SEARCH–BASED PROGRAM TRANSFORMATION FOR AMORPHOUS SLICING AND PROGRAM COMPREHENSION

By
Ayọdèji Fatiregun

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY AT KING’S COLLEGE LONDON STRAND, LONDON, WC2R 2LS, UK SEPTEMBER, 2006

© Copyright by Ayọdèji Fatiregun, 2006
........................................... To the memory of my Dad.
# Table of Contents

Table of Contents ........................................ ii

Abstract ................................................. xi

Acknowledgements ....................................... xii

Declaration .............................................. xiii

1 Introduction ........................................... 1
   1.1 Program Transformation ............................. 1
      1.1.1 An Example of Transformation .................. 2
      1.1.2 A Sequence of Transformations ................. 4
      1.1.3 Applications of Program Transformation ....... 7
   1.2 Metaheuristic Search ............................... 8
      1.2.1 Representation Scheme ........................ 10
      1.2.2 Fitness Function .............................. 11
      1.2.3 Techniques .................................... 11
      1.2.4 Example of a Hill Climbing search for
            Transformation Sequences .................... 12
   1.3 Aims and Objectives of this Thesis ............... 15
   1.4 Contributions of this Thesis ...................... 16
   1.5 Organisation of this Thesis ....................... 17

2 Literature Survey ..................................... 20
   2.1 Introduction ...................................... 20
   2.2 Transformation ................................... 20
      2.2.1 Source–Source Code Transformation .......... 21
      2.2.2 Source–Object Code Transformation .......... 22
      2.2.3 Inter–Language Source Transformations ....... 22
      2.2.4 Survey of Transformation Systems .......... 22
4 Application of SB-Transformations: Amorphous Slicing 78
  4.1 Introduction .................................................. 79
  4.2 Problem ....................................................... 79
  4.3 Reformulating Amorphous Slicing as a Redundancy Removal Problem 80
  4.4 Experimental Study 2 - Search Based Amorphous Slicing ............... 82
    4.4.1 Experimental Setup ....................................... 82
    4.4.2 Test Objects ............................................. 84
    4.4.3 Results .................................................. 86
    4.4.4 Output Slices ............................................ 97
    4.4.5 A Surprising Result ...................................... 106
    4.4.6 Statistical Analysis ..................................... 107
    4.4.7 Conclusions of Experimental Study 2 ...................... 107
  4.5 Summary ..................................................... 108

5 Variable Proximity based Program Comprehension 110
  5.1 Introduction .................................................. 111
  5.2 Problem and Motivation ...................................... 113
  5.3 Transforming for Comprehension ................................ 113
  5.4 Context Sensitive Comprehension ................................ 115
    5.4.1 Define-Use (DU) Based Fitness ............................ 116
    5.4.2 Lexical Based Fitness Function ........................... 117
  5.5 Search Based Heuristics ...................................... 118
    5.5.1 Multi-Objective Optimization ............................... 118
  5.6 Experimental Study 3: Measures of variable proximity for program comprehension .................................. 120
    5.6.1 Experimental Setup ....................................... 121
    5.6.2 Test Objects ............................................. 123
    5.6.3 Fitness Function .......................................... 123
    5.6.4 Results and Discussion .................................... 124
    5.6.5 Program Outputs .......................................... 130
    5.6.6 Statistical Analysis ...................................... 132
  5.7 Summary ..................................................... 132

6 Reflections 134
  6.1 Introduction .................................................. 134
  6.2 Kinds of Transformations ...................................... 135
  6.3 Choice of Search Algorithms .................................. 137
  6.4 Generic Solution ............................................. 139
  6.5 Fitness Function ............................................. 140
## CONTENTS

6.6 Search Landscape ........................................ 141
6.7 Genetic Operators ........................................ 142
  6.7.1 Choice of Crossover Operator ......................... 142
  6.7.2 Selection of Mutation Operator ....................... 144
6.8 Threats to Validity ....................................... 145
  6.8.1 Internal Threats ..................................... 145
  6.8.2 External Threats ..................................... 146
6.9 Summary .................................................. 146

7 Conclusions And Future Work 148
7.1 Summary of Achievements ................................. 148
  7.1.1 Is it possible to fully automate a generalised program transformation using a search-based framework? 149
  7.1.2 Which search-algorithms produce better solutions to program transformation problems? ..................... 150
7.2 Conclusion ................................................ 151
  7.2.1 SB-code optimisations ................................. 152
  7.2.2 SB-Amorphous Slicing ................................ 152
  7.2.3 SB-Program Comprehension ........................... 153
7.3 Restrictions and Limitations ............................. 154
  7.3.1 Types of Programs .................................... 154
  7.3.2 Scalability ............................................ 154
7.4 Summary of Future Work ................................ 155
  7.4.1 Inter-procedural and Object-Oriented Test Objects 155
  7.4.2 Extending the Search-Framework ....................... 155
  7.4.3 Static Analysis to find areas of promising optimisations ........................................ 156
  7.4.4 Inclusion of future transformation rules ............. 156
  7.4.5 Field Study in Program Comprehension ............... 156

Bibliography .................................................. 157

A Source Description for Test Objects 174
A.1 Surface Area .............................................. 174
A.2 Odds / Even ............................................... 175
A.3 Tax ......................................................... 176
A.4 Student-Marks ............................................. 179
A.5 Calendar ................................................... 184
A.6 Rear-Wing Defroster ...................................... 186
A.7 Braking Controller Defroster ............................ 186
A.8 Test Object for Program Comprehension .......................... 187
List of Figures

1.1 Examples of program transformation rules ................................. 4
1.2 Source fragment: source program before sequence is applied ........ 4
1.3 Output fragment: resulting program from the application of sequence
[T4, T6] to source program in Figure 1.2 ................................. 5
1.4 Example Transformations .................................................... 6
1.5 Hill Climbing landscape. ................................................... 12
1.6 Framework for Search-Based Program Transformation. ................. 15

2.1 Iterative stages of a Genetic Algorithm, from the initial Population,
crossover through to the application of mutation operators. ............ 33
2.2 Illustration of single point crossover. ................................... 34
2.3 Different landscapes showing appropriateness of Hill Climbing to each
landscape. ........................................................................ 38
2.4 Transformation of a search landscape. Image taken from [83] ........ 41

3.1 First-Ascent Hill Climbing Algorithm structure. ........................ 54
3.2 Genetic Algorithm structure. ................................................ 56
3.3 Greedy Algorithm structure. ................................................ 57
3.4 Sample test programs, showing the increasing number of moves to the right that a good sequence would require to produce an optimal solution. Program one requires a sequence of absorb transformations to produce the optimal result. In order to simplify program two and three, the algorithms need to be ‘intuitive’ to realise where potential optimisations may be applied, e.g., in program three, the program cursor would eventually need to go DOWN the Abstract Syntax Tree (AST) for the program, into the IF-block for further simplifications to occur. 65

3.5 if-tests: Exceptional ‘toy’ test case showing Random Search outperforming both hill climbing and genetic algorithm. 69

3.6 Comparison of performance of GA, HC and RS on ‘toy’ test 1 object. 71

3.7 Comparison of performance of GA, HC and RS on ‘toy’ test 2 object. 71

3.8 Comparison of performance of GA, HC and RS on ‘toy’ test 3 object. 72

3.9 Comparison of performance of GA, HC and RS on ‘toy’ test 4 object. 72

3.10 Comparison of performance of GA, HC and RS on ‘toy’ test 5 object. 73

3.11 Comparison of performance of GA, HC and RS on ‘toy’ test 6 object. 73

3.12 Comparison of performance of GA, HC and RS on ‘toy’ test 7 object. 74

3.13 Results of Wilcoxon significance tests showing strong significance in the results with the GA outperforming both random search and the HC. 75

4.1 Illustrative example showing Amorphous Slicing Producing Thinner Slices by Removing Syntactic Restrictions. Slice computed w.r.t. variable slice at end. 80

4.2 Re–structuring Source Program for Search–Based Transformations . 81

4.3 Results of executing different search techniques to compute amorphous slice and the analytic amorphous slicing system showing average percentage reduction in slice size 87

4.4 Rear-Window defroster test Object. 88

4.5 Calendar test object. 89

4.6 Odds / Even test object. 89
LIST OF FIGURES

4.7 Surface Area test object. ........................................... 90
4.8 Braking System Controller test object. .............................. 91
4.9 Student Marks test object. .......................................... 91
4.10 Tax test object. ........................................................ 92
4.11 bar plot for surface area test object. ............................... 93
4.12 box plot for odds/even test object. ................................. 94
4.13 boxplot for calendar test object. ................................. 94
4.14 boxplot for student marks test object. .......................... 95
4.15 boxplot for tax test object. ........................................ 95
4.16 boxplot for braking system controller test object. .............. 96
4.17 boxplot for rear-window defroster test object. .................. 96
4.18 Source code for tax program before transformation. .......... 98
4.19 Output code slice for tax program returned by the Greedy Algorithm.
   Slicing criterion is variable personal at end of program. ...... 99
4.20 Output code slice for tax program returned by the Genetic Algorithm.
   Slicing criterion is variable personal at end of program. .... 101
4.21 Output code slice for tax program returned by Random Search. Slicing
criterion is variable personal at end of program. ............... 102
4.22 Output code slice for tax program returned by Hill Climbing Algo-
rithm. Slicing criterion is variable personal at end of program. .. 103
4.23 Output code slice for tax program returned by LinIAS. Slicing criterion
   is variable personal at end of program. ....................... 104
4.24 Output code slice for tax program constructed by human. Slice repre-
   sents the optimum solution for the program based on variable personal
   as slice criterion. ...................................................... 105
4.25 Results of Wilcoxon significance statistical tests. The pair analyses
   significant differences between results from pairs of techniques. .. 108

5.1 Program transformed for comprehension .......................... 115
5.2 Example illustration distance-based DU Fitness .................. 117
LIST OF FIGURES

5.3 Example illustration distance-based Lexical Fitness .................. 118
5.4 Example showing multi-objective optimization ....................... 119
5.5 Graph of (x,y) equally weighted optimisation ..................... 125
5.6 Graph of x-bias mean fitness results ................................. 127
5.7 Graph of y-bias mean fitness results ................................. 129
5.8 Pareto Front for the three search based techniques .................. 129
5.9 Source code for program before transformation ..................... 130
5.10 Comparison of output programs produced by the GA, HC, RS and Human transformation ............................................. 131
5.11 Results of Wilcoxon significance tests showing strong significance in the results with the GA outperforming both random search and the HC.132

6.1 Performance of the different search techniques over the three experiments performed. P.T. represents program transformation, A.S., represents amorphous slicing and P.C. represents program comprehension. 139
6.2 Illustration of uniform crossover ........................................ 144
Abstract

This thesis addresses the automation of program transformation using search-based heuristics: a genetic algorithm, a hill climbing algorithm, a greedy search algorithm and random search.

Program transformation is a useful technology that has successfully been applied to many Software Engineering sub-areas such as program comprehension, reverse engineering, re-engineering, software testing, maintenance, compiler optimisation and so on. In all these applications, transformation algorithms are constructed by hand for each different transformation goal. Loosely speaking a transformation algorithm defines a sequence of transformation steps to apply to a given program. It is notoriously hard to find good transformation sequences automatically, and so much (costly) human intervention is required.

In this thesis, we analysed the transformation problem and reformulated it as a search problem, where search techniques systematically and intelligently traverse through a large number of potential candidate solution in a search space to find one which is the best solution, or where that is not possible, one solution which is at least good enough. We show how search-based meta-heuristic algorithms can be used to automate, or partly automate the problem of finding good transformation sequences.

We show that the approach is sufficiently general to find sequences of transformation rules for different transformation goals. These transformation goals represent areas within Software Engineering where program transformation has been successfully applied. We provide experimental results regarding the effectiveness of the search in the sub-areas of re-engineering, program slicing and program comprehension. For all test objects considered and test goals considered, the meta-heuristic search methods were able to find good sequences and, as a basic requirement, outperformed random search.
Acknowledgements

First and foremost, I would like to thank God for all his mercies and for strength to have carried on thus far.

Special thanks to both my supervisors - Professor Mark Harman for your unrivalled and unparalleled supervision. You gave me much belief, support and motivation towards this project and research in general, and Professor Robert M. Hierons for your contributions and suggestions throughout the duration of this work. I am greatly in your debt.

I would like to thank Derek Jones for your very helpful communication regarding program comprehension.

I would like to thank my examiners Drs. Sebastian Danicic and Sue Black for your suggested comments on improving the quality of my research. Dr. Danicic for kindly supplying WSL test programs used in the experiments. Also like to thank Daimler Chrysler for supplying test programs also used in the experiments.

To my parents, family and friends for all your encouragement, love and for putting up with so much over the last few years.

I gratefully acknowledge financial support by the department of computer science at King’s College, London.

Lastly, and most certainly not least, very special thanks to all members of CREST (formerly King’s GESTERS). Your inputs have been invaluable and I owe you all a great debt.
Declaration

The work presented in this thesis is original work undertaken between November 2002 and September 2006 initially at Brunel University and subsequently completed at King’s College, University of London. Some of the work presented in this thesis has previously been published in the following papers:


All of the work contained within this thesis represents the original contributions of the author.
Chapter 1

Introduction

In order to automate program transformation, this thesis presents a framework for performing source-code transformation using search-based heuristics. In this dissertation, I describe different experiments that employ a number of search techniques, such as genetic algorithms and hill climbing to achieve different program transformation goals. This chapter explains the need for program transformation, some applications of program transformation, meta-heuristic search algorithms and how meta-heuristic search may be applied to program transformation (searching through sequences of meaning-preserving transformation rules). Lastly, this chapter presents an overview of the dissertation.

1.1 Program Transformation

Program Transformation is a widely researched area in software engineering. It is used in various software engineering sub-disciplines such as program synthesis, optimisation, refactoring, reverse engineering, re-engineering and program comprehension. Transformation has also been shown to be a useful supporting technology for search-based software testing using evolutionary search techniques [56, 57, 58, 61, 83].
However, in all these applications the transformation of a source program $P$ into a desirable target program $P'$ is an expensive process insofar as so much (costly) human intervention is required. The transformation algorithms are constructed by hand for each different transformation goal. Loosely speaking, a transformation algorithm defines a sequence of transformation steps to apply to a given program and it is notoriously hard to find good transformation sequences automatically [27].

The purpose of this thesis is to investigate the automatic generation of transformation sequences for any given program and for any transformation goal that can be formally expressed as an objective function. This is a difficult problem, because, for any program of any realistic size and a transformation system with an equally realistic number of transformation rules, the number of permutations / combinations of each rule in a sequence is too large to be searched exhaustively.

For these reasons, we define the transformation problem as a search problem and consider the application of metaheuristic search techniques, such as genetic algorithms and hill climbing algorithms to search the domain of transformation rules for good transformation sequences.

1.1.1 An Example of Transformation

The program fragments described in this thesis are written in the Wide Spectrum Language (WSL) syntax.

The essential idea in semantic preserving program transformation is to change the representation of a given piece of code, while still preserving its functional behaviour. This is done by applying certain rules of manipulation that show how a given fragment of code may be converted into another form. Figure 1.1, shows a list of some rules
that can be used in program transformation. Consider the rule $T_1$ which can be used to transform the code fragment below:

\[
\text{IF (x=y) THEN p:=1 ELSE p:=2 FI;}
\]

This fragment could be re-written as:

\[
\text{IF (x<>y) THEN p:=2 ELSE p:=1 FI;}
\]

without affecting its meaning.

Similarly, the rule $T_4$ suggests that if we had a code fragment such as:

\[
a:=x+y; b:=u+v;
\]

Then these two statements could be written as:

\[
b:=u+v; a:=x+y;
\]

The inversion of both statements in this case is clearly only possible because there are no existing dependencies between both statements.

A rule can only be applied to a code fragment if it applies to the terms of the fragment. The terms of the fragment refer to the structure or pattern of the code fragment. In our first example fragment, the terms of the rule $T_1$ pattern match with the constructs of the code fragment, therefore, $T_1$ is applicable and is said to be valid for that code fragment. However, $T_1$ though valid as a transformation rule, is not applicable to the second code fragment because its terms do not pattern match with the terms in the code fragment.
T_1: IF (e1) THEN s1; ELSE s2; FI; \Rightarrow IF (\neg e1) THEN s2; ELSE s1; FI;
T_2: IF TRUE THEN s1; ELSE s2; FI; \Rightarrow s1;
T_3: x:=2; x:=x - 1; y := 10; x := x + 1; \Rightarrow x := 2; y := 10;
T_4: x := x + 1; y := y + 1; \Rightarrow y := y + 1; x := x + 1
T_5: FOR (s1; e2; s2) DO s3; OD; \Rightarrow s1; WHILE (e2) DO s3; s2; OD;
T_6: x := x + 1; x := x + 1; \Rightarrow x := x + 2;

Figure 1.1: Examples of program transformation rules

1. x := x+1;
2. y := y+1;
3. x := x+1;
4. IF (x>y) THEN
5. \quad a := a+1;
6. ELSE
7. \quad b := b+1;
8. FI;
9. a := a+1;

Figure 1.2: Source fragment: source program before sequence is applied

1.1.2 A Sequence of Transformations

Transformation rules such as those such in Figure 1.1 are seldom applied in isolation. Alone, they serve little or no purpose whatsoever. Rather, these rules are applied in a sequence, one after the other on a target program to achieve an overall desired effect. The application of one transformation rule would normally create an opportunity for a subsequent rule to be applied and so on. It is this “domino effect” that generates substantial syntactic differences between the original and transformed versions of code.

Consider the code fragment shown in Figure 1.2. Suppose we want to apply our transformation rules from Figure 1.1 to this fragment, we show the application of a
1. \( y := y + 1; \)
2. \( x := x + 2; \)
3. IF \( x > y \) THEN
4. \( a := a + 1; \)
5. ELSE
6. \( b := b + 1; \)
7. FI;
8. \( a := a + 1; \)

Figure 1.3: Output fragment: resulting program from the application of sequence \([T4, T6]\) to source program in Figure 1.2

sequence of transformations \([T4, T6]\) - assuming that the current program cursor position along the program Abstract Syntax Tree (AST) is on line 1 (the head node) - to the fragment and the resultant output program shown in Figure 1.3. The program cursor position indicates the program node along the AST to which the selected transformation rule would be applied.

An AST is an internal structure in memory, which is a representation of the source code. The following shows an example of an Abstract Syntax Tree.

A merge transformation rule, where possible, combines two adjacent nodes into a single node. The following fragment shows an example of the effect of a merge transformation rule.
A move transformation rule would swap the position of two adjacent nodes if such a swap could occur without breaking any dependencies. For example:

\[
\begin{align*}
a : b + c; \\
d := e + f;
\end{align*}
\]

\[
\begin{align*}
d := e + f; \\
a := b + c;
\end{align*}
\]

Another example that illustrates the transformation of a given program using a sequence of simple transformation rules is given in Figure 1.4.

By examining the program in Figure 1.4 and assuming we had a current cursor position on line 1 of the source program, observe that lines 1 and 2 may be merged into one statement corresponding to line 1 in prog2(b). However, we would need to skip over lines 3 and 4 in order to apply more optimisations to the source, placing the cursor position on line 5. Two further merge operations would simplify lines 3 through 5 in prog2(a) to one statement (line 4) in prog2(b). Therefore, the following transformation rules merge, move-right, move-right, merge, merge would
result in the transformation of prog2(a) to prog2(b). Further simplification may be performed on prog2(b) and for completion we show the final transformation of prog2(a) as prog2(c) obtained by the transformation sequence: \textit{merge, move-right, move-right, merge, merge, back, back, move-right, move-right, merge}.

Importantly, this is what this thesis tries to tackle: the training of heuristic search algorithms to build up sequences of low level transformation rules which when applied to a source program result in a functionally equivalent, optimised, transformed version.

\subsection*{1.1.3 Applications of Program Transformation}

Program Transformation has several applications including:

- **Program Synthesis** [79, 116], which is the generation of executable programs from specifications.

- **Optimizations** [30]: These techniques are geared toward improving resources during program execution. Programs can be optimised to improve speed, memory consumption, etc.

- **Refactoring** [84, 99]: This is referred to as methods for restructuring an existing body of code. In this work, we use the term transformation to refer to simple code restructurings while refactoring involves restructuring at a higher level. For instance, a simple refactoring may involve moving a method from one class to another, which would not necessarily equate to a simple transformation. Such a refactoring process might require multiple transformation rules to be applied.

- **Program Maintenance** activities are those operations performed after software
has been deployed, and serve as corrective measures to the system. [13, 48]

- Reverse-Engineering [25, 67, 113], which can be described as the generation of more abstract representations, such as, generating a program specification or design artefacts from the code.

- Program Comprehension [96], which is used to refer to the readability and understandability of a program.

1.2 Metaheuristic Search

Search techniques are those that, through investigation, attempt to find good answers or solutions to a particular problem description. Searching is the activity of looking thoroughly in order to find or compute an appropriate answer. A search strategy is described as an approach or framework which a particular search techniques employs to find these good solutions. It defines how a particular search should be performed and what new areas of the search space need to be explored. They are used widely in Artificial Intelligence to find a sequence of steps to move from an initial state, to a goal state.

There are different types of search techniques and the choice of which strategy to opt for usually depends on the search space being explored. When search is performed, the general idea is to look for a universally defined ‘correct’ solution to a problem. This ‘correct’ solution is also referred to sometimes as the optimal solution and where there does not exist such an optimal solution, then the user of the search paradigm may have to settle for a solution which is merely “good enough” - as close to optimal as possible.
For simple problems, we can systematically and exhaustively search through all possible solutions in the search space to find the best solution and relatively simple search algorithms such as Depth-First Search and Breadth-First Search tree traversal algorithms would be appropriate. However, for more complex problems, there may be a very large number of potential solutions in the search space, or a large number of paths to traverse. In such cases, it would be infeasible to search through the entire space. We apply heuristic search techniques to these types of problem.

Metaheuristic search algorithms are search algorithms that have been widely used to solve a range of problems across multiple domains [5, 23, 38, 40, 78, 86]. They are a class of search paradigms that require some level of intelligence built into the search. The useful feature in a meta-heuristic search procedure is the ability to predict from a current possible candidate solution, another candidate which is better than the current one so that the entire search procedure is guided. Rather than search the entire space of possible solutions exhaustively, the techniques focuses only on those solutions that seem to tend toward the optimal solution.

