$	Average
Synthetic Good Solutions	0.68 (96)	**0.70 (96)**	**0.71 (96)**	0.66 (96)	0.59 (96)	**0.71 (96)**	**0.72 (96)**	**0.71 (96)**	0.687
Synthetic Mixed Solutions	0.69 (96)	**0.72 (96)**	**0.72 (96)**	0.69 (96)	0.66 (96)	**0.72 (96)**	**0.74 (96)**	**0.73 (96)**	0.710
Historic A vs. B	0.69 (48)	**0.73 (48)**	0.71 (48)	**0.73 (48)**	0.70 (48)	**0.74 (48)**	**0.72 (48)**	0.71 (48)	0.717
Historic B vs. A	0.73 (48)	**0.81 (48)**	**0.80 (48)**	0.78 (48)	0.76 (48)	**0.82 (48)**	**0.82 (48)**	**0.81 (48)**	0.791

All correlations significant at the $p < 0.01$ level

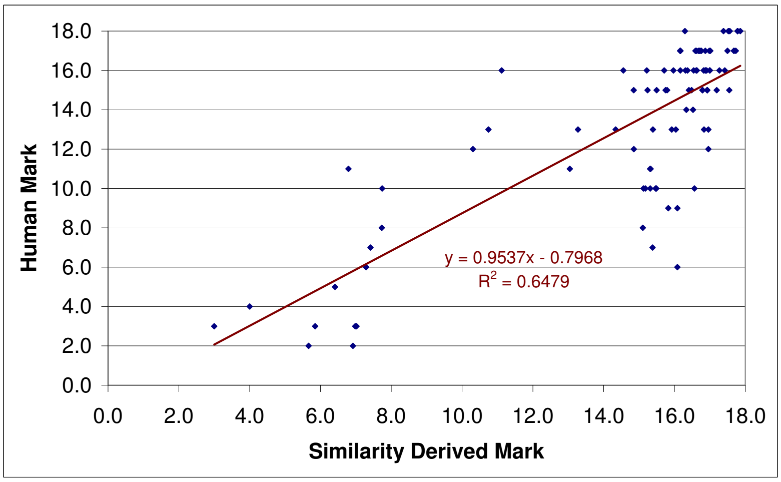
Bold indicates correlation greater than row average

