THE UNIVERSITY OF HULL

USE OF PROGRAM AND DATA-SPECIFIC HEURISTICS FOR AUTOMATIC SOFTWARE TEST DATA GENERATION

being a Thesis submitted for the Degree of

Doctor of Philosophy

in the University of Hull

by

Mohammad Alshraideh

April 2007

Abstract

The application of heuristic search techniques, such as genetic algorithms, to the problem of automatically generating software test data has been a growing interest for many researchers in recent years. The problem tackled by this thesis is the development of heuristics for test data search for a class of test data generation problems that could not be solved prior to the work done in this thesis because of a lack of an informative cost function. Prior to this thesis, work in applying search techniques to structural test data generation was largely limited to numeric test data and in particular, this left open the problem of generating string test data. Some potential string cost functions and corresponding search operators are presented in this thesis. For string equality, an adaptation of the binary Hamming distance is considered, together with two new string specific match cost functions. New cost functions for string ordering are also defined. For string equality, a version of the edit distance cost function with fine-grained costs based on the difference in character ordinal values was found to be the most effective in an empirical study.

A second problem tackled in this thesis is the problem of generating test data for programs whose coverage criterion cost function is locally constant. This arises because the computation produced by many programs leads to a loss of information. The use of flag variables, for example, can lead to information loss. Consequently, conventional instrumentation added to a program receives constant or almost constant input and hence the search receives very little guidance and will often fail to find test data. The approach adopted in this thesis is to exploit the structure and behaviour of the computation from the input values to the test goal, the usual instrumentation point. The new technique depends on introducing program data-state scarcity as an additional search goal. The search is guided by a new fitness function

made up of two parts, one depending on the branch distance of the test goal, the other depending on the diversity of the data-states produced during execution of the program under test.

In addition to the program data-state, the program operations, in the form of the program-specific operations, can be used to aid the generation of test data. The program-specific operators is demonstrated for strings and an empirical investigation showed a fivefold increase in performance. This technique can also be generalised to other data types. An empirical investigation of the use of program-specific search operators combined with a data-state scarcity search for flag problems showed a threefold increase in performance.

Acknowledgements

First and foremost I thank my supervisor, Dr. Leonardo Bottaci, who has been a constant source of stimulation and encouragement throughout my research. His patience advice and constructive criticism helped me to stay on track and remain motivated, whatever difficulties I faced, as well as providing an example of scholarship that will remain with me throughout my life. Without his guidance, this work could not have been accomplished.

I would also like to thank Dr Yiannis Papadopoulos for his helpful support and comments throughout this project.

I would to thank Dr. Marc Roper for for useful and helpful discussions.

Thanks also go to all the staff in the Department of Computer Science in particular Dr Chandra Kambhampati, Helen El-Sharkawy, Joan Hopper and Colleen Nicholson for their help.

Thanks also to my best friend Woakil Ahamed for his support.

Finally, I would like to thank my parents and brother for their love and support. Thanks to my wife, for her love, understanding and assistance during the preparation of this thesis.

Thanks must also go to The university of Jordan for funding this research work.

Publications

- Mohammad Alshraideh, Leonardo Bottaci

  Automatic Software Test Data Generation For String Data Using Heuristic Search with Domain Specific Search Operators. Proceedings of the Third UK Software Testing Workshop (UKTest 2005), University of Sheffield, UK, September 5-6, 2005, pp.137-150.

- Mohammad Alshraideh, Leonardo Bottaci

  Search-based Software Test Data Generation For String Data using program-Specific Search Operators. Special Issue of Software Testing, Verification and Reliability devoted to Extended Papers from the Third UK Testing Conference (UKTest 2005), 16(3), September 2006,pp.175-203.

- Mohammad Alshraideh, Leonardo Bottaci

  Using program data-state diversity in test data search. Testing: Academic & Industrial Conference practice and research techniques (TAIC), London, UK, August 29 -31, 2006, pp. 107 -114.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software testing covers a range of activities aimed at evaluating an attribute or capability of a program or system (Beizer, 1990). Usually, it is a process of executing a program or system with the intent of finding failures (Myers, 1979). Because software and digital systems are not continuous, testing boundary values is not sufficient to guarantee correctness. All the possible values need to be tested and verified, but obviously, for a realistic software module, complete testing is impractical. Therefore testing must be selective.

Testing is done at different levels of the software:

1. Unit testing: which refers to the individual testing of separate units of a software system. In object-oriented systems, these units typically are classes and methods.

   Unit testing may be structural (white box) or functional (black box).

   (a) Structural testing: test case selection that is based on an analysis of the internal structure of the component, typically the source code. Different approaches to how test cases should be selected lead to different coverage criteria, statement coverage, branch coverage etc.

   (b) Functional testing: testing based on an analysis of the specification of a piece of software without reference to its internal workings. The goal is to test how well the component conforms to the published requirements for the component.

2. Integration Testing: exposes faults during the process of integration of software components or software units and is specifically aimed at exposing faults in their interactions. The integration approach could be either bottom-up, top-down or a mixture of the two.

3. System testing: testing that attempts to discover defects that are properties of the entire system rather than of its individual components.

4. Regression testing: retesting a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made.

This research is concerned with the automation of structural unit testing, in particular, the automatic generation of test data.

1.1 Approaches to Automatic Test Data Generation

Test data generation in white-box testing (source-code based testing) is a process of finding program input on which a selected element (e.g. a not yet covered statement) is executed. Finding such input test data manually can be very labour intensive and expensive.

A number of different automatic software test data generation methods have been investigated (McMinn, 2004). These methods may be placed into one of two broad categories known as static methods and dynamic methods. Static methods aim to analyse the static structure of the program under test in order to compute suitable test cases. Static methods exploit control and data-flow information and may use symbolic execution (King, 1975), (King, 1976), (Beizer, 1990) but the program under test is not executed.

Dynamic methods aim to exploit information gained by execution of the program under test. The most basic dynamic method is random test data generation (Duran and Ntafos, 1984). In this method, test data is generated randomly. Each test case

```
void func(int a) {
    if (a == 0) {
        //execution required
    }
}
```

Figure 1.1: Simple predicate example

is then executed and either retained or discarded according to whether it executes any test goals not executed by any other so far retained test case. Unfortunately, the likelihood that a test, generated randomly, will execute a difficult to reach branch is very low. As an example, consider the problem of generating an input to execute the target branch of the program fragment shown in Figure 1.1.

The probability that a randomly generated input will set the variable `a` to be equal to `0` may be very small. In general, random test data generation performs poorly and is generally considered to be ineffective at covering all branches in realistic programs (Coward, 1991).

1.2 Applying Heuristic Search to Test Data Generation

Heuristic search techniques such as genetic algorithms, simulated annealing and tabu search are high-level frameworks which use heuristics to find solutions to problems without the need to perform a full exhaustive enumeration of a search space (Reeves, 1995). They have been used to find acceptable approximations to the solution of many NP complete problems.

Software testing normally aims to achieve certain measurable objectives. In fact, many test generation techniques are based around some notion of the coverage of the code. This coverage can be measured and incorporated into an objective or cost function. Better test values should be rewarded with lower cost values, whereas poorer test values should be penalized with higher cost values. With feedback from the cost function, the search looks for better tests based on a heuristic evaluation

of existing tests.

For example, in the program of Figure 1.1 suppose a test case is required to execute the true branch. If the branch is not executed, many test cases will cause `a == 0` to be false. The value of `abs(a - 0)` increases as `a` becomes far from `0`. A value of `4` has a better objective value than that of `10`, since the objective function is better (4 is more close to 0 than 10). The search is encouraged to search around the value of `4`, possibly encountering further "better" values, for example the values `1` or `2`.

1.3 Current Problems with Search Methods Applied to Test Data Generation

A number of different decisions have to be made in order to adapt a heuristic search technique to a specific problem, e.g. the way in which solutions should be represented so that they can be handled by the search. A good choice of encoding, for example, will ensure that similar solutions in unencoded space are also neighbours in representational space and the search will be moved easily from one solution to another that has similar properties. Most important, however, a search can solve a problem only if the cost function is informative. In many test data generation problems it is difficult to find an informative cost function. The following section gives some examples of such problems.

1.3.1 String test data

Current work in test data generation has been largely limited to programs whose predicates compare numbers (Baresel et al., 2001), (Harman et al., 2002). Using the numerical diffrence between numbers, it is easy to define cost functions for these predicates. It is not obvious, however, how to compare non-numeric data types. An example is the string data type.

Consequently, a problem that needs further research is how to automatically generate software test data for character strings. A simple branch coverage problem is illustrated in Figure 1.2. The problem is to find an input string `s` so that the

```
  ...
if (s == "World Cup") {
  //TARGET
}
```

Figure 1.2: Need to measure similarity of string `s` to `World Cup` in order to measure the cost of failure to execute the required branch.

required branch is executed. If `s` is such that the predicate fails, a cost is associated with `s`. This cost is used to guide the search. Given the use of a particular search technique such as a genetic algorithm, a key problem is how to compute a useful cost for this predicate failure. For example, for two test cases `s1 = Wirld Cup` and `s2 = World Cap` the problem is to find which one, if any, should have the lower cost. Until the problem of a cost function for string equality is solved it overly reduces software testing approaches for applications in practice since string predicates are widely used in programming.

1.3.2 Loss of Information

The computation produced by many programs leads to a loss of information. The use of flag variables, for example, can lead to information loss. A flag variable is any variable that takes on one of a few discrete values . A boolean is a special case of a flag variable. Using flag variables in the predicate of conditions in programs produces flat cost function landscapes when computing the branch distance measure. If the program has only relatively few input values which make the flag variable adopt a desired value, it will be hard to find, without guidance, an input to set the flag to the desired value.

The lack of information at the branch distance function causes plateaux in the objective function landscape, one plane corresponding to the undesired value, or all input vectors that do not execute the target, and one plane corresponding to the desired value, or the required test data. No guidance is provided to the search as to how to move from one plane in the objective function landscape to the other. This is true in the example of Figure 1.3. The plateaux corresponding to the "false" value of the flag can be seen clearly in Figure 1.4. If the search compares any input

```
void Subject(int x) {
   // ASSIGNMENT TO flag HERE
   if (flag) {
      //execution required
   }
}
```

Figure 1.3: Need to measure the flag to execute the required branch.

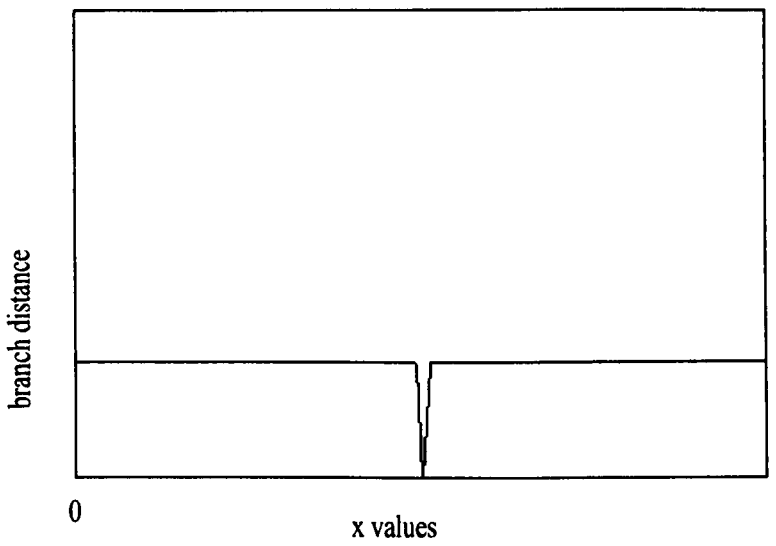


Figure 1.4: Objective function landscape for the example program with flag variable of Figure 1.3

x with its neighbours, it does not know whether to move towards increased values or decreased values, since the objective values returned for these inputs are exactly the same. The search therefore becomes random.

1.4 Research Problem

The problem tackled by this thesis is the development of heuristics for test data search for a class of test data generation problems that currently can not be solved because of a lack of an informative cost function. These are the problem of generating string test data and the problem of generating test data for programs whose coverage criterion cost function is locally constant.

1.5 Contributions of this Thesis

1. Presents new cost functions that can be used to instrument programs where the test data is intended to cover program branches which depend on string predicates such as string equality and string ordering.

2. New techniques for instrumenting programs where "flag variable" problem exists that depend on data-state scarcity.

3. Use of program-specific search operators to improve performance of test data search in general.

1.6 Overview of the Structure of this Thesis

This thesis is organized as follows:
Chapter 2 surveys the literature in automatic generation test data of structural (white-box) testing. Chapter 3 introduces and describes the string problem. Some potential string cost functions and corresponding search operators are presented, program-specific search operators for string are introduced, then the program-specific search operators are generalised to non-string data types. Chapter 4 considers the internal variable problem, with particular focus on the flag problem. Further work in the literature of relevance to the problem is evaluated. A new technique for directing the search when the function that instruments the test goal is not able to discriminate candidate test inputs is presented. The new technique depends on introducing program data-state scarcity as an additional search goal. The search is guided by a new evaluation (cost) function made up of two parts, one depending on the conventional instrumentation of the test goal, the other depending on the diversity of the data-states produced during execution of the program under test.

Finally Chapter 5 summarises the conclusions and suggests some areas for further work.

Chapter 2

Automatic Test Data Generation for Structural unit Testing

Automatic test data generation research has concentrated on structural testing probably because in some ways it is easier to automatically generate test data for structural testing than functional testing. Structural testing allows visibility into internal data structures and control flow. This ability to "see inside" the test object allows the test generator to identify near miss test cases and thus guide a search process. Automatic functional testing requires access to a machine readable specification which is often absent.

2.1 Methods for Data Generation

Approaches for finding test data that execute structural elements according to some coverage criteria are classed as either static or dynamic (McMinn, 2004).
The early years of structural test data generation were dominated by static methods, which do not actually execute the code to be tested. Instead they attempt to find the conditions that bring about the traversal of a given statement or path in the program. Test data is then derived from these conditions. Dynamic methods on the other hand actually execute the software under test, and employ analytical or search techniques in order to find the relevant test data. Early dynamic methods employed simple local searches, whereas more current methods use global search techniques

such as evolutionary algorithms.

2.1.1 Static methods

Symbolic execution is a concept that was first introduced by King (King, 1976). It is based on the idea of executing a program without providing values for its input variables. The output will then in general be a term depending on these input variables, rather than an actual value. This is usually described as a symbolic input value and in turns produces a symbolic output value.

Clarke's work (Clarke, 1976) was one of the earliest in symbolic execution. For the execution of a given path, symbolic execution works by statically traversing the path in the code, building up representations of internal variables in terms of the input variables. Branches within the code introduce constraints on the variables. If these constraints are linear in nature they can be solved through the use of linear programming techniques to find input data. If not, conjugate gradient methods are used; however these require human interaction. For paths where the set of linear constraints generated are found to be inconsistent, the path is deemed infeasible. Clarke's work has several limitations. It requires the user to completely specify all desired test paths. If array indices depend on input data, the system cannot resolve the reference; the same problem occurs with pointers.

The approach of Ramamoorthy (Ramamoorthy and Chen, 1976) which was built on Clarke's work suggested possible solutions to the array problem. Ramamoorthy's approach delays the symbolic execution of an array when it is dependent on input data until the constraint satisfaction stage. Values for variables upon which array access depends are determined first. This allows the array access to be completely determined. According to this method a new symbolic instance of the array will be created which is identical to the previous instance except for the element in question. However, this method significantly increases the complexity and memory requirements of the symbolic execution system.

DeMillo and Offutt (DeMillo and Offutt, 1991) developed a test data generation

1- Each variable is assigned a domain that includes all possible values for that type.

2- Do
{
by using information in a constraint, reduce variable's domain, this done as follows(X, Y are variables, C is constant and ***R*** is a relational operator):
a - for constraints of the form X ***R*** C : reduce Domain of X.
b - for constraints of the form X ***R*** Y : reduce domain of X and Y.

} until (no further reductions are possible)

3- Input variable with the smallest domain is chosen, and a random value is assigned to it.
4- Do from step 2 for all remaining variables using the assigned value(s).
5- If all variables have been assigned a value then success is achieved, otherwise restart from step1.

Figure 2.1: Domain reduction algorithm

technique called Constraint-based testing with the goal of killing program mutants as part of a mutation testing strategy. A constraint system is derived which incorporates reachability and necessity constraints. Reachability constraints represent conditions under which a given statement will be executed, and necessity constraints describe conditions under which a mutant will be killed.

Symbolic execution is used to rewrite the constraints in terms of the input variables. A search procedure known as domain reduction is used to attempt a solution to these constraints. The domain reduction works by using information in the constraint system to reduce the domains of the variables as shown in Figure 2.1.

This method is hampered by similar problems involving loops, viewing the array as one variable, i.e. does not differentiate between individual elements in the array, etc. Offutt et al. introduced a new technique (Offutt et al., 1997) with the intent of solving some of the problems with constraint-based testing known as dynamic domain reduction (DDR).

The DDR procedure starts with several pieces of information: a flow graph, the initial domains for all input variables, and two nodes representing the initial and goal nodes. The first step is that a finite set of paths from the initial to the goal node is determined. Then each path is analysed in turn. The path is traversed and symbolic execution is used to progressively reduce the domains of values for the input variables. When choices for how to reduce the domain must be made, a search process is used to split the domain of some variable in an attempt to find a set of values that allows the constraints to be satisfied. Finally, a test case is chosen arbitrarily from within the reduced input domains. The DDR procedure has partially solved the problems of the arrays and loops.

In summary, the major weakness of symbolic execution is the insufficient handling of loops, dynamic data structure, arrays and procedures.

2.1.2 Dynamic methods

In actually executing the software under test, dynamic methods manage to overcome some of the problems associated with static methods. The simplest dynamic test data generation is to generate test data randomly. A random test data generator randomly generates a set of inputs and then runs the program with these inputs with the expectation that it might execute the selected target (statement, branch, path). The disadvantage of this method is that it is not very effective in finding test data for complex targets. Because of this disadvantage, random test data generation is not considered suitable for real application programs.

The next attempt at test data generation based on actual executions of the program was in 1976 (Miller and Spooner, 1976), when Miller and Spooner put forward the idea that test data generation could be formulated as a numerical optimisation problem. To utilise the method, the user must select a path in the program and then produce a straight-line version of that program containing only that path. Any conditionals are replaced by constraints. A function is then derived which provides a real value estimate of how close the constraints are to being

satisfied. This value is negative when any of the constraints are not satisfied and positive when all constraints are satisfied. Numerical maximisation techniques can then be employed using this function to find inputs to the program that satisfy the constraints for the path.

The idea of Miller and Spooner was extended by Korel in 1990 (Korel, 1990). It is based on actual execution of the program under test and function minimisation algorithms. Test data are developed using actual values of input variables. The program is executed by these input variables and its execution flow is observed. If an undesirable execution flow is taken at some branch, then a real value function is assigned to this branch. Function minimisation methods are used to automatically find the values of input variables which cause this function to become negative and the branch predicate to be true. This method has the advantage that it handles loops, dynamic data structures and array.

In 1992 Korel (Korel, 1992) developed what became known as the goal-oriented approach. The goal-oriented approach alleviates the path infeasibility problem encountered by eliminating the path selection stage. First, the test data generator executes the program with an arbitrary input. While the program is being executed the program execution flow is also observed. Then the search procedure decides whether the execution should continue through the current branch or an alternative branch should be taken because the current branch does not lead to the execution of the selected statement. If an undesirable execution flow at the current branch is observed, then a real-value function is associated with this branch. A function minimisation search algorithm is used to find a new input that changes the execution flow at this branch.

It was shown that the problem of finding input data can be reduced to a sequence of subgoals where each subgoal is solved using a function minimisation method that uses branch predicates to guide the search process. The branch predicate is assumed to be of the form $E1$ *op* $E2$, where $E1$ and $E2$ are arithmetic expressions and *op* is a relational operator $<$, $\leq$, $>$, $\geq$, $=$, $\neq$. A function of the form F *rel* 0 can

then be derived, as shown in Table 2.1:

Branch Predicate	Branch Function F	Relation rel
E1 > E2	E2 -E1	$<$
E1 $\geq$ E2	E2- E1	$\leq$
E1 < E2	E1 -E2	$<$
E1 $\leq$ E2	E1 -E2	$\leq$
E1 = E2	abs(E1-E2)	$=$
E1 <> E2	abs(E1-E2)	$\leq$

Table 2.1: Deriving a cost function from branch predicates from (Korel, 1990)

F is a real-valued function, referred to as fitness function, which is positive or zero if rel is $<$ when a branch predicate is false or negative or zero if rel is $=$ or $\leq$ when a branch predicate is true. The fitness function is evaluated for a program input by executing the program and is used to guide the search. The problem is to find a value of x that causes $F(x)$ to be negative or zero.

In 1996 Korel proposed the chaining approach (Korel and Ferguson, 1996). In the chaining approach, the mechanics of a path search takes place using the notion of an event sequence. This incorporates some of the ideas of the influences graph. An event sequence is basically a succession of program nodes. Each program node has a set of variables associated with it called a constraint set, which can not be modified from one node to the next (i.e. a definition-clear path has to be found for these variables from one node in the sequence to the next). Formally, an event sequence is a series of tuples $e_i = (n_i, S_i)$ where n_i is a node of the program graph and S_i is the constraint set of variables associated with that node. The chaining approach works as shown in Figure 2.2.

The chaining mechanism is therefore a secondary means of trying to force execution of test goals. If inputs cannot be found to change the execution of some problem node by means of a local search, the method implicitly tries to change the

1. Initial sequence is <(s, Φ), (g, Φ)>, where s is start node, g is the goal node.
2. Assume that at node p, the normal function minimisation method cannot bring about the execution of the program to g (p becomes the problem node).
3. If the search fails, find the last definitions of the variables used at p. (LD(p) = (d1, d2,..,dn)
4. a number of new event sequences are generated by inserting the problem node p and one of the last definition nodes, with the constraint set as follows:
   E1 = <(s, Φ), (d1, D(d1)), (p,Φ), (g,Φ)>
   E2 = <(s, Φ), (d2, D(d2)), (p,Φ), (g,Φ)>
   ...
   En = <(s, Φ), (dn, D(dn)), (p,Φ), (g,Φ)>
   From node di the control flow should progress to node g along a definition clear path with respect to D(di).

Figure 2.2: Chaining approach

execution at the node by considering other nodes that can determine its outcome. For this reason Korel suggested the use of a global optimisation technique for future work, as local search methods tended to get stuck in local optimum.

Korel uses gradient decent combined with an exploratory phase. The exploratory phase helps avoid getting stuck at a local optima. Other search methods are also able to avoid local optima. These include simulated annealing and genetic algorithms(GA). This leads to the developments of a collection of work known as Evolutionary Testing.

2.2 Evolutionary Testing

Evolutionary algorithms (EAs) are search methods that take their inspiration from natural selection and survival of the fittest in the biological world. EAs differ from more traditional optimization techniques in that they involve a search from a "population" of solutions, not from a single point. The best-known kind of evolutionary

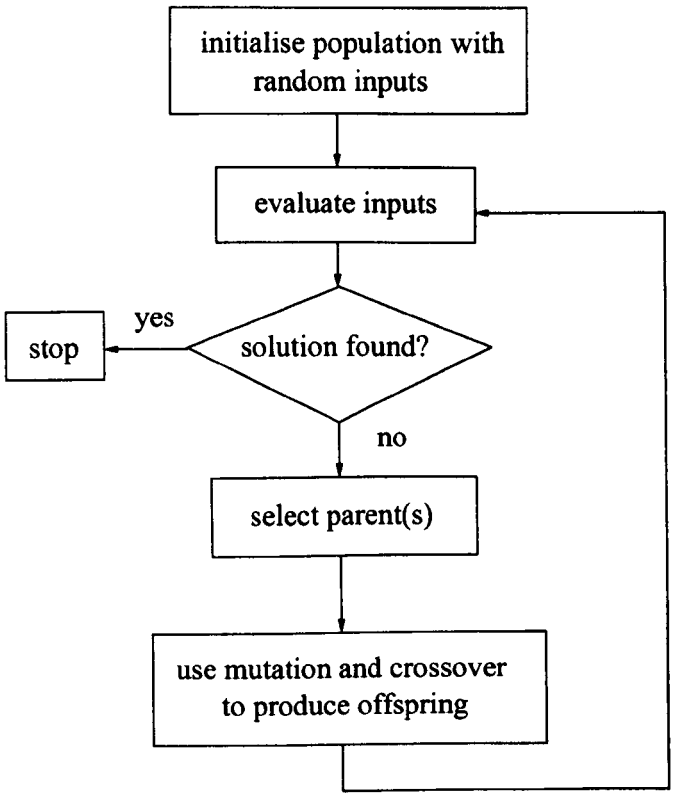


Figure 2.3: Flowchart of test data generation using a genetic algorithm.

algorithms is genetic algorithms. Figure 2.3 shows the basic steps of a genetic algorithm. First the population is initialised, either randomly or with user-defined candidates. The genetic algorithm then iterates through an evaluate-select-produce cycle until either a solution is found or some other stopping criterion applies. The effectiveness of a genetic algorithm depends crucially on the reliability of the guidance provided by the cost function. The cost function is a metric that evaluates each candidate in terms of its "closeness" to a solution.