Clearly, as can be seen from the description of meta-heuristic search, there is a need to distinguish between the quality of different possible candidate solutions in the search space in order to determine which is better or worse and in which direction the search should move toward. Some meta-heuristic search algorithms include: Genetic Algorithms [47], Genetic Programming [73], Evolutionary Strategies [10], Hill-Climb Algorithms [86], Simulated Annealing [70] and Tabu Search [45]. They are attractive for problem domains where it is quite difficult to compute an exact solution. They also perform well for problem domains that have a large amount of possible solutions hence the search for the ‘best’ [27].
The ability to search through a myriad of potential solutions in order to find potentially good solutions to problems and the flexibility of these techniques wherein they can be adapted to different problem domains makes them very attractive. In their purest form, they require no domain knowledge. These characteristics have led to the emergence of a fairly new sub-area of software engineering referred to broadly as Search-Based Software Engineering (SBSE). The goal is to be able to perform traditional software engineering tasks such as resource allocation, work breakdown, test-data generation, etc., which have hitherto been carried out using systematic, analytical methods by using search-based techniques [27]. One common feature in software engineering is the lack of a clearly defined solution to a problem but rather the existence of several possible combinations of requirements from a generally large space of possibilities. This domain therefore, is well suited for the application of search-based approaches.

The subject of this thesis, Search-Based Program Transformation, is an example of using search heuristics to solve software engineering problems.

1.2.1 Representation Scheme

The representation scheme refers to the manner in which a candidate solution is encoded. Search techniques rely on being able to manipulate input sequences. Our input sequences are our transformation rules (Figure 1.1) which we apply to source programs in sequence. Below is an example of a transformation sequence.

\[ [T_1,T_2,T_3,T_4,T_4,T_3,T_3,T_1,T_4] \]
1.2.2 Fitness Function

The fitness function is arguably the most important element of a meta-heuristic search method. The fitness function describes the quality of potential solutions to the problem and it is the basis on which comparisons are made between several different candidate solutions. Therefore, it can be said that the fitness function drives the search toward finding the presence of an ideal solution.

1.2.3 Techniques

The hill climbing algorithm is one meta-heuristic search technique that is very simple. The basic idea is to start with an initial solution (usually selected randomly) and to always move toward a nearby solution that is better than the current one according to the fitness function. This process is repeated until no better nearby solutions can be found. The algorithm does not attempt to search through every possible solution in the space. There are two main variations to how the algorithm selects the next better candidate solution. These are the next-ascent and the steepest ascent techniques. In the next-ascent techniques, the algorithm compares the current solution with one nearby solution and selects the better solution of the two as the current best. The steepest ascent technique compares the current solution with a range of nearby solutions and selects the best solution from the group. The next-ascent variation of the hill climbing algorithm can be summarised as follows:

1. Start with initial solution (current solution) and calculate its fitness score.

2. REPEAT UNTIL no better solution is found
   
   (a) Get a nearby (neighbour) solution and calculate its fitness score.
If fitness value of neighbour is better than fitness value of current solution, then neighbour becomes current best solution.

Graphically, this can be seen in Figure 1.5.

![Hill Climbing landscape](image)

**Figure 1.5:** Hill Climbing landscape.

### 1.2.4 Example of a Hill Climbing search for Transformation Sequences

Consider again the example in Figure 1.4, recall that we showed how a transformation sequence: \([\text{merge}, \text{move-right}, \text{move-right}, \text{merge}, \text{merge}]\) would transform the source program into the intermediate transformed program. The following simple example illustrates how we may search and arrive at this ideal solution from another possible solution using the hill climbing mechanism.

Imagine at the start of the search, if we had an initial randomly generated sequence of transformation rules, such as, \([\text{merge}, \text{move-right}, \text{move-right}, \text{move-down},\)
move-left], this sequence would transform the source program ($prog2(a)$) in Figure 1.4 into the transformed program ($prog2(b)$) also shown in Figure 1.4. In this example, the goal of the transformation exercise is to obtain an equivalent program with fewer lines of code. The fitness function value for a transformation sequence here is:

\[ \text{fitness} = \text{Size of input program} - \text{Size of output program (LoC)} \]

A fitness value of 10 is better than 8 or 7 or 6 and so on. The initial transformation sequence would generate the following program:

1. $x := x + 2$;
2. $y := a + b$;
3. $z := c + d$;
4. $x := x + 1$;
5. $x := x + 1$;
6. $x := x + 1$;

Therefore the fitness score of the initial sequence $\text{merge, move-right, move-right, move-down, move-left}$ would be 1.

In the general hill climbing approach, the search identifies a neighbouring candidate solution to the current solution and measures the fitness value for that new solution. We define a neighbour to a transformation sequence as a sequence that is one transformation step different from that sequence. Therefore, sequence $[\text{merge, move-right, move-right, move-down, move-right}]$ is considered a neighbour to $[\text{merge, move-right, move-right, move-down, move-left}]$ because both sequences are similar except for the last transformation rule. Applying the neighbour to the source program, we get the program shown below:
This program is similar to the that generated by the first sequence and has an identical fitness score of 1. Therefore, different transformation sequences can generate similar programs or programs with similar fitness scores. The technique retains the initial sequence as the current best because there has not been a better score.

The search may then select \([\text{merge, move-right, move-right, merge, move-left}]\) as the next nearby solution and this sequence produces a program with five lines of code and consequently a fitness score of 2. This new sequence becomes the current best solution and a new nearby neighbour is selected. This process is repeated until the search selects the sequence \([\text{merge, move-right, move-right, merge, merge}]\), which is a nearby solution. This sequence produces:

1. \(x := x + 2;\)
2. \(y := a + b;\)
3. \(z := c + d;\)
4. \(x := x + 3;\)

The above sequence has a fitness score of 3 and therefore, becomes the current best solution. Other nearby solutions from this sequence repeatedly fail to generate a program with a better fitness value, therefore, the search stagnates and the current best sequence is returned as the optimal solution. Figure 1.6 shows the framework for search-based program transformation.
The aim of this thesis is to investigate the possibility of implementing and analysing the way in which different search algorithms may be used to perform program transformation. This is geared towards ultimately constructing a general purpose transformation system based on search methods, where the majority of human effort is performed working out how to compute the fitness function for the specific goal.

The aim encompasses the following general objectives:

1. To re-formulate the program transformation problem as a search based problem.

2. To identify an appropriate representation scheme for encoding individual solution for manipulation by search-based techniques.
3. To identify the problems caused when applying search-based techniques to program transformations.

4. To compare the effects of searching for transformation sequences for different transformation goals. Comparing search heuristics driven by different fitness functions, which maybe single objective or multi-objective.

5. To implement, compare and analyse different search methods to transform procedural programs written in WSL using transformation rules from the FermaT transformation workbench.

1.4 Contributions of this Thesis

The contributions of this thesis are as follows:

1. The proposal of the search-based approach to carrying out program transformations.

2. Experimentation for the computation of amorphous slices via dependence analysis into a redundancy removal problem which can be tackled using search-based approaches

3. Demonstration that search techniques generate valid amorphous slices.

4. Applying search-based transformation to minimize the proximity of same name variable identifiers in source code.

As well as the above, the approach presented in this thesis can be extended to the state-of-art software engineering technique of refactoring. It would possible to
completely automate code refactorings using search–based methods depending on an appropriate metric for evaluating how good the effects of performing a particular refactoring operation is.

1.5 Organisation of this Thesis

The rest of the thesis is organized as follows:

- Chapter 2, Literature Survey briefly surveys the previous work on program transformation, focusing on the techniques used for formulating transformation strategies. It also surveys the area of metaheuristic search examining two general classes of search algorithms and concludes with a brief survey on program slicing laying particular emphasis on amorphous program slicing, which combines elements of program slicing with program transformation.

- Chapter 3, Transformation Problem, introduces the transformation problem describing the challenge of finding good general purpose transformation algorithms and introducing the expression of the transformation problem as a search–based issue where appropriate heuristic algorithms may be applied.

Three search algorithms; Local, Global and Greedy algorithms are described in some detail.

The chapter concludes with the description of a code compaction experiment, to minimize the number of lines of code of a source program and compares the results of a Hill Climb Algorithm, a Genetic Algorithm and a Random Search Algorithm. For these experiments, the SB-approach should outperform random search.
• In Chapter 4, **SB-Amorphous Slicing**, the search-based transformation idea is extended with a different transformation goal. The Chapter introduces the problem of trying to compute valid amorphous program slices with respect to a slicing criterion using existing transformation rules in our system and our search-based approach. It reformulates the problem of computing amorphous slices as a redundancy removal problem and applies our search approach to find transformation sequences that weed out redundant statements. The chapter concludes with an experiment that uses search techniques to compute amorphous slices and compares the results obtained by a Hill Climb Algorithm, a Genetic Algorithm, a Greedy Algorithm and Random Search. The results obtained are contrasted with those from LinIAS - a benchmark non search based amorphous slicing tool.

• Chapter 5, **Variable proximity based Program Comprehension** introduces another application for search-based program transformation. We describe how the proximity of similar variable names in a program helps in the comprehension of a given piece of code. The short-term memory of a human programmer may be very transient and therefore the ability to remember the use and purpose for a particular variable would be enhanced if references to that variable were brought closer together. We describe this as a transformation problem and carry out experiments to find sets and sequences of transformations that minimize these distances between variables with similar names. The chapter concludes with a discussion of the results from the experiment performed.

• Chapter 6, **Reflections**, describes the lessons learnt while carrying out this
work and presents the author's reflections and experiences. In this chapter, we also analyse the potential threats to the validity of the results obtained throughout this research.

- Chapter 7, **Conclusions and Future Work** concludes this Thesis. Here, we summarize the outcome of and answer to the research questions that this work set out to investigate. We summarise all the results from the different experiments and suggest areas of direction for future research.
Chapter 2

Literature Survey

2.1 Introduction

This chapter reviews work in the field of program transformation - the different forms of transformations and the application of transformation sequences. It also reviews the literature on metaheuristic search approaches for generating transformation sequences and the use of a metaheuristic approach to similar software engineering problems.

2.2 Transformation

Program Transformation can be described in a variety of ways. In transformation, a program is re-moulded to satisfy a particular objective or a group of objectives. The kinds of transformations that are applied differ, based on the languages to which they
are being applied.

Transformations can be source-source or source-object and inter- or intra-language. Program Transformation has been used as a goal in itself to restructure programs into ‘better’ versions and also as a means to an end, where the transformed program becomes only an object required for some kind of analysis which may then be discarded after such analysis is completed.

The overall aim is that the target (output of the transformation process) program observes some property that makes it ‘better’ or more suitable than its original version. Most of the research on source program transformation has historically been applied to programs written in functional languages with the classic examples being research performed in the 1970s by Burstall and Darlington [33, 34] and later by Partsch [94]. There has, however, been some work on imperative languages, while more recently, there have been studies on transforming object-oriented systems [3].

The objectives for a particular transformation system naturally depend on the developers of the system and may be quite low-level such as code optimisations [11, 71], side-effect removal [59], unreachable blocks of codes [18], etc., or they may involve higher-level objectives, such as the need to preserve software quality design principles: modularity [103], coupling and cohesion [104], maintainability [13, 100] and testability [56, 58], etc.

2.2.1 Source–Source Code Transformation

Source–source transformations refer to transformations where both the source and target programs are high-level programming level descriptions.
2.2.2 Source–Object Code Transformation

Source–Object\(^1\) code transformations involve the conversion of a high-level program language source description into machine code executable instructions. Examples of source-object transformation systems are programming language compilers.

2.2.3 Inter–Language Source Transformations

Inter-Language\(^2\) source transformation also sometimes called program translation \([1]\) refers to the transformation from a given high-level language description into another. An example is the transformation of a source program written in Java into C.

2.2.4 Survey of Transformation Systems

Many transformation systems exist that are often specialized for a specific object language and/or kind of transformation. Some are industrial-strength transformation systems, capable of transforming large-scale systems, while others are still relatively small-scale academic inventions for research purposes. This review focuses on the transformation strategies, that is, the control part of transformation systems that determine the order of application of basic transformation steps employed by these transformation systems. In some articles \([8, 68]\), transformation systems are referred to as rewrite systems and transformation rules as rewrite rules.

Many of these systems can be classified based on the kind of program structures they manipulate. These may be terms, trees or graphs.

\(^1\)Source-Object transformations are not included in the scope of this thesis.
\(^2\)Inter-Language transformations are not included in the scope of this thesis.
**Term Rewriting**

Term rewriting systems consist of sequences of discrete transformation steps where one term is replaced by another. The objects in the rules are pattern-matched with the terms in the source and replaced when appropriate.

Examples of transformation systems based on term-rewriting are ELAN [20], ASFandSDF [106] and Stratego [107].

Stratego is a transformation system for performing source-source transformations. The system uses term rewriting rules performed on the Abstract Syntax Tree (AST) to effect the changes. In the implementation of Stratego, there are varying transformation strategies used in deciding which rewrite rules may be applied and when they may be applied. These strategies usually correlate to some method of traversing the AST of the source program. For example, it uses the **innermost strategy** [108] and exhaustively applies each rule down the tree. There is also a bottom-up tree traversal strategy. The application of rules through any method of AST traversal is not iterative.

ELAN is a language environment for specifying and prototyping deduction systems in a language based on rules controlled by strategies. The application of these rules is based on term-rewriting. Several rules may be applied to any one term. By contrast with other transformation systems, the strategies in ELAN are not hardwired into the design of the system. Users can introduce new strategy operators by using strategy construction functions. The ELAN transformation system comes with a set of rules and strategies for selecting which rules can be applied to specific terms and which ought to be ignored.
**Tree Rewriting**

In *Tree Rewriting* systems, the transformations are applied directly to tree structures, such as the abstract syntax tree representation of programs.

The TXL [36] transformation system is a general purpose source transformation system used to support rule-based source to source transformations for a wide range of software maintenance and re-engineering tasks. Source text structures are described using a BNF-type grammar. It uses a pattern-matching technique, that binds names of items in the rules to actual name patterns and replaces it with an instance of appropriate replacements. Rules are applied using a tree search of the input parse tree for instances of the pattern parse tree iteratively until no further patterns can be matched. The transformation strategies in TXL are user programmable. A parse tree is represented as a nested list representation and is also used to represent TXL grammars, rules, patterns, etc.

FermaT [114] is an industrial strength program transformation system based on the WSL language and is aimed at reverse engineering, program comprehension and migration between programming languages. It is based on tree rewriting. The output of a parsed program results in an AST, to which the FermaT transformations are applied. The system is based on the wide spectrum language, which is an intermediate language representation that makes it possible to write source transformations. It uses formally proven program transformation rules, which preserve or refine the semantics of a program while changing its form. It also permits the sequencing of rules to achieve greater transformation power.
Another related system is the OPTIMIX [8, 9] system, designed to facilitate the construction of program analysers and transformers in a unified manner. The system is based on graph rewriting and uses two new classes of graph rewrite systems: Edge Addition Rewrite System and the Stratified Graph Rewriting System. OPTIMIX treats all program objects as graphs and performs program analysis and program transformation using graph rewriting. As with other systems mentioned, the transformations are described as a sequence of these graph rewrite rules. The system defines a Strata, which consists of a range declaration and several graph rewrite rules.

Other related transformation systems include: Tampr [21, 62] (Transformation Assisted Multiple Program Realisation System) is one of the earlier transformation systems and has been available since the early seventies. The DMS [3] re-engineering toolkit is a set of tools for automating customised program analysis, modification or translating of software systems. It is an industrial strength transformation engine that has been used for very large scale, dynamic program analysis across multiple source files and languages.

We refer the reader who is interested in a more exhaustive and indepth analysis of program transformation tools to the program transformation online resource [1].

2.2.5 Goal-oriented Transformation Strategy

Tahvildari and Kontogiannis [104] suggest a method for selecting source-code improving transformations that map directly to non-functional requirement goals. Their approach involves constructing a Soft-Goal Interdependency Graph (SIG) which is a
graph that relates source-code improvement transformations to actual design objectives such as Low Data Coupling, High Cohesion, High Modularity, Low I/O complexity, etc., allocating a probability score for each transformation rule's ability to satisfy the desired metric. They create a probability matrix based on the probability scores for each rule and calculate the optimal path using the Viterbi Algorithm for finding probable states.

### 2.2.6 Transformation Strategies

Program transformation usually requires the syntax tree of a program as a starting point for the transformations that need to be carried out. A transformation rule can be referred to as an operation carried out at a node in the syntax tree that preserves the semantic behaviour of the program. A problem, however, is the traversal of the tree: only some nodes in the tree can be transformed while the tree is being traversed and the challenge is to locate these nodes and apply the transformations.

The goal of a traversal is achieved by visiting each node in a certain visiting order and then applying an appropriate transformation rule to each node.

Among the known traversals are the top-down, bottom-up, left-to-right and right-to-left traversals. For binary trees, there is another method of visiting each node, in-order.

In the top-down traversal strategy, the visiting order is root, left sub-tree, right sub-tree and this continues systematically down the tree. The bottom-up strategy involves visiting the sub-trees first and then the root node.
2.2.7 Search for Transformation Sequences

Cooper et al. [29, 30] introduce the use of evolutionary algorithms for performing compiler optimisations, motivated by the need to generate reduced code size for embedded systems and present results to show that these algorithms provide positive results when compared to a manually pre-determined fixed order of application. The basic compiler optimization structure is one which encodes the optimisations as a fixed sequence of passes. Cooper et al. identify certain features in compilers that may need to be optimised such as speed, space, power and page faults. The ultimate goal of their work was to change the economics of producing high quality compilers by automating much of the work needed to tune a compiler for new circumstances - building retargetable compilers. A significant problem, however, is the complexity presented when searching for ‘best’ sequences of transformations. This is the result of the interplay between transformation rules where one transformation in a sequence creates or hinders an opportunity for subsequent transformation along the sequence. This generally means that the search for best sequences is tailored to specific compiler systems and cannot be generalized for any class of collective systems.

Their approach replaces fixed-order optimizations with a pool of transformations, a “steering algorithm” and an explicit objective function. The steering algorithm is a form of search algorithm that, depending upon the objective function, decides on a sequence or ordering for the transformations. The algorithm introduced by Cooper et al., referred to as a biased random sampling algorithm: essentially a population based search algorithm such as a genetic algorithm, in which, it traverses the search space initially with fairness and randomness since each solution is potentially as good as the other. However, as the search progress, the algorithm builds a history of which are the
better solutions and this affects which regions of the space are sampled subsequently.

Cooper et al. present results on comparing different search strategies. They compare a genetic algorithm and an unbiased random search and find that the GA converges more quickly toward the best solution than random sampling. The experiments were carried out on different objective functions, which is one of the great strengths of this approach, since it allows the search techniques to be applied to essentially different uses. The experiments were carried out to optimise code size, running time and inter-operation name transitions. They found that search-based compilation produces code with a 13% reduction in code size and up to 20% increase in speed than that produced by the compiler’s original compilation sequence. In each case, the search algorithms found sequences that produced better solutions than the original, default fixed-sequence compiler.

However, their work concentrates on what we term holistic transformations. That is, transformation rules that are applied broadly to all statements across a program from top to bottom. Each compiler optimization is applied to the entire test program. In our research, the scope of the transformation rules we examine are much narrower. They are the low-level transformation rules that apply to statement pairs, rather than entire blocks of code. Examples of these rules were shown in Chapter 1. The difference is that the search strategy needs to be much cleverer in identifying regions where potentially good solutions may be present.

Williams, in his PhD thesis [120], describes the application of meta-heuristic and evolutionary search methods to program transformation, in this case applying it to aid automatic parallelisation. He describes several loop changing transformations for use in the REVOLVER system [120] and suggests two different gene representation
techniques within the evolutionary strategies: the Gene-Transformation (GT) representation in which a single gene represents a single optimising transformation to be applied and a Gene-Statement (GS) representation where each gene represents a statement in the program and an entire chromosome represents a sequence of statements. The GT techniques however define absolute addresses for the points of application for the loop transformation, presuming some prior knowledge about the source-code or requiring static analysis of the code to determine exact locations of loops within the code body.

Nisbet [88] focused on using genetic algorithms to find program restructuring transformations for FORTRAN programs to execute on parallel architectures. He introduces the GAPS framework, which uses GA optimisation to determine restructuring transformations. Similarly to the experiments performed by Cooper et al., the objective function for his experiment relates to running time, where the goal of the optimization is to minimize the execution time for a given program. Again, this work compares a genetic algorithm with other more conventional compiler search algorithms. The GAPS framework [88] works on a reduced set of transformation rules, such as loop interchange, loop fusion, loop skewing, etc., part of the goal being to parallelise loop nests. In contrast to simple GAs, which have been described as weak search techniques because they do not include any domain specific knowledge into the search, the Nisbet framework creates hybrid GAs which employ problem-specific knowledge to generate the initial population of solutions.

The results of the experiments conducted by Nisbet show that the GA based approach was slightly better when compared with other techniques. This may be perhaps due to the search based approach not being appropriate or perhaps a better
encoding strategy for solutions would legalize a greater percentage of solutions. Currently, legal solutions (sequence of transformations) account for only 5.5% of solutions generated. This perhaps is due to solution degradation as a result of the application of GA operators such as crossover and mutation. However, in spite of the reduced numbers of legal solutions, the GA still produced notable performance improvements in parallel execution times.

Whilst Nisbet concentrated largely on the sole objective of minimizing execution times for parallel architectures by reconstructing, re-ordering loops, Wolf et al. [121] looked at combining loop transformations for modern micro-processor multi-objective requirements. Their premise is based on the fact that loop transformations such as fission, fusion, tilling, interchanging and so on are known to improve aspects of compiler performance such as cache behavior, instruction scheduling, register allocation, etc. However, these objectives do not occur in isolation, but are rather highly interdependent upon each other. For example, transforming a particular loop may enhance register allocation but hinder cache behavior profusely.

Their work presents a model that estimates total machine cycle time considering cache misses, software pipelining, register pressure and loop overhead. They present an algorithm that searches through various possible transformations, based upon the defined model to select a set of transformations that leads to the best overall performance. Their results show that while their algorithm is expensive due to a large search space, it provides up to 50% improvements in performance within reasonable compile time.

The work described in this thesis differs from the previous examples of searches for
transformation sequences in the literature by focusing on small pair-wise transformation steps. These are rules that transform small portions of code (typically 2 to 3 lines) at a time. Also, this study investigates transformations performed at source-code level. It seeks to implement a generalised program transformation system, which is capable of evolving effective transformation sequences based on meta-heuristic search algorithms. Cooper et al. carry out their investigations using ‘global’ transformation rules. These rules carry out transformations of the entire program at once and their research focused on compiled code. Other researches into search–based source code program transformation have been based on rather specific transformation goals, such as Williams’ transformation of loop structures for parallelisation.

2.3 Metaheuristic Search

Metaheuristic search techniques are high-level frameworks that use heuristics to find solutions to problems without the need to perform full exhaustive enumeration of a search space. These are therefore suited to finding good solutions to combinatorial problems at a reasonable computational cost. These frameworks may be adapted to specific problems. Examples of metaheuristic search methods include:

- Genetic and Evolutionary Algorithms [47, 65]
- Hill Climbing Algorithm [78]
- Simulated Annealing [70]
- Tabu Search [45, 46]
Most of these techniques are classified on the basis of the ability of find either locally good solutions (local optimum) or globally good solutions (global optimum). In subsequent subsections, we will describe what global and local optimums are. The rest of this section provides an overview of each of these techniques to make this thesis self contained.