C_s are conversion functions as described in *Section 5.2.2*

Table 5.3: Correlations of similarity derived marks to human assigned marks



(a) Synthetic Good Solutions

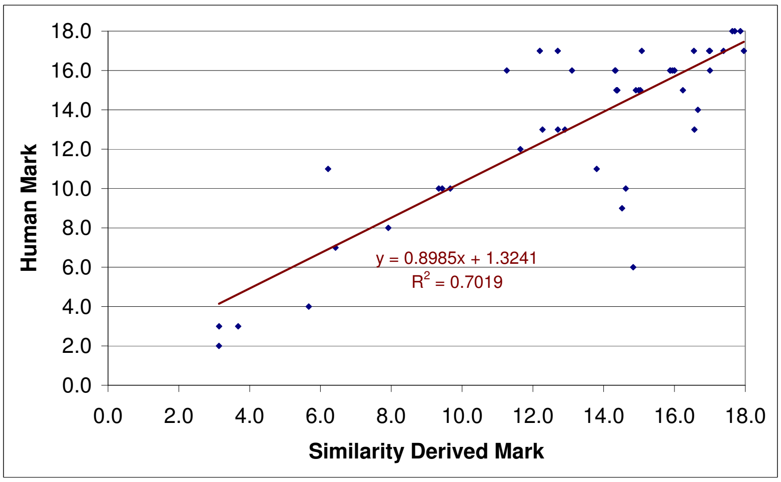


(b) Synthetic Mixed Solutions

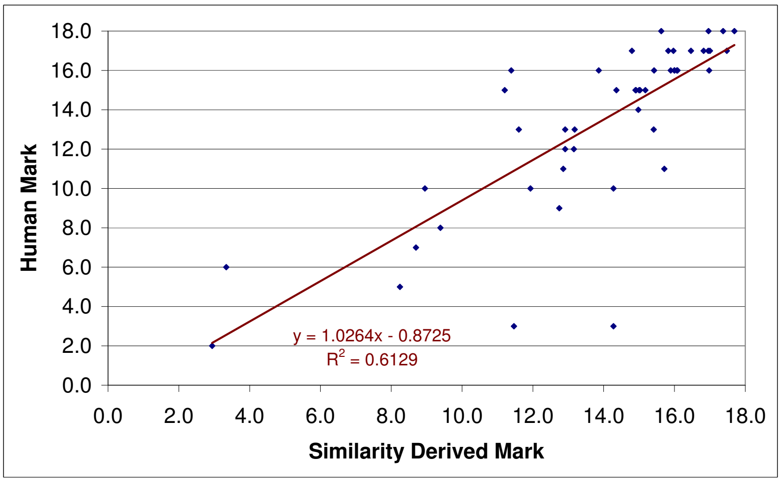
Figure 5.13: Scatter plots for synthetic standard, using $C_{\{1,3,5\}}$

Evidence for this is found in *Figure 5.15*, which shows a large average deviation from human assigned mark. Furthermore, the *synthetic mixed solutions* set, which includes only two examples of poor programs, incurs only small deviations for poor programs. This indicates that its accuracy is improved for knowing examples of poor programs. It also suggests that best accuracy is achieved if the *standard set* includes examples of programs from all quality levels. The historic standard sets have this property, and as shown in *Figure 5.15*, exhibit low average deviations across all quality levels.

Chapter 4 argued that existing graph similarity measures are inappropriate for use in student program assessment. To show that this is the case, the *SimRank* measure was applied to the same data. It is not clear how *SimRank* scores should be converted into marks, but this does not matter since correlation is not sensitive to scaling and linear



(a) Historic A vs. B



(b) Historic B vs. A

Figure 5.14: Scatter plots for historic standard, using $C_{\{1,3,5\}}$

transformations of the data. For this reason, a correlation considering the raw *SimRank* scores is just as meaningful as a correlation using scaled *SimRank* scores.

Table 5.5 summarised the correlations between *SimRank* scores and human marks. However, it proves more useful to discuss this measure in terms of the corresponding scatter plot. *Figure 5.16* shows the scatter plot after applying the technique with the *synthetic mixed solutions* data – the counterpart of which is *Figures 5.13b*.

From the *SimRank* scatter plot it is clear that there is no specific relationship between the human scores and the scores assigned by the *SimRank* measure - *SimRank* assigns scores of about 0.3 regardless of the human assigned score. This does not imply that *SimRank* is a poor graph similarity measure, but rather that it disregards precisely the detail that is important in assessment. In contrast, the *Weighted Assignment Similarity*

	Regression Formula	R^2
Synthetic Good Solutions	$y = 2.0808x - 19.715$	0.5326
Synthetic Mixed Solutions	$y = 0.9537x - 0.7968$	0.6479
Historic A vs. B	$y = 0.8985x + 1.3241$	0.7019
Historic B vs. A	$y = 1.0264x - 0.8725$	0.6125

All significant at the $p < 0.01$ level

Table 5.4: Result of regression analysis

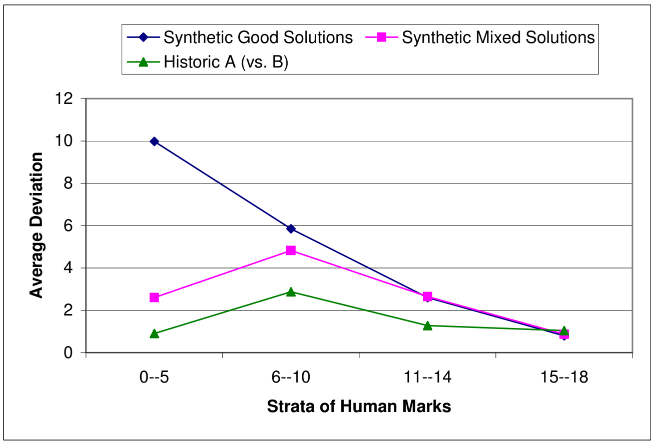


Figure 5.15: Deviations from human marks

scores are meaningfully related to the human scores (*Figure 5.13b*).