2.2.1 Early work

The history of applying genetic algorithms to software testing problems can be traced back to 1992. The earliest referenced paper is Xanthakis et al. (Xanthakis et al., 1992) where the testing prototype TAGGER is introduced. The system was used for generating test data written in Pascal.

The first PhD thesis in the area was by Sthamer (Sthamer, 1995) who studied the use of GA as a test data generator for structural testing. The example programs are small procedures written in ADA, including triangle classification, linear search, remainder calculation, and direct sort. Sthamer applies GA to branch, boundary, and loop testing, and also for mutation testing. He claimed that "GAs show good results in searching the input domain for the required test sets, however, other heuristic methods may be as effective, too".

2.2.2 Dynamic test data generation

Pei et al. (Pei et al., 1994) concentrated on pathwise test data generation. By using test data generation by GA they try to define if the selected subpath is feasible or not. The Pei et al. approach is better than the other systems (e.g. random and TAGGER (Xanthakis et al., 1992)) because it processes the whole path simultaneously.

Watkins (Watkins, 1995) deals with path coverage optimisation by GA, using the popular triangle classification problem as an example. GA reached the same coverage as the random method while sampling a smaller percentage of the complete search space. The objective function penalizes individuals that follow already covered paths, by assigning a value that is the inverse of the number of times the path has already been executed during the search.

Smith and Fogarty (Smith and Fogarty, 1996) studied test coverage optimization by a hybrid version of GA and hill-climbing local search. Their application was also the triangle classification problem. They claim that their system can generate test sets that fully satisfy the given metric and reduce the size of evolved test sets.

In the technique used by Jones et al. (Jones et al., 1996), a path does not need to be selected. The objective function is simply formed from the branch distance of the required branch, no guidance is provided so that the branch is reached within the program structure in the first place. Michael et al. (Michael et al., 1997) have developed what is called the GADGET (Genetic Algorithm Data Generation Tool) system, which is fully automatic and supports all C/C++ constructs. The system is

used to obtain condition/decision coverage. They compared results with the random method. GAs gained a much higher coverage than the random method.

Kasik and George (Kasik and George, 1996) have used a GA for emulating software inputs in an unexpected, but not totally random way. The GA is used as a repeatable technique for generating user events that drive conventional automated test tools, so that the system can mimic different forms of novice user behaviour. The system tries to represent how a novice user learns to use an application. The fitness value is given according to how much the actions performed are guided by the individual to resemble novice-like behaviour. The novice behaviour is described by a special reward system that is built based on observations.

Wegener et al. (Wegener and etl., 1996), (Wegener et al., 1997), (Wegener and Grochtman, 1998), (Wegner et al., 1999), (Wegner et al., 2000), (Baresel et al., 2001) have studied the search of the execution time extremes of real-time software with a GA. They have compared their results to the random testing and static analysis. Their object software has mainly been some small examples or DaimlerChrysler embedded automotive electronics software. They think that the static analysis and evolutionary testing together can effectively find the lower and upper execution time limits. They claim that there is not much support for temporal testing and often testers just use the methods that are designed to test the logical correctness. In their research, GA based testing was found to be much more effective than the random testing and particularly effective when a problem has many variables and a large input domain. Also they introduced the term "evolutionary testing".

Pargas et al. (Pargas et al., 1999) have experimented with genetic algorithm based test data generation for statement and branch coverage using a control-dependence graph to guide optimization. They tested six relatively small test programs and compared the results to the random method. Their approach clearly outperformed the random method for three of the six test programs, for the other three programs both methods found the optimal solution in the initial population. They suggest that the use of a GA could be more beneficial for complex programs. A minimizing version of the objective function of Pargas et al., can be computed as $(dependent - executed)$.

The main problem with this objective function is the coarseness of the resulting fitness landscape and the existence of several fitness plateau. This reduces the ability of the fitness function to guide the search to the target. As many test cases will have the same fitness values, it will have difficult to compare between these test cases.

The PhD thesis by Tracey (Tracey, 2000) deals with automatic test data generation for testing safety-critical systems. He uses simulated annealing and genetic algorithms, but also random search and hill climbing as the optimization methods. He defines the framework on how to use them for generating test data for temporal WCET (Worst Case Execution Time) testing, assertion based testing, and structural testing. In this work Tracey identifies the control dependent nodes for the target structure. If an individual takes a critical branch from one of these nodes, a distance calculation is performed using the branch predicate of the required, alternative branch. This is computed using the functions of Table 2.2. Tracey then uses the number of successfully executed control dependent nodes to scale branch distance values. The published formula used by Tracey for computing the fitness function is :

$$\frac{executed}{dependent} * branchdist,$$

where *branchdist* is the branch distance calculation performed at the branching node where a critical branch was taken. This fitness function can lead to unnecessary local optima in the objective function landscape. $\frac{executed}{dependent}$ increases but should decrease and even $\frac{dependent}{executed}$ although it decreases it can be dominated by branch distance.

In the example of Figure 2.4, the control dependent nodes for the target structure are identified. A distance is calculated using the branch predicate of the alternative branch if an individual takes a critical branch from one of these nodes. In this example, the first and second "if" statements are the critical nodes. A valley exists where $\frac{1}{2} * |x| \leq |y|$ and $x \neq 0$. Figure 2.5 shows the landscape of the example in Figure 2.4.

```
void landscape(int x, int y) {
        if(x == 0){
            if(y ==  0) {
                //required executed
        }
    }
}
```

Figure 2.4: Example to show Tracey's objective function.

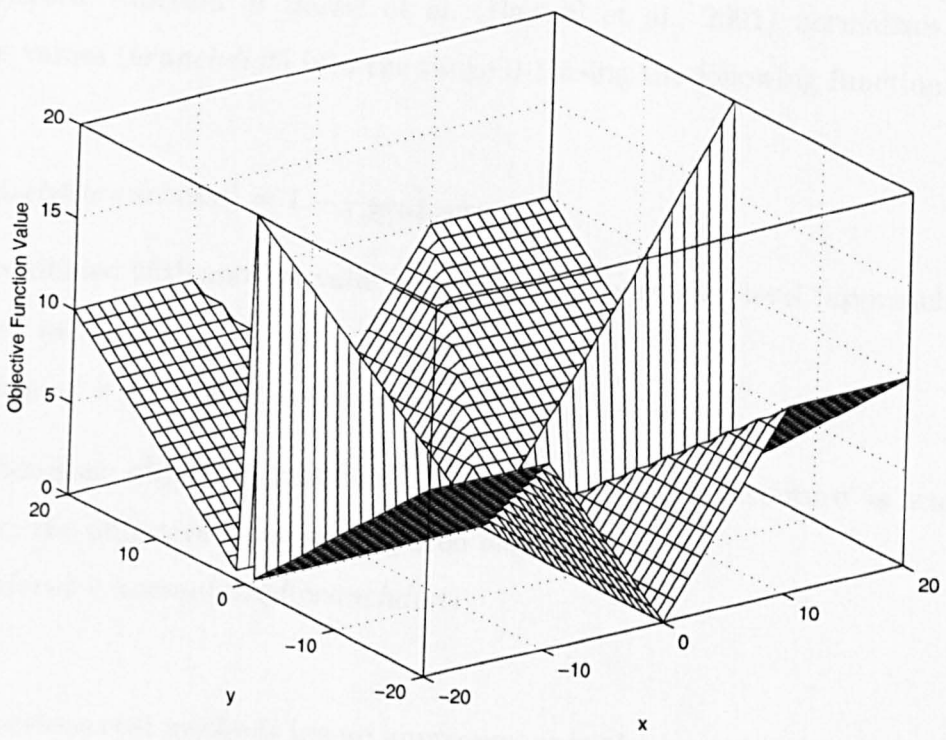


Figure 2.5: Objective function landscape of Tracey (Tracey, 2000) for example of Figure 2.4.

Relational Predicate	Objective Function obj
Boolean	if TRUE then 0 else K
a = b	if abs(a - b) = 0 then 0 else abs(a - b) + K
a $\neq$ b	if abs(a - b) $\neq$ 0 then 0 else K
a < b	if a - b < 0 then 0 else (a - b) + K
a $\leq$b	if a - b $\leq$ 0 then 0 else (a - b) + K
a > b	if b - a < 0 then 0 else (b - a) + K
a $\geq$ b	if b - a $\leq$ 0 then 0 else (b - a) + K
$\sim$ a	Negation is moved inwards and propagated over a

Table 2.2: Tracey's objective functions for relational predicates. The value K, K > 0, refers to a constant which is always added if the term is not true

The objective function of Barsel et al. (Baresel et al., 2001) normalizes branch distance values (*branchdist*) into the range 0-1 using the following function:

$$normalizebd(branchdist) = 1 - \frac{1}{1.001^{branchdist}}$$

This is combined with another value called the approximation level (approach level), calculated as follows:

$$approachlevel = dependent - executed - 1$$

The minimising objective function is zero if the target structure is executed, otherwise, the objective value is computed as:

$$approachlevel + normalizebd(branchdist)$$

All the previous cost methods ignore improvement in the cost of disjunction (*min*) of the more costly operand. Consider, for example, the program fragment in Figure 2.6. To execute the goal the values **x** and **y** should be equal to 0. Suppose there are test cases as shown in Table 2.3. The cost of the predicate is a flat surface; even

```
...
if(x == 0 || y == 0){
   //executed target
}
```

Figure 2.6: Example program using *min* as cost for disjunction

the second test case is more "close" to solution than the others. Clearly, when the operands have different truth values, the cost of the disjunction should be the cost of the true operand. This leaves the cases where the operands have the same truth value. In this case, a popular choice for the cost of a disjunction is the cost of the operand with the lowest cost i.e. the cost function is the *min* function. The common corresponding cost function for the conjunction is the *max* function. The issue of cost functions to use for logical operators *or* and *and* is addressed by Bottaci (Bottaci, 2003). The cost function of *or* and *and* are shown in Table 2.4, where a and b are positive ($false$) and a', and b' are negative ($true$), a, b are never zero.

x	y	*cost*
1	10	1
1	1	1
1	99	1
1	999	1
1	50	1

Table 2.3: Test cases for example in Figure 2.6. Note differences in the larger value are ignored.

These costs can be illustrated with an example showing three failed and two successful attempts to execute the predicate $a \leq b$ for various integer values of a and b as shown in Table 2.5. The predicate cost function $a - b$ when the predicate is

a	b	$a \bigvee b$	$a \bigwedge b$
a	b	$\frac{ab}{a+b}$	$a+b$
a	b'	b'	a
a'	b	a'	b
a'	b'	$a'+b'$	$\frac{a'b'}{a'+b'}$

Table 2.4: Logical *or* and logical *and* cost table (from (Bottaci, 2003))

false and $a - b - 1$ when the predicate is true.

a	b	$cost$	$or - cost$	$and - cost$
8	3	5	5	5
6	3	3	$\frac{15}{8}$	8
5	3	2	$\frac{30}{31}$	10
3	3	-1	-1	10
1	3	-3	-4	10

Table 2.5: Cumulative or-cost and and-cost for the predicate $a \leq b$ for the values listed

Bottaci (Bottaci, 2005) also suggests a new method of finding a cost for branch in the loop as follows: Each reached branch maintains two cost values, both derived from the associated predicate cost function. One cost value is the cost that all attempts to execute the branch are successful. This is called the cumulative $and - cost$. The other cost value is the cost that any attempt is successful, called the cumulative $or - cost$. For example, in the following piece of code:

```
public f(int x){
 for (int i =0; i< 3;i++){
```

```
if (x == 0){//cost1 , cost2, cost3 for each iteration 0,1,2 respectively
      //Target executed
 }
}
```

The accumulative cost according to Bottaci is (cost1 $\vee$ cost2 $\vee$ cost3) or (cost1 $\wedge$ cost2 $\wedge$ cost3), and this can be found depending on the previous equation of disjunction or conjunction.

The value $and - cost$ is positive and the value $or - cost$ is negative when both branches at a conditional node have been executed. The cost values produced by relational predicates are normalised to lie with in [-1, 1] using the following formula (let c be the branch distance value):

$$\begin{cases} 1 - \frac{1}{1+c} & \text{if c} > 0 \\ \frac{1}{1-c} - 1 & \text{if c} < 0 \\ 0 & \text{otherwise} \end{cases}$$

Bottaci incorporates the two notions of branch distance and approximation level into a single predicate expression based on the control dependency condition for a branch. This condition is expressed as a conjunction of the predicates and the evaluation function for AND is used. For each statement in the program under test there is a reachability condition such as: (P $\wedge \sim$ Q $\wedge$ ...) $\vee$ (R $\wedge \sim$ S $\wedge$ ...), where (P $\wedge \sim$ Q $\wedge$ ...) is a single control dependency path to target and (R $\wedge \sim$ S $\wedge$ ...) is an alternative control dependency path to the target. Any non-reached branch predicates for which a cost value is not available are given a maximum predicate expression cost value that depends on the number of relational predicates in the branch predicate. This maximum value is equal to the largest cost value that may be generated at any branch predicate in the program after normalisation. This can be calculated at compile time by counting the number of conjuncts in a conjunctive predicate expression where each conjunct is a relational predicate which has a max cost of 2.0 after normalisation. In the following example there are three relational predicate expressions and so the max is equal 2.0 + 2.0 + 2.0 = 6.0 which is the cost given to this branch if it is not reached.

```
if (a == b && x == y && t == w){
    //do something
}
```

The cost of all the boolean expressions in the empirical work reported in this thesis were calculated according to the above scheme.

Chapter 3

Search-based Software Test Data Generation for String Data using Program-Specific Search Operators

3.1 Introduction

Current test data generation work (Baresel et al., 2001), (Harman et al., 2002), (McMinn et al., 2005), (Korel, 1990) has been limited largely to programs in which predicates compare numbers, as illustrated in the example below.

```
if (y == 30) {
    //TARGET
  }
```

These can be dealt with using cost functions discussed in the previous chapter.

Unlike numeric equality predicates, string equality does not suggest a single straightforward cost function. In research that has considered string predicates, one approach has compared strings by comparing their underlying character bit string representations using the bit Hamming distance as a cost function (Jones et al., 1996). Another approach (Zhao, 2003) reduces the problem of string search to a sequence of character searches. In this approach, only a character matching cost function is used and characters are matched as numbers. There are a number of string matching algorithms, used in information retrieval and biological applications, which may be useful for defining cost functions but as yet none of these have been applied to the problem of searching for string test data.

This chapter consists of two main parts. In the first part, some potential string cost functions and corresponding search operators are considered, including two new cost functions. These cost functions are assessed by comparing their performance on a number of sample test programs. In the second part of this chapter, a new type of search operator is introduced with the aim of biasing the search towards strings that occur as literals in the program under test. The performance of the cost functions when used with the new operators are assessed by again generating test data for the sample programs. The results show that the new search operators provide a substantial improvement in efficiency. The main contribution of this chapter is that it provides a new method for searching for string test data and demonstrates that it is potentially quite efficient.

3.2 Cost Functions and Search Operators for String Predicates

This section considers the extent and nature of the string search space that is relevant to programs in practice. It then considers some existing cost functions and some new cost functions for the string predicates of equality and ordering.

3.2.1 String search space

Modern software uses 16-bit character strings. The space of strings formed from the 16-bit character set is huge, so much so that a search process may be prohibitively slow to be of practical benefit. A preliminary investigation was thus done to try to establish the size and structure of the space of strings that are used in practice. A large body of software, the .NET Framework Shared Source Common Language Infrastructure (SSCLI) (Stutz et al., 2003), was scanned to extract string literals. The strings were in turn scanned in order to produce a frequency distribution for the occurrence of each character in a SSCLI string literal. In over 13 million characters, only 850 were outside the 8-bit range. Over 99% of the characters were within the range of the 95 "printable" characters from the space character to the tilde. In practice, this means that the vast majority of the 16-bit character set need never be explored when searching for test data for typical programs. The examination of the SSCLI source code also showed that about 6% of the predicate expressions are string predicate expressions and about 91% of the string predicates are string equality.

In order to exploit the marked non-uniformity of the distribution of characters typically used in string processing programs, in the work reported here, characters were restricted to have an ordinal value between 0 and 127, i.e. within the lower 7-bits. Characters outside this range were excluded entirely because the vast majority of programs do not require them. Since non-printing characters occur in the strings of only a very small proportion of programs, it is probably better to deal with these as a special case. This might be done by the tester, with a knowledge of the program under test, setting the parameters of the search space to a specific set of characters.

It was also observed that many of the string literals consisted of English or English-like text such as might be used for identifiers, the names given to products, organisations, etc. This means that not all strings in the space of 7-bit character strings are equally likely to be required as test data. It is clearly advantageous to bias the search for strings according to the distribution of strings that occurs in practical programs.

Given a relative frequency table for the occurrence of character pairs in English text, a random English-like string may be constructed as follows. Initially, the first character is selected randomly from the English alphabet. The selection of subsequent characters, however, is biased so that consecutive pairs of characters in the string occur with the same relative frequency as they occur in English language texts, so as example if the first character is 'A' then the second character will not be 'E' because the probability of 'A' followed by 'E' is zero (Table 3.1). This approach can be applied to natural languages other than English of course, providing the language can be identified. Again, this is probably most easily done by the tester. Some strings cannot be generated by this method, e.g. the string "ae", and so an additional string generator generates strings with characters selected independently and with a uniform random distribution.

The search space also depends on the minimum and maximum length of input string that is generated. It is clearly inefficient to set a maximum length that is greater than is required for a particular program. It is thus expected that the tester, with knowledge of the program under test, will specify both the minimum and maximum length of input string.

A random string from the set of "practical" strings may therefore be constructed by selecting a random string length and then selecting, with some specified probability, from strings generated from characters selected from a uniform distribution over the 7-bit character space or from English-like strings.

Search operators and the role of the cost function

The cost function estimates the number of search operations that must be performed to transform the candidate into a solution. Clearly, the number of search operations required to transform a given candidate into a solution depends on the nature of the transformation performed by each operator. This important dependence between the cost function and the search operators can be illustrated by considering a simple search problem over the set of eight 3-bit strings from `000` to `111`. Assume that

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	1	9	15	24	0	5	9	2	18	0	6	26	12	84	0	9	0	5	44	48	4	9	6	0	12	1
B	5	1	0	0	22	0	0	0	3	0	0	8	0	0	9	0	0	5	1	1	10	0	0	0	5	0
C	15	0	2	0	18	0	0	18	4	0	8	4	0	0	21	0	0	5	0	7	3	0	0	0	1	0
D	22	9	4	6	26	5	3	12	23	1	1	5	7	8	19	3	0	7	16	26	5	1	9	0	4	0
E	51	11	21	58	22	14	7	15	19	1	2	25	22	51	15	13	1	78	53	39	3	11	23	4	12	0
F	12	1	1	1	9	5	0	4	11	0	0	4	2	1	18	1	0	8	2	17	4	0	2	0	1	0
G	11	2	1	1	13	1	2	16	8	0	0	3	1	2	11	1	0	6	4	7	3	0	2	0	1	0
H	53	1	1	1	140	1	0	3	45	0	0	1	1	1	23	1	0	3	2	15	4	0	2	0	2	0
I	4	2	15	15	10	8	12	3	0	0	4	17	17	83	11	3	0	13	40	43	0	7	3	1	0	1
J	1	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	4	0	0	0	0	0
K	3	1	0	0	14	1	0	1	7	0	0	1	0	4	2	0	0	0	2	2	0	0	1	0	1	0
L	19	2	1	17	30	5	1	2	22	0	2	26	2	1	19	1	0	1	5	7	4	1	2	0	14	0
M	21	3	0	0	31	1	0	2	11	0	0	1	2	1	14	5	0	1	5	4	4	0	2	0	8	0
N	19	3	12	68	31	4	43	8	15	1	4	4	3	4	28	2	0	1	15	49	3	1	6	0	5	0
O	7	6	6	8	2	37	3	5	6	0	6	12	22	49	17	7	0	39	13	26	53	6	22	0	2	0
P	11	0	0	0	14	0	0	2	5	0	0	7	0	0	10	5	0	8	2	5	3	0	1	0	1	0
Q	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0
R	23	3	5	9	61	4	5	6	23	0	4	5	6	8	25	3	0	6	17	19	5	2	5	0	10	0
S	31	5	8	3	34	5	2	23	23	1	3	6	6	4	26	9	1	2	20	50	9	1	11	0	2	0
T	27	5	5	3	35	4	1	153	35	1	1	8	5	3	53	2	0	13	16	26	8	0	14	0	7	0
U	3	2	6	3	3	1	6	1	4	0	1	16	4	17	0	8	0	20	16	22	0	0	1	0	0	0
V	2	0	0	0	30	0	0	0	5	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0
W	31	0	0	1	18	0	0	20	18	0	0	1	1	5	13	0	0	1	2	2	0	0	1	0	1	0
X	1	0	1	0	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
Y	8	3	3	3	6	3	1	4	5	0	0	2	3	2	17	2	0	1	9	9	1	1	5	0	1	0
Z	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 3.1: Relative frequency of character pairs in English text from (Leon, 2002)

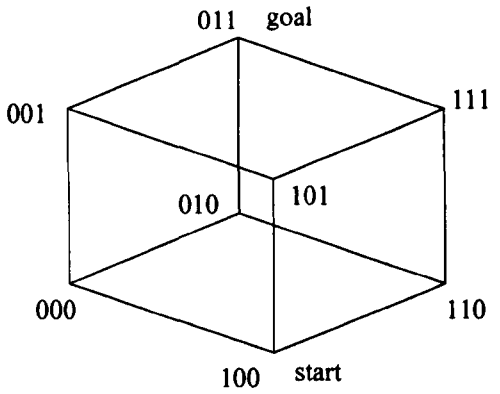


Figure 3.1: The search space defined by the 8 3-bit strings and a single bit inversion search operator.

these strings constitute a set of inputs that is to be searched for the single input, `011`, which executes the target branch as shown below.

```
P(s) {
  t = F(s);
  if (t == 011) {
     //TARGET
```

For simplicity of explanation, a simple hill-climbing search process will be used. In each iteration, each candidate string `s` is submitted to the program P and the string `t` is compared to the required or goal string by computing a Hamming distance. If the candidate is not a solution new candidates are generated from it, in this example by a search operator that inverts each bit to produce new candidate strings. The string (or one of the strings) with the lowest cost is used as the start for the next iteration. In this example, assume, initially at least, that F is the identify function and that the initial candidate is `100` from which the bit-inversion search operator generates the strings, `000`, `110` and `101`. The original string is discarded since the lowest Hamming cost is now 2. One of the new strings is selected and the iteration repeats. It is clear that this search converges rapidly to the goal string of `011`.

The space induced by the bit inversion operator on the set of eight strings is shown in Figure 3.1. Since each edge represents an application of the search operator, the

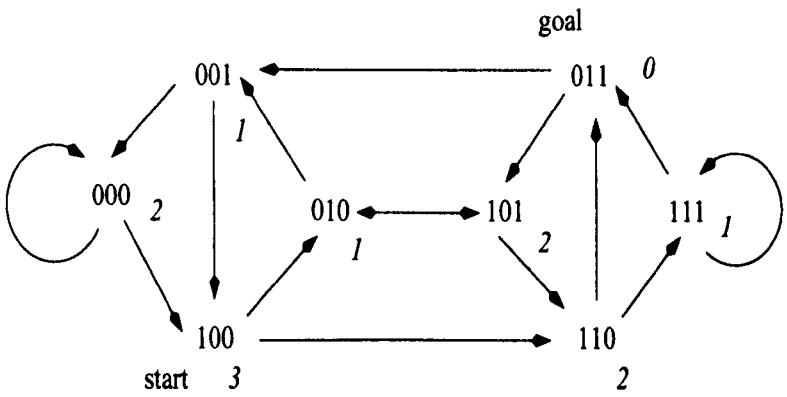


Figure 3.2: The search space defined by the set of eight 3-bit strings and a right bit-insert operator.