2.3.1 Global Search Techniques

Global Search techniques have been used to solve a myriad of extremely difficult problems where good approximate solutions would suffice. A global solution is the best possible solution to a problem. Some examples of global search strategies include the GA and Simulated Annealing.

A *Genetic Algorithm* is a population based search procedure which starts with an initial random population and evolves over a series of populations, such that the overall average fitness of the population in successive generations is better or at least no worse than those in preceding generations. An optimised individual is one which presents a more desirable solution to the given problem.

GAs imitate the natural process of evolution and use the evolutionary operators of crossover and mutation to alter the population across several generations towards optimality. Figure 2.1, graphically illustrates the different stages of the GA process. In the first stage, a population of individual solutions is randomly created as possible solutions to a given problem. Note that some experiments such as Nisbet’s [88] seed the initial GA population with outputs from other prior experiments or the results of technical domain-specific knowledge by experts. These kinds of GAs are generally referred to as *hybrids*. Of the individuals contained in this initial population, some of
them are selected to mate based upon a selection strategy. In the GA literature, there are different selection schemes available like tournament selection and the roulette wheel selection. Mating of two individuals involves the exchange of genetic material between them and is often referred to as crossover.

Crossover facilitates the exchange of information between two chromosomes by swapping components of each. The exchange of components may be carried out differently depending on what points within the chromosome are selected to be exchanged. Crossover strategies includes: Single Point crossover, Two-point crossover and Uniform crossover. Single point crossover works by dividing each chromosome at a selected position and swapping adjacent sides, creating two new offspring chromosomes. Two-point crossover involves selecting two points in each parent individual and swapping them between both parents. In Uniform crossover, offspring chromosomes are created by selecting alternate genes from each parent.
Sometimes, during crossover, potentially good subsequences within the overall sequence may be destroyed if they are chopped in the middle. Therefore, different kinds of crossover approaches have their merits. Uniform crossover for instance, is clearly unsuitable for application where subsequences of chromosomes need to be preserved.

Crossover rate is the probability of crossover execution. Figure 2.2, shows the effect of crossing over materials from two selected individuals referred to as parents to form two new offsprings.

![Illustration of single point crossover](image)

<table>
<thead>
<tr>
<th>crossover point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent 1: 2 14 2 1 22 16 18 12 21 15 9 22</td>
</tr>
<tr>
<td>Parent 2 13 13 7 10 18 7 5 12 8 11 20 15</td>
</tr>
<tr>
<td>Child 1 2 14 2 1 18 7 5 12 8 11 20 15</td>
</tr>
<tr>
<td>Child 2 13 13 7 10 22 16 18 12 21 15 9 22</td>
</tr>
</tbody>
</table>

Figure 2.2: Illustration of single point crossover.

One general problem with genetic algorithms is premature convergence. This happens when the population quickly becomes homogenous. That is, one good individual quickly becomes ubiquitous and dominant. GAs attempt as much as possible to lengthen the time before this happens as it invites the chance that better solutions may be found through designed mechanisms and operators such as crossover. One
operator geared towards maintaining diversity in the population is mutation. GA mutation allocates a chance for a gene in the representation to be mutated. Random Mutation simply means that a selected gene is mutated at random to form another gene. The probability that a given gene with a chromosome (sequence) will be mutated is known as the mutation rate.

GAs are referred to as global search techniques. They employ a population of potential solutions, which reduces the chance of the search getting stuck in local optima because they sample a number of individuals from the population simultaneously. In a GA process, success therefore may depend upon the population size, crossover strategy, crossover rate, mutation rate and encoding system.

Simulated Annealing originated from the physical process of annealing, a process of cooling a material in a bath so that a minimal energy is reached. Simulated annealing can be considered as a local search that provides escapes to avoid getting stuck in local optima. In the overall procedure, one may accept an inferior solution to the current one in the hope that it leads to a global solution. In essence, it may get worse to get better. This contrasts with the GA because the GA samples a whole range of solutions while SA only looks at one individual at a time. The acceptance of inferior solutions is based on the idea of a temperature, which can be altered from high to low. At high temperatures, any inferior solution is accepted while at low temperatures, the probability of accepting an inferior solution is lower. Therefore, at very low temperatures, there is little chance that an inferior solution would be accepted. At a temperature of zero degrees, no inferior solutions are accepted, hence the search becomes a regular gradient ascent (similar to the hill climbing algorithm described in the example in Chapter 1). Simulated annealing processes are dependent upon the
initial temperature, cooling mechanism and encoding structures. For instance, if the initial temperature is set too low, then the search continues within the same local area and can get stuck.

### 2.3.2 Local Search Techniques

Local search procedures involve walking through a rather narrow space within a much larger one. The solution found therefore, may be the best within the area explored but by no means guaranteed to be the best overall. Normally, in a local search method such as *Hill-Climbing* (HC), one guesses an initial random solution within the solution space and then moves towards a better solution closer to the goal. The idea is to start with a sub-optimal solution to a problem and then repeatedly improve the solution until some condition is maximised.

The algorithm iterates through each neighbour to the current position and terminates when no better neighbouring solution can be found. It is referred to as a hill because the search terrain looks like one and the whole process of finding the optimal solution is analogous to ‘climbing a hill’. Each neighbour that presents a better solution than the current one represents a climb up the hill until the search arrives at the peak.

There are primarily two kinds of climbing strategies in a Hill Climbing Algorithm. The steep ascent climbing strategy evaluates all neighbours at the same time, and the neighbouring candidate which offers the best improvement to the current solution is selected as the new current solution. The other climbing strategy referred to as first ascent requires only the evaluation of one neighbour at a time, with the first best neighbouring solution found selected as the new current solution. This technique is
similar to simulated annealing with the initial temperature set to zero.

Hill Climbing is comparatively simple to implement, produces fast results and has been shown to perform better than more sophisticated search techniques such as Genetic Algorithms for certain classes of problems [80]. It is highly effective for search landscapes with a smooth curve and a minimal number of peaks and troughs. Such landscapes naturally allow the search to gradually progress toward optimality. It is also suitable for landscapes which may have lots of peaks and troughs of equal height. In such landscapes, it would not matter where the initial solution was selected from and consequently which hill was being climbed.

Rather notoriously, local search approaches have the inherent problem of getting stuck in local minima from which point the search becomes hindered, returning sub-optimal solutions [78]. This normally occurs in more ‘jagged’ search landscapes with large numbers of spikes where an unlucky random start position results in the search being guided toward a sub-optimal place. One solution to avoid this problem is to carry out random restarts of the hill climbing algorithm to diversify the points where the search is started, so that the search may stake out a previously unsearched region in the search space.

Hill Climb algorithms are unsuitable for search landscapes with long plateaux. The search procedures become stuck when all neighbouring candidates have identical fitness values and thus can not be deemed to offer more improvement than the current solution. Figure 2.3 shows all three landscapes: a smooth landscape one where a hill climbing search might be appropriate, a spiky landscape and a plateaux.

Another example of a local search procedure is tabu search [24, 45]. In tabu search also known as prohibition-based search, the idea is to keep a list of previously
visited locations in the search landscape and to prevent the search from revisiting such locations hence the term ‘taboo’. The presence of this memory list is an essential feature of tabu search.

The search proceeds as normal until it reaches a local optima. In order to escape from this local optima, a worse solution may be accepted if it has not been previously selected on the way to finding the current best solution.

2.3.3 Greedy Algorithms

In a greedy search algorithm, the best option is always the preferred option hence the term ‘greedy’. The algorithm selects any potentially good opportunity as soon as it can, and subsequently continues in this manner until a solution is reached. The choices regarding which solution is selected always depend on the current situation.
2.3.4 Random Search

Random search simply searches the search space for random transformation rules to form a sequence. A random collection of points along the search space is collected to form an individual solution. A test program is executed with the solution returned by the search and the effectiveness of the solution is measured. Subsequent solutions are also generated randomly and independent of previous solutions. The best random selection so far is noted while further searches take place. Obviously, this is a highly unintelligent search approach, requiring no knowledge of previous regions of the search landscape that may have been searched either with the potential for success or propensity for failure. However, certain types of problems and landscapes still present random search techniques with a fair chance of outperforming heuristic approaches. For instance, it is suggested that simple hill climb might perform rather badly with very spiky landscapes, but given a reasonable chance, randomly searching for input sequences may provide better results or at least present a better chance of finding good solutions.

Research regarding the application of search-based heuristic techniques for solving problems such as optimizations employ random search as a baseline technique. This is a rather crude measure of assessing the quality of supposedly more intelligent approaches. It is expected that for these heuristic methods to be considered worthy, they should perform significantly and at least be no worse than random search. This is the primary reason for our interest in random search.
2.3.5 Transforming Landscapes

It has been shown that the success of local search techniques depends largely on the modality of the landscape through which the algorithm traverses [80]. For example, a landscape with lots of spikes naturally reduces the chances of finding a globally optimal solution using a hill climb algorithm. In order to get around the problem of search techniques getting stuck in a cul-de-sac, it might be necessary to transform the landscape from a spiky terrain into a smoother, search-friendly terrain.

Mathias and Whitley [80] provide empirical results that suggest that local optimisation methods perform comparably with genetic algorithms when it is possible to transform the search space. Their findings suggest that with the use of gray coding, local optimisation techniques performed better than genetic algorithms on test functions that seem inherently better suited to GA-based solutions. They found in some cases, that gray coding results in local search techniques optimising faster than the genetic algorithm.

McMinn et al. [83], show that it is possible to transform the search landscape for a problem by applying source transformations. Their approach applied to automatic test data search, perform source transformations to remove predicate nesting. The problem they addressed was one where in the search for test data for executing a conditional branch of a nested IF predicate, the execution depends upon the successful execution of prior conditional predicates. They applied source transformations to alter these multi-layered nested predicates into a single flat level predicate for the execution of the desired branch or block of code. Effectively, the result was the transformation of the search landscape from one which had a large plateaux area with a sudden deep trough into a landscape with gradually undulating leaves which neatly converge at a
mid-point. The landscape clearly provides for a more efficient search.

![Figure 2.4: Transformation of a search landscape. Image taken from [83]](image)

Source transformations have also been applied to the removal of other search hindering program features such as the presence of spikes caused by flag variables [58], altering the search landscape from one with small spikes of high fitness to a much smoother one with a larger basin of attraction toward the global optimum, thus enhancing the search. Figure 2.4 illustrates the transformation of a predominantly flat landscape to a more search-friendly landscape.
2.4 Some Related Applications of Heuristic Search to SE Problems

Clarke et al. [27] make the case for the application of meta-heuristic search techniques in solving software engineering problems. Hitherto they have typically been applied to other more traditional engineering problems such as aircraft wing control systems [90, 93], to bioinformatics problems such as classifying gene expression data [4] and financial stock prediction [74]. However, more recently, there has been some evidence that they can be applied to processes at any stage within the software engineering lifecycle such as project management [5], project cost estimation [19], project planning [7], resource allocation [6] and testing [82]. O’Keeffe and Cinnéide [91] apply heuristic search to a maintenance activity. They argue that the cost of maintenance can be reduced by refactoring object-oriented programs to improve their understandability, adaptability and understanding. These goals are similar to those earlier introduced by Tahvildari and Kontogiannis [104]. The refactoring process is automated through a search approach which is guided by a quality evaluation function. More recently, Seng et al. [100] apply search to recondition the class structure for object-oriented systems. Viewed as a maintenance task, they state that the degradation of a system over time, referred to as system decay, can be improved. Their search approach helps the designer in choosing suitable class refactorings to be applied to the class structure.

2.4.1 Project Planning

Antoniol et al. [7] apply search-based techniques to optimize project planning for a maintenance project. Their research involves finding an optimal or near optimal
order for allocating work packages to programming teams while minimizing project duration. Their findings show that with an appropriate representation scheme, search based techniques, such as, a genetic algorithm and a hill climbing approach provide good results.

2.5 Program Slicing and Amorphous Program Slicing

Program slicing is a technique for extracting parts of a program that affect a chosen set of variables of interest. By focusing on the computation of only a few variables, the slicing process can be used to eliminate parts of the program which cannot affect these variables, thereby reducing the size of the program. The reduced program is called a slice. The traditional, syntax–preserving paradigm of computing slices by a process of statement deletion is well studied, and there are well established algorithms for automated syntax–preserving slice construction [66, 119].

Amorphous Slicing

Amorphous slicing is a form of slicing, in which any transformation can be applied (not merely statement deletion). As a result the slice produced can be a lot smaller, but the algorithms for computing such slices are less well developed than those that use the statement deletion transformation alone. Furthermore, unlike traditional, syntax–preserving slices, the amorphous slices do not preserve the syntactic structure of the original program. All applications of slicing require slices to be as small as
possible and many (such as reuse [63], testing [15] and reverse engineering [40, 55]) do not require syntax-preservation. For those applications where syntax preservation is unimportant, amorphous slicing is clearly attractive.

Slicing was introduced by Mark Weiser in 1979 [118] and has been the subject of extensive study since then. Studies [22, 66, 72, 92] carried out from then through to the early 90s employed statement deletion as the sole simplifying transformation used to create slices.

Several authors have indicated that the syntactic subset requirement of syntax-preserving slicing has been a hindrance to the computation of small slices [26]. Indeed, in his thesis, Weiser immediately recognized and acknowledged ([118], page 6) that it would not always be possible for a slice to be constructed as a purely faithful subset of the original program’s syntax.

Many other authors have suggested ways of combining slicing and transformation for a variety of applications including refining the precision of syntax-preserving slicing[105], assisting testing [52], identifying unobservable components in optimising task scheduling [43], register-allocation optimisation [89], partial evaluation [35], restructuring Cobol [42], parallelization [75] and model checking [31].

Amorphous slicing was first introduced by Harman and Danicic [53], and has been developed by Binkley [16] and Harman, Binkley and Danicic [51] and by Ward [115]. Binkley’s approach uses the System Dependence Graph [66], while Ward’s approach uses a novel syntax-preserving slicing algorithm, which is currently under development into an augmented system for producing semantic slices [115], which are closely related to amorphous slices. Binkley et al. [17, 51] have shown that amorphous slicing aids program comprehension. Hierons, Harman and Danicic [64]
have shown how amorphous slices can be used to (partly) ameliorate the equivalent mutant problem for mutation testing.

### 2.6 Program Comprehension

Program Comprehension refers to the ability to understand what a piece of software intendeds to accomplish. It is the primary requirement for any kind of human analysis of a software system. Program Comprehension has been shown to impact on other software engineering disciplines such as Software Maintenance [110], Reverse Engineering [87] and Program Transformation [54, 117]. The Reverse Engineering process is crucial in the understanding of source programs. It is reported that up to 50% of a software maintainer’s time can be spent determining the intent of source code [87].

There are two main approaches to understanding programs: bottom-up and top-down. The bottom-up process begins with understanding the source code and generating an abstract conceptual description of the system. Reverse Engineering involves generating these conceptual artefacts of the system such as design documents from the source code. Chikofsky and Cross [25] define reverse engineering as:

“The process of analysing a subject system to identify the systems components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.”

The top-down model involves understanding the system at that higher level of abstraction, creating cognitive mappings between the application domain and the program.
2.6.1 Cognitive Models

Several theoretical foundations have been proposed to explain how human beings understand computer programs. For example, in certain instances programmers build a mental representation of a system’s control and data flow from the bottom up. All comprehension theories agree that the programmer uses existing knowledge coupled with a comprehension strategy to achieve new knowledge.

2.6.2 Top-Down Approach

The top-down approach defines a process where one begins to understand the program by first gaining an understanding of the overall goals or purpose of the system. Components are then evaluated on the basis of how they relate with those stated goals. Early pioneers of the top-down model for program comprehension also referred to as the Domain Model were Soloway and Ehrlich [102]. This description of the top-down approach also fits in with Gilmore’s model [44], where there is some understanding of the design decisions of the system and this is later reconciled with the actual performance of the system. The central idea is the formation of an initial hypothesis, which is later upheld or debunked after other parts of the system have been analysed.

2.6.3 Bottom-Up Approach

The bottom-up approach requires the program reader to begin with the detail. The basic idea is reading and understanding simple, low-level fragments. The fragments are composed into large aggregates whose purpose is constructed from its composite
parts. This process is then repeated until the full program function is discovered.

2.6.4 Hybrid Approach

Von Mayrhauser and Vans [109, 111] suggest that programmers actually use a combination of the top-down and bottom-up approaches to understanding a system depending on the skill and experience level. For example, an experienced developer of a stock prediction application written in Java, encountering a similar system written in C++, might employ the top-down model. By understanding the constructs and fundamentals of a stock predicting system, the developer immediately builds a high-level model of the system. The model would need to be validated at a lower-level and this is done on closer inspection of the actual program constructs.

2.7 Summary

This chapter provided an overview of a number of themes that are central to this dissertation. Firstly, we described elements of program transformation and different classes of program transformation. We also mentioned some industrial and academic program transformation tools currently in use. Importantly and most relevant to this work, the section on transformation described current applications of using heuristic techniques to search for sequences of transformation rules and the transformation of search-landscapes via different evaluating functions.

Secondly, we examined heuristic search techniques. The techniques fall into two categories: global techniques and local techniques. These techniques tend to be useful for solving problems that have a fairly large search space.
Thirdly, we reviewed the literature on program slicing and amorphous slicing. In Chapter 4, we show how heuristic search can be used to compute amorphous slices.

Lastly, this chapter concluded with a summary of comprehension and understandability of computer programs and the various approaches undertaken for program comprehension.

In the next chapter, we describe a re-engineering experiment, we compare the performance of three algorithms in performing lines of code optimisation on source code.
Chapter 3

The Transformation Problem

3.1 Introduction

The previous chapter described the different forms of transformations, some of the strategies employed in selecting the transformation rules to be executed and the sequencing of their execution. It also examined two broad classes of metaheuristic search techniques and some application areas where they have been successfully implemented.

In this chapter, we describe the challenge which search-based transformation seeks to address. We describe informally how the heuristics introduced may be extended as a solution to the transformation problem. Section 3.2.2 describes the choice of representation scheme selected for a solution in search-based heuristics. In section 3.7.4, we provide the details of the fitness function. The chapter concludes by describing an experiment designed to show the feasibility of the approach for carrying out general
purpose program transformations.

### 3.2 Challenge

An overall program transformation from one program $p$ to an improved version $p'$ typically consists of many smaller transformation tactics [12, 14]. Each tactic consists of the application of a set of transformation rules. A transformation rule is an atomic transformation capable of performing simple alterations like those captured in the examples $T_1 \ldots T_6$ from Chapter 1. At each stage in the application of these simple rules, there are many points in the program at which a chosen transformation rule could be applied.

There are many points in a program; typically one per node of the Control Flow Graph. The set of pairs of possible transformation rules and their corresponding application point is therefore large. Furthermore, to achieve an effective overall program transformation tactic, many rules may need to be applied, and each will have to be applied in the correct order to achieve the desired result.

This explosion in the possible choices of transformation rules and their appropriate sequencing has led to problems with the generalisation of program transformation. While specialised transformation algorithms exist for dedicated tasks, there are no general purpose transformation algorithms. This is the problem that search–based transformation seeks to address.
3.2.1 The Search–Based Solution

Viewed as a search problem, the space of possible program transformation sequences is ideal for exploration using search–based techniques. The idea would be to use a code-level software metric [41, 101] as a fitness function to guide the selection of good program transformation sequences for a given program. Subsequently the approach could be extended to apply to more sophisticated forms of transformation optimisations, such as the transformation for higher-level maintenance objectives proposed by Tahvildari and Kontogiannis [104] and also by Zou [123].

We show how traditional software metrics associated with the measurement of properties of code can be viewed as fitness functions, used to determine the behaviour of meta-heuristic search algorithms. In this context, the goal of these algorithms is to optimize the subject program with respect to a particular metric. The method is at a sufficiently high level of abstraction that it allows the user to explore the meaning of families of metrics through the lens of code optimized to obtain high metric values (high fitness).

A byproduct of such an investigation is an insight into the nature of the transformation strategies required to re-write programs in a way that improves their value under some metric of interest.

3.2.2 Gene-Transformation Representation

As mentioned earlier, the success of any search-based method depends largely on the representation scheme chosen to describe a solution to the problem.

We considered two methods of representing sequences of transformation rules and the points where they may be applied in the source program.
Relative GT-Representation

The Relative Gene-Transform representation where each gene in a sequence represents a single transformation that could be applied. This means the selection of a correct position to which an optimising rule may be applied is via the inclusion of enabling transformation rules that do not, in themselves, perform any optimisation but facilitate the more potent rules.

Absolute GT-Representation

In the Absolute Gene-Transform representation, each gene consists of a tuple of transformation rule and an absolute address where the transformation rule should be applied. This method of representation removes the need for such enabling transformation rules as described in Section 3.2.2. An example of an absolute GT-representation is shown below:

\[
[(T1,2),(T2,4),(T3,5),(T4,6),(T4,8),(T3,9),(T3,10),(T1,12),(T4,14)]
\]

3.3 Local Search-Based Transformation

Meta-heuristic search algorithms such as hill-climbing may be applied to arrive at an optimum result, or at least, a locally optimal result. Rather than apply the transformations manually, one after the other, we allow the algorithm to pick the best transformations to apply from a given set.

We assume a search space containing all the possible allowable sequences of transformation rules and define our fitness function using an existing software metric [41, 101], for example the size of the program in LoC.
An optimum solution would be a sequence of transformations that results in an equivalent program with the fewest possible number of statements. For instance, in the examples used in Chapter 1, transformation rule $T_3$ clearly shows a reduction in the size of the program from 4 nodes to 2 nodes and so would be selected as one which returned a better program, than, for example, the identity transformation.

Using a simple hill-climbing search algorithm and a size-based metric such as LoC, after a particular transformation sequence is applied, the size of the new program is compared with that of the previous program. A sequence which reduces size is retained and the search continues from the new smaller program found. When no smaller program is found by the application of a rule, the search terminates.

This approach is essentially a local search using hill-climbing; as the program is mutated (using transformation) and better transformation sequences are retained. The approach is similar to that adopted by Williams [120], who considered transformation for auto-parallelisation. In Williams’ case, the metric was execution speed when run in parallel and the optimisation therefore searched for programs which executed faster when executed in parallel. Figure 3.1 shows the hill climbing algorithm.