5.4 Conclusion

In this case study, the conversion of programs into normal form was vital to the feasibility of using the graph similarity measure for assessment. In this regard, a similar mechanism to that used by Saikkonen *et al.* (2001) proved effective in finding patterns in programs that may be transformed into normal form.

Employing the *Weighted Assignment Similarity* measure to assessment is novel, but

	SimRank	p
Synthetic Good Solutions	0.25**(96)	0.013
Synthetic Mixed Solutions	0.25**(96)	0.013
Historic A vs. B	0.09 (48)	0.54
Historic B vs. A	0.06 (48)	0.70

** significant at the $p < 0.05$ level

Table 5.5: Correlations of *SimRank* scores to human assigned marks

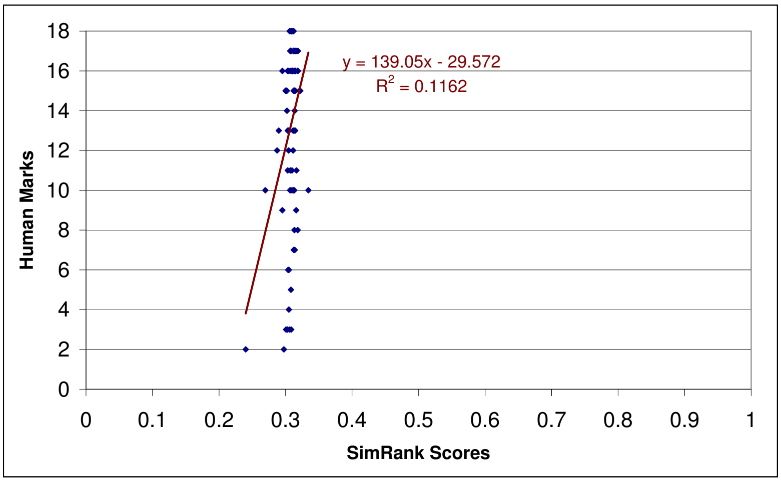


Figure 5.16: Scatter plot after applying *SimRank* to the *synthetic mixed standard* set

some relationships to prior work exist. The PASS system, due to Thorburn & Rowe (1997), relies on a mix of static and dynamic analysis to map functions in the student program to functions in the ideal solution. In a similar way, the *Weighted Assignment Similarity* measure uses purely static structural information to map functions between the programs. However, this similarity measure does not only map similar functions, but similar constructs at a much finer granularity.

The case study investigated in this chapter was small, but still of a sufficient size to show that the measure can feasibly be used for assessment. The most significant remaining problem concerns the development of a standard set against which to grade student submissions. The selection of the standard set most directly affects the accuracy of the technique. However, further work relating to the vertex significance function may also make the technique more accurate.

Chapter 6

Conclusion

A substantial body of research has been conducted regarding the application of computers in assessment. The largest proportion of this research has concerned the assessment of student program code, as this is a significant factor in the lives of many Computer Science academics. With large classes common in introductory programming courses, assessment becomes a substantial burden for academics, and the value of assessments, from a student perspective, occasionally suffers as a result. The problem is accentuated by the need for frequent assessment, necessitated by the practice needed in learning to program.

This final chapter presents an overview of the objectives achieved in this research, as well as a discussion of the contributions made in the problem domain. Further sections discuss the limitations of this research and the opportunities for further work that arise.

6.1 Research Objectives

Chapter 2 provided a thorough review of the very large body of research on Computer Aided Assessment. Douce *et al.* (2005) provides a useful classification of CAA systems, but the distinction between second and third generations (the presence of a web front-end) disregards the assessment method. It was argued that it is more useful to define the third generation according to the special capabilities of the assessment techniques. In particular, it was suggested that the third generation be redefined as those systems capable of assessment at finer than *whole program* granularity.

The literature, indicates that dynamic analysis of student programs is well-understood. The same is true for measures of program quality based on static analysis. However, it was clear that more research was required in the application of static structural analysis to program assessment. The second objective was to investigate the available measures of similarity between structured data. Traditional measures of similarity were largely restricted to tree structures due to their computational complexity. Tree data structures are occasionally used for representing programs, but in each case there is important information (mostly dependency information), that cannot be stored within the normal structural relationships of the tree. A more flexible data structure, such as *directed acyclic graphs* was needed.