Hamming distance between any two strings is precisely the number of applications of the search operator required to "search" for one string from the other. As a consequence of this, the Hamming distance is a reliable cost function for this space.

The reliability of the Hamming distance is, however, dependent on the use of the bit inversion search operator. This can be illustrated by considering an alternative search operator that right shifts a random bit into a given string and discards the rightmost bit. The space produced by this operator is shown in Figure 3.2. Consider again the problem of finding the solution string 011 by applying this new search operator to the initial candidate 100. From Figure 3.2, it can be seen that this can be achieved by two applications of the search operator. Notice, however, that in this case, the Hamming distance (shown against each string) does not correspond to the number of applications of the search operator required to reach the solution. Moreover, if this Hamming distance were to be used to guide the search, the search would follow a path from 100 to 010 to 001 where it would remain stuck at a local minimum. For the right bit-insertion operator, the Hamming distance is not a reliable cost function. An appropriate cost function would count the number of right bit-insertion operations (equivalently, the edges between any two strings in the space shown in Figure 3.2) between a given string and a solution string.

The above example illustrates a general principle concerning the duality between search operators and cost functions. A realistic search problem, however, is com-

plicated by factors omitted from the example. The search operators are applied to values in the input domain but the cost function is applied to values that are arguments to the target branch predicate expression. The example avoids this issue because F is assumed to be the identity mapping; in general, it is not. In the general case, the cost function may lose some of its reliability but often it is still sufficiently reliable to guide the search to a solution.

Mutation operators

There are three basic kinds of string mutation operator: deletion, insertion and substitution. A single deletion operator was defined to delete a random character from a given string. Two insertion operators were defined. The uniform insertion operator inserts a character, selected randomly from the range 0 to 127, into a given string at an insertion position selected randomly. Over time, random insertions within an English-like string will reduce its English-like property. To counter this, an English-like insertion operator was defined. This operator inserts a letter from the English alphabet selected probabilistically according to letters that precede and follow the insertion point. If the insertion point is not preceded or followed by a letter from the alphabet, a random letter is inserted according to the frequency of single English letters.

Two character substitution operators were defined. To accompany the binary Hamming cost function (see below), a binary character substitution operator was defined. This operator inverts a random bit within the 7 low order bits of a character. To accompany the cost functions that compare the ordinal values of characters, a character substitution operator was defined to replace a character with another of a similar ordinal value. The new character is selected from a Gaussian distribution with mean at the given character. If the mean is selected then the new character is chosen randomly from the characters adjacent to the mean. The standard deviation of the distribution was set, rather arbitrarily, at 16.

3.2.2 String equality cost functions

Of the many string matching metrics in the literature (Navarro, 2001), some use a vector space approach in which each string is equated with a point, in a Euclidean space, say and the Euclidean distance between the two points is used to derive a match cost. Another category of string matching metrics produces a distance value in terms of the number of primitive operations, typically, insertion, deletion and substitution of single characters, required to transform one string into another. Functions differ in terms of the particular costs attached to the particular operations. In some applications, for example, it is more important to match digits than letters or leftmost character matches may be more important than matches in other positions. In a spelling correction application, for example, the cost of substituting one character for another may depend on the proximity of the two characters on the user's input keyboard or on-line dictionary. The approach of considering the number of operations required to transform one string into another seems to be the most promising for defining a cost function that estimates the number of search operations required to transform a string to a solution. It is only this second type of metric that is considered in this thesis.

Binary Hamming distance

Traditionally, many genetic algorithms have used a binary string representation for candidates. The mutation operator has been bit inversion and binary Hamming distance (HD) has been the cost function. This representation could be used for character strings by simply working with the underlying bit representation of each character. Once each string is converted to a bit string by concatenating the bit patterns of each character, the Hamming distance of two equal length strings may be easily computed.

Since character strings vary in length, a Hamming distance must be defined for unequal length strings. The comparison of unequal length strings requires consideration of the cost of inserting or deleting characters. Strictly, a binary

representation should allow only single bits to be inserted and deleted, which is clearly inappropriate, but a character insertion can be considered to be 7 consecutive bit insertion operations. On the assumption that each bit operation should have an equal cost, the cost of inserting additional bits or removing excess bits was thus taken to be equal to the cost of mismatched bits. The Hamming distance function was therefore extended so that any bits in one string that extend beyond the length of the shorter string are counted as mismatched. More formally, the distance between two strings A and B is

$$HD(A,B) = (\sum_{j=0}^{minlen} \sum_{i=0}^{i<7} (A_i|B_i)) + 7(maxlen - minlen).$$

where $minlen, maxlen$ are the minimum length and maximum length of string A and B, A_i and B_i are bits number i and | is XOR operator. For example if:

A = "SET"

B = "CASE"

$HD(A,B)$ =(1010011 | 1000011)+(1000101 | 1000001) + (1010100 | 1010011)+ 7*1

$HD(A,B) = 1 + 1 + 3 + 7 = 12$

Strings are left-aligned, which may lead to unrepresentative costs because of a failure to take account of deletion and insertion. An example is HD(`XHELLO`, `HELLO`) where only one `L` matches and the cost is therefore relatively high.

A problem with standard binary encoding is the disparity that can occur between solutions that are close to each other in unencoded solution space, but are far apart in the encoded binary representation. For example in a standard binary encoding the integer 63 is represented as 0111111, yet 64 is represented as 1000000. Therefore, the crossover and mutation operators must change all 7 bits to move from one integer value to the neighbouring other.

Another limitation in this method is the maximum number of fitness values it can produce, which is $\leq$ (7 * $maxlen$), where $maxlen$ is the maximum length of the two strings compared. The example program fragment in Figure 3.3 requires s = "MN" to be executed. If the length of input string s is restricted and equal to 2 and the

```
if (s == "MN") {
    //TARGET
    }
```

Figure 3.3: Simple branch requiring MN to be executed

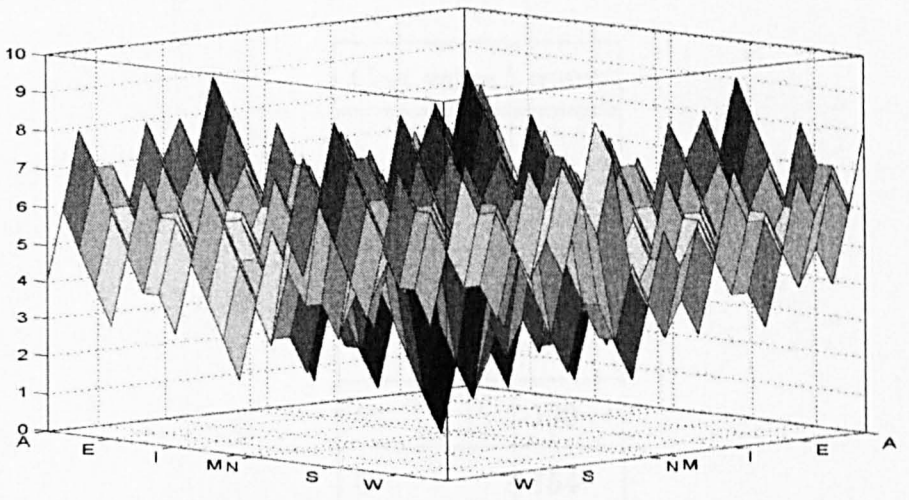


Figure 3.4: Object function landscape for HD in 26 characters domain for example in Figure 3.3. Cost of matching a string of 2 characters with "MN"

domain size is 128, then there are 128 * 128 different combinations of two characters (...,"AA",...,"zz",...). The number of fitness values, however, is 7 * 2 = 14. So there are (128 * 128)/14 $\approx$ 1152 different input strings that have the same fitness value (e.g. "MO" and "MM").

Suppose that the domain is restricted to 26 letters ('A',..., 'Z'). For this domain the possible number of string combinations is 26 * 26 = 676. Table 3.2 shows the cost values and the frequency of this cost value. As shown in this table the frequency varies from 1 to 154 and the average is 676/14 = 48. Figure 3.4 shows the HD cost landscape of program in Figure 3.3.

Cost value	count
0	1
1	8
2	32
3	78
4	130
5	154
6	137
7	86
8	39
9	10
10	1

Table 3.2: The cost values and frequency in 26 characters domain ('A',...,'Z'), number of fitness values = 11 i.e this restricts the search method.

Character distance

Rather than using a binary space in which to map characters and strings, characters may be mapped into an ordinal space according to each character's ordinal value. In this space, a substitution operator would be sensitive to the ordinal value of the character it substitutes. This suggests a new cost function based on the pairwise comparison of character values. This new cost function, called character distance (CD) is defined as the sum of the absolute differences between the ordinal character values of corresponding character pairs. For strings of unequal length, any character without a corresponding character increases the cost by 128, the size of the character search space. More precisely, let string $s = s_0 s_1 \ldots s_{k-1}$ be of length k where s_i is the ordinal value of the ith character. Similarly, let string $t = t_0 t_1 \ldots t_{l-1}$ be a string of length $k \leq l$ then

$$CD(s,t) = \sum_{i=0}^{i=k-1} |s_i - t_i| + 128(l-k).$$

Strings are left aligned and absent characters are treated as null characters. For example if:

s = "SET"

t = "CASE"

$CD(s,t)$ = |*ascii*('S')-*ascii*('C')|

+ |*ascii*('E')-*ascii*('A')|

+ |*ascii*('T')-*ascii*('S')|

+ 128(4 - 3)

$CD(s,t) = |83 - 67| + |69 - 65| + |84 - 83| + 128 = 149$

Figure 3.5 shows the CD cost landscape of the program in Figure 3.3.

CD, like HD, is sensitive to the alignment of different length strings.

Edit distance

Of the many existing string comparison metrics used in information retrieval and biological matching, the vast majority are derived from the edit distance. The edit

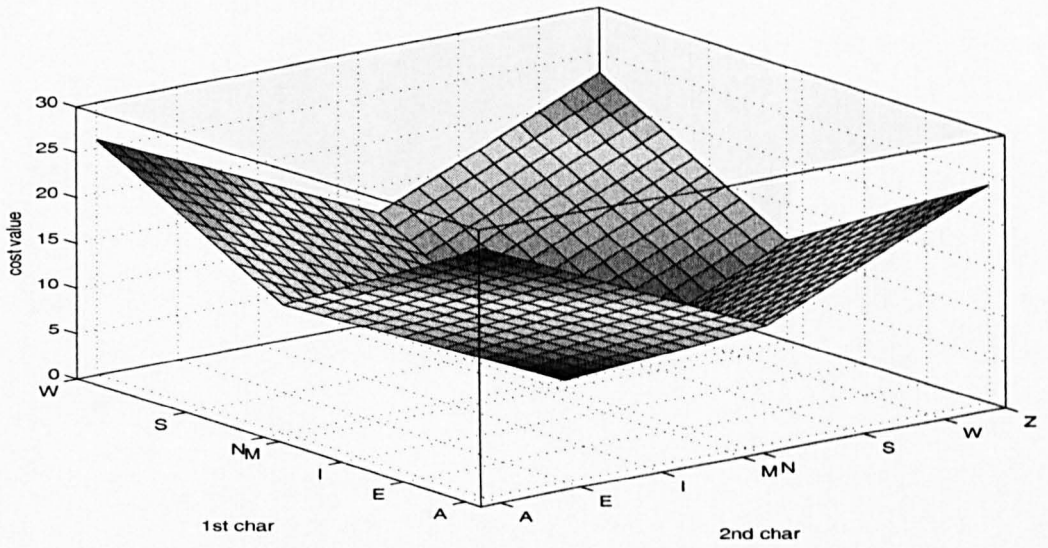


Figure 3.5: Object function landscape for CD in 26 characters domain for example in Figure 3.3. Cost of matching a string of 2 characters with "MN"

distance (ED) (or Levenshtein distance (Navarro, 2001)) is derived explicitly from consideration of three operators that perform character insertion, character deletion and character substitution. The edit distance between two strings is the minimum number of deletions, insertions, or substitutions required to transform one string into another. For example $ED(\texttt{TEST}, \texttt{TOT}) = 2$, because 1 deletion (or 1 insertion) and 1 substitution is sufficient to match the two strings. The edit distance function is defined by the recurrence relation below where $s : a$, $t : b$ are character strings, each consisting of a possibly empty string s, t, followed by the character a, b.

$$ED(s : a, t : b) = min(ED(s : a, t) + 1, ED(s, t : b) + 1, ED(s, t) + ED(a, b))$$

The edit distance of two characters is one unless they are equal, in which case it is zero. The edit distance of an empty string and a given string is the length of the given string.

In considering the suitability of the edit distance as a cost function, the size of the range is important. The range of edit distance values is equal to the maximum length of the two strings compared. Consider for example, that there are over 10^{10} strings in the space of strings of length 5 and yet only 6 edit distance values. This

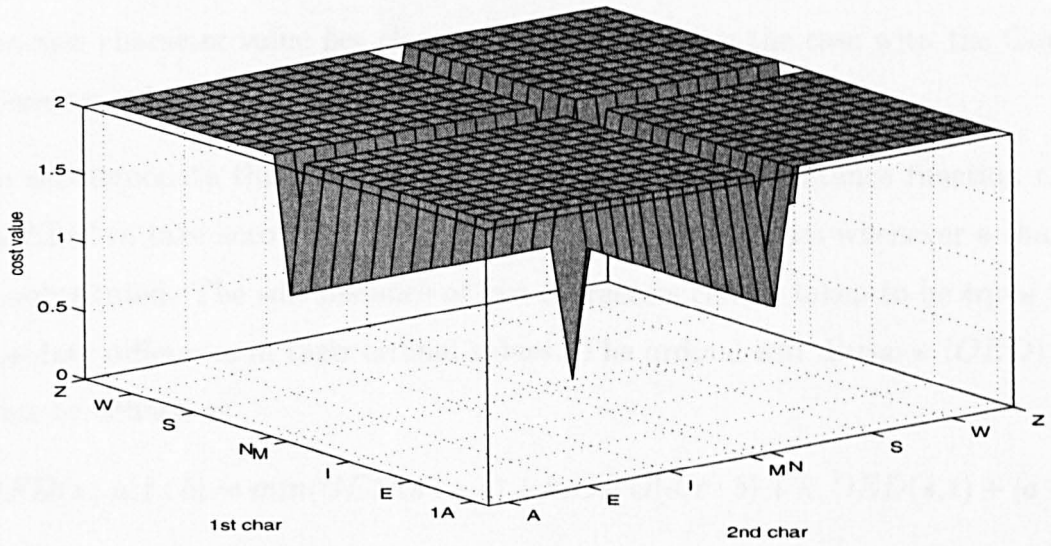


Figure 3.6: Object function landscape for ED in 26 characters domain for example in Figure 3.3. Cost of matching a string of 2 characters with "MN"

means that the "surface" of the cost function produced by evaluating each point of the search space with respect to a given goal will be largely flat.

Figure 3.6 shows the *ED* cost landscape of the program in Figure 3.3, the objective function contain 3 values (0, 1 and 2) and gives no guidance as to change from 'A' to 'M'.

Ordinarily, such an indiscriminate cost function would provide little guidance to the search but the cost function is nonetheless reliable given the search operators it assumes. A cost of one for the substitution of a character by any other character assumes a search operator which generates, in a single step of the search, all strings that may be formed by a single character substitution. In a practical search, the number of successors that such an operator would produce is too large to allow the easy identification of the best successor.

A more practical substitution operator would produce a single string by substituting a given character with a single character. If the new character value is defined to be adjacent to that of the character it replaces, then the difference in ordinal values estimates the number of substitution operations required to substitute one

character for another. The ordinal distance is also a reasonable estimate when the new character value lies close to the original, as is the case with the Gaussian character substitution operator.

To accommodate this kind of search operator, the edit distance function can be modified to take account of the difference in character values whenever a character is substituted. The edit distance of two characters can be taken to be equal to the absolute difference in their ordinal values. The ordinal edit distance (OED) could thus be defined as

$$OED(s:a,t:b) = min(OED(s:a,t)+k, OED(s,t:b)+k, OED(s,t)+|a-b|)$$

where k is the insertion or deletion cost and a, b in $|a-b|$ are interpreted as ordinal values.

To choose a value for k, note that the argument for the practicality of using a character substitution operator that, given a source string, produces a single modified string, applies also to the insertion operation. A practical insertion operator would therefore produce a single string by insertion of a single characterinto the source. The number of insertion operations now required to insert a given character is a function of the character set size. For this reason, the cost of insertion, k, was chosen to be 128. Given that any match that can be achieved by an insertion into one string can also be achieved by a deletion in the other, the cost of deletion was also chosen to be 128.

Using 128 as the cost of insertion and deletion, however, gives OED(`XHELLO`, `HELLO`) = 128 and yet OED(`GDKKN`, `HELLO`) = 5 which is too low since the search effort required to match `GDKKN`, `HELLO`, five substitutions, should be higher than the effort to match `XHELLO`, `HELLO`, a single deletion. The problem is that substitution costs become unreasonably low as corresponding character values approach each other. The low, non-zero substitution costs were therefore offset away from zero while retaining the maximum cost at 128 (the maximum cost is 127 but for consistency 128 is used). This was done by setting the substitution cost to be $128/4 + 3|a-b|/4$ when $|a-b| > 0$ and zero otherwise. This gives

		C -- (67)	A --(65)	S -- (83)	E -- (69)
	0	128	256	384	512
S -- (83)	128	44	172	256	384
E --(69)	256	161.5	79	207	256
T --(84)	284	289.5	207	111.75	239.75

Table 3.3: OED example calculation

OED(`GDKKN`, `HELLO`) = 163.75 which is higher than 128. OED was thus defined as

$$OED(s:a,t:b) =$$
$$min(OED(s:a,t)+128, OED(s,t:b)+128, OED(s,t)+128/4+3|a-b|/4).$$

$\frac{1}{4} : \frac{3}{4}$ is an arbitrary value which was found to be effective, $\frac{1}{3} : \frac{2}{3}$ was also tried and was effective. The precise value is not important providing it is in the range $\frac{1}{5} : \frac{4}{5}$ to $\frac{1}{3} : \frac{2}{3}$.

For example if:

s = "SET"

t = "CASE"

$OED(s,t)$ = 239.75, the solution is shown in Table 3.3.

Figure 3.7 shows the OED cost landscape of the program in Figure 3.3.

A significant difference between HD and CD compared to OED is that the cost of OED does not depend on a left-most alignment of strings. For example, in OED(`XHELLO`, `HELLO`) five characters match.

String ordinal distance

Zhao et al. (Zhao, 2003) propose a string comparison cost function. In this function, each character string is represented by a nonnegative integer. This integer is constructed by regarding each character of the string as a "digit" in a base equal to the character set size. If 128 is the size of the character set then the numerical value of the string $s = s_0 s_1 \ldots s_{k-1}$ is $\xi(s) = 128^{k-1}s_0 + 128^{k-2}s_1 + \ldots + 128^2 s_{k-3} +$

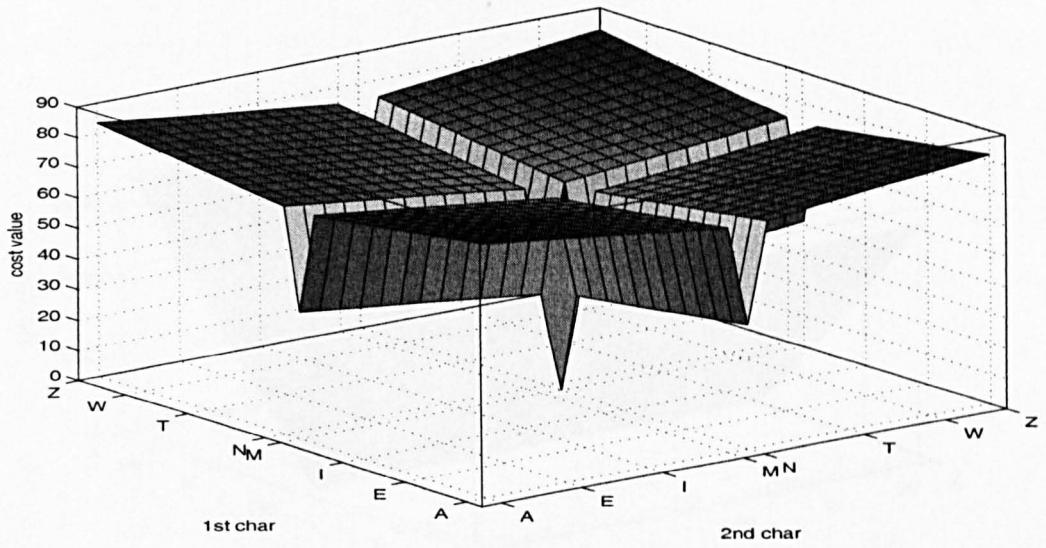


Figure 3.7: Object function landscape for OED in 26 characters domain for example in Figure 3.3. Cost of matching a string of 2 characters with "MN"

$128s_{k-2} + s_{k-1}$. The string ordinal distance cost function (OD) is thus defined for two strings s, t as $OD(s,t) = |\xi(s) - \xi(t)|$.

For example if:

s = "SET"

t = "CASE"

$\xi(s) = 84 * 128^0 + 69 * 128^1 + 83 * 128^2$

$\xi(s)$ =1368788

$\xi(t) = 69 * 128^0 + 83 * 128^1 + 65 * 128^2 + 67 * 128^3$

$\xi(t) = 141584837$

$OD(s,t) = |\xi(s) - \xi(t)| = 140216049$

Figure 3.8 shows the OD cost landscape of the program in Figure 3.3. In this Figure the valley in the fitness landscape along the line 1st char = 'M' appears to be horizontal but is in fact descending towards (1st char = 'M', 2nd char = 'N') because the gradient is only 13. This caused by the dominance of the cost function by the value of the 1st char and illustrating the problem with the cost function. In Figure 3.9, the gradient is visible because the cost varies only with the second character.

In practice, $\xi(s)$ may be very large for long strings. OD may be too large to be

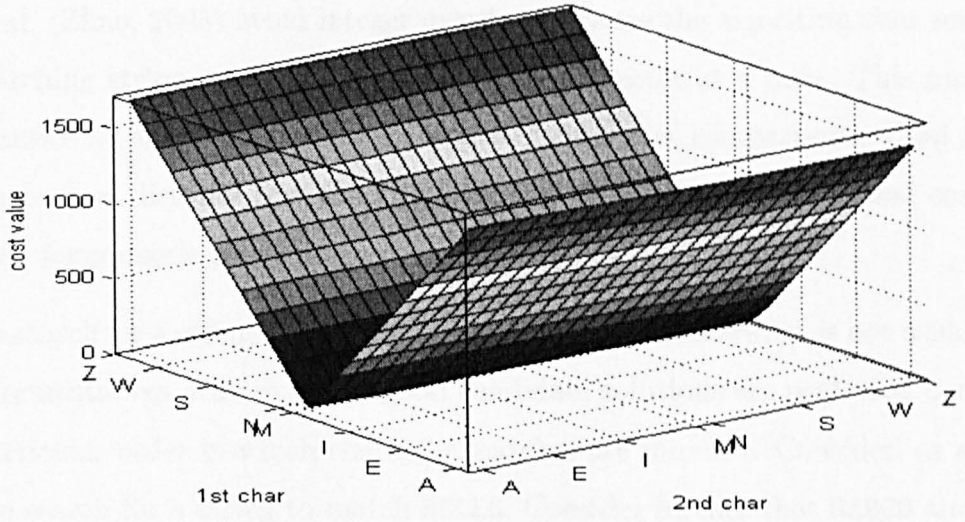


Figure 3.8: Object function landscape for OD in 26 characters domain for example in Figure 3.3. Cost of matching a string of 2 characters with "MN".