### 3.4 Evolutionary Search-Based Transformation

Using the program itself as the individual to be optimised, it is hard to define a crossover operator, making the application of evolutionary search techniques problematic. That is, two programs $p_1$ and $p_2$, with identical behaviour would have to be combined to produce a program $p'$ with identical behaviour to each of $p_1$ and $p_2$. However, equivalence is undecidable, so it will be hard to automate the evaluation of
**Hill Climbing Algorithm:**

```
Begin
while (more restarts) do
    begin
        initialise $P_i$
        evaluate $P_i$
        let currentBest ← $P_i$
        while (not termination condition) do
            select neighbour $P_{i+1}$ according to neighbourhood criteria
            evaluate $P_{i+1}$
            if ($fitness_{P_{i+1}} > fitness_{P_i}$) then
                set currentBest = $P_{i+1}$
            end
        end
    end
end
```

---

Figure 3.1: First-Ascent Hill Climbing Algorithm structure.

crossover, making it impractical.

Ryan [97] describes an approach, called Pargen I, for auto-parallelisation, in which the individual is a program to be transformed, represented as an Abstract Syntax Tree, to which genetic programming mutation and crossover operators are applied. Ryan uses a set of test cases which he incorporates into the fitness function to determine whether the transformation has preserved meaning. This approach can inherently lead to a program which is not faithful to the semantics of the original. In the language of program transformation, this would result in a non-meaning preserving transformation, and would be rejected for all the applications of transformation described in the introduction.

Our approach in this thesis is to use the transformation sequence to be applied to
the program as the individual to be optimised. Using the transformation sequence as the individual makes it possible to define crossover relatively easily. Two sequences of transformations can be combined to change information, using single point, multiple point or uniform crossover. The result is a valid transformation sequence and since all transformation rules are meaning preserving, so are all sequences of transformation rules.

Clearly, as with all search–based techniques, this approach may not find the globally optimal transformation sequence, but any building blocks found will correspond to transformation tactics. Such tactics can be recognised, either automatically by the system itself as evolution takes place, or by a human monitoring the results. Since finding good transformation tactics is a goal of transformation research, even these modest sub-optimal results could be highly beneficial. Figure 3.2, shows a description of the genetic algorithm.

### 3.5 Greedy Algorithm

In this section, we describe the details of a greedy algorithm applied to search for good transformation rules.

From a pool of allowable transformation rules in the system, we build a sequence of rules applied to a source program $p$. Each rule is applied iteratively through a systematic (top-down) traversal of the AST. An optimisation is effected if a transformation rule is valid at the node of application. If the transformation rule fails at a node, then the next transformation rule from the pool is selected and applied at the node. The source program is being transformed as a sequence of applicable
transformation rules is collected on the fly as the algorithm progresses.

In contrast with the implementation for the genetic algorithm and hill climbing algorithm that manipulate fixed length transformation sequences, the systematic search algorithm ‘builds’ an expandable list of transformation rules. The description of the greedy algorithm can be seen in Figure 3.3.

We consider the greedy algorithm to be a systematic structured technique, because of the process by which it generates the transformation sequences. The only transformation rules added to the sequence are those that have been tested and found to be valid at the appropriate node. Therefore, the success of the greedy algorithm will be based on its ability to discard transformation rules that do not make program improvements and those that are not valid at the node of application. This process ensures that ‘bad’ transformation rules are discarded.
3.6 Yielding Insight into the Metrics Themselves

Given a code level metric $m$, a program $p$ defines a fitness landscape. If it is easy to optimise $p$ according to $m$ then the fitness landscape will have certain properties sought after by practitioners of the adaptive search techniques paradigm. For example, if the landscape is unimodal and smooth then even a simple-minded hill climbing approach will perform well in optimization and the original program can be viewed to be highly susceptible to improvement under metric $m$.

If the fitness landscape remains smooth, but is multi-modal, then hill-climbing is likely to become stuck in a local optimum, without ever reaching the global optimum. In this case simulated annealing and genetic algorithms are likely to out-perform simple-minded hill climbing. However, $m$ remains relatively susceptible to improvement according to $m$.

At the other end of the spectrum lie the fitness landscapes which are jagged, discontinuous and those that are predominantly ‘flat’. These landscapes are described in Figure 3.3: Greedy Algorithm structure.

**Greedy Algorithm:**

Begin  
Let $P =$ set of rules  
$N ← 1$  
Repeat until no more rules  
\( ∀T \) in $P$,  
if ($T$ is valid at Node $N$ ) then  
reapply $T$  
$N ← N + 1$  
end
by programs which are extremely hard to optimize using any technique.

By quantifying these various qualitative concepts, such as smoothness, modality and flatness, it will be possible to arrive at a ‘meta–metric’, $\mathcal{M}$, which evaluates a fitness landscape and therefore, implicitly evaluates the ‘optimizability of some program $p$ with respect to some metric $m$’.

3.7 Experiment 1: Lines of Code (LoC) Optimisation

In this section, we present details for an initial experimental study to evolve transformation sequences. The goal is to minimise the size (LoC) of the target program, that is, to compute sequences of rules that result in the smallest possible equivalent target program.

3.7.1 FermaT Transformations

We implemented our algorithms in the WSL [122] programming language using transformation rules from the FermaT [114] transformation workbench. FermaT has a number of built-in transforms that could be applied directly to any point within the program.

The transformations in FermaT could, if their application is possible, either increase or decrease the size of the program. Some transformations may leave the program size unchanged. We do not include those transformations that could transform an entire program in one single step but rather atomic ones that work on pairs
of nodes as previously shown in our examples.

In addition to transformation rules which alter the structure of the source programs in some way, we also include in the sequence, rules such as [right], [left], [up] and [down], that do not change the syntax of the input program but merely allow the movement of the current cursor position along the parse tree for the source program. We term these types of transformations ‘navigational rules’.

The inclusion of such transforms mean that we also need not statically determine the position of application for the optimising transforms but allow the algorithm to walk through the input program to determine when and where a valid transform should be applied. Every other transform apart from the four described above is a WSL-to-WSL transformation of the input source designed to alter the code in some way. In the experiments carried out, the following 20 transformations were used: ¹

¹Source for the description of FermaT transformations compiled from the transformation manual
<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double to Single Loop</td>
<td>Double to Single Loop will convert a double nested loop to a single loop, if this can be done without significantly increasing the size of the program.</td>
</tr>
<tr>
<td>Else-if to Elsif</td>
<td>This transformation will replace an ‘Else’ clause which contains an ‘If’ statement with an ‘Elsif’ clause. The transformation can be selected with either the outer ‘If’ statement, or the ‘Else’ clause selected.</td>
</tr>
<tr>
<td>Elsif to Else-if</td>
<td>This transformation will replace an ‘Elsif’ clause in an ‘If’ statement with an ‘Else’ clause which itself contains an ‘If’ statement. The transformation can be selected with either the ‘If’ statement, or the ‘Elsif’ clause selected.</td>
</tr>
<tr>
<td>Merge-right</td>
<td>This transformation will merge the selected statement into the statement that follows it.</td>
</tr>
<tr>
<td>Merge-left</td>
<td>This transformation will merge the selected statement (or sequence of statements) into the statement that precedes it.</td>
</tr>
<tr>
<td>Remove-redundant-vars</td>
<td>This transformation will remove any redundant variables in the source program.</td>
</tr>
<tr>
<td>Transformation</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>@absorb-right</code></td>
<td>A transformation that would absorb into the current statement, the one that follows it.</td>
</tr>
<tr>
<td><code>@absorb-left</code></td>
<td>A transformation that would absorb into the current statement, the one that precedes it.</td>
</tr>
<tr>
<td><code>@simplify</code></td>
<td>This transformation would simplify any components as fully as possible.</td>
</tr>
<tr>
<td><code>@up</code></td>
<td>Moves up one level on the parse tree from the current item if possible.</td>
</tr>
<tr>
<td><code>@down</code></td>
<td>Moves down one level on the parse tree from the current item if possible.</td>
</tr>
<tr>
<td><code>@right</code></td>
<td>Moves right into next item on program parse tree.</td>
</tr>
<tr>
<td><code>@left</code></td>
<td>Moves left into the next item on program parse tree.</td>
</tr>
<tr>
<td><code>@move-to-right</code></td>
<td>This transformation will move the selected item to the right so that it is exchanged with the item that follows it.</td>
</tr>
<tr>
<td><code>@move-to-left</code></td>
<td>This transformation will move the selected item to the left so that it is exchanged with the item that precedes it.</td>
</tr>
<tr>
<td><code>@add-left</code></td>
<td>This transformation will add the selected statement (or sequence of statements) into the statement that precedes it without doing further simplification.</td>
</tr>
<tr>
<td><code>@delete-all-comments</code></td>
<td>This transformation will delete all the ‘COMMENT’ statements within the selected code.</td>
</tr>
<tr>
<td><code>@combine-wheres</code></td>
<td>Will combine two nested WHERES into a single structure.</td>
</tr>
<tr>
<td><code>@delete-all-redundants</code></td>
<td>Deletes all redundant variables.</td>
</tr>
<tr>
<td><code>@delete-all-skips</code></td>
<td>This transformation will delete all the ‘SKIP’ statements within the selected code.</td>
</tr>
</tbody>
</table>

A test for validity is carried out before each transformation is applied at the
current cursor position. Each transformation is only performed if its application at that point is valid. If the test for the application of a transformation is invalid then it is not applied and the program remains unchanged.

### 3.7.2 Simplify Transformation Rule

The simplify transformation rule takes every expression to which it is applied within the program and attempts to evaluate it as fully as possible. It carries out each evaluation locally, that is, it does not consider surrounding statements in the evaluation of a current statement. The simplify rule can be applied to evaluate assignment and predicate statements.

Consequently, this rule cannot be considered a ‘magic bullet’ because the simplifications achieved are not global but occur at the level of the expression and therefore the fitness value remains unchanged after the application of the rule.

We show examples illustrating the effect of applying simplify to a code fragment. From the following code fragment, assuming that simplify was applied to line 1, we observe that its application has no effect on the code.

\[
\begin{align*}
1. \ x & := x + 1; \\
2. \ x & := x + 1;
\end{align*}
\]

\[
\begin{align*}
1. \ x & := x + 1; \\
2. \ x & := x + 1;
\end{align*}
\]

The next example illustrates how simplify evaluates an expression. However, observe that there is no reduction in fragment size, as measured by lines of code, our fitness measure.

\[
\begin{align*}
1. \ x & := x + 1 + 3; \\
2. \ x & := x + 1;
\end{align*}
\]

\[
\begin{align*}
1. \ x & := x + 4; \\
2. \ x & := x + 1;
\end{align*}
\]
Again, the following example illustrates that applying simplify results in no further reduction the size of the program fragment. Assume that the simplify rule is applied at line 3, then the it is not sophisticated enough to evaluate that predicate expression as false. The consequence therefore, is that the fragment remains unchanged.

\[
\begin{array}{ll}
1. & x := 3; \\
2. & y := 4; \\
3. & \text{IF } x > y \quad \Rightarrow \\
4. & \text{THEN} \\
5. & x := x + 1; \\
6. & x := x + 1; \\
7. & \text{ELSE} \\
8. & x := x + 1; \\
9. & \text{FI}; \\
\end{array}
\]

However, one exception to simplify not affecting the fitness value of a transformation sequence exists. Consider the situation where an IF-predicate can be evaluated fully, without constant propagation because it contains only compile time constraints. As a result, either the THEN or ELSE branches of the IF-statement will be deleted.
This is the only situation in which \texttt{simplify} will affect the value of our fitness function.

### 3.7.3 Test Objects

The experiment was carried out on a number of ‘toy’ test programs, with different program characteristics - Sequential, Conditional and Iterative programs. Examples of these test programs are shown in Figure 3.4. This experiment was performed on single function intra-procedural programs only, which have no function calls and no pointer variables.

A goal of the experiment is to observe the effect of algorithms evolving transformation rules and to apply these rules at precise points in the program. Sometimes, a rule that optimises the program can only be applied after a pre-requisite set of rule. The preceeding rules can be said to provide an opportunity for the optimising rule. The test object were designed to exhibit these properties. For instance, in carrying out any code simplifications inside the a branch of an IF-statement, there are a number of transformation that \textit{must} carry us into the required branch. Other programs demonstrates an increase in the number of moves required before an optimisation can
Search-Based Transformations

Transformation Problem

<table>
<thead>
<tr>
<th>program one</th>
<th>program two</th>
<th>program three</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. x := x + 1;</td>
<td>1. x := x + 1;</td>
<td>1. x := x + 1;</td>
</tr>
<tr>
<td>2. x := x + 1;</td>
<td>2. x := x + 1;</td>
<td>2. x := x + 1;</td>
</tr>
<tr>
<td>3. x := x + 1;</td>
<td>3. x := x + 1;</td>
<td>3. x := x + 1;</td>
</tr>
<tr>
<td>4. x := x + 1;</td>
<td>4. a := a + 1;</td>
<td>4. IF x &gt; y THEN</td>
</tr>
<tr>
<td>5. x := x + 1;</td>
<td>5. b := b + 1;</td>
<td>5. b := b + 1;</td>
</tr>
<tr>
<td>6. x := x + 1;</td>
<td>6. c := c + 1;</td>
<td>6. b := b + 1;</td>
</tr>
<tr>
<td>7. x := x + 1;</td>
<td>7. x := x + 1;</td>
<td>7. x := x + 1;</td>
</tr>
<tr>
<td>8. x := x + 1;</td>
<td>8. ELSE</td>
<td>8. ELSE</td>
</tr>
<tr>
<td>9. x := x + 1;</td>
<td>9. c := c + 1;</td>
<td>9. c := c + 1;</td>
</tr>
<tr>
<td>10. x := x + 1;</td>
<td>10. c := c + 1;</td>
<td>10. c := c + 1;</td>
</tr>
<tr>
<td>11. FI;</td>
<td>11. FI;</td>
<td>11. FI;</td>
</tr>
<tr>
<td>12. x := x + 1;</td>
<td>12. x := x + 1;</td>
<td>12. x := x + 1;</td>
</tr>
<tr>
<td>13. x := x + 1;</td>
<td>13. x := x + 1;</td>
<td>13. x := x + 1;</td>
</tr>
<tr>
<td>14. x := x + 1;</td>
<td>14. x := x + 1;</td>
<td>14. x := x + 1;</td>
</tr>
</tbody>
</table>

Figure 3.4: Sample test programs, showing the increasing number of moves to the right that a good sequence would require to produce an optimal solution. Program one requires a sequence of absorb transformations to produce the optimal result. In order to simplify program two and three, the algorithms need to be ‘intuitive’ to realise where potential optimisations may be applied, e.g., in program three, the program cursor would eventually need to go DOWN the Abstract Syntax Tree (AST) for the program, into the IF-block for further simplifications to occur.

3.7.4 Fitness Function

We measure the fitness of a potential solution as the difference in lines of code between the source program and the new, transformed program created from the execution of a particular sequence.

This is evaluated by:

1. Computing the length of the input source program.
2. Generating transformation sequence (randomly, through crossover and mutation in the case of the GA or next neighbour in the HC).

3. Applying the transformation sequence to input source code.

4. Computing new length of target source program after sequence has been applied.

5. Computing the fitness of individual (old length - new length).

### 3.7.5 Experimental Setup

We carried out experiments to investigate the potential for finding good transformation sequences that optimise the size of the source code. We arbitrarily defined a fixed length of 20 genes for our transformation sequence and compare the results of our search-based techniques against a random search technique.

We implemented a standard genetic algorithm using single point crossover, a crossover rate of 100% and a mutation rate of 7%. We adopted a tournament selection strategy for choosing mating parents and created a single offspring which was re-inserted into the population to replace the weaker of its parents. We also employed an elitist strategy keeping the best individual across successive generations. There was a constant population size of 50 individuals and we ran the algorithm over 200 generations.

We implemented a hill-climbing (HC) algorithm with multiple restarts also recording the best solution found for each restart. The solution returned is that with the best fitness value from the different hill climbs. The HC algorithm uses the first-ascent technique, where the first better solution found becomes the new current position. The algorithm is restarted 10 times with a new random individual each time, so that
the algorithm begins at the base of a different ‘hill’ with each restart. The restarts are required in the hope that the search may escape from the peak of any small ‘hills’ it may find.

The algorithm for random search generates solutions randomly and iteratively and each solution is applied to the test programs. The best solution across each iteration is recorded. Each algorithm is granted the same total number of 3,000 fitness evaluations. The results presented are averaged over 10 runs for each algorithm.

The target program is kept static under all implementations, meaning that each new sequence of transformation rules is applied to the same starting program. Once a sequence has been applied and observed, its effects are discarded before the next selection is applied.

We observe two outputs from the execution of our algorithms: the most desirable sequence of transformations that it finds and the number of fitness evaluations taken to arrive at that solution.

### 3.7.6 Results

Figures 3.5 to 3.12 illustrate the result of the experiment carried out on the various test objects. Each graph shows the result of executing each of the three search algorithms on a single test source program. The mean fitness value over 10 runs is plotted on the y-axis against the number of fitness evaluations required by the techniques to achieve the fitness values on the x-axis.

The results from this experiment show that the genetic algorithm outperforms both the HC algorithm and the random search algorithm in the search for good minimising transformation sequences, which reduce the number of lines of source code.
code. An exception to this is Figure 3.5, which shows the result on a test object with IF statements included. In addition, as the source files increase in size, the gap in the performance between the genetic algorithm and the hill climbing algorithm also increases.

We observe that it takes fewer fitness evaluations for the GA to match the fitness values for programs returned by random search and hill-climbing.

Interestingly, we noticed that the hill-climbing algorithm repeatedly performed worse than random search over different test criteria. We believe that this might be due to specific implementation details in the hill-climbing algorithm, which inhibits its ability to find desirable solutions. The representation scheme chosen for the HC algorithm implements a rather tight neighbourhood condition. This restrictive constraint may prevent a wider traversal of the search space.

One test case where random search outperformed both heuristic methods shown as Figure 3.5 provided us with a feeling for how difficult it was for the heuristic methods to find even reasonably good solutions.

That is, the implementation scheme for the transformation rules in our search procedure requires that before an “optimising” transformation rule can be applied, that is, a transformation rule which is guaranteed to produce a reduction of at least 1 statement in the source program, our search procedure first requires that it finds the precise location where this optimisation is required. Hence, the search is inherently complicated. First, we need to find the appropriate location and then apply an appropriate optimising transformation rule. This is especially true because of the nature of the transformation rules we are searching for and illustrates the difficulty of the problem.
Consider the following test program:

1. \( x := x + 1 \);
2. IF \( y > z \) THEN
3. \( x := x + 1 \);
4. \( x := x + 1 \);
5. \( x := x + 1 \)
6. ELSE
7. \( y := y + 1 \)
8. FI

We can clearly observe that some optimisations can be performed at line 3. However, in the implementation, the application of transformation rules begins on line 1. Therefore, the search requires transformation rules that would lead it into line 3 before the appropriate transformations can be effected. The IF-structure requires a series of ‘navigational’ transformation rules to occur before cursor control is transferred into
either the THEN or ELSE branches of the IF-structure. This is an example situation that illustrates the need for good building blocks (subsequences).

We observe that the random search algorithm outperforms both the hill climbing algorithm and the genetic algorithm for test programs such as these which contain IF predicates. In the case shown in Figure 3.5, the GA and HC were not ‘moving’ toward these areas where potential optimizations may occur. However, the nature of random search allows for a certain amount of luck in finding an initial transformation sequence that moves the cursor to a fertile location for optimisation.

Characteristically, the GA is described as a weak search procedure because it does not require any domain knowledge to be encoded into the structure of potential solutions. Even where domain knowledge is encoded into the structure, it may or may not be preserved when GA operations such as crossover take place. In analysing our implementation for the GA, we believe that the GA potentially kills off good subsequences of transformations during crossover. These subsequences, can then be treated as building blocks which feed into later stages of the GA search process. With the presence of domain knowledge encoding and a rather clever crossover scheme, it might be possible to improve the overall quality of solutions returned by the GA by preserving the building blocks.

Finally, we conjecture that the presence of certain features intrinsic in some programs would allow random search to perform surprisingly well when compared against supposedly more ‘clever’ approaches. For instance, consider program one in Figure 3.4, a sequence of six merge-right transformations would result in the following program:
Figure 3.6: Comparison of performance of GA, HC and RS on ‘toy’ test 1 object.

Figure 3.7: Comparison of performance of GA, HC and RS on ‘toy’ test 2 object.
Figure 3.8: Comparison of performance of GA, HC and RS on ‘toy’ test 3 object.

Figure 3.9: Comparison of performance of GA, HC and RS on ‘toy’ test 4 object.
Figure 3.10: Comparison of performance of GA, HC and RS on ‘toy’ test 5 object.

Figure 3.11: Comparison of performance of GA, HC and RS on ‘toy’ test 6 object.
**Figure 3.12:** Comparison of performance of GA, HC and RS on ‘toy’ test 7 object.

\[
x := x + 7;
\]

Similarly, for program two, a sequence of two \texttt{merge-right}, three \texttt{move-right} and then more \texttt{merge-right} transformation rules would produce the following optimum equivalent program:

\[
\begin{align*}
a & := a + 1; \\
b & := b + 1; \\
c & := c + 1; \\
x & := x + 7;
\end{align*}
\]

while for program three, a sequence of two \texttt{merge-right} transformation rules and one \texttt{down} transformation rule would allow the current cursor position be placed inside the IF-block of code where further simplifications may then be performed. Further \texttt{merge-right} transformation rules may then be applied while in the IF-block to produce the following output:
3.7.7 Statistical Analysis

We performed the Wilcoxon test for significance. This is a non-parametric test to check the significance of the difference between two samples of results. The \( p \)-values resulting from the application of the test are presented in Figure 3.13, showing strong significance in the results across the different test cases. The comparison between GA and Random shows at least a 99.3\% confidence that the GA outperforms random search. For the test programs, there is also at least 89\% confidence that the GA outperforms HC.

<table>
<thead>
<tr>
<th>Test</th>
<th>GA v. Random</th>
<th>GA v. HC</th>
<th>Random v. HC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.007</td>
<td>.011</td>
<td>.033</td>
</tr>
<tr>
<td>2</td>
<td>.004</td>
<td>.005</td>
<td>.010</td>
</tr>
<tr>
<td>3</td>
<td>.003</td>
<td>.004</td>
<td>.010</td>
</tr>
<tr>
<td>4</td>
<td>.004</td>
<td>.005</td>
<td>.010</td>
</tr>
<tr>
<td>5</td>
<td>.004</td>
<td>.004</td>
<td>.011</td>
</tr>
<tr>
<td>6</td>
<td>.004</td>
<td>.005</td>
<td>.010</td>
</tr>
<tr>
<td>7</td>
<td>.005</td>
<td>.005</td>
<td>.011</td>
</tr>
</tbody>
</table>

Figure 3.13: Results of Wilcoxon significance tests showing strong significance in the results with the GA outperforming both random search and the HC.
3.7.8 Conclusions of Experimental Study 1

The experiments highlight the difficulty involved with training search heuristics to find sequences of appropriate transformation rules. It showed that it was not always possible for heuristic techniques to optimise programs. However, in many cases, the genetic algorithm outperformed the random method and the hill climbing algorithm. The hill climbing algorithm performed poorly when compared against the random method.