Several more recent graph similarity measures, based on propagating similarity scores through a product graph, are promising as they operate on general directed graphs. However, each of the published methods produce final similarity scores that may be compared with one another, but do not have intrinsic meaning. This latter requirement is essential if the similarity scores are to be treated as percentages.

6.2 Research Contributions

The main contribution of this research is a new graph similarity measure, called the *Weighted Assignment Similarity* measure. This is based on similarity propagation and is somewhat similar to the *SimRank* measure (Jeh & Widom, 2002). The novel feature of the new measure is that similarity only propagates across the locally optimal neighbour assignment, rather than across all possible neighbour assignments. This fact is sufficient to grant the similarity scores intrinsic meaning, and can reasonably be regarded as percentages.

Associated with this measure are several additional contributions:

- An important feature of the graphs that represent programs and graphs in many other application domains, is that important semantic detail is captured by attributes in vertices and edges. Prior authors, for example Jeh & Widom (2002), have suggested that these details may possibly be used to influence the similarity propagation measure. The first general technique by which this may be done is detailed in the current study.

- The attributes used in program graphs usually capture fine detail, such as literal values and the use of identifiers. A major problem in comparing programs is that they very often use distinct sets of identifiers. A further contribution of this research is the general technique for applying the similarity measure in a two step process. The first step finds the optimal mapping between identifiers in the two programs. In the second step, these mappings become relevant attribute detail that can steer the similarity measure.

- Since similarity propagation measures are iterative techniques, some assurances of timely convergence are needed for practical use. A proof of convergence over DAGs is offered, as well as a very efficient algorithm for applying the measure to DAGs.

In prior work such as that of Truong *et al.* (2004), the use of structural analysis was restricted to considering only small *fill-in-the-gaps* exercises, as it was not widely considered feasible to assess anything larger. An important contribution of this research is to show that it is indeed feasible to use structural similarity in the assessment of modest student programs.

6.3 Limitations of Research

Programming assignments are usually formulated as a description of the objectives and tasks the program must perform. However, this is not the only kind of programming assignment. Students may also be given partially completed programs, or be asked to extend a working program to add features. Sometimes students might be given a program with a variety of bugs, and the student's task is to identify and remove the bugs. A limitation of this research is that it cannot easily be applied to these kinds of questions, but its application is presently restricted to traditional programming assignments (program creation).

In the assessment of these kinds of assignments, an assumption had to be made, namely when humans assign marks to an ideal program solution, the marks are evenly distributed throughout the program. In general this is not the case. In particular, after students reach some proficiency, educators may wish to begin taking some programming details for granted and only assign marks to those regions they consider pertinent.

A further limitation is simply that the assessment contains no true intelligence. When it finds structural similarities, they accrue scores. Suppose a student does not really know how to solve a given problem, but submits a program with a moderate number of arbitrarily arranged constructs. Some of these may match elements of the structure of the ideal solution, accruing marks that a human marker would not assign. This is because human markers are able to extract meaning from the choice of variable names, string constants, and comments in the program. Such knowledge is simply not available to any current computer aided assessment system.

6.4 Suggestions for Future Research

A number of open problems remain. For example, a great deal of similarity between programs in the *standard set* is common. This is because each local variation must be represented by a whole new program. It seems clear that local variations may be expressed and managed more efficiently.

Program normalisation is important as it reduces the variety of program solutions that must be considered. An opportunity exists to investigate the relative value of different program transformations. Each have associated costs, but their benefits are (in general) not formally understood.

An important problem that remains is to determine to what extent this assessment method scales to larger problems. Are there heuristics that may be used to alert educators that an assignment is too complex to support assessment by structural similarity? Answers for these problems are expected, but their form is currently unknown.

Another large research opportunity is the investigation of feedback mechanisms based on the assessment technique. As the similarity measure accrues similarity scores at each vertex of the program graph, there is a wealth of information available to construct meaningful commentary for a program.

6.5 Conclusion

This current research addressed a problem that has received much attention in the academic community. The important approaches to the problem rely on either testing programs for correctness, or examining the code using heuristics that measure program quality. The application of structural similarity (another static analysis technique) to program assessment has seen some attention, but this has been largely restricted to program critiquing.