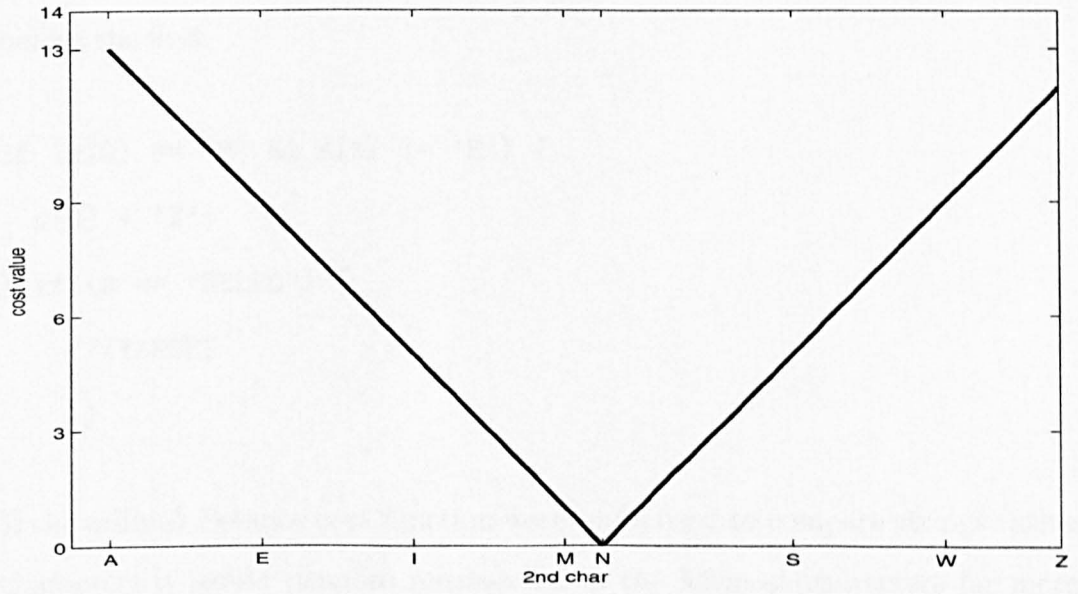


Figure 3.9: Object function landscape for OD in 26 characters domain for example in Figure 3.3. The first character is 'M', the cost varies only with the 2nd character.

represented using programming language provided integer data types even when two strings differ in only a single character, if it is the leftmost character. Zhao et al. (Zhao, 2003) avoid integer overflow because the algorithm that searches for a matching string searches for only a single character at a time. This means that in practice all string comparisons are in fact character comparisons. Used in this way, the ordinal distance cost function becomes equal to the conventional cost functions used for numeric predicates.

To search for a string match, one character at a time, however, is not usually efficient. Circumstances arise in which good candidate solutions are neglected because of the particular order in which character matches are pursued. Consider, as an example, the search for a string to match `HELLO`. Consider further that `GABCD` and `XELLO` are two candidates in this search but the first character only is being used to give a cost for the match. Because `H` is closer to `G` than to `X`, the string `XELLO` is discarded in favour of `GABCD`. This particular example is clearly a mistake. The general problem is that a character by character search imposes the additional search problem of finding an order in which character positions are to be matched. The example program fragment below requires that the second character of the string be matched before the first.

```
if (s[0] == 'H' && s[1] != 'E') {
  s[0] = 'X';
} if (s == "HELLO") {
     //TARGET
     }
```

If the ordinal distance cost function were to be used to compare strings rather than characters it would penalise mismatches in the leftmost characters far more than mismatches in the rightmost characters. Given that each character is equally likely to be modified by a search operator, this difference in cost values renders the cost function unreliable in many situations. The ordinal distance cost function is also very sensitive to the lengths of the strings compared. For example, the distance between

`HELLO` and `XHELLO` is an order of magnitude higher than the distance between `HELLO` and `GABCD`. For these reasons, the string ordinal distance was considered unsuitable as a cost function for string equality.

3.2.3 String ordering

String test data may be required to satisfy string ordering predicates such as $s < t$. String ordering may be determined from the ordinal value of each character in the character set or it may be determined using language or culture-specific rules. It is only the ordering induced by the character ordering that is considered here. But this does not limit the work reported because the work relies only on the existence of an ordering.

Ordinal value ordering

For a given maximum string length, and when all strings are right null padded if required to achieve the maximum length, a set of strings may be totally ordered according to the ordinal value of the string. When two strings are out of order, the difference in ordinal values may be used as the cost OV, thus

$$OV(s,t,<) = \xi(s) - \xi(t) + 1, \qquad s \geq t$$
$$OV(s,t,<) = -OV(t,s,\leq), \qquad s < t$$
$$OV(s,t,\leq) = \xi(s) - \xi(t), \qquad s > t$$
$$OV(s,t,\leq) = -OV(t,s,<), \qquad s \leq t$$

This cost function has disadvantages, however. The problem of the very large values it produces has already been mentioned. In addition, it is not clear that the cost values produced are a reasonable estimate of the cost of satisfying a string ordering predicate.

Table 3.4 shows how the cost function is computed for `s < "T"` where `s` is a string variable. Before the comparison right fill short strings with 0 to make the strings's length equal. Different values with different length are assigned to s to illustrate the cost.

(s < "T")			
s	target	value	Result
"Z"	"T"	90 -84 + 1	7
"ZZ"	"T0"	(90 + 90 *128) - (84*128+ 0) + 1	859
"XX"	"T0"	(88 + 88 * 128) - (84 *128) + 1	601
"WZ"	"T0"	(90 +87 * 128)- (84*128) + 1	475
"WAA"	"T00"	(65 + 65 * 128+ 87 * 128 * 128) - (84 * 128 *128) +1	57538
"TB"	"T0"	(66 + 84 * 128) - (84 * 128) + 1	67

Table 3.4: String ordering example using ordinal method

In the majority of cases, when two random strings are compared for a given ordering, it is only the most significant character of each string that is relevant. Consider, for example, the cost of satisfying the predicate expression `XYA` $<$ `NKL`. This predicate may be satisfied by modifying only the first character in either string. This observation motivated the definition of the following string order cost function.

Single character pair ordering

The cost of satisfying the predicate expression `XYA` $<$ `NKL` should depend on the cost of satisfying `X` $<$ `N` since this is how the string predicate expression may be solved with the least modification to the strings. A predicate such as `X` $<$ `N` may be satisfied by modifying either character. In general, the cost of satisfying a character predicate is calculated according to the cost functions for numerical predicates (Bottaci, 2003). In the case of $<$, it is the difference in ordinal values plus one, in the case of $\leq$ it is the difference in ordinal values.

For some string predicate expressions, there may not be a choice of character that may be modified. For example, if the string alphabet consists only of the 26 letters from `A` to `Z` then two characters must be modified to satisfy `Z` $<$ `A`. Because twice as many character modifications are required, such predicates are considered more costly to satisfy.

In general, the cost of satisfying a string predicate expression may be based on the

number of character pairings where a single character modification is sufficient to satisfy the string predicate expression. In the example `XYA <NKL`, a single character modification is sufficient to satisfy the predicate expression only in the first pairing. Until the first pairing is at least equal, modifying characters of subsequent pairings is futile. In the example `NNP <NNC`, a single character modification in any three of the character pairings is sufficient to satisfy the predicate expression and consequently the cost of this example should be lower. The cost is highest when no such character pairings exist.

On the basis of the above observations, a cost function, known as single character pair (SCP) cost was defined. A cost for an unsatisfied string ordering predicate expression is calculated as follows: for each character pair formed from corresponding characters in two strings, left aligned and right null padded where necessary to be of equal length, a character pairing cost is calculated. A character pairing cost is calculated as follows: if no single character modification in that pair can satisfy the string predicate expression, the cost for that pair is 2×128. 2×128 was chosen to be significantly larger than 128 which is the largest possible cost for a single character modification. If modification of a single character in a pair, a, b may satisfy the string predicate then the cost for the pair is $a - b + 1$ assuming $a < b$ is required.

The cost of the string predicate expression is the sum of the character pair costs divided by the length of the longest string to give an average cost for the modification of a character in a character pair. When two empty strings fail a string ordering predicate, the cost is 2×128. For example the cost of is "$a0x00d$" < "$a0xffc$" ($0x00$ is null or a character of zero ordinal value and $0xff$ is a character of 127 ordinal value) equal to ((a -a + 1) + 2* 128 + (d - c + 1))/3. For consistency with the cost function for logical negation (Bottaci, 2003), the cost of a satisfied string ordering predicate expression is the arithmetic negation of the cost of the logical negation of the string predicate expression.

Table 3.5 shows how the cost function computed for `s < "T"` where `s` is a string variable.

(s < "T")			
s	target	value	Result
"Z"	"T"	(90 -84 +1)/1	7
"ZZ"	"T0"	((90 -84 +1) + (90 -0 +1))/2	49
"XX"	"T0"	((88 - 84 +1) + (88 - 0 + 1))/2	47
"WZ"	"T0"	((87- 84 +1) + (90 -0 +1))/2	47.5
"WAA"	"T00"	((87 -84 +1) + (65 -0 +1) + (65 - 0 +1))/3	45.33
"TB"	"T0"	((84 - 84 + 1) + (66 -0 +1))/2	34

Table 3.5: String ordering example using single character pair method

3.3 Empirical Assessment of String Search Operators and Cost Functions

In order to assess the reliability of the cost functions introduced in the previous section, an empirical investigation was done. A number of test programs were assembled and for each program, an attempt was made to generate inputs to achieve branch coverage. These programs, which include predominantly string predicates, are described in Table 3.6. The size of each program is given as lines of code (LOC). For each program, the total number of relational predicate expressions (RPE) in each program is listed, and in parentheses, the number of them that are string relational predicate expressions; the remaining relational predicates are numeric. One or more relational expressions may be combined with logical connectives into a branch predicate. The number of branches is also listed. The source code for each test object can be found in Appendix A.

Calc

This test object consists of one function, which takes an operator as string and two numeric operands. The function returns the result if the current inputted operator and operands are valid otherwise the function returns invalid-operation. The code is modified; instead of using a symbol operator (e.g "+","-", "*", "tan",...), the

code uses full operator name ("plus", "minus", "multiply", "tangent", ...). Also there is a constraint to be sure that there is consistency between the operand and the operators. For example if the operator is "sqrt" then the operand must have a non-negative value; if the operator is "divide" the second operand must be non-zero. The ranges used for the integer array values in the experiments were -10,000 to 10,000.

Cookie

This test object takes three input variables, name, val and site as string. The purpose of this test object is to read a specific name-attribute pair from a cookie and compare these values with input parameter values.

DateParse

This test object takes Day-name and Month-name as string parameters. The function validates name of day of the week and decodes name of month.

FileSuffix

The purpose of this test object is to check whether that the file suffix is consistent with directory. This function takes directory and filename as string parameters, then splits filename into (sub)strings at all occurrences of "." delimiter, returns a vector of (sub)Strings (fileparts). It assigns the last part of fileparts to Suffix then validates directory with the specific application name and validates Suffix with data format which is consistent with the application name.

Order4

This function takes 4 string parameters and the main purpose of this function is to check if 4 argument strings are in a specific increasing or decreasing order.

Pat

This test function takes two string parameters then checks for the presence of an argument string x, within another argument string y, the presence of the reverse of x, both x and its reverse and the two palindromes formed by concatenating the x with its reverse.

Stem

This module is an implementation of the Porter stemming algorithm (or Porter stemmer) which is a process for removing the most common morphological and

inflexional endings from words in English. Its main use is as part of a term normalisation process that is usually done when setting up Information Retrieval systems. The code is adapted from (Martin, 2005). This test function consists of six functions, the main function Subject takes one string argument.

Txt

The main goal of this test object is to convert English text into mobile telephone txt by substituting abbreviations for common words (e.g "two" to "2", "you" to "u", "bye the way" to "btw", ...). This test object takes 3 string parameters and depending on the values of these parameters, translates the words and phrases into an abbreviated txt form.

Title

The test object consists of one function which takes sex and title as string parameters. The main purpose of this function is to validate that a person's title and sex are consistent(e.g "male" and " "mr").

Name	LOC	RPE (str)	Branches
Calc	46	11 (11)	22
Cookie	23	6 (5)	10
DateParse	52	19 (19)	26
FileSuffix	40	11 (10)	22
Order4	15	14(6)	6
Pat	62	14(10)	28
Stem	44	11(8)	16
Txt	29	11(11)	14
Title	36	21(21)	12

Table 3.6: The JScript functions used for empirical investigation.

3.3.1 Experimental parameters

Each of the cost functions and associated search operators were implemented in a prototype test data generation tool. The tool has been constructed by modifying the JScript (JavaScript) language compiler within the SSCLI and can therefore be used to test functions within programs written in the JScript language. The program must include directives to specify any input domain constraints that are to be applied. The program is then parsed and semantic analysis is done. The tool then inserts instrumentation code at each branch in the function. This instrumentation code calculates the cost of each branch predicate whenever it is executed, for more information refer to (Bottaci, 2005).

The cost of each relational predicate expression was calculated according to the cost functions given in the previous section. Where branch predicate expressions consist of two or more relational predicates joined by logical connectives, *and*, *or* and *not*, the cost values were combined according to the scheme given in Bottaci (Bottaci, 2003). In the case of logical *and*, for example, the costs of the constituent operands are added whenever they are both false. For nested branches, the costs of the branches in the control dependency condition of the target branch were similarly combined to provide an overall cost value for the candidate input. Unexecuted branches were assigned a high fixed cost (see Chapter 2, page 21).

Input domains

As described in Section 3.2.1, all character values were restricted to an ordinal range from 0 to 127 inclusive. A maximum string length of 20 was used to create a large search space and thereby reduce, to a very low probability, the possibility of achieving branch coverage by random data generation.

Genetic algorithm

The search was directed to generate data for one branch at a time. The order in which the branches of the program were targeted was arbitrary except that no nested branch was targeted before the containing branch. This is not, in general, a good strategy since the search will become stuck at an infeasible branch but it is adequate for the experimental purposes of this research given that all the branches in the sample programs are feasible.

A steady-state style genetic algorithm, similar to Genitor (Whitley, 1989), was used in this work. The cost function values computed for each candidate input were used to rank candidates within the population in which no duplicate genotypes are allowed. A probabilistic selection function selected parent candidates from the population with a probability based on their rank, the highest ranking having the highest probability. More specifically, for a population of size n, the probability of selection is

$$\frac{2(n - rank + 1)}{n(n-1)}$$

A single tree-structured representation was used, both for candidate inputs (phenotype) and for crossover and mutation (genotype). At the top-level, a candidate is an array of objects in 1-1 correspondence with the parameters of the program under test. Each object may be a primitive value, i.e. a number, character or string, or an array. This representation has the advantage that all candidates have the same structure. Candidates differ only in the lengths of strings and these occur only at the leaf nodes of the structure.

Single point crossover was used. A cut-point within the tree structure was selected randomly and the resulting parts were exchanged. If the cut-point fell within a string and beyond the length of the shorter string then the single suffix from the longer string was transferred to the shorter string as illustrated below.

```
parents          offspring
-----------------------------
```

```
GENER|ATION          GENER

DATA |               DATAATION
```

A genetic algorithm has a number of parameters that may be modified to suit a given problem. The size of the population and the frequency with which selected candidates are mutated are two examples. In the context of test data generation, a search algorithm must be able to perform effectively without significant human intervention as such intervention is not cost effective hence no parameter was "tuned" to suit any particular program under test. In the work reported here, a population size of 100 was always used. At each evaluate-select-produce cycle, either mutation or crossover was applied with equal probability. This means that a third of selected candidates were mutated since two candidates are selected for a single application of crossover. The mutation of a candidate consisted of a mutation to a single randomly selected primitive value, in this case a string or a number. Of the two character substitution mutations, only the binary bit inversion operator was used when searching with the Hamming distance (HD) cost function, otherwise the Gaussian substitution operator was used.

3.3.2 Preliminary results

There are three kinds of string mutation that may be applied: deletion, insertion or substitution. Initially, the choice of which particular mutation to apply was determined randomly with equal probability. During a number of preliminary runs of the genetic algorithm, however, it was noticed that the effectiveness of the different kinds of mutation operator (i.e. the rank of the offspring produced) varied according to the stage in the search. The bar chart of Figure 3.10 shows the mean rank of the offspring produced by each kind of mutation for successive periods of the search to find data to execute a single branch in the ***Calc*** program.

The three different kinds of mutation operator are broadly equally effective in the early part of a search but in the latter stages, only the substitution operator is effective and no offspring produced by the deletion or insertion operator enter the

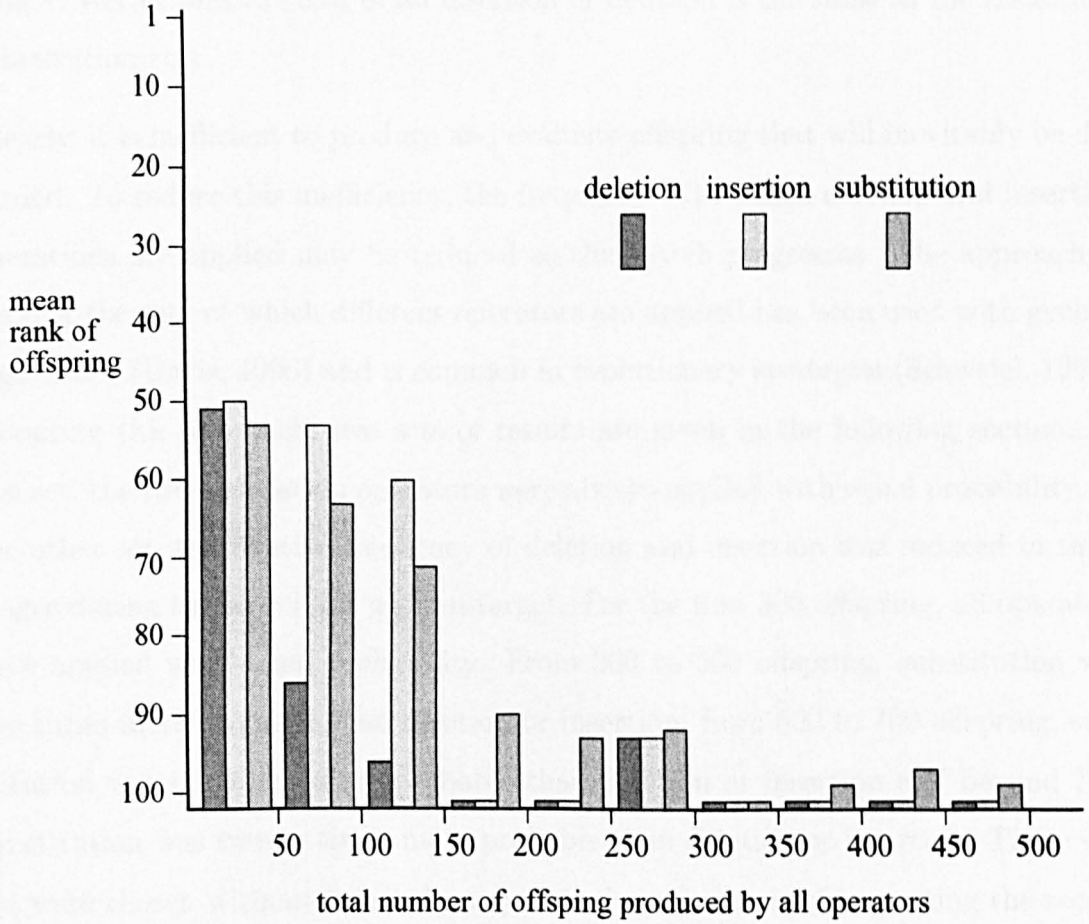


Figure 3.10: The mean rank of offspring produced by each kind of mutation operator during successive periods of search. The population size is 100 and a rank of 101 indicates offspring not sufficiently fit to enter the population.

population.

In the latter stages of a search for a given target, all the candidate strings will usually be of the same length as the solution. Some of the characters will be correct and others will be close in value. In such a situation, inserting or deleting a character from any candidate string increases its cost beyond that of the lowest ranked candidate. Recall that the cost of an insertion or deletion is the same as the maximum substitution cost.

Clearly, it is inefficient to produce and evaluate offspring that will inevitably be discarded. To reduce this inefficiency, the frequency with which deletion and insertion operations are applied may be reduced as the search progresses. The approach of varying the rate at which different operators are applied has been used with genetic algorithms (Davis, 1996) and is common in evolutionary strategies (Schwefel, 1995). Adopting this approach, two sets of results are given in the following section. In one set, the three mutation operators were always applied with equal probability. In the other set, the relative frequency of deletion and insertion was reduced in three stages during the search for a given target. For the first 300 offspring, all operators were applied with equal probability. From 300 to 500 offspring, substitution was five times more probable than deletion or insertion, from 500 to 700 offspring, substitution was ten times more probable than deletion or insertion and beyond 700, substitution was twenty times more probable than deletion or insertion. These values were chosen without detailed analysis and on the basis of inspecting the record of mean offspring rank for each of the three mutation operators for some of the branches in the sample programs. In addition, whenever, the probability of substitution was increased, the standard deviation of the Gaussian substitution operator was reduced in order to localise the search. The initial reduction was from 16 to 10, the second reduction was to 6 and finally to 3. Again, no detailed analysis was done to choose these figures, they were chosen only on the basis that they provide a progressive decrease to a small value.

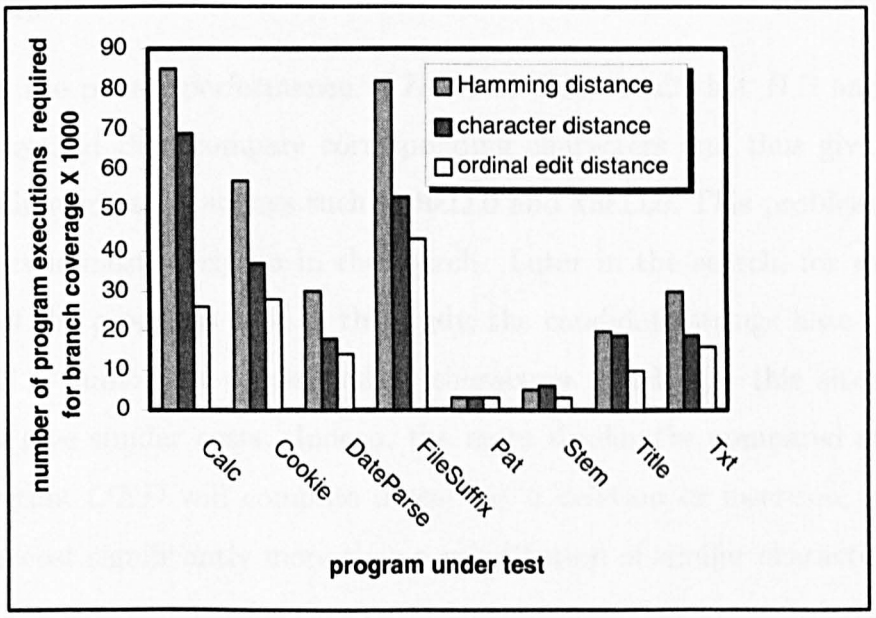


Figure 3.11: The number of executions of the program under test required to find test data to achieve branch coverage (average over 20 trials). Equal probability of character insertion, deletion and substitution.

3.3.3 Results

For each type of string predicate, results are given for the number of test program executions required to find test data to cover all branches.

Equality

The results shown in the bar chart of Figure 3.11 show the number of program executions required to find input data to achieve branch coverage, averaged over 20 trials. The probability of character insertion, deletion and substitution was equal throughout the search. These results provide some evidence that OED is the most efficient of the three cost functions and that CD is more effective than HD. Overall, OED is about a third more efficient than CD. The relative performance of OED is consistent across all the programs apart from ***Pat***. ***Pat*** does not require its arguments to be any specific string as it is attempting to find two randomly chosen strings that are equal in part. As such, ***Pat*** presents relatively weak demands on

the test data.

To explain the poorer performance of HD and CD, recall that HD and CD left-align strings and then compare corresponding characters and thus give relatively high costs in comparing strings such as `HELLO` and `XHELLO`. This problem, however, tends to occur most, early on in the search. Later in the search, for most of the branches of the programs used in the study, the candidate strings have the correct length and a number of corresponding characters match. In this situation, CD and OED give similar costs. Indeed, the more similar the compared strings, the less likely that OED will compute a cost via a deletion or insertion, since these operations cost significantly more than a substitution of similar characters.