3.8 Summary

This chapter described the transformation problem and provided some justification for the application of search-based methods to solve the problem. The size of the search space of possible sequences of transformation rules suggests that exhaustive search is infeasible and that advanced techniques might be appropriate. The chapter described the local and global search techniques and the appropriateness of the representation selected in carrying out operations such as crossover.

The issue of searching for transformations has already been examined in Chapter 2. Cooper et al.[29, 30] searched for sequences of transformation for optimising whole programs. Furthermore, Nisbet[88] and Williams[120] examined the problem of searching for specifically loop reordering transformations to facilitate parallelisation. However, this chapter investigated the search for low-level source transformations that rather than optimising the entire program at once, optimise small segments of the program at a time.

We conclude that our search-based methods offer a viable approach to performing
program transformation for optimising source code size. In the next chapter, we extend the search-based approach to solve a different problem validating the robustness claim of the approach.
Chapter 4

Application of SB-Transformations:

Amorphous Slicing

In the experiment described in Chapter 3, we used a genetic algorithm and a multi
Hill Climb algorithm to identify semantically equivalent programs with fewer lines of
code than the original program.

In this section, we extend the methods described in Chapter 3 and apply search-
based techniques to compute amorphous slices. An amorphous slice is a subset of the
program which preserves the behaviour of the program with respect to the variable
of interest. The structure of the slice may not be similar to that of the program.
Harman et al. [60] developed an amorphous slicing system LinIAS based on depen-
dence reduction transformations and traditional slicing, which is a tailor-made system
designed for computing amorphous slices. Our approach in this thesis, is to reformu-
late the the problem of computing these slices into a redundancy removal problem.
Section 4.3 shows how this is carried out. The chapter concludes by describing an experiment using our search based algorithms for amorphous slicing. The experiment uses a greedy algorithm. The results from these experiments are compared against those returned by LinIAS for the same test objects.

### 4.1 Introduction

Inherent in the idea of amorphous slicing, is the application of transformation rules such as constant propagation, statement deletion, etc., in unfolding expressions. These specific transformation rules are selected from within a pool using an appropriate selection strategy. Typically if the transformation sequences are of fixed length, then the size of the search space is exponential.

In this chapter, we report the results of our second empirical study. We use our search algorithms as a technique for tackling a current practical problem, the creation of valid amorphous slices. We reformulate the problem of computing valid amorphous slices for our programs under test and carry out redundancy removal using a fitness function that is similar to that described in chapter 3. The results in this chapter demonstrate that search-based approaches can successfully be applied to solving real problems where tailor-made analytic algorithms have already been examined.

### 4.2 Problem

Amorphous slices have traditionally been produced by using dependence relations in a program dependence graph or a similar code artefact and applying meaning
preserving transformations. The challenge of this experiment is to compute these amorphous slices without the use of any such design artefact. The experiment aims to determine whether valid amorphous slices can be generated simply by using existing transformation rules. Figure 4.3, shows the difference between a syntax-preserving slice and an amorphous slice.

| TA:= 2*a;      | TA:= 2*a;                  | slice:=(-b+sqrt((b*b) - (4*a*c)))/(2*a); |
| b_sq:= b*b;    | b_sq:= b*b;                |                                           |
| temp := b_sq * TA; | root := sqrt(b_sq-(4*a*c)); |                                           |
| root := sqrt(b_sq-(4*a*c)); | var := -b*root; |                                           |
| var := -b*root; | x:= var/TA;                | slice := x;                               |
| slice := x;    |                            |                                           |

Original | Syntax-Preserving slice | Amorphous slice

Figure 4.1: Illustrative example showing Amorphous Slicing Producing Thinner Slices by Removing Syntactic Restrictions. Slice computed w.r.t. variable slice at end.

4.3 Reformulating Amorphous Slicing as a Redundancy Removal Problem

In order to compute valid amorphous slices that preserve the computation for a single variable of interest (slicing criterion) using a search-based approach, we re-formulate the problem as a redundancy removal problem by appending killing assignment statements to the program and then optimise by removing (transforming out) these redundant statements.

\[1\] Assigning a constant value to a variable identifier, which invalidates previous assignments to that variable identifier.
Given a variable \( x \) captured by our slicing criterion and a set of assigned variables \( V \) in a program, such that, \( V' = V - x \). In order to obtain a slice \( S \) which represents a projection of the semantics of a program \( P \) with respect to \( x \) in the slicing criterion, we need to ensure that our algorithm preserves the presence of \( x \) in \( S \). For all variables \( v \) in \( V' \), we append killing assignments such that: \( v = C \) (where \( C \) is some arbitrary constant value).

This provides the search techniques with variable differentiation, which distinguishes the slicing criterion from other variables defined within the test program. The search for transformation sequences and the specific transformation rules being tested are performed on the reconstructed program (Figure 4.2).

---

<table>
<thead>
<tr>
<th>Source Program: slicing criterion is</th>
<th>Re–Constructed Program with killing variables appended to end-of-program</th>
</tr>
</thead>
<tbody>
<tr>
<td>( D := 2*r; )</td>
<td>( D := 2*r; )</td>
</tr>
<tr>
<td>( \text{FaceArea} := \pi<em>r</em>r; )</td>
<td>( \text{FaceArea} := \pi<em>r</em>r; )</td>
</tr>
<tr>
<td>( C := \pi*D; )</td>
<td>( C := \pi*D; )</td>
</tr>
<tr>
<td>( \text{SArea} := 2*\text{FaceArea}+h*C; )</td>
<td>( \text{SArea} := 2*\text{FaceArea}+h*C; )</td>
</tr>
<tr>
<td>( \text{slice} := \text{SArea}; )</td>
<td>( \text{slice} := \text{SArea}; )</td>
</tr>
</tbody>
</table>

Figure 4.2: Re–structuring Source Program for Search–Based Transformations
4.4 Experimental Study 2 - Search Based Amorphous Slicing

This section describes the experiments carried out using 3 different search-based heuristic techniques for computing amorphous slices. The objective of the experiment is to determine whether search-based techniques could be used to compute valid amorphous slices. The algorithms presented were implemented in the Wide Spectrum Language (WSL) using transformation rules from the FermaT transformation workbench [114]. Fatiregun et al. [39] lists some of the transformations rules implemented in the FermaT system and used in this work. The results obtained are compared with those obtained from LinIAS [60] - a tailor-made amorphous slicing tool. The LinIAS system is seen as providing an ideal upper bound.

4.4.1 Experimental Setup

We examined the application of search-based methods to find transformations which would optimise programs to generate amorphous slices with respect to a given slicing criterion. The research included comparing the results from four search methods: Genetic Algorithm, Hill–Climb Algorithm, Greedy Algorithm and Random Search.

In order to apply our search methods, there is the need to distinguish the slicing criterion from other assigned variables within the source by restructuring the program by way of adding killing assignments to the source (as described in section 4.3). For the individual based search methods, we define arbitrarily a fixed length of 20 genes per individual for the transformation sequences.

We implemented a standard Genetic Algorithm using single point crossover and
a mutation rate of 7%. In our implementation, we carry out selection using the tournament selection method for choosing mating parents and create a single offspring from the mating. The offspring replaces the parent with the worse fitness value in the new population. We define a constant population size of 200 and run the algorithm over 100 generations. The individual with the best fitness value is replicated across successive generations.

The experiment with the hill–climb algorithm was set up with multiple restarts, storing the best individual over the different restarts. The HC Algorithm uses a first ascent technique. A neighbouring individual is analysed and its fitness value computed. If the fitness of the neighbour is better than that of our current position, the neighbour becomes our new current best position. Neighbourhood in our hill–climb algorithm is defined as the mutation of a single gene in an individual (all other genes remain unchanged). The algorithm is restarted 10 times with a new random individual each time.

With the greedy algorithm, each transformation rule is applied iteratively at each node in the program. Because the algorithm works with single genes rather than individuals, a collection of sequences is built up as the algorithm progresses through each program node.

Lastly, a random search for transformation sequences was implemented using identical metrics. Random transformation sequences of fixed length 20 genes are computed and applied to the source program. The fitness values for these individuals are noted. The experiment is terminated after a 50000 of iterations.
4.4.2 Test Objects

During the experiment, the algorithms described were tested using 7 programs including 2 from a large industrial automobile company: a simple odds and evens program, a surface area program, a simplified UK tax program, a program computing student marks on a given course, a calendar program, a rear-end window defroster and a braking system controller. Each algorithm is terminated after a fixed number of fitness evaluations.

An amorphous slice was computed for each test program w.r.t. every assigned variable in the source code from 4 slices per technique for the Area program to over 60 slices computed per techniques for the braking controller.

We observe two outputs from each simulation of the algorithms: the amorphous slice and the number of lines of code for the slice. In the implementation, we keep the source program static. This means that each new application of an individual is applied to the same original source program. The effects on a program of previous applications of transformation sequences are discarded before the next sequence is applied.

This experiment was performed on single function intra-procedural programs only, which have no other function calls. The following is a description of the programs contained in the second suite of test programs:
<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surface Area</td>
<td>A simple program that implements the formula for calculating the surface area of an object.</td>
</tr>
<tr>
<td>Odds/Evens</td>
<td>A simple decision program to determine whether a number is an odd number or an even number. It contains 3 IF predicates.</td>
</tr>
<tr>
<td>Tax</td>
<td>A simplified United Kingdom tax program. This test object contains multiple nested IF statements.</td>
</tr>
<tr>
<td>Student Marks</td>
<td>Program which calculates the grades that a student should receive for a particular score.</td>
</tr>
<tr>
<td>Calendar</td>
<td>A leap year validator. It determines whether a given year is a leap year.</td>
</tr>
<tr>
<td>Rear-Window Defroster</td>
<td>One of the test programs from the automobile company. The “rear-window controller” is an embedded program original written in C. It was translated into the WSL language before the experiment could be performed.</td>
</tr>
</tbody>
</table>
Braking System Controller | The “braking system controller” is another of the program from the automobile company and is responsible for controlling the braking operations of the car. It was originally written in C, and contains a large number of control predicates. It was translated into the WSL language before the experiment could be performed.

4.4.3 Results

Figure 4.3 summarises the results of the experiment. It shows the average percentage reduction in size of the source code. We see that the greedy algorithm and the hill climbing algorithm on the Odds/Even test object result in about an 80% reduction in size. The table shows the average reduction in program size, when a slice is constructed for every defined variable in the test program.

Figures 4.4 – 4.17 show the results of applying different heuristic search approaches to amorphous slicing and are compared against the analytic-based LinIAS System. The results are based on slicing w.r.t. every every assigned variable in the test program and the average slice size for each technique applied.

We observe that the search heuristics achieved success in slicing the program under test with respect to the slicing criterion, returning in some cases slice sizes that match the LinIAS system.
Figure 4.3: Results of executing different search techniques to compute amorphous slice and the analytic amorphous slicing system showing average percentage reduction in slice size

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (LoC)</th>
<th>Search Technique</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>S. Area</td>
<td>5</td>
<td>GA</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gr</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HC</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LinIAS</td>
<td>1</td>
</tr>
<tr>
<td>OddEven</td>
<td>41</td>
<td>GA</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gr</td>
<td>79</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HC</td>
<td>82</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LinIAS</td>
<td>87</td>
</tr>
<tr>
<td>Tax</td>
<td>77</td>
<td>GA</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gr</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HC</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LinIAS</td>
<td>34</td>
</tr>
<tr>
<td>Calendar</td>
<td>87</td>
<td>GA</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gr</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HC</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LinIAS</td>
<td>53</td>
</tr>
<tr>
<td>Defroster</td>
<td>123</td>
<td>GA</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gr</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HC</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS</td>
<td>71</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LinIAS</td>
<td>78</td>
</tr>
<tr>
<td>Marking</td>
<td>155</td>
<td>GA</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gr</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HC</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LinIAS</td>
<td>61</td>
</tr>
<tr>
<td>Braking Clr.</td>
<td>326</td>
<td>GA</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gr</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HC</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RS</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LinIAS</td>
<td>72</td>
</tr>
</tbody>
</table>

There were wide variations in the results obtained from the various search techniques. The Hill–Climb Algorithm performed better with the small sized Area program than it did with the larger sized test programs while the Greedy Algorithm performed more consistently across the various tests. This perhaps suggests that a hybrid algorithm may combine the best aspects of each of these individual approaches to produces even better results.

Figures 4.4 to 4.10 shows the graphs of the experiments carried out on each test object. Each point on the graphs represents the value for executing a particular technique on the test object with respect to a single slicing criterion. A lower point on the graph indicates a better result than one higher up in position.

Figure 4.4 which illustrates the results from the test on the rear-end window controller show the Greedy Algorithm considerably reducing the program size to a level almost comparable to LinIAS. On each vertical array of points, we see that LinIAS returns the best slice sizes followed by the greedy algorithm. Encouragingly,
Figure 4.4: Rear-Window defroster test Object.

the average size for slices produced by search techniques are better than those returned by random search and a syntax-preserving slicer.

In the Calendar program test case, we see in the graph shown in Figure 4.5 that there is a clear difference between the slice sizes returned by both greedy algorithm and LinIAS when compared against the remaining techniques. Across all slicing criteria, we found that these two implementations consistently outperform the others. This is clearly seen in Figure 4.3, where if we consider the Calendar test case, we see both greedy algorithm and LinIAS achieving 51% and 53% in average percentage slice size reduction. The other algorithms only result in about 26% average slice size reduction.

Figure 4.6 shows the graph of the Odds/Even test case and again identifies that LinIAS performs better. In this case, we see that both the genetic algorithm and random search perform worst of all the techniques investigated.

Figure 4.7 shows the results of the experiments on the relatively small sized Area
Figure 4.5: Calendar test object.

Figure 4.6: Odds / Even test object.
In Figure 4.8, we show the result of applying the slice construction techniques to the Braking System Controller test object. In this experiment, we constructed lots of slices, which explains the number of points on the graph. However, from the graph, it is clear that for each slicing criterion, linIAS performs best. Again we observe that of the search based implementations, greedy algorithm outperforms all the other search based methods. There is also an evident tracking between the results of the different approaches for each slicing criteria. For instance, for two slicing criteria $sc_1$ and $sc_2$, if linIAS returned a better result for $sc_1$ than $sc_2$, then the other techniques tend to follow a similar trend of results. That is, the greedy algorithm would also return a better result for $sc_1$ than $sc_2$. Of course, this is logical because the variable being sliced on in $sc_2$ might have more dependencies in the program and result in larger slice size irrespective of the different techniques.

Figure 4.9 shows the graph for the Student Marks test program. Again, we notice
Search-Based Transformations

Figure 4.8: Braking System Controller test object.

Figure 4.9: Student Marks test object.
that LinIAS returns the best results and across all the different slicing criteria, we see that the genetic algorithm and random search perform worst of the different techniques.

In Figure 4.10, which represents the graph of results for the Tax program, we see that there variation in slice sizes by the different techniques is small. For each slicing criterion, each algorithm appears to return slice sizes that are in about the same region.

Interestingly, we observe that in one test case, both the Random Search and the Hill–Climb return smaller slices than the LinIAS amorphous slicing system. We provide an analysis of this test case in Section 4.4.4.

Figure 4.11 is a bar-chart of the result from executing the algorithms on the surface area test object. We notice that hill climbing and LinIAS provide almost identical slice sizes.
Box Plots

Figures 4.12 to 4.17 illustrates the results of the experiment as boxplots, which show the distribution of the results for each technique per test object. Each box represents the distribution of the different slice sizes for each slicing criterion. It shows the upper and lower quartiles, while the thick black line across the box indicates the median value. Outliners are shown as asterisks on the graph.

Consider Figure 4.13, the greedy algorithm performs much better than any other search technique in computing amorphous slices for every defined variable in that test object. The results from the greedy algorithm are comparable with those from LinIAS.

Significantly, although the LinIAS system on the average returns smaller slices than the search–based approaches, it has a seemingly unfair advantage. It uses as one of its transformations, a traditional syntax-preserving static slicer, which the search methods did not have available within their pool of transformations. The presence
Figure 4.12: box plot for odds/even test object.

Figure 4.13: boxplot for calendar test object.
Figure 4.14: boxplot for student marks test object.

Figure 4.15: boxplot for tax test object.
Figure 4.16: boxplot for braking system controller test object.

Figure 4.17: boxplot for rear-window defroster test object.
of such a traditional slicing transformation rule in the pool of transformation rules available to the search algorithms may result in even more inspiring results than those already observed. However, it is very encouraging that these techniques still return good and competitive results despite this handicap.

### 4.4.4 Output Slices

An case study of one of the input programs used in the experiment is described along with the output slices generated by techniques. In Figure 4.18, we see the un-transformed version of the tax test program. An amorphous slice of this program is constructed with respect to each variable defined in the program. Therefore, we construct a slice for variables such as `married`, `pc10`, `personal` and so on. We used our search-based implementations to generate transformation sequences to transform this program, as well as, applying the LinIAS system to construct amorphous slices.

Figures 4.20 through 4.23 show the output slice produced by greedy, genetic algorithm, random search, hill climbing algorithm and LinIAS respectively. For this particular slicing criterion, we see clearly that both the greedy algorithm and genetic algorithm perform badly and only make limited optimisations. The code portions highlighted in red could have been deleted but were missed by the algorithms. However, in Figure 4.21, we observe the output slice constructed by Random Search and note that it performs better than both greedy and the genetic algorithm. Random Search was able to delete large chunks of code missed by the greedy and genetic algorithms. One reason why this may arise, is the large number of potentially good solutions present in the search space, which increases their chances of being selected randomly.
age := 40;
WHILE age < 0 DO
  age := read; read := read + next OD;
income := 23000;
WHILE income < 0 DO
  income := read; read := read + next OD;
marrried := 1;
WHILE married <> 0 AND married <> 1 DO
  married := read; read := read + next
OD;
widow := 1;
WHILE widow <> 0 AND widow <> 1 DO
  widow := read; read := read + next OD;
blind := 0;
WHILE blind <> 0 AND blind <> 1 DO
  blind := read; read := read + next OD;
IF age >= 75
  THEN personal := 5980
ELSE IF age >= 65
  THEN personal := 5720
  ELSE personal := 4335 FI FI;
IF age >= 65 AND income > 16800
  THEN IF 4335 > personal - (income - 16800) / 2
    THEN personal := 4335
    ELSE personal := personal - (income - 16800) / 2
    FI IF;
IF blind = 1
  THEN personal := personal + 1380 FI;
IF married = 1 AND age >= 75
  THEN pc10 := 6692
ELSE IF married = 1 AND age >= 65
  THEN pc10 := 6625
  ELSE IF married = 1 OR widow = 1
    THEN pc10 := 3470
    ELSE pc10 := 1500 FI FI FI;
IF married = 1
  AND age >= 65
  AND income > 16800
  THEN IF 3470 > pc10 - (income - 16800) / 2
  THEN pc10 := 3470
  ELSE pc10 := pc10 - (income - 16800) / 2
  FI FI;
IF income <= personal
  THEN tax = 0
ELSE income := income - personal;
IF income <= pc10
  THEN tax := (income * 10) / 100
ELSE tax := (pc10 * 10) / 100;
  income := income - pc10;
IF income <= 28000
  THEN tax := tax + (income * 23) / 100
ELSE tax := tax + (28000 * 23) / 100;
income := income - 28000;
tax := tax + (income * 40) / 100
FI FI FI;
IF blind = 0 AND married = 0 AND age < 65
THEN code := 1
ELSE IF blind = 0 AND age < 65 AND married = 1
  THEN code := 2
  ELSE IF age >= 65
    AND age < 75
    AND married = 0
    AND blind = 0
    THEN code := 3
    ELSE IF age >= 65
      AND age < 75
      AND married = 1
      AND blind = 0
      THEN code := 4
      ELSE code := 5 FI FI FI FI;
age := 1;
income := 1;
tax := 1;
code := 1;
read := 1;
marrried := 1;
widow := 1;
blind := 1;
pc10 := 1;
slicevar := personal

Figure 4.18: Source code for tax program before transformation.
Figure 4.19: Output code slice for tax program returned by the Greedy Algorithm. Slicing criterion is variable personal at end of program.
Figure 4.22 shows the output amorphous slice produced by the hill climbing algorithm. In this particular example, we also see hill climbing perform significantly better than both greedy and genetic algorithms. It was also able to delete chunks of code missed by both. HC produced a slice similar but not identical to random search. LinIAS produced an amorphous slice that was also considerably smaller than the greedy and genetic algorithms as shown in Figure 4.23. Crucially, we see that the slice produced by LinIAS appears to be much worse than those by random search and HC. However, upon closer inspection of the slice in Figure 4.23, we note that the chunk highlighted in blue can be deleted as it appears after the assignment:

\[ \text{slice} := \text{personal} \]

which is the slicing criterion.

Although hill climbing, random search and LinIAS produced good slices w.r.t. variable \texttt{personal} as the slicing criterion, none of the three techniques returned the smallest possible slice (best solution). Figure 4.24 shows the optimum slice for the problem, constructed by a human. All three techniques missed the following statements:

\begin{verbatim}
married := 1;
WHILE married <> 0 AND married <> 1 DO
married := read; read := read + next OD;

widow := 1;
WHILE widow <> 0 AND widow <> 1 DO
widow := read; read := read + next OD;
\end{verbatim}
Figure 4.20: Output code slice for tax program returned by the Genetic Algorithm. Slicing criterion is variable personal at end of program.
Figure 4.21: Output code slice for tax program returned by Random Search. Slicing criterion is variable `personal` at end of program.
age := 40;
WHILE age < 0 DO age := read; read := next + read OD;
income := 23000;
WHILE income < 0 DO
    income := read; read := next + read OD;
married := 1;
WHILE married <> 0 AND married <> 1 DO
    married := read; read := next + read OD;
widow := 1;
WHILE widow <> 0 AND widow <> 1 DO
    widow := read; read := next + read OD;
blind := 0;
WHILE blind <> 0 AND blind <> 1 DO
    blind := read; read := next + read OD;
IF age >= 75
    THEN personal := 5980
ELSE IF age >= 65
    THEN personal := 5720
    ELSE personal := 4335 FI FI;
IF income > 16800 AND age >= 65
    THEN IF personal < 4335 THEN personal := 4335 FI FI;
IF blind = 1
    THEN personal := personal + 1380 FI;
slice := personal

Figure 4.22: Output code slice for tax program returned by Hill Climbing Algorithm. Slicing criterion is variable personal at end of program.
```plaintext
age := 40;
income := 23000;
WHILE age < 0 DO
  age := read; read := next + read OD;
married := 1;
blind := 0;
WHILE income < 0 DO
  income := read; read := next + read OD;
widow := 1;
WHILE married <> 0 AND married <> 1 DO
  married := read; read := next + read OD;
WHILE widow <> 0 AND widow <> 1 DO
  widow := read; read := next + read OD;
IF age >= 75
  THEN personal := 5980
  ELSE IF age >= 65
    THEN personal := 5720
    ELSE personal := 4335 FL FL;
WHILE blind <> 0 AND blind <> 1 DO
  blind := read; read := next + read OD;
IF income > 16800 AND age >= 65
  THEN IF personal < 4335
    THEN personal := 4335
    ELSE personal := personal FL FL;
IF blind = 1
  THEN personal := personal + 1380 FL;
slice := personal;
IF married = 1 AND age >= 75
  THEN pc10 := 6692
  ELSE IF married = 1 AND age >= 65
    THEN pc10 := 6625
    ELSE IF married = 1 OR widow = 1
      THEN pc10 := 3470
      ELSE pc10 := 1500 FL FL;
IF married = 1
  AND income > 16800
  AND age >= 65
  THEN IF pc10 < 3470
    THEN pc10 := 3470
    ELSE pc10 := pc10 FL FL;
IF income > personal
  THEN income := income - personal FL
```

Figure 4.23: Output code slice for tax program returned by LinIAS. Slicing criterion is variable `personal` at end of program. 