This study makes several valuable and novel contributions to computer aided assessment. It introduces a novel similarity measure and shows that it can be successfully applied to marking student programs, yielding marks strongly correlated to those assigned by a human marker.

The primary research question of this work regarded the extent to which the structural similarity between candidate and ideal programs can be used in assessment. This research confirms that student programs may be assessed in this way. It also shows that the most accurate assessments are achieved when programs are compared with a standard set of mixed-quality solutions, rather than only comparing against good solutions.

This study represents the first successful application of structural similarity to program assessment.

Bibliography

ALA-MUTKA, K. 2005. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, **15**(June), Pages 83–102.

ALLALI, J., & SAGOT, M-F. 2005. A New Distance for High Level RNA Secondary Structure Comparison. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, **2**(1), Pages 3–14.

ARCHER HARRIS, J., ADAMS, E. S., & HARRIS, N. L. 2004. Making program grading easier: but not totally automatic. *J. Comput. Small Coll.*, **20**(1), Pages 248–261.

BEATY, S. J. 2001. Programs, not code. *J. Comput. Small Coll.*, **17**(1), Pages 278–283.

BERRY, R E., & MEEKINGS, B. A. E. 1985. A style analysis of C programs. *Commun. ACM*, **28**(1), Pages 80–88.

BLONDEL, V. D., & VAN DOOREN, P. 2004. A Measure of Similarity between Graph Vertices: Applications to Synonym Extraction and Web Searching. *SIAM Rev.*, **46**(4), Pages 647–666.

BLONDEL, V. D., NINOVE, L., & VAN DOOREN, P. 2004. Convergence of graph similarity algorithms. *Proceedings of the 23rd Benelux Meeting on Systems and Control*, Paper FrP06–4.

BLONDEL, V. D., NINOVE, L., & VAN DOOREN, P. 2005. An affine eigenvalue problem on the nonnegative orthant. *Linear Algebra and its Applications*, **404**, Pages 69–84.

BLOOM, B. 1956. *Taxonomy of Educational Objectives: The Classification of Educational Goals*. Susan Fauer Company, Inc. Pages 201–207.

BUTTLER, D. 2004. A Short Survey of Document Structure Similarity Algorithms. *Pages 3–9 of: International Conference on Internet Computing*.

CARTER, J., ALA-MUTKA, K., FULLER, U., DICK, M., ENGLISH, J., FONE, W., & SHEARD, J. 2003. How shall we assess this? *Pages 107–123 of: ITiCSE-WGR '03: Working group reports from ITiCSE on Innovation and technology in computer science education*. New York, NY, USA: ACM Press.

CHANON, R. N. 1966. Almost alike programs. *Pages 215–222 of: Proceedings of the 1966 21st national conference*. New York, NY, USA: ACM Press.

CHAWATHE, S. S., & GARCIA-MOLINA, H. 1997. Meaningful change detection in structured data. *Pages 26–37 of: SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press.

CHAWATHE, S. S., RAJARAMAN, A., GARCIA-MOLINA, H., & WIDOM, J. 1996. Change detection in hierarchically structured information. *Pages 493–504 of: SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press.

COBERTURA. 2006. *Cobertura*. `http://cobertura.sourceforge.net`, accessed on 19 October 2006.

DALY, C. 1999. RoboProf and an introductory computer programming course. *SIGCSE Bull.*, **31**(3), Pages 155–158.

DOUCE, C., LIVINGSTONE, D., & ORWELL, J. 2005. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, **5**(3), Page 4.

EDWARDS, S. H. 2003. Improving student performance by evaluating how well students test their own programs. *J. Educ. Resour. Comput.*, **3**(3), Page 1.

ELLSWORTH, C. C., FENWICK, J. B., & KURTZ, B. L. 2004. The Quiver system. *Pages 205–209 of: SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer Science education*. New York, NY, USA: ACM Press.

ENGLISH, J. 2004. Automated assessment of GUI programs using JEWL. *Pages 137–141 of: ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education*. New York, NY, USA: ACM Press.