It should be noted that one of the most difficult search problems, which was presented by the ***FileSuffix*** program, was not related to the performance of any cost function. A fragment of this program is shown below.

```
fileparts = file.Split(".");
lastpart = fileparts.Length - 1;
if (lastpart > 0) {
  ...
  if (fileparts[lastpart] == "exe") {
      //TARGET
  }
```

In this program, an input string `file` is split into substrings at each occurrence of the dot character. A branch predicate is then satisfied if at least two substrings are produced and the last substring is equal to the string `exe`. Until a string is generated that contains a dot, only one substring is produced and `lastpart > 0` produces a constant cost of 1 and so the search receives no guidance. Once a dot was inserted, the search progressed steadily under the guidance of the cost function.

The results shown in the bar chart of Figure 3.12 again show the number of program executions required to find input data to achieve branch coverage, averaged over 20

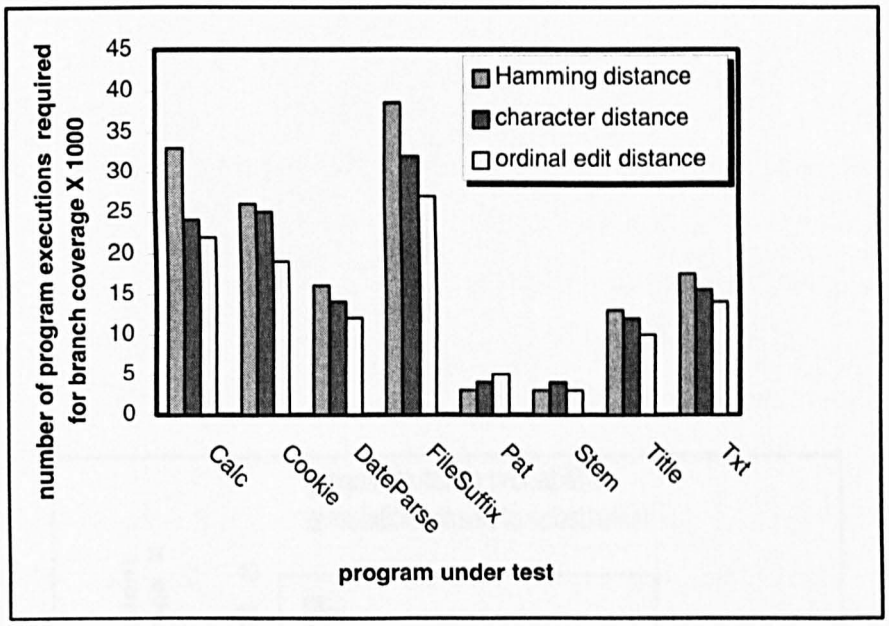


Figure 3.12: The number of executions of the program under test required to find test data to achieve branch coverage (average over 20 trials). Progressive increase in the probability of character substitution.

trials but in this case the probability of character substitution was progressively increased and the standard deviation of the distribution used by the Gaussian substitution operator to select the replacement character was reduced. Comparing these results with those of Figure 3.11, Figure 3.13 shows the average number of executions over all programs according to cost function and mutation probabilities. It is clear that there is a significant improvement in efficiency to be gained from increasing the probability of a substitution mutation. Note, however, that the superiority of the ordinal edit cost function declines when the probability of a substitution mutation is increased. This can be explained by noting that the ordinal edit distance will give relatively low costs to matches such as (`XHELLO`, `HELLO`) compared to matches such as (`GDKKN`, `HELLO`). Awarding (`XHELLO`, `HELLO`) a low cost is ineffective, however, if deletion or insertion is rarely applied.

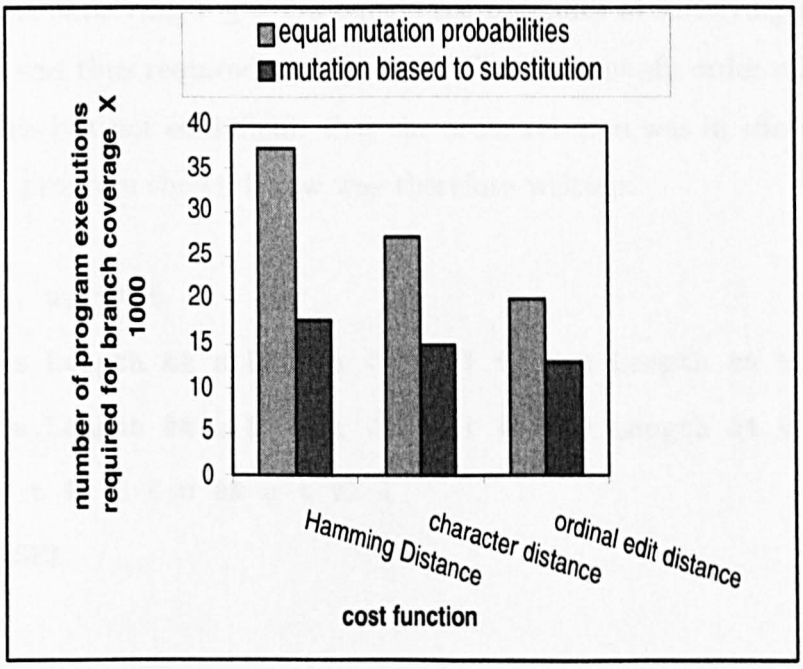


Figure 3.13: The average number of executions of the program under test required to find test data to achieve branch coverage for sample programs (average over 20 trials).

String ordering

The empirical assessment of string order relations such as $\leq$ is not straightforward. There is a reasonable probability of satisfying such an order relation given two randomly selected strings, something that is most unlikely for an equality predicate. A situation in which a predicate such as $\leq$ is difficult to satisfy is when a constraint such as $s \leq t \leq u$ applies. Here the value of t may be difficult to find if s is close to u. This problem becomes more difficult as s approaches u and in the limiting case, the difficulty of satisfying $s \leq t \leq u$ equals the difficulty of satisfying $s = t = u$. A test program was thus required to impose a difficult to satisfy order relation on the program inputs but not so difficult that the order relation was in effect an equality relation. The program shown below was therefore written.

```
Order4(s, t, u, v) {
  if ((4 < s.Length && s.Length < 7) || (4 < t.Length && t.Length < 7)
      (4 < u.Length && u.Length < 7) || (4 < v.Length && v.Length < 7)) {
    if (s < t && t < u && u < v) {
      //TARGET
    }
    else if (s > t && t > u && u > v) {
      //TARGET
    }
  }}
```

The lengths of the strings were restricted in order to make the target branches more difficult to satisfy. Table 3.7 shows the number of executions required to find test data to execute all branches averaged over 50 trials using each of the candidate cost functions for string ordering. The results are given with and without the bias to the substitution mutation operator. It also shows the average number of executions required to find test data when a 2-valued cost function was used, i.e. a single cost value for true and a single cost value for false. This indicates the difficulty of finding

program name	no mutation bias		mutation bias		random
	Ordinal value	single character pair	Ordinal value	single character pair	
Order4	1711	1622	1702	1813	23115

Table 3.7: The number of executions required to find test data to achieve branch coverage (average over 50 trials).

test data by random search. 50 trials were used to distinguish more accurately the performance of the cost functions compared to random search.

There is no evidence to suggest that SCP is more or less efficient than OV in terms of performance. There is also no advantage in increasing the probability of the substitution operator. This is understandable given that the search is not aiming to generate a fixed length string. As can be seen from the number of random candidates generated before satisfying the order predicates, the order relations are not particularly difficult to satisfy in this example and this may be true or order predicates more generally. SCP has practical advantages, though; it is easier to implement since it does not require additional work to represent large numbers that exceed the capacity of the native numerical types.

3.4 Program-dependent Search Operators

Many of the programs that process strings contain string literals. The examination of the SSCLI code showed that about 65% of string predicate expressions contain a string literal, also in (DeMillo and Offutt, 1988), DeMillo and Offutt showed that 58% of clauses are of the form $x\ R\ c$, where x is variable, c is a constant and R is relational expression. So a program may match a string literal with a string input, or a string derived from the input.

```
function f(s:string) {
   if (s.Equals("CHILD"){
        ...
   }
}
```

The first branch of this program is true when `s = "CHILD"`. This suggests a heuristic to guide the search for values for the strings `s`, namely, set `s` to a string literal that appears in the program under test.

Another example is given below.

```
P(s) {
  t = F(s);
  if (t == "AC") {
    //TARGET
```

If the string literal, i.e. `AC` were to be generated as a candidate solution and F is the identity function, the search would produce a solution immediately. In general, however, the relationship between the input string and the string comparison in a branch predicate may not be so direct. In practice, F may not be the identity function and the input may be processed by any number of statements before a string comparison is made in a branch predicate. The effect of these statements is to add a transformation to the search space.

The reliability of the cost function need not necessarily suffer as a result of such transformations. To illustrate this, assume the function F reverses its input. In this case, it is the input `CA` that executes the target branch, not the program literal `AC`. For simplicity, assume a search space of only 9 strings as shown in Figure 3.14. The bidirectional edges of the graph indicate possible string modifications by a search operator that may only "increment" or "decrement" a single character in the string. Against each string is shown the cost of that string compared to `AC`. The string

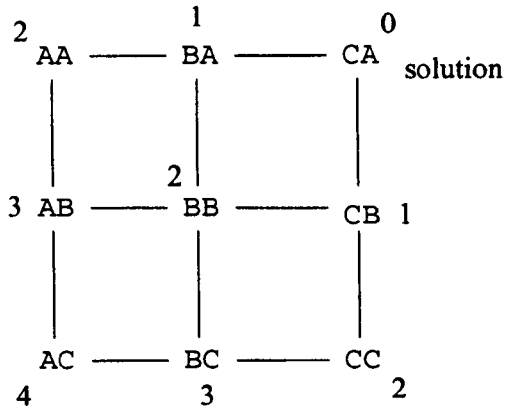


Figure 3.14: A small search space of 9 strings with increment and decrement character mutations. The cost of each string compared to CA is shown.

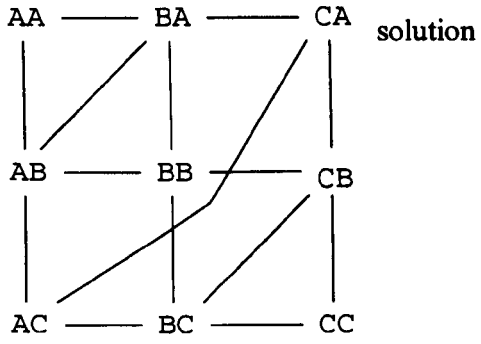


Figure 3.15: The search space after the addition of a reverse search operator.

reverse operation performed by F does not reduce the reliability of the cost function since the costs decrease steadily towards the solution CA.

In this example, using the program literal string AC as a candidate solution provides the worst possible start for the search as the minimum distances from other strings are all shorter. This observation prompted consideration of applying a search operator to counter the effect of F. Adding a string reverse search operator (reverse is its own inverse) leads to the space shown in Figure 3.15. The minimum number of applications of a search operator necessary to transform the initial input string AC to the solution CA is now just one. Additional search operators reduce path lengths but they do so at the expense of increasing the number of paths. In the particular case of moving from AC to CA, however, the mean number of operations is reduced. Although the addition of the reverse operator increases the number of edges, they

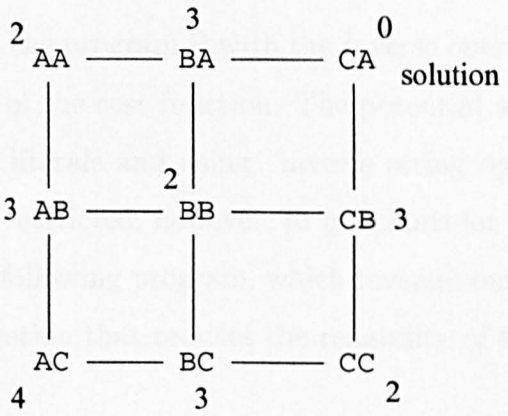


Figure 3.16: The program Q renders the cost function unreliable. (costs to the soultion shown against each node)

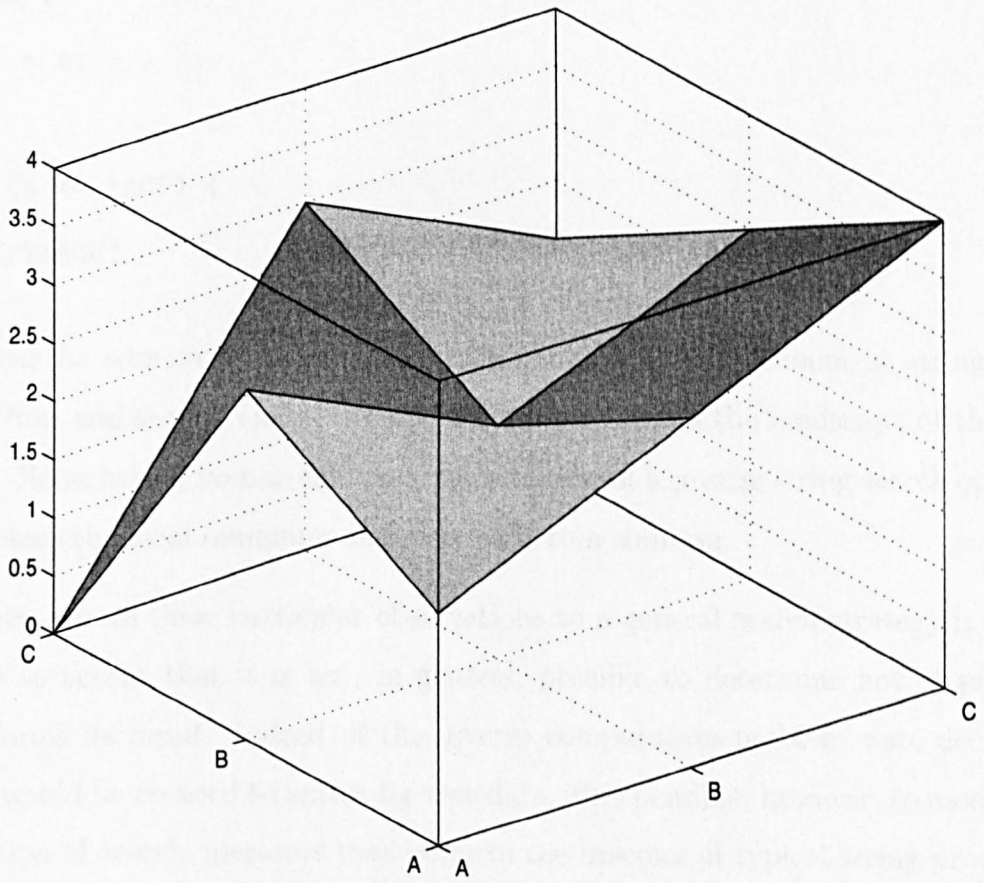


Figure 3.17: The landscape of program Q.

are all shortcuts on paths from `AC` to `CA`.

It could be argued that the program P with the reverse operator F is unusual in not reducing the reliability of the cost function. The potential advantage of seeding the population with string literals and using "inverse string operations" as additional search operators is not restricted, however, to programs for which the cost function is always reliable. The following program, which reverses only selected strings, is an example of a transformation that reduces the reliability of the cost function.

```
Q(s) {
  if (s[0] == 'A' || s[1] == 'C' || (s[0] == 'C' && s[1] == 'A')) {
    t = Reverse(s);
  }
  else {
    t = s;
  }
  if (t == "AC") {
    //TARGET
```

This can be seen from Figure 3.16 which shows a local minimum at strings with equal first and second characters and Figure 3.17 shows the landscape of this program. Nonetheless, even in this case, the addition of a reverse string search operator overcomes the local minimum and also leads to a solution.

In moving from these particular observations to a general search strategy it is necessary to accept that it is not, in general, possible to determine how a program transforms its input. Indeed, if the inverse computation problem were decidable, there would be no need to search for test data. It is possible, however, to assemble a collection of search operators that perform the inverses of typical string processing functions such as string concatenation, insertion and deletion. It is hoped that the use of such operators, together with any string literals drawn from the program under test, should, in general, improve the efficiency of the search.

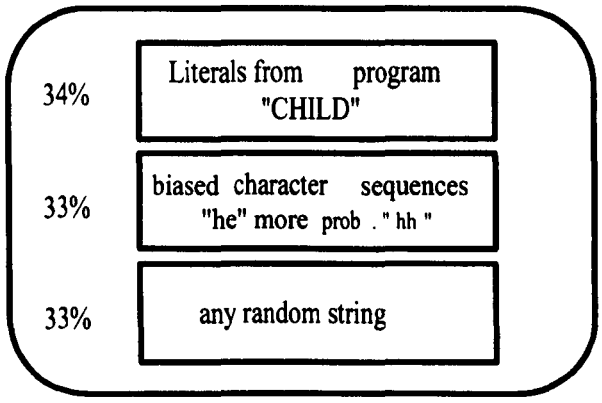


Figure 3.18: The string domain

3.4.1 String operations biased towards program string literals

To exploit the observations of the previous section, the random string generator used to generate initial candidate solutions was extended to comprise two components. One component is the former string generator which selected strings from two distributions, a uniform distribution of strings with characters with a range of ordinal values from 0 to 127 and an English-like distribution. The second component generates strings that are either program literals or formed by concatenating these literals. The reason for concatenating literals is that programs often test if a string is a substring of another. Concatenating literals, rather than inserting a single literal into an arbitrary string, increases the chances of selecting the required literal and is also useful in the case in which the test program requires more than one literal to be a substring of a string.

The current mutation operators will, over time, decrease the proportion of candidates in the population that contain a program string literal. Consequently, three additional mutation operators were defined. One operator deletes a program literal from a given string if such a literal exists. An operator inserts a program literal or the concatenation of two literals into the string it is mutating. Another operator replaces a random substring of the string it is mutating with a program literal or the concatenation of two literals. The length of the substring replaced is equal to

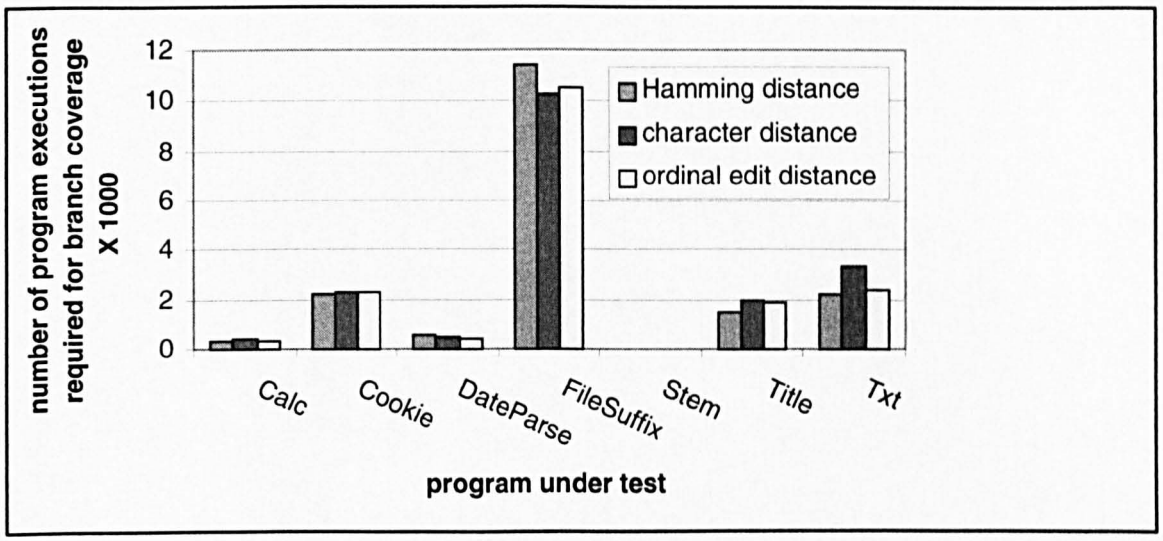


Figure 3.19: The number of executions of the program under test required to find test data to achieve branch coverage (average over 20 trials) using program-specific search operators.

that of the string to be inserted so that overall there is no change in length.

The reason for replacing an equal number of characters follows from a characteristic of the search that was discussed earlier, i.e. the convergence of the search to a population of strings with the same length, If a mutation operator modifies the length of a candidate string then the cost function is likely to penalise it to the extent that it does not enter the population.

3.4.2 Empirical assessment of program-specific search operators

The programs listed in Table 3.6, except ***Order4*** and ***Pat*** which contain no string literals, were used to assess the performance of the program-specific search operators. The test tool collects program literals during a traversal of the program abstract syntax tree. Character literals are also collected and treated as strings. The random string generator was set to generate each type of string, i.e. 7-bit character, English-like and literal, with equal probability as shown in Figure 3.18.

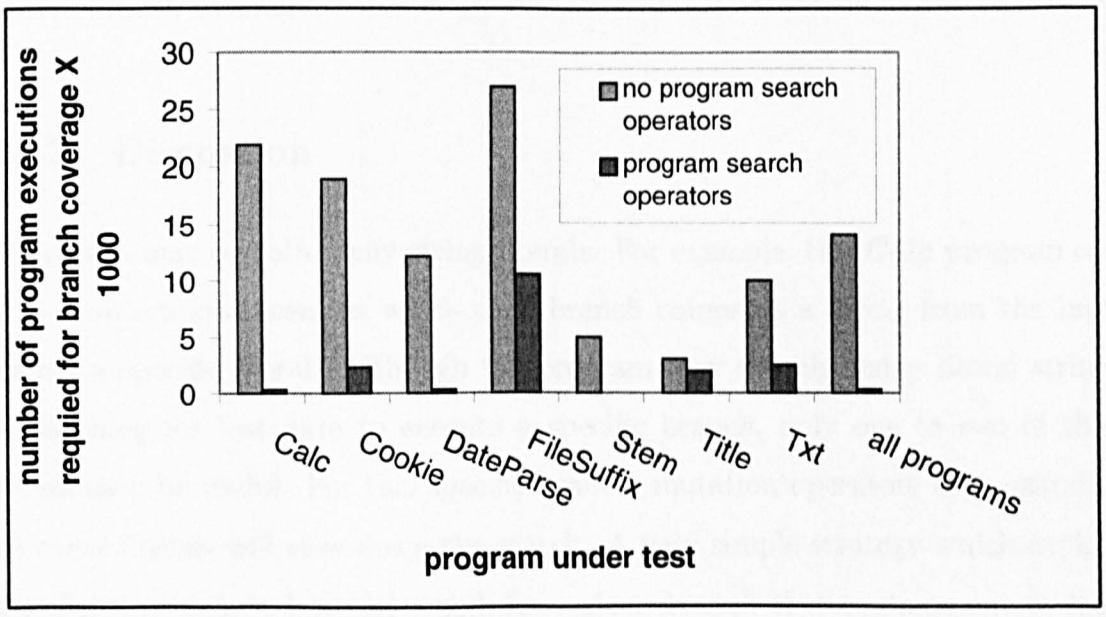


Figure 3.20: A comparison of the number of executions of the program under test required to find test data to achieve branch coverage (average over 20 trials) with and without program-specific search operators. Only the values obtained with the ordinal edit distance are compared and mutation is biased towards substitution.

Again, the aim was to find input data to execute all the branches in each of the programs. For each program and cost function, 20 trials were done. The average number of program executions required to achieve branch coverage over 20 trials is shown in the bar chart of Figure 3.19. These results show a significant improvement in performance compared to the results of the previous section, as can be seen in Figure 3.20 which compares the results of the ordinal edit function with respect to the use of program-dependent search operators.

Overall, the use of program-specific search operators leads to about a fivefold improvement in search efficiency. Note that the results in Figure 3.19 show the performance of the various cost functions to be broadly similar. This is probably explained by the fact that, with program-specific search operators, much less search is performed and hence the cost function is likely to have less influence on the overall performance. In the case of ***Stem***, for example, no guided search was required. The initial population strings, created from the program literals, were sufficient to

achieve branch coverage.

3.4.3 Discussion

A program may contain many string literals. For example, the ***Calc*** program contains a switch statement in which each branch compares a string from the input against a specific literal. Although the program may contain many literal strings, in searching for test data to execute a specific branch, only one or two of these literals may be useful. For this specific branch, mutation operators that introduce the other literals will slow down the search. A very simple strategy which exploits this observation is to bias the search for a given branch that contains one or more literals to those literals. This strategy would have improved the performance of the program-specific operators for almost all of the sample test programs. More generally, the literals that should be used to bias the search are those that appear in any statement that may influence the branch predicate expression. Such literals could be identified from a dataflow analysis of the program.