104
age := 40;
blind := 0;
income := 23000;
WHILE age < 0 DO
  age := read; read := next + read OD;
IF age >= 75
  THEN personal := 5980
ELSE IF age >= 65
    THEN personal := 5720
    ELSE personal := 4335 FL FL;
WHILE income < 0 DO
  income := read; read := next + read OD;
WHILE blind <= 0 AND blind <= 1 DO
  blind := read; read := next + read OD;
IF income > 16800 AND age >= 65
  THEN IF personal - (income - 16800) / 2 < 4335
     THEN personal := 4335
     ELSE personal := personal - (income - 16800) / 2 FL FL;
IF blind = 1
  THEN personal := personal + 1380 FL;
slice := personal

Figure 4.24: Output code slice for tax program constructed by human. Slice represents the optimum solution for the program based on variable personal as slice criterion.
4.4.5 A Surprising Result

It has been widely observed that search techniques are good at producing unexpected answers. This happens because the techniques are not hindered by implicit human assumptions. One example is the discovery of a patented digital filter using a novel evolutionary approach [98]. The other example is the discovery of patented antenna designs [76] which are available commercially. The human formalises their (explicit) assumptions as a fitness function. The machine uses this fitness function to guide the search. Should the search produce unexpected results then this reveals some implicit assumptions and/or challenges the human’s intuition about the problem.

Unlike human–based search, automated search techniques carry with them no bias. They automatically scour the search space for the solutions which best fit the (stated) human assumptions captured by the fitness function. This is one of the central strengths of the approach.

In a small way, this insight-yielding advantage of search was observed in our experiments on search–based amorphous slicing. For one of the examples — the braking system controller program in Figure 4.16 — the amorphous slice found by the search techniques was smaller than that produced by the analytic algorithm, LinIAS. This seemed impossible at first, because the analytic algorithm is designed to produce excellent results with precisely the type of programs under study. Closer examination revealed that the syntax–preserving slices (upon which LinIAS is based) were over-large. For the particular program and slicing criterion, LinIAS therefore missed a chunk of code that could be deleted. The search based approach did not miss this and as a result produced a smaller amorphous slice than LinIAS.
4.4.6 Statistical Analysis

Figure 4.25\(^2\) shows the result of the statistical analysis test on the output data from the experiment. We performed the Wilcoxon pair test on related samples. Figure 4.25 compares the significance of the results between a pair of techniques used for constructing the amorphous slices. In the Surface Area test program, we observe that there is no significant difference between the results produced by both Genetic Algorithm and Greedy Algorithm. However, if we observe the Tax test program, we notice that there is a significant difference between the results produced by both Genetic Algorithm and Hill Climbing, a 99.8% confidence level. Consider the Braking Controller test object, we notice an 86.2% confidence level in the results from HC and random. Generally, confidence values lower than 95% are not considered statistically significant.

4.4.7 Conclusions of Experimental Study 2

Experimental Study 2 shows that transformation sequences can be generated that produce valid amorphous slices with respect to a particular slicing criterion through the use of search-based heuristics.

Greedy algorithm outperforms other heuristic approaches employed in this experiment and performs well when compared with LinIAS. However, it is important to note that although it did not conclusively perform better than LinIAS, the results are not disappointing. Considering the results from random search as a base line and those from LinIAS as the benchmark standard, the greedy algorithm produces results.

\(^2\)GA = Genetic Algorithm, HC = Hill Climbing, Gr = Greedy Algorithm, Li = LinIAS, Ra = Random Search
Search-Based Transformations

Amorphous Slicing

<table>
<thead>
<tr>
<th></th>
<th>GA v. Gr</th>
<th>GA v. HC</th>
<th>HC v. Gr</th>
<th>GA v. Li</th>
<th>Gr v. Li</th>
</tr>
</thead>
<tbody>
<tr>
<td>S. Area</td>
<td>1.000</td>
<td>.066</td>
<td>.066</td>
<td>.066</td>
<td>.066</td>
</tr>
<tr>
<td>OddEven</td>
<td>.001</td>
<td>.002</td>
<td>.154</td>
<td>.001</td>
<td>.001</td>
</tr>
<tr>
<td>Tax</td>
<td>.005</td>
<td>.002</td>
<td>.008</td>
<td>.004</td>
<td>.317</td>
</tr>
<tr>
<td>Calendar</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
</tr>
<tr>
<td>Defroster</td>
<td>.063</td>
<td>.016</td>
<td>.138</td>
<td>.007</td>
<td>.001</td>
</tr>
<tr>
<td>Marking</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
</tr>
<tr>
<td>Braking Clr.</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>HC v. Li</th>
<th>GA v. Ra</th>
<th>HC v. Ra</th>
<th>Gr v. Ra</th>
<th>Li v. Ra</th>
</tr>
</thead>
<tbody>
<tr>
<td>S. Area</td>
<td>1.000</td>
<td>1.000</td>
<td>.066</td>
<td>1.000</td>
<td>.066</td>
</tr>
<tr>
<td>OddEven</td>
<td>.046</td>
<td>1.000</td>
<td>.002</td>
<td>.001</td>
<td>.001</td>
</tr>
<tr>
<td>Tax</td>
<td>.005</td>
<td>1.000</td>
<td>.002</td>
<td>.005</td>
<td>.004</td>
</tr>
<tr>
<td>Calendar</td>
<td>.000</td>
<td>1.000</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
</tr>
<tr>
<td>Defroster</td>
<td>.009</td>
<td>.001</td>
<td>.001</td>
<td>.012</td>
<td>.001</td>
</tr>
<tr>
<td>Marking</td>
<td>.000</td>
<td>.180</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
</tr>
<tr>
<td>Braking Clr.</td>
<td>.000</td>
<td>.000</td>
<td>.180</td>
<td>.000</td>
<td>.000</td>
</tr>
</tbody>
</table>

Figure 4.25: Results of Wilcoxon significance statistical tests. The pair analyses significant differences between results from pairs of techniques.

closer to the benchmark than the baseline. Thus, there is great optimism for the improvement and refinement of the technique.

4.5 Summary

In this chapter the search techniques were used to compute valid amorphous slices for test programs using every defined variable in the test program as the slicing criteria. The problem of using search to compute these amorphous slices was reformulated into a simpler problem of redundancy removal by reconstructing the test program. The chapter showed examples illustrating the transformation of the test program into the reconstructed program. Existing search-based transformation techniques are used to find transformation sequences on the reconstructed programs. The results from the
experiment show that search can be used to produce valid amorphous slices. We present one case where a search technique outperforms an analytical approach albeit due to a bug in the implementation of the analytic algorithm which the search based approach highlighted.

The experiment presented successfully uses search techniques guided by a single objective fitness function. The subject of the next chapter is the application of search to another problem domain, which requires a multi-objective approach. An analysis of this domain and the application of search algorithms within them is given in Chapter 5.
Chapter 5

Variable Proximity based Program Comprehension

In our previous experiment in Chapter 4, we used search techniques to compute valid amorphous slices by searching for sequences of reducing transformations. We then compared the results from the experiment with those obtained by computing amorphous slices from LinIAS, a tool geared specifically towards computing amorphous slices. In this chapter, we will look at another application area for program transformation. We examine how program transformation may be used to improve program comprehension by reducing the proximity between variable definition and use. These transformations are carried out using our Genetic Algorithm (GA), Hill Climb Algorithm (HCA) and Greedy Algorithm (GrA). The results are discussed in the following sections.
5.1 Introduction

One of the first tasks a software maintainer is required to carry out is to try and understand a piece of software, usually written by someone else. This is even before any other activities can be performed and is a notoriously hard task to achieve. It is hard because, in some sense it involves having to retrospectively get into the mind of the original developer, to decide what they were thinking when they wrote a particular section of code. The comprehension of a piece of code is further hampered if it is written in a style which is not too obvious. For instance, many developers write sections of code that perform multiple calculations almost within the same block of code resulting in interleaved code [96]. Therefore, understanding such code sections would involve disentangling the interwoven pieces of code to reveal code that is more contiguous and elegant.

Many psychological factors affect the software development process. They can affect the way in which a system is perceived and used by its user community and the way in which the system is developed and managed by the development community. Miller [85] in his famous experiment describes the magic ‘number’: $7 \pm 2$, which is the Short Term Memory (STM) capacity for a large proportion of the human population. This number is replicated whether the subject tried to recall numbers, letters or words. Psychological researchers use the word ‘chunk’ to refer to the unit of data stored in short term memory.

There are several artefacts present after the complete design, implementation and deployment of a system that later serve as tools for comprehending what the system does. In understanding the overall functions of a system, it is of course important to peruse the requirement specifications, detailed and abstract design documents
that describe the system architecture in some detail, implementation manuals, etc. However, after several evolutions of the system, it is common to observe that these documents become less useful. Therefore, the source code is usually the last source of documentation for the system and it is not uncommon for maintainers of systems to decipher code functionality purely from the source files. This is further demonstrated by an increase in research into reverse engineering source code to generate higher level abstraction artefacts such as the design documents, code visualizations, mental models, etc., all in order to improve the comprehensibility of the code [85].

When trying to understand how a program works, we base most of our understanding on the appearance and position of variable names in our program. For example, when we see an \texttt{x} on the left side of an assignment operator, we realize that some value is been stored in it and when we see \texttt{x} on the right side of an assignment operator, we know we are using the value from \texttt{x} in some form of computation.

**Recency**

Recency refers to how soon after seeing an item we observe another reference to the same item. Our ability to recall what an item means improves with the nearness of the most recent reference to that item.

Our ability to recall what may have happened to the value contained in a variable name such as \texttt{x} or indeed recall what the variable name stands for depends largely on how recently / frequently we encountered that variable name while reading the source code. Naturally, we read source code from top to bottom keeping in our STM, variable names we see along the way.
5.2 Problem and Motivation

When we analyze code, we want to observe the flow of data through variable names along the program. The challenge of the study reported in this chapter is to design a transformation system that re-organizes the structure and position of variable identifiers, such that, the flow between a definition and corresponding use of a variable identifier is reduced in distance; uses for variable identifiers are brought as close as possible to their definitions. This is based upon the premise that comprehension of the source code is improved by observing similar things sooner and thereby aiding memory retention [69]. The experiment aims to verify whether these sorts of reconstructions can be performed using search-based approaches. The motivation for using search-based heuristic techniques remains the presence of a large search space that cannot be traversed exhaustively.

5.3 Transforming for Comprehension

It is widely believed that one of the causes of a lack of understanding of programs is interleaving [96], which is the weaving together of code with multiple purposes into a single section. This may sometimes occur as a result of some optimisations where the coder wants to exploit efficiency gains. An example of such might be the implementation of a single loop to calculate the sum and product of array elements. This interleaving may make it harder to understand the source as the reader is required to keep track of at least two calculations. We realise that the simplest means of performing calculations and keeping track of values through code is by way of the variable name. A desired characteristic for well written programs is the employment
of meaningful variable names that give an indication of their purpose, type, bounds, etc.

We challenge our short-term memory severely to remember important characteristics such as type and use regarding variable identifiers when these variable identifiers appear in several sections of code separated by large gaps between each occurrence. This may be a sign of the presence of interleaved code.

The problem of remembering and consequently understanding the purposes of a variable clearly multiplies when the number of variables under consideration increases and can be extremely difficult even for fairly simple straight line programs. In the example in Figure 5.1, we see in progA, code with nine different variable names. Take for instance the variable w defined on line 1. Looking down the code lexically, we only see a reference to w again on line 6 after seeing references to a, b, foo, c, d, b, x and y. Further reference to w occurs on line 9 after more references to a, z, foo, i, op and b. These interleavings at best might be benign and not in any sense hinder comprehension but even in an average case would confuse the casual code observer. ProgB is a transformed version of ProgA which attempts to minimize the interleaving between each variable name. References to w found on lines 1, 6 and 9 in progA have been brought as close together as possible and similarly for other variables represented in the source. We conjecture, based on elements of cognitive psychology, that progB would be more easily comprehensible than progA or at least it should be no worse than progA.
1. \( w := a + b; \)
2. \( \text{foo} := c + d; \)
3. \( b := x + 1; \)
4. \( x := x + 1; \)
5. \( x := x + 1; \)
6. \( y := w - a; \)
7. \( z := \text{foo} + i; \)
8. \( \text{op} := b; \)
9. \( \text{print} (w); \)

\( \text{progA} \): Source Program with interleaved define-use variable name positions.

<table>
<thead>
<tr>
<th>1. ( w := a + b; )</th>
<th>1. ( w := a + b; )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. ( \text{foo} := c + d; )</td>
<td>2. ( y := w - a; )</td>
</tr>
<tr>
<td>3. ( b := x + 1; )</td>
<td>3. ( \text{print} (w); )</td>
</tr>
<tr>
<td>4. ( x := x + 1; )</td>
<td>4. ( \text{foo} := c + d; )</td>
</tr>
<tr>
<td>5. ( x := x + 1; )</td>
<td>5. ( z := \text{foo} + i; )</td>
</tr>
<tr>
<td>6. ( y := w - a; )</td>
<td>6. ( b := x + 1; )</td>
</tr>
<tr>
<td>7. ( z := \text{foo} + i; )</td>
<td>7. ( \text{op} := b; )</td>
</tr>
<tr>
<td>8. ( \text{op} := b; )</td>
<td>8. ( x := x + 1; )</td>
</tr>
<tr>
<td>9. ( \text{print} (w); )</td>
<td>9. ( x := x + 1; )</td>
</tr>
</tbody>
</table>

\( \text{progB} \): Transformed Program with similar variable names in close proximity.

Figure 5.1: Program transformed for comprehension

### 5.4 Context Sensitive Comprehension

Clearly the way in which the reader reads and understands a piece of code depends on a number of factors such as their job level / description, level of experienced, motivation for performing comprehension activity, etc. A more experienced programmer might understand a previously unobserved piece of code by recognising patterns in the code which the novice programmer might not. This fact was observed in studies, which showed that expert programmers used a strategy of reading a program in the order in which it would be executed [32]. Also, we might pre-suppose that we always read code lexically from top to bottom which might not be the case. For example, a reader trying to understand the functionality of an IF -statement on recognizing that the guard condition is FALSE intuitively skips reading the THEN branch of the code and naturally continues from the ELSE branch. Therefore, when we analyze how readers comprehend code, it is important to factor into the reasoning process, the background and psychology of the reader. We suppose there are at least two methods

115
for reading through source code:

1. through the flow of data - we refer to this as DU-fitness.

2. lexically.

The next two sections introduce candidate fitness functions to assess optimality with regard to these two reading methods.

### 5.4.1 Define-Use (DU) Based Fitness

The DU-based fitness measure calculates the distance between a definition (an assignment) to a variable and a corresponding subsequent use of that variable in the code. In this work, we make no distinctions to the kinds of use for the variable, therefore the use of the variable might be a computational use, for example on the right side of an assignment operation or a predicate use in a conditional predicate. This measure assumes that when we read code, what we want to observe is the flow of data through a variable along the program. Consider the example in Figure 5.2. Observe an assignment to the variable name \( w \) on line 1 and the next use for \( w \) is a computational one on line 6. We say we have a DU pair because there has not been any other redefinitions to \( w \) in the intermediate lines between [2]. This DU pair (for \( w \) in lines 1 and 6) has a distance value of 5, which is the number of lines between the definition and use.

Therefore, the fitness function is constructed to minimize this distance between DU pairs for a variable. If the transformations that reduce distances in DU pairs are possible, which means the program can be reconstructed without breaking any dependencies, then such transformations are valid and executed.
Rather obviously, with this approach, there is an assumption that program comprehension is enhanced, the closer a use to its point of definition.

5.4.2 Lexical Based Fitness Function

The lexical based fitness measure calculates the distance between any mention of the variable of interest. Here, we do not distinguish between an assignment to a variable and a use for that variable. Rather, this measure simply scans the source file lexically and measures the distances between identifiers with similar names. This is based upon the premise that comprehending the source file is improved by observing similar things sooner aiding memory retention. This measure assumes that the code observer, simply reads the code in lexical order and perhaps might not need to do any kind of data-flow analysis. In Figure 5.3, we show an example where we transform a sample program, minimizing the distance between different references to the variable \( w \).
5.5 Search Based Heuristics

In trying to achieve our goal of transforming programs to enhance comprehension, we intend to use search based heuristics as a means of optimization. In order to employ search techniques, we are required to formulate a fitness function which would guide the search hopefully in a direction which produces optimal (or as close to optimal as possible) results. As before, we have a large search space that cannot be explored exhaustively.

5.5.1 Multi-Objective Optimization

Because we are not simply interested in optimizing for a single variable identifier such as \( x \) in Figure 5.4 but rather for a whole range of identifiers - references to all variable identifiers in the source code, our transformation becomes a multi-objective problem. The search goal is to find transformation sequences that minimize the distance between references to several variable identifiers. This usually results in a competition...
for good optimizations between the different criteria. A positive optimization position for one identifier might threaten to jeopardize an already good optimization for another identifier. Consider the example in Figure 5.4 and imagine we were interested in optimising the distances for variables $w$ and $m$. By swapping lines 6 and 7, we optimise for $m$ while making $w$ worse.

Typically when such scenarios arise, the search procedure finds a compromise solution. This solution is based on some predefined measure for what constitutes an optimal solution. Some multi-objective search problems are resolved by computing the different objective values and combining each in some way based on some prior reasoning about the domain and computing a single overall fitness score.
5.6 Experimental Study 3: Measures of variable proximity for program comprehension

This section describes the experimental studies performed, including the design, subjects, measurements and results.

Given a source program $P$, a list of assigned variables $A$ in $P$ and a list of used variables $U$ in $P$, the objective of this experiment was to find a sequence of transformation rules $T$ that would transform $P$ into a program $P'$ such that the DU distance measure for two variables, $v_i$ and $v_j$ is smaller in $P'$ than $P$. Variables $v_i$ and $v_j$ are elements of $A$ and $U$.

We define the distance between a Define-Use pair for an identifier $x$ as the number of statements between a Define (an assignment) to $x$ and a reference to $x$ provided there are no other intermediate definitions to $x$.

An important point to note for this experiment is that the transformations required are only statement re-ordering transformations and not those which perform any other kinds of simplifications to the program. In essence, both $P$ and $P'$ should have an identical number of line of code LoC. Of course, a possible counter-example is the following transformation:

\[
\begin{align*}
v & := y; \\
z & := z + 1; \\
z & := z + 1; \\
\text{foo} & := v;
\end{align*}
\]

\[
\begin{align*}
v & := y; \\
z & := z + 1; \\
z & := z + 2; \\
\text{foo} & := v;
\end{align*}
\]

However, transformations capable of merging statements are not permissible. The aim is to improve comprehension only by statement re-ordering. A post-condition of
the transformation for comprehension is that the number of lines of code for both the source and transformed versions of the test object be identical.

As with experiments already described in previous chapters, the search algorithms described were implemented in the WSL using the transformation rules from the FermaT workbench [114].

5.6.1 Experimental Setup

We examined the application of search–based methods to find transformations that would optimise programs and generate functionally equivalent programs that have a reduced distance between a DU pair for pre-identified variable identifiers.

The following search methods were implemented during this experiment: a Genetic Algorithm, a Hill-Climb Algorithm and Random Search.

Applying Genetic Algorithms

We apply our population based genetic algorithm to traverse the search space for possible good solutions, that is, good transformation sequences that may be applied to our source program.

Applying Hill Climbing

We apply a hill climb algorithm to search through the search space of possible solutions to find a sequence of transformations that would satisfy our fitness requirements.
Applying Random Search

We apply random search as a base line technique. Fixed length sequences of transformations are generated at random and their fitness values evaluated.

Implementation

The implementation details for the GA include a fixed sequence length of 40 genes per individual, a high mutation rate of 30%, constant size population of 100 individuals and the algorithm was run over 600 generations. We used tournament selection technique to select two parents to mate during the process of crossover to produce a single offspring, which, if better than at least one parent, replaces the worse parent in the population. We employ an elitist strategy that keeps the individual with the best fitness score across all generations. The results presented are averaged over 10 runs with a new random seed for each iteration.

The Hill–Climb algorithm was implemented to search a fixed-length sequence of 40 genes per individual. Each gene represents a transformation rule. The HC algorithm uses first-ascent technique, which selects the first best neighbouring solution found. The neighbourhood is described as the mutation of a single transformation rule to any other permissible transformation rule. The HC algorithm performs 20 restarts with a new random start and the best solution over the different restarts is noted. The results presented are averaged over 10 runs of the HC algorithm with a new random seed for each run.

Lastly, a random search for transformation sequences is also implemented using identical metrics, such as a fixed-length sequence of 40 genes. The best solution after 20,000 random iterations is selected. The results presented are averaged over 10 runs.
The algorithms were executed on a Linux operating system, Pentium 4 with a processor speed of 3GHz.

5.6.2 Test Objects

The experiment was performed on a specially constructed ‘toy’ program to investigate the problem characteristics. The experiment was constructed to examine the relationship between the movement of variable identifiers in a program. The following are assumptions which hold true for the test programs used in this study:

- We care only for a single DU pair per variable.
- We care only about computational uses for variables and not predicate uses.
- Where an identifier occurs as a use in a computational sense, that is, on the right side of an assignment, we assume that it occurs alone on the right side of that assignment statement.

During the experiment, each of the three algorithms described was tested using the ‘toy’ program. The optimisation was with respect to two variable identifiers in the source code.

5.6.3 Fitness Function

We measure the fitness of a potential solution by measuring the distance between the definition and use for all the variables of interest. Subjective weightings are placed on each distance value and are part of the research questions to be investigated.