FORSYTHE, G. E., & WIRTH, N. 1965. Automatic grading programs. *Commun. ACM*, **8**(5), Pages 275–278.

GANESAN, P., GARCIA-MOLINA, H., & WIDOM, J. 2003. Exploiting hierarchical domain structure to compute similarity. *ACM Trans. Inf. Syst.*, **21**(1), Pages 64–93.

GAREY, M. R., & JOHNSON, D. S. 1990. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. Chichester, England: Ellis Horwood Limited.

GOLDWASSER, M. H. 2002. A gimmick to integrate software testing throughout the curriculum. *Pages 271–275 of: SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer Science education*. New York, NY, USA: ACM Press.

GRUNE, D., & JACOBS, C. J. H. 1990. *Parsing techniques a practical guide*. Chichester, England: Ellis Horwood Limited.

HEXT, J. B., & WININGS, J. W. 1969. An automatic grading scheme for simple programming exercises. *Commun. ACM*, **12**(5), Pages 272–275.

HIGGINS, C., SYMEONIDIS, P., & TSINTSIFAS, A. 2002. The marking system for CourseMaster. *Pages 46–50 of: ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology in computer science education.* New York, NY, USA: ACM Press.

HIGGINS, C. A., GRAY, G., SYMEONIDIS, P., & TSINTSIFAS, A. 2005. Automated assessment and experiences of teaching programming. *J. Educ. Resour. Comput.*, **5**(3), Page 5.

HOLLINGSWORTH, J. 1960. Automatic graders for programming classes. *Commun. ACM*, **3**(10), Pages 528–529.

JACKSON, D. 1996. A software system for grading student computer programs. *Comput. Educ.*, **27**(3-4), Pages 171–180.

JACKSON, D. 2000. A semi-automated approach to online assessment. *Pages 164–167 of: ITiCSE '00: Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education.* New York, NY, USA: ACM Press.

JACKSON, D., & USHER, M. 1997. Grading student programs using ASSYST. *Pages 335–339 of: SIGCSE '97: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education.* New York, NY, USA: ACM Press.

JCOVERAGE. 2006. *jcoverage (sic).* `http://www.jcoverage.com`, accessed on 19 October 2006.

JEH, G., & WIDOM, J. 2002. SimRank: a measure of structural-context similarity. *Pages 538–543 of: KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining.* New York, NY, USA: ACM Press.

JOY, M., & LUCK, M. 1998. Effective electronic marking for on-line assessment. *Pages 134–138 of: ITiCSE '98: Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education.* New York, NY, USA: ACM Press.

LASS, R. N., CERA, C. D., BOMBERGER, N. T., CHAR, B., POPYACK, J. L., HERRMANN, N., & ZOSKI, P. 2003. Tools and techniques for large scale grading using Web-based commercial off-the-shelf software. *Pages 168–172 of: ITiCSE '03: Proceedings of the 8th annual conference on Innovation and technology in computer science education.* New York, NY, USA: ACM Press.

MCCABE, T. J. 1976. A complexity measure. *Page 407 of: ICSE '76: Proceedings of the 2nd international conference on Software engineering.* Los Alamitos, CA, USA: IEEE Computer Society Press.

MELNIK, S., GARCIA-MOLINA, H., & RAHM, E. 2002. Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. *ICDE*, **00**, Pages 117–128.

MORRIS, D. S. 2003. Automatic grading of student's programming assignments: an interactive process and suite of programs. *fie*, **3**, Paper S3F–6.

MUNKRES, J. 1957. Algorithms for the Assignment and Transportation Problems. *Journal of the Society of Industrial and Applied Mathematics*, **5**(1), Pages 32–38.

NIERMAN, A., & JAGADISH, H. V. 2002. Evaluating Structural Similarity in XML Documents. *In: Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002)*.

NOUNIT. 2006. *NoUnit*. `http://nounit.sourceforge.net`, accessed on 19 October 2006.

PRESTON, J. A. 1997. Evaluation software: improving consistency and reliability of performance rating. *Pages 132–134 of: ITiCSE-WGR '97: The supplemental proceedings of the conference on Integrating technology into computer science education: working group reports and supplemental proceedings*. New York, NY, USA: ACM Press.