3.5 Program-specific Search Operators for Non-string Data Types

Although the usefulness of program-specific operators has been demonstrated for strings it seems clear that the technique generalises to other data types. This is illustrated for the numeric data type in the example in Figure 3.21. Although none of the three integer values 5, 10 and 20 that occur in the program are input values that execute the target branch (to execute the target branch `a = 10` and `b = 15`) they do provide reasonable starting points for a guided search. To get the variable `b = 15`, simply inverse the arithmetic operation plus (`v = v + b`) which is (20 = 5 + `b`) then `b` = 20 - 5.

Numerical types may be converted from integer and double as required by the input

```
void  f1(int a, int b) {
   int v = 5;
   if (a == 10) {
       v = v + b;
   }
   if (v == 20) {
      // Target executed with a = 10, b = 15
   }
}
```

Figure 3.21: Alternative internal variable example

domain. Numerical types may be also converted to character data type and vice versa if possible. In general, adding the literal that appears in the program is not straightforward. If the data types of the input parameters are integer but the literal collected from the program is double, the input domain is different from the literal data type: the literal data types are converted to confirm with the input domain data type.

Figure 3.22 shows a program in which the first branch is executed when `a = 25`. The required branch is executed only when the first "if statement" is executed and $sin(b) = 1$. This happens only when `b` = $\frac{\Pi}{2}$. It is easy to execute this branch if the mutation operator sequence of $sin^{-1}(6.0 - 5.0)$ is used to create a value for `b`. The arithmetic operations in this program are : Plus and *sin*, the proposed mutation operators to execute the required branch might be Minus and Arcsin.

This has motivated the introduction of additional genetic operators to increase the performance of searching the program under test by analysis and extracting arithmetic operators from the program under test, then reversing these operators to induce the mutation operators.

```
void InverseSin (double a, double b) {
    double v = 5.0;
    if (a == 25.0) {
        v = v + sin(b);
    }
    if (abs(v - 6.0) <= Double.Epsilon) {
        //Target executed
    }
}
```

Figure 3.22: To execute the target, `b` equal to $\frac{\Pi}{2}$

In general, when any arithmetic operator[1] or trigonometric function[2] occurs in the program under test, this operator or function and its inverse are used as mutation operators (e.g. Plus and Minus, *sin* and Arcsin). Note that Arcsin and Arccos are used when the parameter is in the range (-1 to 1) only.

The polygon classification program can be seen in Appendix A. The program has an array of 6 or 8 of positive real numbers. The length of the array represents the figure shape: 6 means the figure might be Triangle, 8 means the figure might be Square or Rectangle or other Polygon. The first half of the input parameters (3 or 4) represents the angles and the rest represents the side lengths of the figure sides. The goal of the program is to determine the figure, Square, Rectangle, Triangle or other shape and also if the figure is Triangle, to categorize the triangle type. The program consists of 22 branches, all the branch cost functions have a gradient which illustrates the usefulness of program-specific search operators for program where branch coverage can be found by straight forward branch cost distance instrumentation. No branch has a branch distance cost which is locally flat. The program was executed by GAs

[1]Plus, Minus, Multiply, Divide, PostIncrement, PostDecrement, Pow, Sqrt, Modulus and Absolute value

[2]*sin*, Arcsin, cos, Arccos, tan and Arctan

with and without using program-specific search operators and over an average of 20 trials the number of executions required to find test data to achieve all branch coverage without using program-specific search operators was 43872; the number of executions required to find test data with program-specific search operators was 1542. It is clear that there is a significant improvement in performance by using program-specific search operators.

3.6 Summary

This chapter considers the problem of generating test data where the test data is intended to cover programs branches which depend on string predicates such as string equality and string ordering. Current work in automatic test data generation has been limited largely to programs containing predicates that compare numbers and almost no work has been done on generating test data to satisfy string predicates.

A dynamic test data generation approach is adopted and the problem is seen as one of defining appropriate search operators and corresponding cost functions with which to guide a search.

A relatively simple but important aspect of the search for string data is the definition of the search space of strings. The space of 16-bit character strings is far larger than the space that need be searched in practice, being the space restricted to strings containing characters in the seven low-order bits.

For string equality, an adaptation of the binary Hamming distance was considered, together with two new string specific match cost functions. New cost functions for string ordering were also defined.

For string equality, a version of the edit distance cost function with fine-grained costs based on the difference in character ordinal values was found to be the most effective in a small empirical study. In addition, a progressive increase in the probability with which the character substitution mutation operator is applied has also been shown to improve the performance of the search. Two functions for string ordering

were investigated but there was no significant difference in their performance in the limited empirical investigation.

The most significant improvement in performance, however, was obtained by exploiting the presence of string literals in programs that process string data. This chapter presents program-dependent string search operators that focus the search in the region of such string literals. In the empirical investigation, the use of these operators was shown to give a fivefold increase in performance. The use of program-dependent string search operators has been shown to be far more important than the particular choice of cost function that guides the search.

3.7 Estimating Number of Search Operator Invocations

Recall that the fitness function should estimate the number of search operators from a given candidate to the solution. After the completion of the experiments in the previous section, the question of estimating the cost of various search operators was considered empirically. Given a specific search operator, it is possible to investigate empirically, for a sample of candidates and goals, the actual number of invocations of the operator required to generate a given goal from a given candidate can be counted. To investigate this idea in concrete terms, the search operator OED was considered.

As explained in Section 3.3.1, the cost of OED is computed as given:

$OED(s:a, t:b) =$

$min(OED(s:a, t) + 128, OED(s, t:b) + 128, OED(s, t) + 128/4 + 3|a - b|/4).$

The previous experimental results show that if different length strings are compared then any deletions or insertions are as effective as a single mutation. However, when a candidate string has the same length as the goal string then a deletion must be followed by an insertion. For example, OED(RAG, RARE) = 161.5, OED(ROAR, RARE) = 129. This means "ROAR" is more close to

"RARE" than "RAG", even though moving from "ROAR" to "RARE" needs 3 substitutions and moving from "RAG" to "RARE" needs 1 insertion and 1 substitution. The question is which path, 3 substitutions or 1 insertion and 1 substitution is more likely to lead to the solution. The higher probability path should have the lower cost in order to guide the search to path that is likely to lead to the solution. To answer this question the probability of insertion, deletion and substitution must be compared. Figure 3.23 shows the possible mutation operators to move from candidate string to the goal string and the probability of each operator is computed as follows:

p(deletion of character i) $=\frac{1}{3} * \frac{1}{n}$, where n is the string length.
p(insertion of character c at i) $= \frac{1}{3} * \frac{1}{n+1} * \frac{1}{128}$.

It is very difficult to find theoretically the probability of substitution by using a Gaussian distribution mutation operator because the new value may be right or left of the mean (given character) and the search ignores the new value if it is not better than the existing one. To find the probability of this mutation, a practical experiment of 100 trials was done to find the actual cost to move from one character to another with 7-bits domain(0 - 127). The actual result is shown in Figure 3.24 and a heuristic approximation is computed and is shown in the same Figure. The heuristic approximation cost to move from one character to another character is given by the following equation:

$\frac{character\ difference}{3} + 11$. So the final cost of EOED (Empirical ordinal edit distance) will be computed as follows:

$$EOED(s:a, t:b) =$$
$$min(EOED(s:a, t) + 128, EOED(s, t:b) + 128, EOED(s, t) + 11 + |a - b|/3).$$

The probability of deletion followed by insertion is :

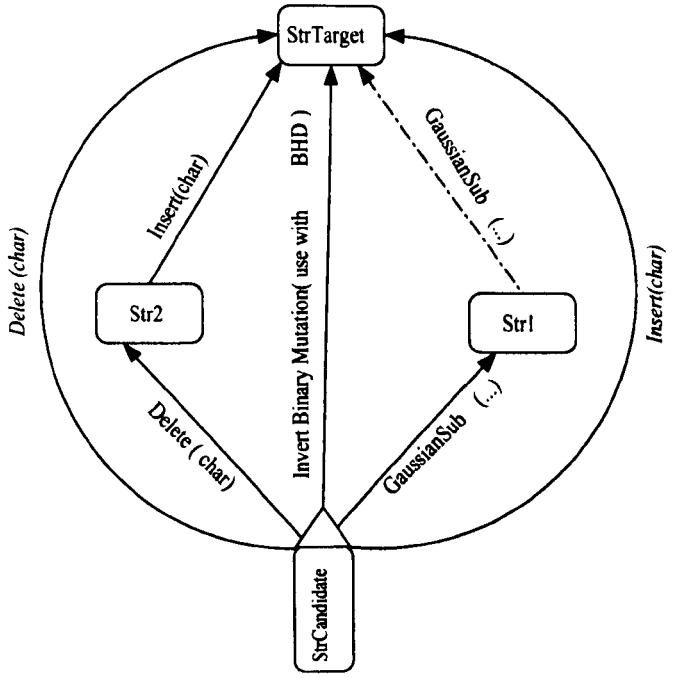


Figure 3.23: The possible mutation operator to move from candidate to goal string.

$(\frac{1}{3} * \frac{1}{n}) * (\frac{1}{3} * \frac{1}{n+1} * \frac{1}{128})$ which is less than probability of substitution.

The results in Figure 3.25 show the number of program executions required to find input data to achieve branch coverage, average over 20 trials while the probability of character insertion, deletion and substitution was equal throughout the search. This figure also shows comparison between number of program executions required to find input data using OED and EOED. These results show that EOED is more efficient than OED.

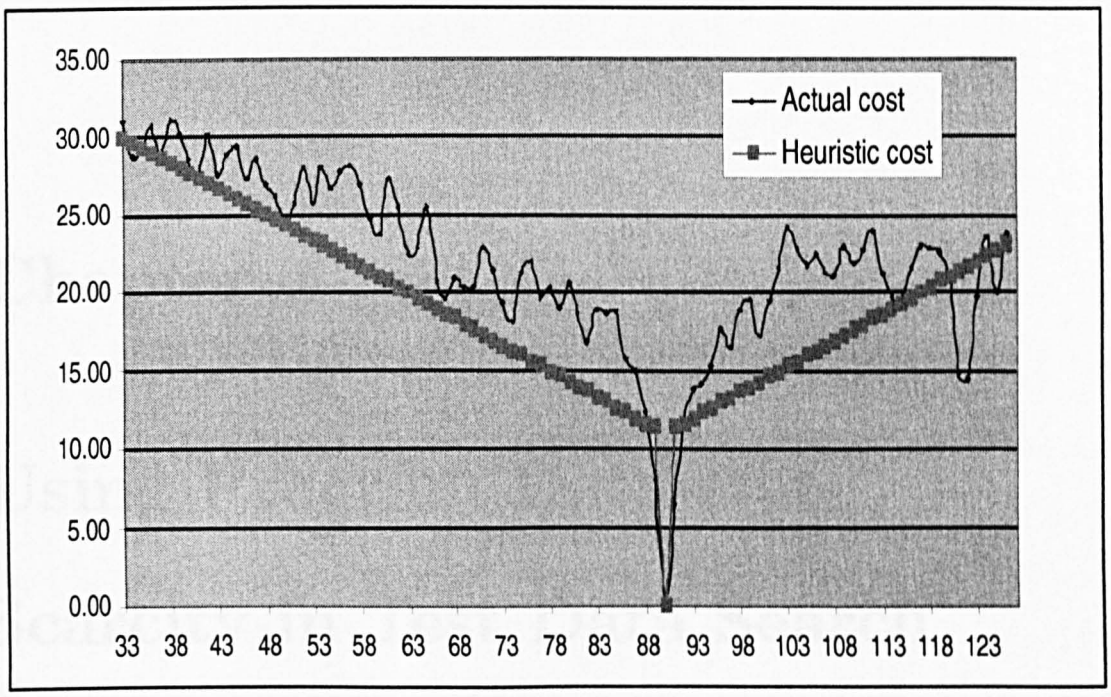


Figure 3.24: The actual and heuristic cost of gaussian mutation operator to move from one character to z with the 7-bits domain.

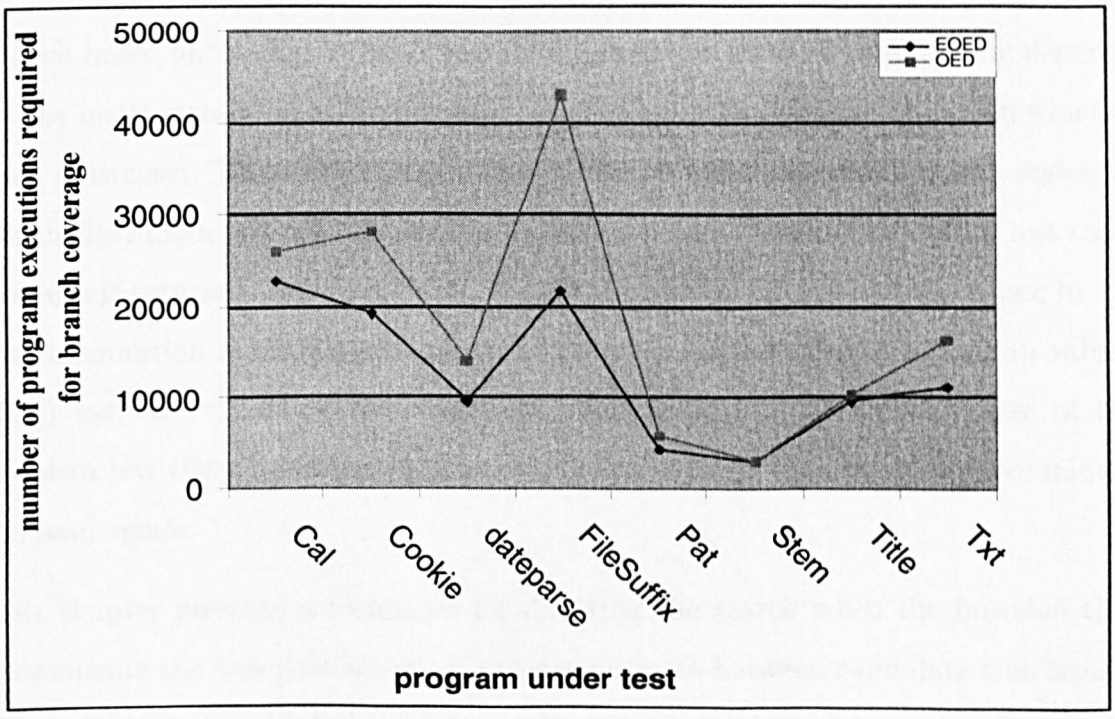


Figure 3.25: The number of executions of the program under test required to find test data to achieve branch coverage by using OED and EOED.

Chapter 4

Using Program Data-state Scarcity in Test Data Search

4.1 Introduction

Search-based automatic software test data generation for structural testing depends on the instrumentation of the test goal to construct a many-valued function which is then optimised. The method encounters difficulty when the search is in a region in which the function is not able to discriminate between different candidate test cases because it returns a constant value. A typical example of this problem arises in the instrumentation of branch predicates that depend on the value of a boolean-valued (flag) variable. Existing transformation techniques can solve many cases of the problem but there are situations for which the existing transformation techniques are inadequate.

This chapter presents a technique for directing the search when the function that instruments the test goal is not able to discriminate between candidate test inputs. The new technique depends on introducing program data-state scarcity as an additional search goal. The search is guided by a new evaluation (cost) function made up of two parts, one depending on the conventional instrumentation of the test goal,

the other depending on the diversity of the data-states produced during execution of the program under test. The method is demonstrated for a number of example programs for which existing methods are otherwise inadequate.

4.2 Problem of Flag Cost Function

A flag variable is any variable that takes on one of a few discrete values . A boolean is a special case of a flag variable. Where the program has only relatively few input values which make the internal flag variable adopt a desired value, it will be hard to find these inputs using random search, see the program in Figure 4.1. A predicate which tests a flag, produces a fitness function that yields either maximal fitness for the special values or minimal fitness for any other value. The landscape induced by the fitness function provides no guidance from lower fitness to higher fitness and hence it is difficult to find inputs to execute such branches. In geometric terms, the surface of values produced by the cost function for different inputs is flat. In such situations, the heuristic search performs no better than a random search.

```
boolean flag = false;
if (x == 3) {
   flag = true;
}
... //ASSIGNMENTS TO flag
if (flag) {
   //TARGET BRANCH
}
```

Figure 4.1: Program fragment for which branch coverage data must be generated.

4.3 Existing Techniques for Instrumenting Boolean Flag Variables

A number of techniques have been proposed to tackle programs that contain boolean flag variables. Bottaci (Bottaci, 2002) proposes a solution for programs in which the flag variable is assigned a predicate expression (as opposed to a constant true or false) as shown in the example program of Figure 4.2. This program iterates through an array of 64 boolean values and determines if the values are all true.

```
AllTrue(boolean[] a) {
  boolean alltrue = true;
  for (i = 0; i < 64; i++) {
     alltrue = alltrue && a[i];
  }
  if (alltrue) {
     //TARGET BRANCH
  }
}
```

Figure 4.2: Example program with a flag variable problem.

In (Bottaci, 2002) it is suggested that the predicate expression that is used to set the flag value is instrumented and the flag variable is replaced by a many-valued variable that can hold the instrumented value of the predicate expression. Any predicate expressions that use the boolean flag variable are rewritten to test the instrumentation value. These predicate expressions can then be instrumented in the usual way. The transformation is illustrated in Figure 4.3.

The logical constants are instrumented as -1.0 and 1.0. By defining a suitable cost function to instrument the logical-and (Bottaci, 2003), where `costAnd` is an operator defined by Table 2.4, page 21, a cost value can be accumulated as the loop iterates.

```
AllTrue(boolean[] a) {
  double alltrue = -1.0;
  for (i = 0; i < 64; i++) {
     alltrue =   costAnd(alltrue, a[i]);
  }
  if (alltrue < 0) {
     //TARGET BRANCH
  }
}
```

Figure 4.3: Transformation of example program with a flag variable problem from (Bottaci, 2002), alltrue = -0.015384615 when all elements in the array are true.

The technique of replacing boolean values with cost values is inapplicable, however, when the flag variable is assigned a constant true or false value, as occurs to the flag variable in the program of Figure 4.1. In this case, Harman *et al.* (Harman et al., 2002) (Harman et al., 2004) suggest the use of a program transformation to remove internal flag variables from branch predicates, replacing them with the expression that led to their determination. In the transformed version of the program, the branch predicate is flag-free and can therefore be instrumented in a straightforward way. Their approach, however, does not handle assignment to flags within loops. In particular, 5 levels of program difficulty are identified and the given transformations are effective only for the first 4 levels. The fifth level consists of programs in which assignments are made to flag variables inside a loop that does not also contain the target branch. The example program of Figure 4.2 is an example of a level 5 problem.

A testability transformation for loop assigned flags is, however, given by Baresel *et al.* (Baresel et al., 2004) who extend the transformation approach for internal flags assigned within loop structures. Two approaches are presented - a "coarse-grained" transformation and a "fine-grained" transformation. Both forms of transformation

```
AllTrue(boolean[] a) {
  ...
  alltrue = true;
  int counter = 0;
  double fitness = 0.0
  for (i = 0; i < 64; i++) {
    if (alltrue && a[i]) {
      alltrue = true;
      fitness ++;
    }
    else {
      alltrue = false;
    }
    counter++;
  }
  if (fitness == counter) {
     //TARGET BRANCH
  }
```

Figure 4.4: A testable transformation of the program shown in Figure 4.2

replace the original condition using the flag variable with a predicate of the form $counter = fitness$, where *counter* is a variable incremented on each iteration of the loop, and $fitness$ is a variable which is incremented if a loop iteration was evaluated in a "desired" manner. Figure 4.4 shows the result of applying the transformation to the program of Figure 4.2.

A loop iteration is executed as desired when the flag is assigned the desired value. For example, an iteration which assigns a false value to a flag required to be true would not result in an increment of the fitness variable; whereas the avoidance of

the assignment would. In this way, the search receives a higher level of guidance to the input values which evaluate the original condition using the flag in the desired manner. This is because the objective function landscape now corresponds to the predicate $counter = fitness$ rather than the landscape containing the flag, which contains plateaux. The difference between the coarse-grained transformation and the fine-grained transformation lies in the increment of the fitness variable within the loop. The coarse-grained transformation simply increments the counter in a uniform fashion. The fine-grained approach uses distances of key branch predicates used within the loop to assign flag values.

The chaining method of Korel (Korel and Ferguson, 1996) is an effective technique for some programs that contain flag variables. The chaining method attempts to find inputs to execute a path from each last definition[1] of each variable used in the unsatisfied branch predicate. In broad terms, the heuristic is that the execution of a new path may produce a different value at the goal branch expression. In the example `AllTrue` program, there is only one path through the program up to the target branch and hence the chaining method is ineffective.

For the transformation of Baresel *et al.* (Baresel et al., 2004) to be applicable, it must be possible to identify the desired and undesired assignments to a flag variable. Sometimes this is not possible. As an example, consider the `Orthogonal` program shown in Figure 4.5. This program determines whether two binary vectors are orthogonal by computing the inner product. Each of the two input arrays consists of integers with the value 0 or 1. The target branch is difficult to execute because for almost all random inputs, the value of the integer `product` is set to 1. Even though `product` is not a boolean variable, a "flag" variable problem arises because `product` may take one of only two integer values. The transformation in (Baresel et al., 2004) requires that assignments to flag variables are replaced by conditional statements in which the flag is set true and also set false and the predicate of the conditional is

[1] A last definition statement is simply a program node n that assigns a value to a variable which may be potentially used at the problem node p. For it to be a last definition therefore, a definition-clear path must exist between n and p

```
Orthogonal(int []a, int []b) {
  // a[i] and b[i] in [0, 1]
  int product = 0;
  for (i = 0; i < 64 && product == 0; i++) {
    product = a[i] * b[i];
  }
  if (product == 0) {
    //TARGET
  }
```

Figure 4.5: A difficult to execute branch in a program with an integer "flag" variable

taken from the expression assigned to the flag. This is clearly inapplicable for the Orthogonal program of Figure 4.5 since **product** is not a boolean variable.

The problem with the transformation of Baresel *et al.* (Baresel et al., 2004) is that it attempts to move the computation done by a program from the sequence of data-states it generates to the sequence of control-states produced and this movement is not practical when the set of possible data-states is large.

The method of data-flow graph search, of Korel (Korel et al., 2005) searches for paths that are selected from examination of the data dependence graph of the variables that appear in the target branch predicate expression. In the example program of Figure 4.5, all such paths begin with the initial assignment to **product** and then take one or more iterations of the loop. By searching the space of paths, which in this case is small, the solution is found. However these paths, except for the solution path, all produce the same branch distance value and so the search is random. This would be a problem if the arrays were very large.

4.4 Data-state Scarcity as a Search Strategy

It is clear that it is not only programs that use boolean variables that, when instrumented, produce cost functions that are locally flat. The program fragment of Figure 4.6 computes the log_{10} of **x** and then converts this value to an integer which is used to access an array. In this program, the target branch cost function is constant for all input values of **x** except for the single value **x** = 1.

```
void Log10(int x){
    //x in [1, 100,000]
    a[0] = 0;
    a[1] = a[2] = a[3] = a[4] = a[5] = 1;
    double y = log10(x);
    int k = ceiling(y);  //y in [0, 5]
    if (a[k] == 0) {
      //TARGET BRANCH
    }
}
```

Figure 4.6: A difficult to execute branch in a program ($\mathbf{Log_{10}}$) for which no existing technique is effective.

Note that no existing technique is applicable to this program. The technique of substituting cost values for boolean values (Bottaci, 2002) is not applicable since there are no boolean expressions that can be effectively instrumented. There are no variables that can be used in the transformation of Harman (Harman et al., 2004). There is a single path through the program and so the path search methods of Korel (Korel and Ferguson, 1996; Korel et al., 2005) are not applicable.

A necessary condition for the cost function to be able to guide the search is that it should produce more than one value as it is applied to different inputs. This suggests a possible search strategy, namely search for inputs that produce a range

of values for the variable `a[k]` in the predicate expression.