The fitness of each solution is given by:
\[ \text{distance}(\alpha) = |\text{def}(\alpha) - \text{use}(\alpha)| \]
\[ \text{distance}(\beta) = |\text{def}(\beta) - \text{use}(\beta)| \]
\[ \text{fitness}_{\text{sequence}} = (\text{distance}(\alpha) \ast C1) + (\text{distance}(\beta) \ast C2) \]

where \( C1 = C2 = 0.5 \)

The function is a minimizing function with solution \( A \) better than solution \( B \), if and only if, \( \text{fitness}(A) < \text{fitness}(B) \).

### 5.6.4 Results and Discussion

The experiments involved one WSL program as test case (Appendix A.8). Each search algorithm was executed 10 times.

**x, y Equal Weighting**

Figure 5.5 show the graphs of the fitness metrics with equal weightings for both \( x \) and \( y \) variable identifiers. The line graph depicts the plot of the average fitness value scored by an algorithm on the y-axis against the number of fitness evaluations by each algorithm on the x-axis. We expect that there ought to be a reduction in the fitness score the further along an algorithm is in its execution.

Therefore, a single point on the graph shows the average fitness score for each sequence of transformations found after a number of fitness evaluations, as the algorithms move along their execution path. The subject program initially had an equal distance measure between the definitions and uses for both variables \( x \) and \( y \).

As the graph in Figure 5.5 indicates, there is a substantial difference between the mean fitness values returned by the GA and those obtained by both HC and
RS. However, there is no apparent, noteworthy difference in the trend of results between the Hill Climb and Random Search although certain individual results from HC appear slightly better than those from Random Search. The box-plot in Figure 5.5 illustrates the distribution of the results over the entire duration of execution for each algorithm. Clearly, we see why the GA outperforms HC and RS. Over the duration of execution, the GA is able to sample more solutions within the search space which have better fitness scores. HC and RS appear to sample a similar region within the search space. We observe this trend repeated in other experiments carried out in this work.
variable x bias

In this experiment, we increased the weightings on variable x. The aim of this was to empirically discover the effects of the coefficients of a multi-objective optimisation problem on an aggregated singular objective function. In this experiment, the fitness equation is given by the following:

\[
\text{distance}(x) = |\text{def}(x) - \text{use}(x)| \\
\text{distance}(y) = |\text{def}(y) - \text{use}(y)| \\
\text{fitness}_{(\text{sequence})} = (\text{distance}(x) \times 0.8) + (\text{distance}(y) \times 0.2)
\]

Figure 5.6 shows the performance of each of the three search algorithms. We observe that again, as with an equal bias, the genetic algorithm outperforms both the hill climb algorithm and random search. In fact, the trend in both line graphs (Figures 5.5 and 5.6) can be seen to be very similar.

variable y bias

In this particular experiment, we reversed the weighting given to each variable of interest. Variable x was allocated a 20% weight, while variable y had an 80% weight. The fitness value for a sequence is thus given by the following:

\[
\text{fitness}_{(\text{sequence})} = (\text{distance}(x) \times 0.2) + (\text{distance}(y) \times 0.8)
\]

Figure 5.7 shows the graph for the y-bias experiment. We observe again a similar trend in the results as shown by the other previous graphs. This provides evidence of the GA’s dominance for this particular kind of optimisation. However a point to note, is that while the graphs follow a similar trend pattern, they are not exactly the same for both the x-bias and y-bias experiments.
Since the test program had the same distance measure between both x and y references, the initial expectation was that presenting x with the same high distance reduction chance as y should provide identical results. We note however, that further along the execution of the genetic algorithm, that the GA provides more individuals with a reduced (thus better) fitness scores in the y-bias experiment than in the x-bias. This is because the GA selects transformation rules that result in a better score for the y-bias optimisation than for x-bias.

**Pareto Optimality**

We also experimented with Pareto Optimality as an alternative approach [28, 37]. Here, rather than aggregate the scores for each of our variables of interest into a single fitness value, we separate out our concerns as a tuple and aim to optimise both
halves of the tuple simultaneously.

Re-using the distance equations previously shown as:

\[ distance(\alpha) = |def(\alpha) - use(\alpha)| \]
\[ distance(\beta) = |def(\beta) - use(\beta)| \]

We say that the fitness of a solution \( f_i \) is given by the tuple, \((distance(\alpha_i), distance(\beta_i))\) and that:

\[ f_i \geq f_j \iff distance(\alpha_i) \geq distance(\alpha_j) \land distance(\beta_i) \geq distance(\beta_j). \]

Figure 5.8 shows the graph of all points on the x and y plane. The graph plots the distance measure for variable y on the y-axis and the distance measure for variable x on the x-axis. Therefore, a single point on the graph represents the x and y distance value obtained as a result of the application of a single sequence of transformation rules. We can see that more points from the GA’s execution dominate\(^{1}\) those from both the HC and Random Search. The GA appears to move the pareto front further along than either of the other techniques. We notice that no GA point on the pareto front is dominated by any points obtained from the execution of the other searches.

---

\(^{1}\)A point on the graph is non-dominated if no other point appears both lower and to its left.
Figure 5.7: Graph of y-bias mean fitness results

Figure 5.8: Pareto Front for the three search based techniques.
For this single execution, we notice that hill climbing performs better than the ge-
in this section, we show present examples of the output programs produced from a
The input test program (before transformation occurs) is shown in Figure 5.9. In this
program, the goal was to bring the assignments and references to variables $x$ and $a$
as close together as possible by repeatedly swapping positions with the assignments
$y := c$. The output programs described in this section were transformed using an
aggregated fitness function, with equal weightings for both variables to be optimised.

Figure 5.10 shows the output program produced by the search based techniques.
For this single execution, we notice that hill climbing performs better than the ge-
etic algorithm. From the figure, hill climbing moved the assignment to $a$ ($a := f$)
much further along than the GA, while the position of $x := e$ is no worse in HC
than GA. By using the x/y equal bias aggregated function, we observe that the outputs produced by both random search and hill climbing have identical fitness scores. These two outputs would be incomparable using pareto fitness function because neither algorithm makes the position of both variables x and a better than the other. Hill Climbing improves variable a compared with random search but the position of variable x is worse in hill climbing than random search.

<table>
<thead>
<tr>
<th>y := c;</th>
<th>y := c;</th>
<th>y := c;</th>
<th>y := c;</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := e;</td>
<td>x := e;</td>
<td>x := e;</td>
<td>x := e;</td>
</tr>
<tr>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
</tr>
<tr>
<td>a := f;</td>
<td>a := f;</td>
<td>x := e;</td>
<td>y := c;</td>
</tr>
<tr>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
</tr>
<tr>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
</tr>
<tr>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
</tr>
<tr>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
</tr>
<tr>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
</tr>
<tr>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
</tr>
<tr>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
</tr>
<tr>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
</tr>
<tr>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
<td>y := c;</td>
</tr>
<tr>
<td>y := a;</td>
<td>y := a;</td>
<td>y := a;</td>
<td>x := a;</td>
</tr>
<tr>
<td>z := x</td>
<td>z := x</td>
<td>z := x</td>
<td>z := x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>programA</strong></th>
<th>Genetic Algorithm</th>
<th><strong>programB</strong></th>
<th>Hill Climbing</th>
<th><strong>programC</strong></th>
<th>Random Search</th>
<th><strong>programD</strong></th>
<th>Human Transformation</th>
</tr>
</thead>
</table>

Figure 5.10: Comparison of output programs produced by the GA, HC, RS and Human transformation.

None of the three search based techniques found the optimum solution. There are a number of possible optimum solutions to this particular optimisation problem and in Figure 5.10, **programD**, we show one such optimum output constructed by a
human.

5.6.6 Statistical Analysis

Figure 5.11 shows the result of performing the Wilcoxon test for significance on our sample data. We observe that the results are strongly significant for each of the three different cases (a-bias, x-bias and even-bias). This shows that GA clearly and significantly outperforms random search and HC at the 100% level of confidence.

<table>
<thead>
<tr>
<th></th>
<th>GA v. Random</th>
<th>GA v. HC</th>
<th>Random v. HC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y-Bias</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
</tr>
<tr>
<td>X-Bias</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
</tr>
<tr>
<td>Even Bias</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
</tr>
</tbody>
</table>

Figure 5.11: Results of Wilcoxon significance tests showing strong significance in the results with the GA outperforming both random search and the HC.

5.7 Summary

In this chapter, the search techniques were used to provide solutions to a multi-criteria optimisation problem. We showed how a solution can be represented and performed experiments on a 2-variable dimension, where the aim was to reduce the distance between the node of definition for a variable and its corresponding node of reference. The experiments were performed using two approaches: an aggregated scalar fitness function as a guide for the search and a pareto optimality metric.

The results from the experiments and the analysis indicate that the Genetic Algorithm performs much better than Hill Climbing Algorithm and Random Search. However, there was no significant difference found between the performance of the
Hill Climbing Algorithm and Random Search. The choice of different weightings for the distance measures for each variable of interest did not result in any significant difference in the efficiency / effectiveness of the algorithms.
Chapter 6

Reflections

6.1 Introduction

This chapter provides some general observations regarding the findings of the thesis. It includes general reflections about methods and alternative techniques that might have been implemented within the framework of the studies carried out. Here, we explain some of the decisions taken in completing the research. The hope is that this will, firstly, explain why specific methods and directions were taken. Secondly, it is hoped that these comments will point to some areas of future research.

At the outset, the initial problem we addressed was how to construct a general purpose automated program transformation system which was based upon a search framework. We soon discovered that it was a hard problem, partly because of the nature of the transformation rules with which we sought to experiment.
6.2 Kinds of Transformations

In earlier chapters, we described the choice of transformation rules that were available to the search framework. We chose not to include rules that affect the entire program in a singular operation. In some programming language implementations and particularly in object code optimisations [30], it is possible to have optimisation rules that have a global optimisation characteristics. Consider the following example:

```
1. x := x + 1;
2. x := x + 1;
3. y := y + 1;
4. z := z + 1;
5. x := x + 1;
6. x := x + 1;
7. x := x + 1;
8. z := z + 1;
```

At a macro level, it may be possible to have a single transformation rule `[simplify-code]`, which takes the above fragment and returns the following output:

```
1. x := x + 5;
2. y := y + 1;
3. z := z + 2;
```

In a sense, a subgoal was to strip off from the transformation rules, such near omni-potent capabilities. Such a rule as above, would have the ability to simplify the computation on the variables x, y and z simultaneously. In furtherance of another of the stated subgoals (Chapter 3, Section 3.4), which was the desirability of good `sub-sequences`, which serve as building blocks, we took the view that in this particular instance, the end would not necessarily justify the means. That is to say that, while
the above example transformation rule might in practice produce simpler code rather quickly, in principle, it fails to encapsulate the ideas behind search-based approaches, one of which, is the generation of these sub-sequences.

The presence of these macro level transformation rules capable of performing huge optimisation in a single step reduces the ability of the search procedure by generating a fractious landscape. We consider this effect unacceptable for the kinds of investigation required in this dissertation.

Consequently, we chose to search for transformation rules that only optimise a little at a time. These allow the search landscape to have a better structure and enable the fitness function to form a trajectory through the landscape. By grouping a collection of these ‘small-power’ transformations together, we are able to achieve similar effects as the more powerful, global transformation rules.

One of the challenges for the search methods was to locate portions of code where optimisations could be performed. For example, consider the following code fragment:

```
1. a := a + 1;
2. b := b + 1;
3. y := y + 1;
4. z := z + 1;
5. x := x + 1;
6. x := x + 1;
7. x := x + 1;
8. z := z + 1;
```

In order for code simplifications to have taken place with the appropriate transformation rule, it was necessary that the cursor position was situated at a ‘fertile’ point. Observe that no optimisations are possible from lines 1 through to 4. It is only after we arrive at line 5 that an optimising rule can be applied to simplify line
search space.

One clear alternative approach to using ‘navigational’ transformations rules, would have been to determine statically the node to which each transformation rule could be applied. Williams [120] employed this approach when searching for loop re-ordering transformations (see Chapter 3). Whilst absolute referencing might have been appropriate for the experiments he carried out, we believe that they would not prove as expedient in our case. This is because for absolute referencing to be worthwhile, it would require an initial pre-processing phase where the correct points for optimisation are statically analysed and recorded. We see no reason for this extra iteration but rather build the location of suitable addresses into our search procedure by navigating freely through the program. In a small way, absolute referencing is against the principle of an un-hindered, near-natural search because of the restrictions it places on points of optimisations.

6.3 Choice of Search Algorithms

In the search literature, there is a myriad of ‘competing’ search techniques for performing combinatorial optimisation problems. We use the term ‘competing’ search techniques rather loosely. It has been suggested that no particular technique ought to be in competition with the others. Rather, that search approaches can complement each other in a number of ways.

The techniques in search are classed as global, local or greedy and we chose to
implement one algorithm from each broad classification. We implemented a genetic algorithm, a hill-climbing algorithm and a systematic search algorithm. These design decisions, taken at the onset of the research, were intended to facilitate a fair comparison between the different classes and techniques. From the results found, we state that an understanding of the problem description should inform the choice of search algorithm to be implemented. There is usually a bias against local search techniques. A knee-jerk solution to most search problems tended to be the introduction of a genetic algorithm implementation. Rana [95] show that genetic search tend to be hindered by similar problems that affect local search. Mathias and Whitley [80] also suggest that local search outperform genetic search for certain classes of problems. Furthermore, as has been shown by Mahdavi [77], we found that local search techniques such as hill climbing with multiple restarts worked well - sometimes better than genetic algorithms.

It is our opinion, where possible, no algorithm should be used in isolation. This means that when the problem description is well understood, each algorithm may be used at different stages of solving the problem. For instance, a genetic algorithm could be used at the start, to search through a huge population. The result of this process may then be fed into a hill-climber and so on. The outcome of this is a hybridized algorithm with an increased probability of finding excellent solutions in a landscape where the chances of finding the global optimum are low.
6.4 Generic Solution

The motivation for this research and subsequent direction was finding a generic solution to program transformation. The idea was that any code-level software metric could be encoded as a fitness function and used to drive a search mechanism that would produce good enough transformation sequences. As a start and to provide some initial results and justification for the approach, we decided to experiment with LoC optimisation. We still believe conceptually that any formalisable metric for code such as McCabe’s cyclomatic complexity metric [81], Halstead metric [50], etc., may be used as search drivers for good solutions. For example, any sequence of transformation which produces a functionally equivalent program with a lower McCabe cyclomatic complexity value may be considered better than one with a higher value.
6.5 Fitness Function

During the design of experimental study 1 (Chapter 3), we implemented a fitness function that measured the sizes of the actual programs i.e., the input and output programs from the transformation process.

However, initially we experimented with a different fitness function, that worked on the basis of allocating a Coefficient Point Weighting (CPW) as a value to each transformation rule. Each rule would have an associated fixed weighting. For example, for code compaction experiments, within an overall sequence, a transformation rule such as [merge] that clearly reduces the size of the program has a higher point score than a ‘navigational’ rule such as [right], which has no effect on program size.

The total fitness value for the whole sequence is thus an aggregation of the different weighted values of the transformation rules that it is comprised. Consider as an example the sequence:

[T1,T4,T4,T6,T2].

Assume that the CPW for T1 is 3, T4 is 1, T6, is 2 and T2 is 3. Then the fitness score for that sequence score is 10 and is given by the following equation:

\[ \text{fitness}_{\text{sequence}} = \sum_{i=1}^{n} W_{T_i} \]

where \( n \) is the length of the sequence and \( W_{T_i} \) is the weight for the transformation rule.

By applying this fitness measure, a transformation sequence \( S_i \) is better than \( S_j \) iff, \( \text{fitness}_{S_i} \) is greater than \( \text{fitness}_{S_j} \).

Early studies carried out using this aggregated CPW fitness metric produced identical results to taking the difference in the lines of code between the output and input programs. We found that the genetic algorithm outperformed the hill climbing
algorithm and random search in these studies.

6.6 Search Landscape

In this section, we justify the results presented in earlier chapters and provide an idea of the sort of landscape created by the fitness function.

One of the justifications for the application of search-based heuristics to our class of problems in the presence of a large search space that is infeasible to traverse exhaustively. The experiments carried out had $40^20$ different possible solutions.

We found that several transformation sequences produced identical fitness scores. There were many transformation rule present in a sequence which did not contribute to improving the fitness score such as rules which swapped statement positions or those which moved the cursor position. The presence of these rules in the search space was still important as they fulfilled other vital roles, such as, facilitating the application of a suitable transformation rule.

However, the presence of at least one ‘contributory’ transformation rule, applied in an appropriate position makes a significant improvement to the fitness score. We believe the landscape is one of large regions of low-lying plateaux - justified by many transformation sequences with identical fitness values - surrounded by the presence of random sharp spikes. We note that transforming the test program in Chapter 3 is easily computable by a human and we can determine the best transformed version. Therefore, there is an upper bound (a best fitness score) achievable search.

In the code simplification experiment carried out in Chapter 3, we note that only 4 out of 22 transformation rules could potentially result in an optimised program.
Therefore, within a transformation sequence, the probability of selecting one of these ‘good’ transformation rules is $\frac{4}{22}$. If the algorithm is executed 1000 times, then there is at least $1000 \times \frac{4}{22}$ potentially good transformation rules that could be selected. It has been suggested that when searching randomly, the size of the search space is not particularly a key feature to a successful search. Rather, a more important characteristic is the ratio of good solutions to bad ones. This gives random search a good opportunity for finding good solutions the higher this ratio. Consequently, a realistic random sample of points along such a landscape provides a very good chance of selecting a point on any spike.

### 6.7 Genetic Operators

In this section, we provide a justification for the operators used in implementing our Genetic Algorithm. We also provide an analysis of alternative methods that could have been used highlighting relative strengths and weaknesses.

#### 6.7.1 Choice of Crossover Operator

Crossover describes the way by which ‘mating’ or recombination takes place between two candidate solutions (individuals) within a population. We already noted that Genetic Algorithms are population based search procedures. In most cases, two individuals are selected according to a determined selection method and genetic material between each individual are exchanged. Variants of crossover approaches exist and the success of the algorithm depends on which variant is selected.
It is important to highlight that fact that there are no guarantees that the offsprings of two parents would be better than either or both parents. Indeed, it is possible that good parents could produce offsprings that are worse after mating. However, there is also a likelihood that good parents might exchange the better elements of their constituent make-up and the result would be a viable offspring. It is this chance that facilitates the use of GAs and encourages the careful selection of a crossover operator.

In the experiments conducted in this thesis, we used the single point crossover - described previously in Chapter 2, pg. 32. However some possible approaches to this would have been uniform crossover, \( n \)-point crossover.

**Uniform Crossover** entails the swapping of adjacent nodes between two parents. Figure 6.2 illustrates uniform crossover. Each alternate node of the offspring is selected from different parents. As we alluded to previously, one of the key features in our application in the presence of building blocks. These are sub-sequences of transformation rules that result in the entire sequence getting a good score. It is important to keep these sub-sequences together through the entire GA cycle. For instance, consider Figure 6.2 and assume that transformation 9 in Parent 1 was a good transformation rule but required rules 2, 4, and 5 to be executed first, that is, rules 2, 4, 5, 9 was a reasonable sub-sequence that needs to kept together, then applying uniform crossover would most certainly disrupt this sub-sequence. This is clearly unattractive for our application of Genetic Algorithms and it is for this reason we do not use uniform crossover.

The next possibility would be to use \( n \)-point crossover, in which case, we elect \( n \) points on the two parents and swap adjacent sub-structures. If \( n = 1 \), then what we
get is equivalent to single point crossover and if as $n$ becomes larger, we get close to uniform crossover. It is important to get the value of $n$ correct, depending upon the specific application domain which the GA is being applied.

For these reasons, we selected single point crossover because we believed it provided the best case for creating offspring which were better than their parents. It was also very important that certain transformation subsequences be preserved during the mating process.

![Figure 6.2: Illustration of uniform crossover.](image)

### 6.7.2 Selection of Mutation Operator

In conducting GA experiments, it is important to preserve diversity within the population. One method for doing this is by having a random chance for mutating individuals within the population. The purpose of having mutation is to prevent premature convergence. This is where, the population quickly homogenises and an
global optimum unreachable.

In our experiment, we select different mutation rates, which is the rate at which we carry out mutation. This was to get an idea of the effects of different values upon the population.

### 6.8 Threats to Validity

This section discusses some of the potential threats to the validity of the studies reported in this dissertation.

#### 6.8.1 Internal Threats

In the studies conducted in this research, the greatest concern as an internal threat to validity involved getting atypical results. We reduced the chances of getting these misrepresentative atypical results by presenting the mean results from 10 runs of the search algorithms. Each new run is carried out with a different seed to the random number generator and each algorithm uses the same start seed. To our knowledge, the studies could not have produced any internal threats due to instrumentation effects. The quality of the results measured is guaranteed to be even across the different techniques because the measures produced are outputs of a function. The experimental details are included in this thesis in order to facilitate replicability studies.
6.8.2 External Threats

Threats to external validity are conditions that limit generalisations from the results. The primary threat to external validity for the studies in this research involve some of the artefacts used in the experiments. The experiments were largely carried out on synthetic programs exhibiting the presence of a particular feature of interest. It is therefore impossible to state as a general conclusion that genetic algorithms outperform hill climbing algorithms for all program when conducting lines of code optimisations. This threat can only be addressed through additional studies and by analysing a large number of widely distributed programs.

The transformation rules used in these studies are also an artefact of the transformation engine FermaT and the source language WSL. Additional studies would need to be performed on other high level source languages such as C, C++ and Java, all of which may have a different set of transformation rules through which to search.

6.9 Summary

The purpose of this chapter was to present an analysis of central issues key to the studies carried out in this thesis. This was to provide some explanation for the approaches adopted as well as outlining any other possible alternative approaches. Retrospectively, we found that the initial problem of searching for transformation sequences turned out to be much harder than was envisaged. This is largely due to the rather coarse-grained landscape generated by the fitness function. This was due in-part to the selection of transformation rules available to the search framework. The chapter also provides some justification for the particular search algorithms considered.
in this work and concluded with a presentation of an alternative fitness calculation scheme which was considered and tested but found to produce results little different from the main fitness approach applied in the research.
Chapter 7

Conclusions And Future Work

In this chapter, we provide a summary of the outcomes of the research. Our motivation was geared toward providing a fully automated, general purpose transformation system, where most effort would be spent computing the fitness function that drives the direction of the search. Section 7.1 explains the contribution of our work and Section 7.2 summarises the work carried out and concludes the thesis. In Section 7.3, we describe some of the limitations of the methods proposed in this thesis, and further work to be carried out is outlined in Section 7.4.