PRESTON, J. A., & SHACKELFORD, R. 1999. Improving on-line assessment: an investigation of existing marking methodologies. *SIGCSE Bull.*, **31**(3), Pages 29–32.

REEK, K. A. 1989. The TRY system -or- how to avoid testing student programs. *Pages 112–116 of: SIGCSE '89: Proceedings of the twentieth SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM Press.

REES, M. J. 1982. Automatic assessment aids for Pascal programs. *SIGPLAN Not.*, **17**(10), Pages 33–42.

RICH, C., & WILLS, L. M. 1990. Recognizing a Program's Design: A Graph-Parsing Approach. *IEEE Softw.*, **7**(1), Pages 82–89.

ROSÉ, C. P., ROQUE, A., BHEMBE, D., & VANLEHN, K. 2003. A hybrid approach to content analysis for automatic essay grading. *Pages 88–90 of: NAACL '03: Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology*. Morristown, NJ, USA: Association for Computational Linguistics.

SAGER, T., BERNSTEIN, A., PINZGER, M., & KIEFER, C. 2006. Detecting similar Java classes using tree algorithms. *Pages 65–71 of: MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*. New York, NY, USA: ACM Press.

SAIKKONEN, R., MALMI, L., & KORHONEN, A. 2001. Fully automatic assessment of programming exercises. *SIGCSE Bull.*, **33**(3), Pages 133–136.

SAIP, H., & LUCCHESI, C. 1993. Matching algorithms for bipartite graphs. *Technical Report DCC-03/93 (Departamento de Ciência da Computação, Universidade Estudal de Campinas).*

SCHORSCH, T. 1995. CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. *Pages 168–172 of: SIGCSE '95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education.* New York, NY, USA: ACM Press.

SELKOW, S. M. 1977. The tree-to-tree editing problem. *Information Processing Letters*, **6**, Pages 184–186.

SHASHA, D., & ZHANG, K. 1989. Fast parallel algorithms for the unit cost editing distance between trees. *Pages 117–126 of: SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures.* New York, NY, USA: ACM Press.

SITTHIWORACHART, J., & JOY, M. 2004. Effective peer assessment for learning computer programming. *Pages 122–126 of: ITiCSE '04: Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education.* New York, NY, USA: ACM Press.

TAI, K-C. 1979. The Tree-to-Tree Correction Problem. *Journal of the ACM*, **26**(3), Pages 422–433.

THOMAS, P., WAUGH, K., & SMITH, N. 2005. Experiments in the automatic marking of ER-diagrams. *Pages 158–162 of: ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education.* New York, NY, USA: ACM Press.

THORBURN, G., & ROWE, G. 1997. PASS: An automated system for program assessment. *Computers & Education*, **29**(4), Pages 195–206.

TRAYNOR, D., BERGIN, S., & PAUL GIBSON, J. 2006. Automated assessment in CS1. *Pages 223–228 of: CRPITS'52: Proceedings of the 52nd conference on Computing education 2006.* Darlinghurst, Australia: Australian Computer Society, Inc.

TREMBLAY, G., & LABONTÉ, É. 2003. Semi-automatic marking of Java programs using JUnit. *Pages 42–47 of: EISTA '03: Proceedings of the International Conference on Education and Information Systems: Technologies and Applications.* International Institute of Informatics and Systemics.

TRUONG, N., ROE, P., & BANCROFT, P. 2004. Static analysis of students' Java programs. *Pages 317–325 of: ACE '04: Proceedings of the sixth conference on Australasian computing education.* Darlinghurst, Australia: Australian Computer Society, Inc.

VON MATT, U. 1994. Kassandra: the automatic grading system. *SIGCUE Outlook*, **22**(1), Pages 26–40.

YANG, R., KALNIS, P., & TUNG, A. K. H. 2005. Similarity evaluation on tree-structured data. *Pages 754–765 of: SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press.

ZELKOWITZ, M. V. 1976. Automatic program analysis and evaluation. *Pages 158–163 of: ICSE '76: Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press.

ZHANG, K., STATMAN, R., & SHASHA, D. 1992. On the edit distance between unordered labeled trees. *Pages 133–139 of: Information Processing Letters*, vol. 42.