To produce a greater range of values for `a[k]`, the search may be guided towards inputs that produce values for `a[k]` that are different from those that have so far been produced in the search. The search may be guided in this way providing inputs that produce so far unencountered values for `a[k]` are given a lower cost whenever they are encountered. In practice, a collection of random inputs will not produce diverse values at `a[k]`. Voas (Voas and Miller, 1995) introduces the notions of information loss and the domain/range ratio. The information loss of a mapping is the ratio of the size of the domain to the size of the range. The ratio can also be applied to any subset of a mapping. The information loss from the input to the value of `a[k]` is extreme for a very large part of the input domain. The information loss from `x` to `k` is not so extreme in that only 90% of the inputs map to a single value `k` = 5, 9% of the inputs map to a single value `k` = 4 and so on. Figure 4.7 shows the distribution of these values. In a population of 100 individuals, selected randomly, there is a reasonable probability of encountering inputs that produce values of `k` that are 4 or 3. If the search is directed to these inputs, then inputs that produce a value of 2 will be found. Such inputs are rarer than those that produce 4 or 3 and so the search is directed to consider inputs similar to those that produced a value of 2, with the result that inputs producing 1 and finally 0, will eventually be found.

Pursuing data-state scarcity is also an effective strategy for finding an input to execute the required branch in the `Orthogonal` program of Figure 4.5. This program implements a mapping from integer array pairs (`a, b`) to the integer `product`, a constant mapping for the vast majority of the input domain. A problem arises if the values of the variable `product` are examined in the predicate expression of the target branch; here rare values are in practice never encountered. Consider, however, the values assigned to `product` within the loop. A number of zeros and possibly a single one may be assigned here. Without considering probabilities in detail, it is clear that the number of inputs assigning a given number of zeros to `product` decreases as the number of zeros increases (see Figure 4.8). These inputs may be identified as producing rare data-states and hence the search may be directed to the region in the

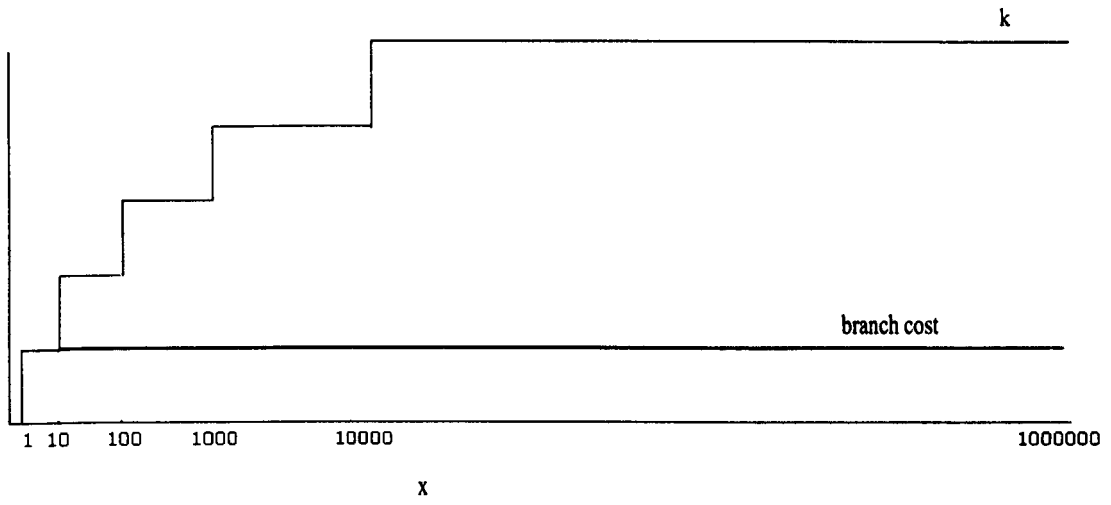


Figure 4.7: The distribution of the $\log_{10}$ values. Figure shows that function that computes `k` is less locally constant than branch cost function; hence it is easier to search for different values of `k`.

vicinity of these rare inputs. Directing the search towards inputs that produce rare patterns of assignments is an effective strategy for changing the final value assigned since this value changes when the maximum number of zeros is assigned.

Consider another example for which existing techniques are inadequate, the program `Mask` shown in Figure 4.9. This program checks that each character in an array conforms to the bit mask `1010101`. There are 16 values that may be assigned to `x`. In general, we may expect the bits within `x` to tend towards zero as the array is iterated (assuming the data in the array is relatively random) since once a bit in `x` becomes zero, it will remain so. This means that although `x` within the loop may take a number of values, `x` in the branch predicate expression is almost constant with the value 0.

Although `x` in the predicate expression is almost constant, a greater range of values is assigned within the loop where it is thus possible to identify relatively rare values. In a reasonably large population of inputs, a small number of inputs will assign the relatively rare value of `1010101` a higher number of times than is typical among the inputs that have so far been executed. By directing the search towards these rare inputs, the search is directed to inputs that produce more than a single value

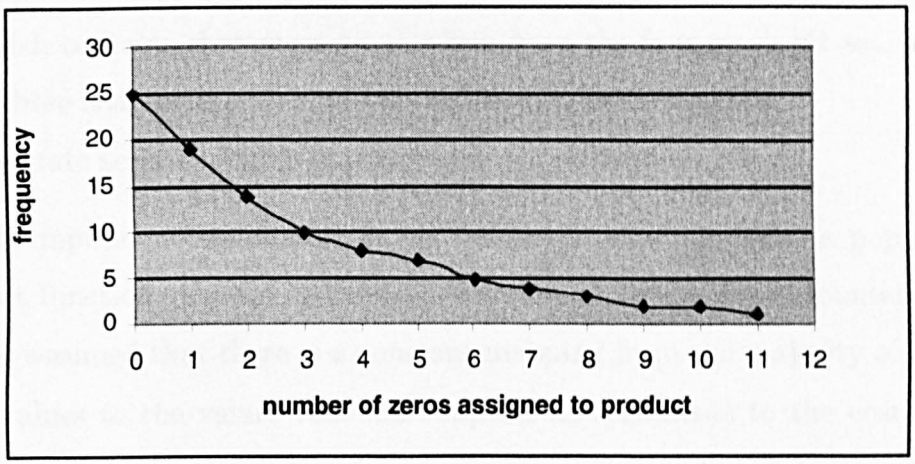


Figure 4.8: Number of test cases decreases as the number of zeros assigned to `product` increases.

```
Mask(char[] a) {
   char x = 0x55;  // 1010101
   for (i = 0; i < 10; i++) {
      ...
      x = x & a[i];  //bitwise and
   }
   if (x == 0x55) {
      //TARGET BRANCH
   }
}
```

Figure 4.9: Program `Mask` checks that each character in an array has each odd bit set.

for `x` in the branch predicate expression. Once this occurs, the cost function at the branch predicate expression may guide the search to a solution. The solution is any array which contains characters all of which have the four mask bit set, and there are 8 of these characters.

The data-state scarcity search strategy may be outlined as follows:

When the inputs encountered so far in the search have produced a population of equal cost function values, i.e. the cost function surface so far encountered is flat, then it is assumed that there is a constant mapping from the majority of the input domain values to the values that are supplied as arguments to the cost function. Such a mapping cannot fail to produce an almost flat cost surface, irrespective of the cost function used. If the mapping is implemented in a progressive manner with respect to information loss, i.e. the information loss from input values to intermediate values, such as `k` in Log_{10} is not as high as that from input values to cost function inputs, then it is possible to instrument the intermediate values. If the information loss for different regions of the input domain is not uniform then a few intermediate values will predominate and other values will be rare. As shown in the Log_{10} example, the probability of value `k = 5` is 90% but the probability of value `k = 0` is $\frac{1}{100000}$. To guide the search to generate new intermediate values (values of `k` = 4, 3, ... in the example of Figure 4.7), the search must be directed to inputs that increase the diversity of intermediate data-state values. By guiding the search towards inputs that produce diverse, as yet unencountered and therefore rare intermediate values, the likelihood increases of finding an input that produces as yet unencountered values as arguments for the cost function.

Directing the search towards inputs that produce rare intermediate values is not necessarily directing the search towards inputs that solve the test goal. The cost functions are certainly different. The purpose of directing the search towards inputs that produce rare intermediate values is to provide inputs that produce a variety of arguments to the cost function that instruments the test goal. Only when this cost function receives a range of values can it produce a non-flat cost surface.

4.5 Data-state Scarcity Search by Maintaining Data-state Diversity

The concept of diversity in the population has been studied in the literature. For example, when the GA fails to find the global optimum, the problem is often attributed to premature convergence, which means that the sampling process converged on a local rather than the global optimum. Several methods for maintaining population diversity have been proposed to combat premature convergence in conventional GAs. These methods rely on various of similarity between individuals in the population. In this thesis, the purpose of investigation existing diversity measures is to consider how they can be used to augment the existing fitness function, which consists of the branch distance, to guide the search towards rare data-state values.

4.5.1 Existing diversity measures and methods

Biological diversity denotes the differences among individuals in a population, which in nature connotes structural and behavioural difference. The term "variety" was used by Koza (Koza, 1992) to represent the number of different genotypes in a population. In simple form, genotype diversity measures the number of unique individuals (Langdon, 1999). Genotype diversity does not consider fitness. Two individuals are equal if they contain exactly the same structure and content.

An edit distance based on string matching was used by O'Reilly (O'Reilly, 1997). He uses single node insertions, deletions and substitutions to transform two genotype trees to be equal in structure and content. De Jong et al. (de Jong et al., 2001) used Levenshtein distance (see Chapter 3), which matches two trees at the root node. If the two different nodes match, they score a distance of 0, otherwise they score a distance of 1. The Levenshtein distance can be normalised by dividing the sum of all different nodes by the size of the smaller tree. The measure represents the number of node changes that need to be made to either tree to make them equal in structure and content.

Keijzer (Keijzer, 1996) used ratio of unique genotype subtrees over total subtrees to measure the subtree variety and the ratio of the number of unique individuals over the size of the population as program variety. Keijzer also used a distance measure between two individuals as the number of distinct genotype subtrees the individuals share.

In addition to genotype diversity, fitness diversity has also been investigated. Fitness diversity, also known as phenotype diversity measures the number of unique fitness values in a population. Fitness entropy is calculated by grouping the fitness values into equivalence classes (Rosca, 1995). Given k classes in the current population, let p_i be the proportion of the population which belongs to class i. Fitness entropy is then defined as,

$$-\sum_{k} p_k.logp_k$$

Figure 4.10 shows how entropy increases as number of classes increases and decreases as distribution becomes less uniform. The high fitness entropy in genetic algorithms describes the presence of many unique fitness values in the population, where the population is evenly distributed over those values. Low fitness entropy describes a population which contains fewer unique fitness values as many individuals have the same fitness.

4.5.2 Diversity control methods

Given the various similarity measures discussed in the previous section, they may be incorporated into the genetic algorithm in different ways. Some common methods are listed below:

1. restricting the selection procedure (crowding models) (Booker, 1982). Crowding induces niches by forcing new individuals to replace those that are similar genotypically. This is completed by using a "steady-state" GA which cre-

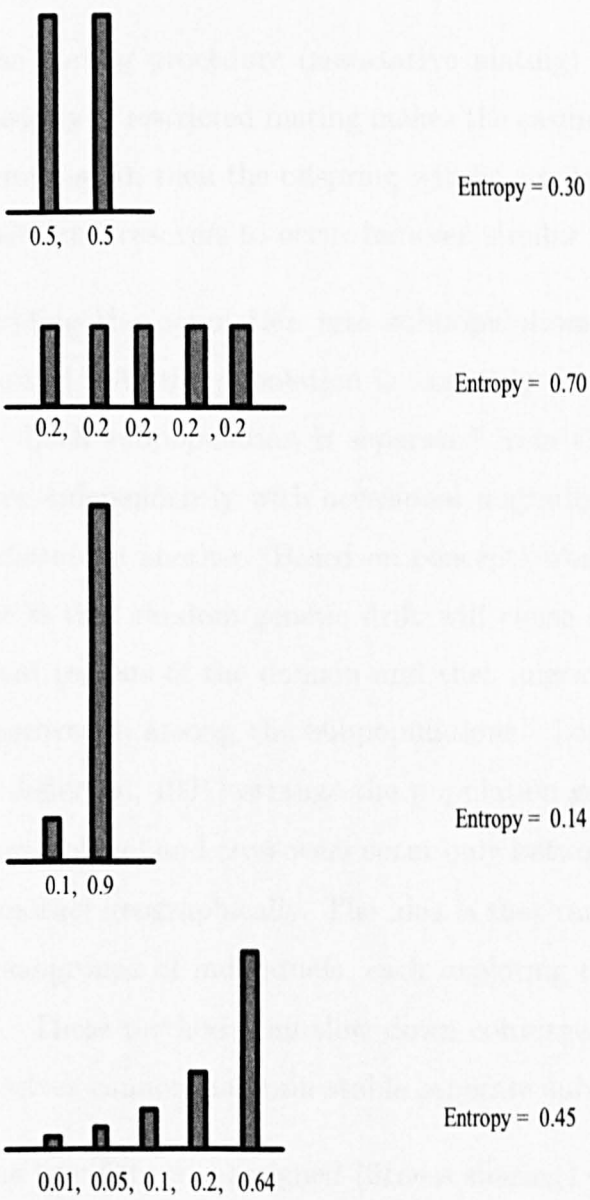


Figure 4.10: Four different distributions showing how entropy varies according to number and distribution of classes.

ates new individuals one at a time, inserting them into the population by replacement of existing individuals. In the crowding algorithm, an individual is selected for replacement by selecting a subset of the population randomly and then selecting the member of that subset that is similar to the individual.

2. restricting the mating procedure (assortative mating) (DeJong, 1975). The general philosophy of restricted mating makes the assumption that if two similar parents are mated, then the offspring will be similar. Assortative mating algorithms restrict crossovers to occur between similar individuals.

3. explicitly dividing the population into subpopulations (common in parallel GAs). In parallel GAs the population is explicitly divided into smaller subpopulations. Each subpopulation is separated from the others in the sense that it evolves independently with occasional migrations of individuals from one subpopulation to another. Based on concepts from population genetics, the idea here is that random genetic drift will cause each subpopulation to search different regions of the domain and that migration will communicate important discoveries among the subpopulations. Local mating algorithms (Collins and Jefferson, 1991) arrange the population geometrically (e.g., in a two-dimensional plane) and crossovers occur only between individuals that are "near" one another geographically. The idea is that random genetic variation will lead to subgroups of individuals, each exploring different regions of the search space. These methods can slow down convergence time dramatically, but by themselves cannot maintain stable separate subpopulations.

4. modifying the way fitness is assigned (fitness sharing) (Goldberg, 1989). Fitness sharing (Deb and Goldberg, 1989), (Goldberg and Richardson, 1987) induces subpopulations by penalizing individuals for the presence of other similar individuals in the population, thereby encouraging individuals to find productive uncrowded niches. Fitness sharing leaves the standard GA unchanged and simply modifies the way in which fitness values are computed[2].

[2]The sharing function, introduced by Goldberg (Goldberg, 1989), is a function used to explicitly define the degree of sharing and maps genotype similarity into the degree of sharing:

It is clear that the existing methods for maintaining diversity discussed in the previous section are inadequate in themselves for the problem of a locally constant branch distance function. Phenotype diversity will make little impact when the space of individuals is large relative to the population. Fitness diversity is ineffective because it assumes that the search is able to find inputs that have a variety of fitness values. In the problems described in this thesis, the cost function is locally constant and such inputs are very difficult to find.

The problem that must be solved is that the fitness function, if it is only the branch distance, is locally constant. The solution is to extend the fitness function to include a measure of scarcity of data-state values.

4.5.3 Data-state distribution diversity metric

Recording program data-state values

At a single assignment statement in the program under test, the values assigned may be stored in a histogram. The domain of the histogram is the domain of the variable to which values are assigned, although of course only non zero frequencies need be stored. For each candidate test case, the data-state distribution is recorded in a histogram of data-state values assigned to each relevant variable in the program under test. Initially, these are the variables that are in the branch predicate. This means that for a single program execution the data-state distribution is recorded as a set of histograms. A histogram of a given variable assignment containing rare data-state values is called a rare histogram and so it is possible to speak of histogram scarcity compared to the histograms of the same variable but in other individuals in the population. Figure 4.11 shows the relation between population and histograms.

shared fitness $= \frac{fitness\ of\ individual}{Total\ degree\ of\ sharing}$

$f_s(x_i) = \frac{f(x_i)}{\sum_{j=1}^{n} s(d(x_i, x_j))}$, where, s is the predefined sharing function and d is the phonotype or genotype similarity(distance) function.

A simple sharing function s would be the identity function.

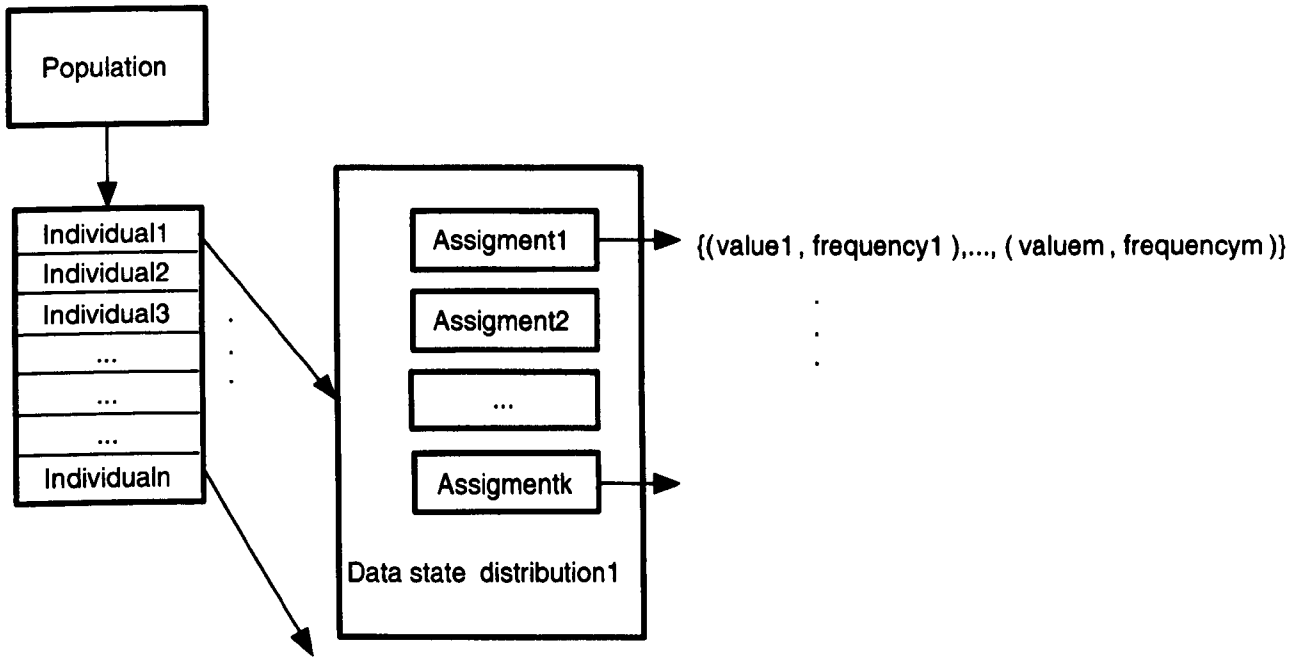


Figure 4.11: The structure of data-state distribution inside population

Data-state values are recorded for discrete value types only (e.g. integer, character, string, etc.). The values assigned to floating point variables are not recorded because it is most unlikely that two or more randomly selected floating point inputs will be equal. In addition, real-valued arithmetic operations and functions applied to these inputs are unlikely to produce equal results. When the floating point values assigned to a variable are unique across all assignments to that variable across the population, it can be argued that diversity is present. In addition, the cost functions for the relational operators cannot produce constant values unless they receive constant inputs. In the Log_{10} example program (Figure 4.6 as seen in page 84), random inputs lead to unique (across the population) values assigned to y. The distribution of values assigned to y is heavily skewed but this alone is not responsible for the locally constant cost values. In fact the histograms of values assigned to y are unique across the population.

Methods for measuring data-state distribution diversity in the population

The aim is to rank individuals of equal branch cost on basis of similarity of data-state distribution to other individual in the population. This requires a measure of similarity or distance between data-state distributions.

A simple method to identify test cases which produce rare data-state values, is to group test cases into classes on the basis of equal histogram sets. This is a binary-valued distance measure, equal or not equal. Two histogram sets are equal if the histograms they contain are equal. This is a strong criterion for grouping histogram sets but is justified on the grounds that only one of the variables used by the cost function need take a different value in order to modify the computed cost.

The fitness function should measure both branch distance and data-state scarcity. For constant branch distance, the fitness of an individual is a function of the data-state diversity it contributes to the population. In the case where the branch distance values in the population are all equal, the size of the data-state distribution equivalence class to which the individual belongs is used for population member ranking.

More formally, let the equivalence classes of individuals under equal data-state distribution be grouped into sets of equal sized classes and then let these sets be ordered according to increasing equivalence class size to produce a data-state equivalence class sequence (as shown Figure 4.12).

The rank of an individual is the position in this sequence of the set that contains the data-state distribution equivalence class to which it belongs. The fitness function for an individual with a data-state distribution in a given class is:

$$branchCost + classSize - 1$$

This is zero when a solution is found because the branch cost is zero and the histogram of values assigned is unique, this assumes that the GA has not already found the solution and so the equivalence class size is one.

Population before ranking

Individuals	Fitness	Data state distributions
i1	c1	{v1, f1}
i2	c1	{v2, f2}
i3	c1	{v1, f1}
i4	c1	{v2, f2}
i5	c1	{v1, f1}
i6	c1	{v4, f4}
i7	c1	{v2, f2}
i8	c1	{v1, f1}
i9	c1	{v1, f1}
i10	c1	{v3, f3}
i11	c1	{v2, f2}
i12	c1	{v1, f1}
i13	c1	{v4, f4}
i14	c1	{v4, f4}
i15	c1	{v3, f3}
i16	c1	{v1, f1}
i17	c1	{v1, f1}
i18	c1	{v5, f5}
i19	c1	{v4, f4}
i20	c1	{v2, f2}

Class	count	Rank
{v5, f5}	1	1
{v3, f3}	2	2
{v4, f4}	4	3
{v2, f2}	5	4
{v1, f1}	8	5

Population after ranking

Individuals	Rank
i18	1
i10	2
i15	2
i6	3
i13	3
i14	3
i19	3
i2	4
i4	4
i7	4
i11	4
i20	4
i1	5
i3	5
i5	5
i8	5
i9	5
i12	5
i16	5
i17	5

Figure 4.12: Population before and after ranking using data-state distribution equivalence class size.

Other dissimilarity distance measures may be defined. In general, assume that there is a distance function $d(x_1, x_2)$ defined between any two histograms x_1 and x_2. This distance function is required to have the following properties for all x_1 and x_2:

1. Symmetric, $d(x_1, x_2) = d(x_2, x_1)$,
2. Non-negative, $d(x_1, x_2) \geq 0$,
3. Zero for identical histograms, $d(x_1, x_2) = 0$ if $x_1 = x_2$.

Two methods were considered to measure the distance between x_1 and x_2. These methods are common in the literature on population diversity (Mattiussi et al., 2004), and are applied to phenotype and genotype diversity, but here they are applied to data-state distributions.