7.1 Summary of Achievements

The research carried out and documented in this thesis set out to answer the following research questions:

1. Is it possible to fully automate a generalised program transformation using a search-based framework?
2. Which search-algorithm produces better solutions to program transformation problems?

7.1.1 Is it possible to fully automate a generalised program transformation using a search-based framework?

In this thesis, I identified some of the problems that transformation tools currently encounter. One of these, is the prescription by the designers of the tool as to exactly how the transformation rules should be applied. At best, this approach is inflexible and is certainly not guaranteed to produce optimal results. There is also some need for human involvement in the transformation process.

We performed three sets of experiments with different transformation goals. This was done to show the flexibility and robustness of the technique. Rather than pre-fix transformation sequences, we spent more effort in the construction and formalisation of the objective function and allowed this to search for ideal sequences. The different experiments were designed to illustrate the generality of the approach. The techniques themselves are cheap and effective. At a high level, the results show that it is possible to employ these heuristic methods to solving program transformation problems. We believe this is a first step toward achieving the overall goal of a fully automated, general-purpose program transformation framework and tool.

The primary achievements in this regard are:

- Formulating the program transformation problem as a search problem. This includes identifying an appropriate representation scheme for individual solutions manipulated by the heuristic methods.
• Identifying some problems that hinder the effective application of search-based techniques to source program transformation. For example, one limiting factor of Search-Based approach is finding good sub-sequences (building blocks) that lead the general search towards fertile areas of optimisation.

7.1.2 Which search-algorithms produce better solutions to program transformation problems?

To answer this question, during our experimentation, we considered different heuristic search techniques adapted for the three different problems. The experiments compared the performance of a genetic algorithm, a hill climbing algorithm and a greedy algorithm. An implementation for random search was used as a baseline measure.

However, we found that the techniques performed differently for each experiment. GA was found to outperform HC and Random search in the re-engineering and comprehension experiments. It produced solutions with better mean fitness values per number of fitness evaluations. The difference in performance between GA and HC was found to be considerable. However, the GA implementation has a longer execution time.

In the comparison between greedy technique, GA, HC and random (Section 4.4.3), the greedy technique outperforms all other approaches in terms of the final output solution. In this particular experiment, GA underperforms when compared against HC. One possibility in order to get a truly optimal solution, is to use a hybrid solution - a mixture of the different techniques. Applying hill climbing to the output of the GA would produce a solution no worse than the best solution in GA process.
We have been able to design and implement a search framework for conducting singular objective and multiobjective optimisations in relation to source-source program transformation.

7.2 Conclusion

The work presented here has been a combination of existing technologies and innovative ideas by the author. The existing technologies include:

- Code level program transformation rules.
- Meta-heuristic search algorithms

It is important to mention that these studies reuse previously existing transformation rules that can be applied at source-code level. These rules have already been proven to be meaning preserving [112]. Consequently, it is outside the scope of this work to prove correctness of each transformation rule. We assume that the transformed programs generated by our approach are equivalent to the original source-code.

The innovative contributions of this work include:

- Search-based source-source transformations for lines of code optimisation.
- Computing amorphous slices using search-based transformations.
- Multiobjective search in program transformation. Simultaneous optimisation of the distance between definitions and uses for variable identifiers in order to aid overall program comprehension.
With the framework for search techniques developed, we conducted several experiments in order to verify the applicability of search to program transformation. The successful application of searching in general for sequences of transformation rules that results in a smaller transformed program was shown in Experimental Study 1 (Chapter 3).

### 7.2.1 SB-code optimisations

By referring to smaller size, we mean the number of lines of code in the transformed version should be fewer than in the original version.

#### Originality

This work proposes an application of existing search-based techniques to perform source-source program transformations, through the application of low-level meaning preserving transformation rules. As shown in Section 2.2.7, automated search-based techniques have already been applied to carry out source transformations for specific targets such as increasing parallelisation. Previous research has also investigated compiler optimisations using these search-based approaches. As far as the author is able to establish, there has not been any prior work investigating generic source-source optimisation via search-based program transformation.

### 7.2.2 SB-Amorphous Slicing

Experimental Study 2 (in Chapter 4) demonstrated that the technique could be applied to addressing a real goal in software engineering. Prior work has been carried
out on slicing and some of the benefits of program slicing have been well documented [40]. We showed that the search-based transformation technique is suited to the task of computing amorphous slices and empirically validated an idea already put forward to transform the initial problem into a solvable one of redundancy removal. To our knowledge, this is the first set of experiments performed to compute amorphous slices using heuristic search. The search approaches performed encouragingly when compared to a specially constructed analytical tool for constructing amorphous slices.

7.2.3 SB-Program Comprehension

Experimental Study 3 (in Chapter 5) broadened the scope of the application of the search framework to include experiments in proximity-based measures of variable identifiers.

Originality

Previous experimental studies focused on singular objective functions. Experimental Study 3 examined a multi-objective transformation problem and addressed an existing problem in software engineering. Previous work in program comprehension has shown that variable proximity enhances or hinders the recollection of the use for a variable. Using the search framework, we were able to reduce the distance between references to similar variable identifiers. This experiment was designed to explore multi-objectivity in search-based source program transformation. We are not aware of any other research that combines program transformation and multiobjective optimisation.
7.3 Restrictions and Limitations

This section outlines some of the restrictions and limitations of the approach described.

7.3.1 Types of Programs

The method is currently limited to intra-procedural test objects written in the WSL language. The approach does not handle function calls in inter-procedural modules. Further, the approach has not been tested with unstructured programs containing forward or backward jumps, for example through the use of break or goto statements.

7.3.2 Scalability

The method, as it currently stands, has some issues with scalability. There is an obvious question about whether or not the approach scales to larger industrial programs. The experiments were performed on relatively small programs. Further research needs to be performed on larger programs. By extending the approach to larger programs, there might be a need to include more powerful transformation rules, those that are capable of performing multiple transformation steps in one step in the sequence. Cooper et al. [29] use such optimisation rules in their research. However, in this thesis, we restrict the use of these larger-scale rules as the focus, as much as full optimisation, was also to investigate the directional nature in which program transformation occurs. Thus, the need to have a means of guiding the program cursor to fertile areas in the code for optimisation. The low-level transformations make it difficult for this approach to scale to much larger programs.
7.4 Summary of Future Work

This section suggests some possible future research directions.

7.4.1 Inter-procedural and Object-Oriented Test Objects

The focus of this research was confined to analyse intra-procedural imperative style programming languages. Therefore, a natural extension to the research would be to examine other genres of programming languages. Firstly, extending the framework to deal with inter-procedural code which contains function calls, parameters, etc., and includes transformation rules that work on such features. Similarly, it would be interesting to perform this kind of research on object-oriented programs using similar low-level transformation rules. O’Keeffe and O’Cinneide [91] performed some initial experiments on refactoring object oriented programs using search-based algorithms. They focused on searching for refactorings such as moving a method from one class to another, deleting a method, etc. However, these refactorings are at a higher level of abstraction than those applied in our research.

7.4.2 Extending the Search-Framework

It is suggested that no one specific search algorithm can outperform all others for a given problem. Each individual search algorithm such as hill climbing, simulated annealing, genetic algorithms, Grover search [49], etc., all have their intrinsic advantages. Further experimentation may need to be performed using search algorithms other than those already employed in this research. It is important to know how each search algorithm performs for program transformation. This would ultimately inform
the adoption of a hybrid-search algorithm.

7.4.3 Static Analysis to find areas of promising optimisations

Further static analysis could be carried out to find areas where the application of transformation rules may have potentially greater significance. Currently as the approach stands, the search techniques spend a large amount of effort (and is often mis-directed) trying to find ‘fertile’ portions of code, where possible.

7.4.4 Inclusion of future transformation rules

As we stated earlier, the effectiveness of the search approach depends largely on the quality of the transformation rules present. Further transformation rules could be included in the overall pool of possible transformation for the search to go through.

7.4.5 Field Study in Program Comprehension

It has been suggested intuitively that similar variable identifiers appearing close together help understanding of the functionality of the program. Jones\(^1\) reported in studies carried on experienced programmers that it was easier to understand code with nearer variable distance measures. Further investigation is suggested for the output produced by the technique. We suggest a set-up of expert and inexperienced programmers for both the experimental study and the control group looking at the source program. The experiment can serve as a replication study to any similar prior studies.

\(^1\)Private Communication.
Bibliography


Appendix A

Source Description for Test Objects

A.1 Surface Area

Below is the source code description for the surface area program used in experiments described in Chapter 4.

\[
D := 2\times r; \\
\text{FaceArea} := \pi \times r \times r; \\
C := \pi \times D; \\
\text{SurfaceArea} := 2 \times \text{FaceArea} + C \times h; \\
\text{slice} := \text{SurfaceArea}; \\
C := 1; \\
\text{FaceArea} := 1; \\
D := 1
\]
A.2 Odds / Even

Below is the source code description for the odds/even program used in experiments described in Chapter 4.

```
total := 0;
i := 0;
evens := 0;
noevens := 0;
odds := 0;
noodds := 0;
n := n0;
WHILE i <= n DO
    evenflag := A[i] / 2;
    evenflag := 0;
    IF FALSE THEN
        evens := evens + A[i];
        noevens := noevens + 1;
    ELSE
        odds := odds + A[i];
        noodds := noodds + 1;
    FI;
    total := total + A[i];
i := i + 1;
OD;
IF noevens <> 0 THEN meaneven := evens/noevens;
ELSE meaneven := 0; FI;
IF noodds <> 0 THEN meanodd := odds/noodds;
ELSE meanodd := 0; FI;
mean := total/(n+1);
n := n + 1;
evendifference := abs(meaneven-mean);
odddifference := abs(meanodd-mean);
slice := evendifference;
total := 1;
i := 1;
evens := 1;
noevens := 1;
odds := 1;
noodds := 1;
n := 1;
evenflag := 1;
meaneven := 1;
meanodd := 1;
mean := 1;
odddifference := 1
```
A.3 Tax

Below is the source–code description for the Tax test object used in the experiment described in Chapter 4.

C: "blind=1 AND widow=1 AND age<50";
PRINT("Enter age: ");
C: "age := read"; age := 40;
C: "read:=read+next";
WHILE age<0
DO
  PRINT("Enter age again: ");
  age:=read;
  read:=read+next
OD;

PRINT("Enter Personal Income: ");
C: "personal := read"; income := 23000;
C: "read:=read+next";
WHILE income<0
DO
  PRINT("Enter personal income again: ");
  income:=read;
  read:=read+next
OD;

PRINT("Enter 1 if you’re married zero otherwise: ");
married := 1;
C: "read:=read+next";
WHILE married<>0 AND married<>1
DO
  PRINT("Enter 1 if you’re married zero otherwise: ");
  married:=read;
  read:=read+next
OD;

PRINT("Enter 1 if you’re a widow zero otherwise: ");
widow := 1;
C: "read:=read+next";
WHILE widow<>0 AND widow<>1
DO
  PRINT("Enter 1 if you’re married zero otherwise: ");
  widow:=read;
  read:=read+next
OD;

PRINT("Get someone else to Enter 1 if you’re blind zero otherwise: ");
blind := 0;
C: "read:=read+next"
WHILE blind<>0 AND blind<>1 DO
  PRINT(“Enter 1 if you’re blind zero otherwise: ”);
  blind:=read;
  read:=read+next
OD;

  IF (age>=75) THEN personal := 5980 ELSE IF (age>=65) THEN personal := 5720 ELSE personal := 4335 FI;
FI;

  IF (age>=65 AND income >16800) THEN IF (4335 > personal-((income-16800)/2)) THEN personal := 4335 ELSE personal := personal-((income-16800)/2) FI ELSE personal := personal-((income-16800)/2) FI;
FI;

  IF (blind =1) THEN personal := personal + 1380 FI;

  IF (married=1 AND age >=75) THEN pc10 := 6692 ELSE IF (married=1 AND age >= 65) THEN pc10 := 6625 ELSE IF (married=1 OR widow=1) THEN pc10 := 3470 ELSE pc10 := 1500 FI ELSE pc10 := 1500 FI;

  IF (married=1 AND age >= 65 AND income > 16800) THEN IF (3470 > pc10-((income-16800)/2)) THEN pc10 := 3470 ELSE pc10 := pc10-((income-16800)/2) FI ELSE pc10 := pc10-((income-16800)/2) FI;

  IF (income <= personal) THEN tax := 0 ELSE income := income - personal; IF (income <= pc10) THEN tax := (income * 10)/ 100
ELSE tax := (pc10 * 10)/100;
income := income - pc10;
IF (income <= 28000)
THEN tax := tax + (income * 23)/100
ELSE tax := tax + (28000 * 23)/100;
income := income - 28000;
tax := tax + (income * 40)/100
FI
FI
FI;

IF (blind=0 AND married =0 AND age < 65)
THEN code := "L"
ELSE IF (blind=0 AND age < 65 AND married=1)
THEN code := "H"
ELSE IF (age >= 65 AND age < 75 AND married=0 AND blind=0)
THEN code := "P"
ELSE IF (age >= 65 AND age < 75 AND married=1 AND blind=0)
THEN code := "V"
ELSE code := "T"
FI
FI
FI
FI
PRINT("age=",age);
PRINT("income= ",income);
PRINT("married= ", married);
PRINT("blind= ", blind);
PRINT("widow= ", widow);
PRINT("personal= ",personal);
PRINT("pc10= ", pc10);
PRINT("code= ", code);
slicevar:=personal;

age := 1;
income := 1;
tax := 1;
code := 1;
read := 1;
mixed := 1;
widow := 1;
blind := 1;
pc10 := 1
A.4 Student-Marks

Below is the source code description for the student marks test object used in the experiment described in Chapter 4.

```plaintext
VAR <
sum:=0,
lowest:=100,
highest:=0,
sum1:=0,
lowest1:=100,
highest1:=0,
sum2:=0,
lowest2:=100,
highest2:=0,
sum3:=0,
lowest3:=100,
highest3:=0,
numberofpasses:=0,
numberoffails:=0,
numberoffirsts:=0,
numberofupperseconds:=0,
numberoflowerseconds:=0,
numberofthirds:=0,
numberofpassdegrees:=0
>
PRINT("enter number of students");
numberofstudents := 50;

WHILE numberofstudents<=0
DO
PRINT("Please enter a positive number this time");
numberofstudents:=read;read:=read+next;
OD;
i:=0;

WHILE i<numberofstudents
DO
PRINT("enter mark1 for student: "); mark1 := 85;
PRINT(i+1);
C: " read>=0 AND read<=100;mark1:=read;read:=read+next;";
IF mark1>=0 AND mark1<=100
THEN
sum1:=sum1+mark1;
IF mark1<lowest1
THEN lowest1:=mark1
ELSIF mark1>highest1
THEN highest1:=mark1
```
FI
FI;
WHILE mark1<0 OR mark1>100
DO
PRINT("Please re-enter a mark between 0 and 100");
mark1:=read;read:=read+next;
IF mark1>=0 AND mark1<=100
THEN
sum1:=sum1+mark1;
IF mark1<lowest1
THEN lowest1:=mark1
ELSIF mark1>highest1
THEN highest1:=mark1
FI
FI
OD;
PRINT("enter mark2 for student: ");
mark2 := 60;
PRINT(i+1);

    IF mark2>=0 AND mark2<=100
THEN
sum2:=sum2+mark2;
IF mark2<lowest2
THEN lowest2:=mark2
ELSIF mark2>highest2
THEN highest2:=mark2
FI
FI;
WHILE mark2<0 OR mark2>100
DO
PRINT("Please re-enter a mark between 0 and 100");
mark2:=read;read:=read+next;
IF mark2>=0 AND mark2<=100
THEN
sum2:=sum2+mark2;
IF mark2<lowest2
THEN lowest2:=mark2
ELSIF mark2>highest2
THEN highest2:=mark2
FI
FI;
OD:
PRINT("enter mark3 for student: "); mark3 := 70;
PRINT(i+1);

    IF mark3>=0 AND mark3<=100
THEN
sum3:=sum3+mark3;
IF mark3<lowest3
THEN lowest3:=mark3
ELSIF mark3>highest3
THEN highest3:=mark3
FI
FI;
WHILE mark3<0 OR mark3>100
DO
PRINT("Please re-enter a mark between 0 and 100");
mark3:=read;read:=read+next;
IF mark3>=0 AND mark3<=100
THEN
sum3:=sum3+mark3;
IF mark3<lowest3
THEN lowest3:=mark3
ELSIF mark3>highest3
THEN highest3:=mark3
FI
FI
OD;
mark := (mark1 + mark2 + mark3)/3;
sum:=sum+mark;
IF mark<lowest
THEN lowest:=mark
ELSIF mark>highest
THEN highest:=mark
FI;
IF mark<35
THEN
PRINT("FAIL");
numberoffails:=numberoffails+1;
ELSE
numberofpasses:=numberofpasses+1;
IF mark<40
THEN
PRINT("PASS DEGREE");
numberofpassdegrees:=numberofpassdegrees+1;
ELSIF mark<50
THEN
PRINT("THIRD");
numberofthirds:=numberofthirds+1
ELSIF mark<60
THEN
PRINT("Lower Second");
numberoflowerseconds:=numberoflowerseconds+1
ELSIF mark<70
THEN
PRINT("Upper Second");
numberofupperseconds:=numberofupperseconds+1
ELSE
PRINT("FIRST");
numberoffirsts:=numberoffirsts+1
FI
FI;
i:=i+1
OD;

average:=sum/numberofstudents;
average1:=sum1/numberofstudents;
average2:=sum2/numberofstudents;
average3:=sum3/numberofstudents;

    IF average1<35
THEN
averagegrade1:=fail
ELSIF average1<40
THEN
averagegrade1:=pass
ELSIF average1<50
THEN
averagegrade1:=third
ELSIF average1<60
THEN
averagegrade1:=lowersecond
ELSIF average1<70
THEN
averagegrade1:=uppersecond
ELSE
averagegrade1:=first
FI;

    IF average2<35
THEN
average2:=fail
ELSIF average2<40
THEN
averagegrade2:=pass
ELSIF average2<50
THEN
averagegrade2:=third
ELSIF average2<60
THEN
averagegrade2:=lowersecond
ELSIF average2<70
THEN
averagegrade2:=uppersecond
ELSE
averagegrade2:=first
FI;
IF average3 < 35
THEN
  averagegrade3 := fail
ELSIF average3 < 40
THEN
  averagegrade3 := pass
ELSIF average3 < 50
THEN
  averagegrade3 := third
ELSIF average3 < 60
THEN
  averagegrade3 := lowersecond
ELSIF average3 < 70
THEN
  averagegrade3 := uppersecond
ELSE
  averagegrade3 := first
FI;

PRINT(average);
PRINT(highest);
PRINT(lowest);
PRINT(numberofpasses);
PRINT(numberoffails);
PRINT(numberoffirsts);
PRINT(numberofupperseconds);
PRINT(numberoflowerseconds);
PRINT(numberofthirds);
PRINT(numberofpassdegrees);
PRINT(average1);
PRINT(average2);
PRINT(average3);
PRINT(averagegrade1);
PRINT(averagegrade2);
PRINT(averagegrade3);
PRINT(highest1);
PRINT(highest2);
PRINT(highest3);
PRINT(lowest1);
PRINT(lowest2);
PRINT(lowest3);
slicevar := lowest1;
ENDVAR
A.5 Calendar

Below is the source-code description for the Calendar test object used in the experiment described in Chapter 4.

read := 0;
inputyear := read;
WHILE (inputyear < 0)
DO
PRINT(”please re-enter”);
inputyear := read;
read := read + next
OD;
inputmonth := read;
NOT (inputmonth < 1 OR inputmonth > 12);
read := read + next;
WHILE (inputmonth < 1 OR inputmonth > 12)
DO
PRINT(”please re-enter”);
inputmonth := read;
read := read + next
OD;
inputday := read;
read := read + next;
NOT (inputday < 1 OR inputday > 31);
WHILE (inputday < 1 OR inputmonth > 31)
DO
PRINT(”please re-enter”);
inputday := read;
read := read + next
OD;

startyear := 1;
startmonth := 1;
startday := 0;
days := 0;
leapyear := 1;
century := 1;
fourthcentury := 1;
leap := 0;
dayofweek := 0;
dofweekofstartday := 5;
k := 0;
year := startyear;
month := startmonth;
WHILE (year <= inputyear)
DO
IF (leapyear = 0 AND (year < 1800) OR NOT(century = 0) OR fourthcentury = 0))
THEN leap:=1
ELSE leap:=0
FI;
IF (year < inputyear)
THEN IF (leap=1)
THEN days:=days+366
ELSE days:=days+365
FI
FI;
IF (leapyear=3)
THEN leapyear:=0;
IF (century=99)
THEN
century:=0;
IF (fourthcentury=399)
THEN fourthcentury:=0;
ELSE fourthcentury:=fourthcentury+1;
FI
ELSE
century:=century+1;
fourthcentury:=fourthcentury+1;
FI
ELSE
leapyear:=leapyear+1;
century:=century+1;
fourthcentury:=fourthcentury+1;
FI;
year:=year+1;
OD;

WHILE(month < inputmonth)
DO
IF ( month=1 OR month=3 OR month=5 OR month=7 OR month=8 OR month=10 OR month=12)
THEN days:=days+31
ELSIF ( month=4 OR month=6 OR month=9 OR month=11)
THEN days:=days+30
ELSIF (leap=1)
THEN days:=days+29
ELSE days:=days+28
FI;
month:=month+1;
OD;
days:=days+inputday;
IF (inputyear>1752 OR (inputyear=1752 AND inputmonth>9) OR (inputyear=1752 AND inputmonth=9 AND inputday > 13))
THEN
days:=days-11
FI;
WHILE (days-k > 6)
DO k:=k+7;
OD;
dayofweek:=days-k+dofweekofstartday;
IF (dayofweek>6)
THEN dayofweek:=dayofweek-7
FI;
PRINT(days);
PRINT(dayofweek);
IF dayofweek=0
THEN
PRINT("SUNDAY")
ELSIF dayofweek=1
THEN
PRINT("MONDAY")
ELSIF dayofweek=2
THEN
PRINT("TUESDAY")
ELSIF dayofweek=3
THEN
PRINT("WEDNESDAY")
ELSIF dayofweek=4
THEN
PRINT("THURSDAY")
ELSIF dayofweek=5
THEN PRINT("FRIDAY") ELSE PRINT("SATURDAY") FI;
slicevar:=month

A.6 Rear-Wing Defroster
We are unable to provide source code description for the rear-wing defroster module due to contractual obligation with our industrial partner.

A.7 Braking Controller Defroster
We are unable to provide source-code description for the braking controller module due to contractual obligation with our industrial partner.
A.8 Test Object for Program Comprehension

The following test object describes the source code used in conducting the experiment in Chapter 5.

\[
x := c; \\
a := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := c; \\
y := a
\]