1. Hamming distance: Each histogram is a set of (value, frequency) pairs. Two histograms may be compared by the number of pairs in each histogram that

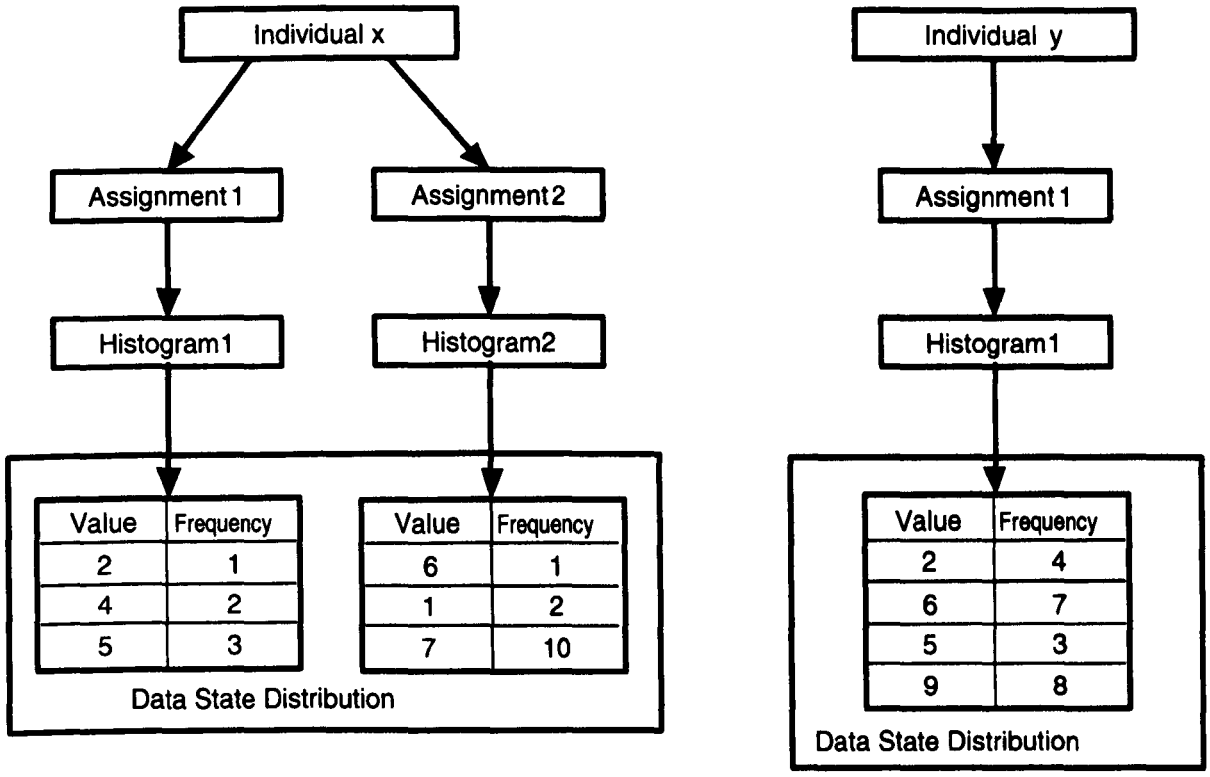


Figure 4.13: The data-state distributions of two individuals

are absent from the other. For example if

$x_1 = (v_1, f_1), (v_2, f_2), (v_3, f_3)$ and

$x_2 = (v_1, f_1), (v_2, f_4), (v_4, f_5)$ then

$HD(x_1, x_2) = 4$, since (v_2, f_2) and (v_3, f_3) are absent from x_2 and (v_2, f_4) and (v_4, f_5) are absent from x_1. So the definition will be:

$HD(x_1, x_2) = |x_1 \backslash x_2| + |x_2 \backslash x_1|$.

Extent x_1 with $|x_2 \backslash x_1|$ and give 0 counts.

Extent x_2 with $|x_1 \backslash x_2|$ and give 0 counts.

2. Euclidean distance: For each pair (v, f) in x_1, if (v, g) is present in $x2$ let $L_1 = \frac{(f-g)^2}{2}$ else $L_1 = f^2$. For each pair (v, f) in x_2, if (v, g) is present in x_1 let $L_2 = \frac{(f-g)^2}{2}$ else $L_2 = f^2$; then $ED(x_1, x_2) = L_1 + L_2$.

The difference between Euclidian distance and the Hamming distance is that the Euclidean distance is more sensitive to the frequency of data-state values.

The number of histograms inside the data-state distribution may be different from one individual to other as this depends upon the number of relevant assignment statements executed during the program execution. Figure 4.13 shows two individuals, the first one executed two relevant assignment statements (assignment1 and assignment2) then the data-state distribution consist of two histograms; the second individual executed one relevant assignment statement (assignment1) then the data-state distribution consists of only one histogram. To find the distance between these two individuals, the cost of insertion or deletion of a histogram will be considered next. Assume a single, unmatched histogram, e.g. Assigment2 in Figure 4.13 compared to the empty histogram shown as $y\phi$.

The Hamming distance between individual x and individual y is:
$HD(x,y) = HD(|x_1 \backslash y_1|) + HD(|y_1 \backslash x_1|) + HD(x_2 \backslash y\phi|) * 2$

$HD(x,y) = 11$, since (2, 1) and (4, 2) are absent from y_1 and (2, 4), (6, 2) and (9, 1) are absent from x_1 in addition to the cost of all pairs in x_2 which are absent from $y\phi$ (the cost = 6).

The Euclidean distance between individual x and individual y calculated as follows:
$ED(x,y) = ED(x_1,y_1) + ED(x_2,y\phi)$
$ED(x,y) = \frac{(1-4)^2}{2} + 2^2 + \frac{(3-3)^2}{2} + \frac{(4-1)^2}{2} + 7^2 + \frac{(3-3)^2}{2} + 8^2 + 1^2 + 2^2 + 10^2$
$ED(x,y) = \frac{9}{2} + 4 + 0 + \frac{9}{2} + 49 + 0 + 64 + 1 + 4 + 100$
$ED(x,y) = 231.$

The total distance between individual i and all other individuals in the population called the population distance Pd_i can be defined as:

$Pd_i = \sum_{j=1}^{n} d(i,j)$, where n is the population size, d is a distance function (e.g. Hamming distance or Euclidian distance) between two individuals i and j.

The sum of the distances from each individual to all the other individuals in the population is a measure of how similar an individual is to the population as a whole.

Pd_i can be used to rank individuals within the population, i.e. the individual with the largest Pd_i has the highest rank. This is called the maximum population

distance measure. During the search, this leads to a replacement strategy based on the contribution of diversity of the offspring to the population where it will be included. An individual of the population with a lower contribution of data-state distribution diversity than the one provided by the offspring will be replaced.

In more detail, let us assume that an offspring, x, is returned from the recombination phase and let i_{min} be an individual in the current population P which has the minimum population distance, let its distance be Pd_i. Consider now the population obtained by removing i_{min} and adding x. Call this population P' and let i'_{min} be the individual in the population P' that has the minimum population distance and let its distance be Pd'_i. If $Pd'_i > Pd_i$ then i_{min} is replaced by x, otherwise x is discarded.

4.5.4 Data-state distribution histogram

The fitness function based on the size of the equivalence class of data-state distributions directs the search to those individuals that are in equivalence classes of smallest size. After some time the number of large equivalence classes will have been progressively removed to make way for new individuals in smaller equivalence classes. This may continue for a while until all the individuals are in singleton equivalence classes and yet no solution has been found. The fitness function will then assign the same rank to all individuals since the data-state equivalence classes in the population are all the same size. In this case, the GA parent selection mechanism is random which will lead to random search. Such a situation occurs in the `Orthogonal` program, when the size of the array increased to be 128 instead of 64. In practice, a population in which all equivalence classes have the same size is likely to occur only when the number of classes is equal to the population size. Otherwise, different data-state values are likely to have non-uniform distributions. For example, in $\mathbf{Log}_{10}$, it is very unlikely that the equivalence class for $k = 5$ will contain as many individuals as the equivalence class for $k = 3$, say.

A possible remedy is to increase the population size and it would perhaps be useful

Histogram	Count	InCurrentPop
A{0 1 2 3 4 5 6 7 8 17 23 20 20 21 67 4 70 6 }	3	True
B{0 2 3 4 5 6 7 39 19 15 66 27 15 67}	7	True
C{0 1 2 3 4 5 6 7 15 20 15 6 66 73 28 26}	4	True
D{ 0 1 2 3 4 5 6 7 48 5 4 15 97 4 70 6 }	19	True
E{0 1 2 3 4 5 6 39 19 15 27 82 67 1}	8	False

Table 4.1: Example of DSD histogram used to rank individuals.

to repeat these experiments with a variety of different population size. Actually this solution is impractical when there is a very large number of data-state distributions. A solution is to record the histogram sets of data-state values produced by any individual during a particular search. These data-state distributions are recorded in a histogram called a data-state distribution histogram or DSD histogram for short. For each individual, a data value histogram set is stored in the DSD histogram as shown in Table 4.1. If the same histogram set is produced for different individuals, then the frequency count is increased. If an individual that produced a histogram set is in the current population then *InCurrentPop* flag is set true. Fitness function now depends on the size of the histogram set frequency count in the DSD histogram rather than the size of the class in the population. If a new individual has a fitness (branch cost and histogram scarcity) equal to the fitness of the other members of the population then it is added to the population, replacing an individual with the most common histogram, even if the size of the histogram class is one. The fitness cost for an individual with a data-state distribution that has a frequency count of f is

$$branchCost + f - 1$$

This value is zero when the first solution is found.

In the case of the `Orthogonal` program, where lengths of the arrays are larger than

the population size, the DSD histogram is constructed when the individuals in the population all belong to their own equivalence class (of size 1). New individuals with histogram sets that are not in the DSD histogram are added to the population and an arbitrary individual is removed, since all individuals in the population have the same histogram scarcity. This part of the search is random. After some time, however, new individuals may have histogram sets that have already been seen before, i.e. they are present in the DSD histogram. These individuals are less fit than any in the current population and are not added to the population because their histogram sets have been generated at least twice and all individuals in the population have a histogram set that has been seen just once only. The repeat occurrence of the histogram sets are noted, however, in the DSD histogram by incrementing the frequency count for the relevant histogram set. In this way, a variety of frequency counts will arise in the DSD histogram and lead to the individuals in the population have different fitness values. Once this occurs, the GA fitness based selection of the individuals for reproduction is restored and the GA search can find a solution.

DSD histogram can be used from the start but it increases the memory requirements and the search time.

4.5.5 Clustering histograms

There is a potential problem with the use of the DSD histogram is that it can become very large. An alternative method to the DSD histogram is to cluster the individuals in a population once the individuals are all in equivalence classes of size 1. Clustering is the classification of similar data-state distributions into classes so that the set of data-state distributions share some common trait. If the number of clusters is less than the population size, then cluster size can be used to measure the scarcity of the histograms of the individuals that it contains. This does not ensure that all cluster sizes will not be equal but it does decrease the likelihood. This method requires more time to compute the clusters but only a fixed amount of additional memory.

The rank of an individual in the population is the same as illustrated in Section 4.5.3 except that the size of the cluster to which an individual belongs is used instead of the size of the equivalence class to which it belongs.

To perform clustering a distance measure between data-state distributions is required. The simplest measure would be using the Euclidean distance as described in Section 4.5.3.

In the following sections, it is shown how two common clustering techniques may be adapted to cluster the individuals in the population.

Hierarchical clustering

The hierarchical clustering algorithm takes as input the number of desired classes k, and the distances between every pair of individuals. The algorithm is as follows:

- Start with n classes (population size), each containing a single data-state distribution.
- For i = $n - 1$ down to k
  - Find the closest pair of data-state distributions, call these A and B, and remove them from the set of classes.
  - Generate a new class C, containing the data-state distributions A and B.
  - Generate new distances from class C to all the other remaining classes. The distance between class C and some other class D is the average distance between the elements of C and the elements of D.

Figure 4.14 shows a possible clustering for a set of 6 data-state distributions. The fitness function for an individual with a data-state distribution in a given cluster is:

$$branchCost + clusterSize - 1$$

The value may not be zero when a solution is found but it does decrease towards individuals that have dissimilar data-state distributions.

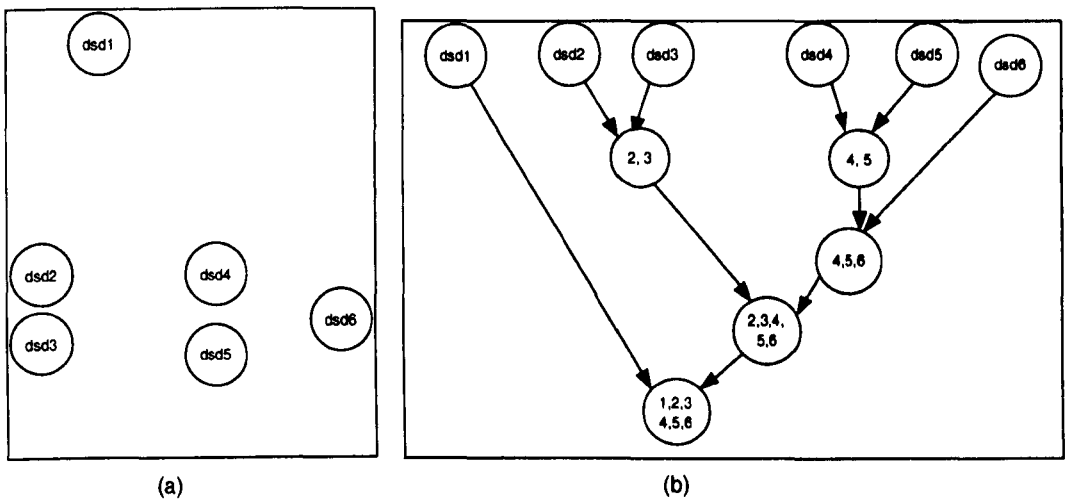


Figure 4.14: Hierarchical clustering example, using 6 data-state distributions

K-means algorithm

K-means (MacQueen, 1967) is one of the simplest unsupervised learning algorithms for clustering. The procedure sets a certain number of clusters (k clusters) fixed at the onset. The main idea of the K-means algorithm is to assign each point to the cluster whose centre is nearest. The centre is the average of all the points in the cluster. Example: The data set has three dimensions and the cluster has two points: X = (x_1, x_2, x_3) and Y = (y_1, y_2, y_3). Then the centroid Z becomes Z = (z_1, z_2, z_3), where $z_1 = (x_1 + y_1)/2$, $z_2 = (x_2 + y_2)/2$ and $z_3 = (x_3 + y_3)/2$.

The algorithm can use the square Euclidean distance between the data-state distributions as follows:

- Choose the number of classes, k.
- remove k individuals, chosen randomly, from the population to form the centres of k classes.
- for $i = 1$ to $n - k$
    1. Assign each remaining individual to the nearest class centre.
    2. Recompute the new class centres.

The main advantage of this algorithm is its simplicity. Its disadvantage is that it does not yield the same result with each run, since the resulting clusters depend on the initial k random selections. To solve this problem, the initial k individuals can be selected according to the total Euclidean distance (Pd_i) from those individuals to all other members of the population. Starting the clusters with the most distant individuals is a heuristic for selecting widely spaced clusters.

4.6 Instigating Data Scarcity Search

Data scarcity search should be instigated only when the branch cost function has become constant (locally flat) and program transformation techniques and path search techniques are either inapplicable or ineffective. Detecting that the search has stopped converging, stagnation, can be done by monitoring the average or best population fitness. In the work reported here, stagnation was defined as no improvement in the best cost value after 50 offspring. At this point, data-state scarcity search was introduced immediately since it was known that for the selected example programs, no transformation technique was applicable.

4.7 Sampling the Data-state to Produce the Data-state Distribution

Recording data-state information is a computational cost that need not be incurred until data-state scarcity search is instigated. This can be done by re-instrumenting the program under test. To sample the data-states, the variables that appear in the predicate expression of the target branch are identified and from the data dependency graph, the variables that affect the variables of the predicate expression are also identified. Input variables are excluded. In order to identify rare data-state values it is necessary to instrument the distribution of values that are assigned to these variables. To do this, each definition of a variable (e.g. an assignment state-

ment), providing it is not the assignment of a compile-time constant, is associated with a histogram in which is recorded the values assigned and the number of times any particular value is assigned. The instrumentation of the program shown in Figure 4.6 is shown in Figure 4.15. `Inst()` returns the value of its first argument after adding this value to the histogram associated with the variable definition that is labeled by the second argument. Note that the array `a` is not instrumented since it is assigned only constant values. In general, however, the instrumentation of values assigned to an array is a problem that was ignored for this work.

```
void Log10(int x){
   //x in [0, 100000]
   a[0] = 0;
   a[1] = a[2] = a[3] = a[4] = a[5] = 1;
   double y =Inst(log10(x), "y1");
   int k = Inst(ceiling(y), "k1");
   if (a[k] == 0) {
      //TARGET BRANCH
   }
}
```

Figure 4.15: Data-state instrumentation of the program from Figure 4.6.

A program with a loop may generate a large number of different values for a particular variable assignment which will lead to an impractically large number of histogram classes. To limit the number of classes in the histogram, the rate at which assigned values are sampled is progressively reduced as the number of classes increases. Initially all values assigned are recorded until the number of assignments (each assignment is recorded in a histogram) $\times$ `k` equals a positive constant s, set to 1000 for this work, where `k` is the maximum number of values stored in one histogram. At this point, the sampling rate is halved so that only each second value is recorded. This does not directly limit the number of new classes but it does reduce

the rate at which they may be created. If the number of class equals $2s$ then the sampling rate is again halved, and so on. This scheme biases data-state sampling to states that are produced early on in the computation. There will be programs for which this bias is advantageous and programs for which it is not. Given that the performance implications of the scheme are unclear, at the moment the scheme can be justified only on the basis of the simplicity of implementation.

4.8 Empirical Investigation

The data-state scarcity strategy was investigated by generating test data for the example programs, `AllTrue`, `Orthogonal`, `Log10`, `Mask` and the three programs `Error`, `CountEqual` and `FloatRefEx`, of which the latter two are described later on.

4.8.1 Experimental setup

The tool used here is the same as described in Section 3.3, Page 48. In the work reported here, a population size of 100 was always used. This parameter was not "tuned" to suit any particular program under test. In a steady state update style of genetic algorithms (as used in this work), new individuals that are sufficiently fit are inserted in the population as soon as they are created. Full branch coverage was attempted for each of the programs under test. Each branch was taken as the individual target of the search, unless it was fortuitously covered during the search for test data for another branch.

GAs search generates inputs for the function containing the current structural target. A vector of floating point, integer, characters and string variable values corresponding to the input data is optimized. The ranges of each variable are specified. The test subject is then called with this input data. The criterion to stop the search was set up to terminate the search after 100,000 executions of the program under test if full coverage was not achieved. Individuals were recombined using binary and real-valued (one-point and uniform) recombination, and mutated using real-valued

mutation. Real-valued mutation was performed using "Gaussian distribution" and "number creep".

4.8.2 Experimental programs

The program `CountEqual` shown in Figure 4.16 determines if more than half the characters in a string are equal to the respective preceding character.

```
CountEqual(char[] a) {
  int equal = 0;
  for (i = 0; i < 64; i++) {
    string s = match(a[i] + "+", a, i);
    equal = equal + s.Length - 1;
    i = i + s.Length - 1;
  }
  if (floor(equal / 32) == 1) {
    //TARGET
  }
```

Figure 4.16: A difficult to execute branch in a program

The variable `equal` is likely to be zero or close to zero. In such cases, the value of `equal / 32` is invariably zero. For randomly selected inputs, the histogram of values assigned to `equal` will be skewed towards zero. Directing the search towards inputs that produce rare histograms will direct the search towards inputs in which the histogram of values assigned to `equal` is less skewed towards zero, which increases the probability of finding an input with a relatively high value for `equal`. The target branch is executed when `equal` has the value 32.

The program `FloatRegEx` is shown in Appendix A and its flowchart is shown in Figure 4.18. This program implements a finite state machine to recognise floating point numbers with an optional exponent. The state transitions of the finite state

```
Error(int[] a) {
  //a[i] in [-100000, 100000]
  int error = 0;
  int errorsum = 0;
  for (i = 0; i < 16; i++) {
    error = abs(a[i]) - i;
    errorsum = errorsum + min(1, abs(error));
  }
  if (floor(errorsum / 4) < 1) {
      //TARGET
  }
}
```

Figure 4.17: Program compares a set of points with the sequence 0, 1, 2, ..., 15 and executes the target branch when fewer than one quarter of the points disagree with the sequence.

machine are defined in an array. The program is a single loop that reads each character of the input string and, together with the current state, accesses the next state from the array. The target branch is executed when the state corresponding to a number with an exponent is reached. For random character strings, this is a difficult state to reach.

Clearly, the set of test programs assembled is a biased collection but the purpose of the investigation is to show the effectiveness of data scarcity search for a class of program for which existing techniques are not effective.

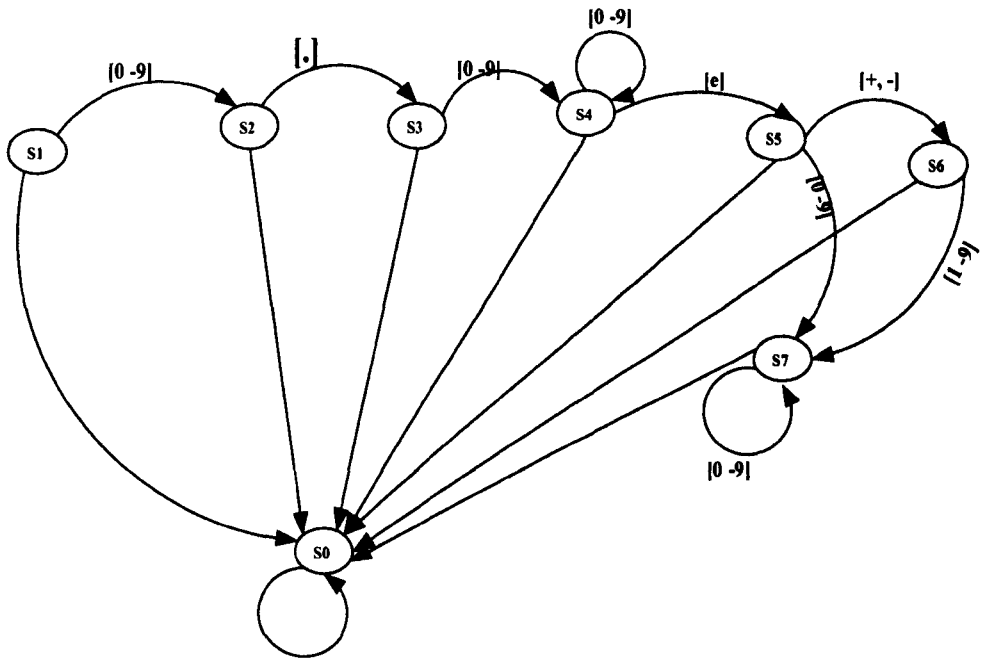


Figure 4.18: Flowchart of FloatRegEx example

4.8.3 Results

Number of data-state distributions is less than population size

In order to assess the fitness function when the branch distances of the population are all equal and the number of the data-state distributions is less than population size, in this case, the size of the data-state distribution equivalence class to which the individual belongs and the population distance Pd_i are used for population member ranking. Test data was generated for each program and the number of program executions required to find data for a given program was noted. This was done for 50 trials and the average taken.

The results in Table 4.3 show the average number of executions required to find test data when equivalence class and distance between data-state distributions (Hamming distance and Euclidean distance) were used. There is no evidence to suggest that one method is more or less efficient than the others in terms of performance but using equivalence class for population member ranking needs less computation than using Hamming distance and Euclidean distance.

Program	Successes	Equivalence class	HD	ED
`AllTrue`	50	4651	6563	8218
`Orthogonal`	50	8004	10814	9325
`Log10`	50	2184	1641	1641
`Mask`	50	1119	3251	2074
`CountEqual`	50	9421	9652	9936
`Error`	50	8719	12362	12251
`FloatRegEx`	50	11081	12354	11832

Table 4.2: The number of successful trials and the average number of test program executions out of 50 required to find test data to achieve coverage of the target branch.

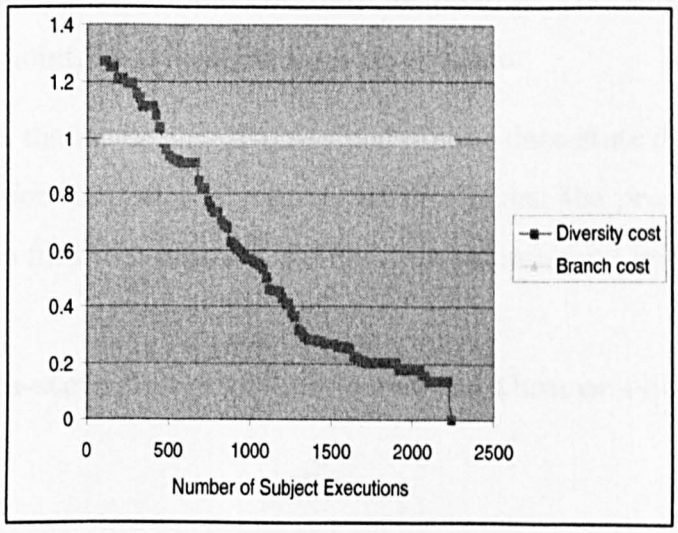


Figure 4.19: Plot showing the fitness landscape for the branch cost and the diversity cost in the program $\texttt{Log}_{10}$. For clarity of presentation, the diversity cost plotted is the maximum entropy value (obtained when the solution is found) less the entropy value after a given execution of the program.

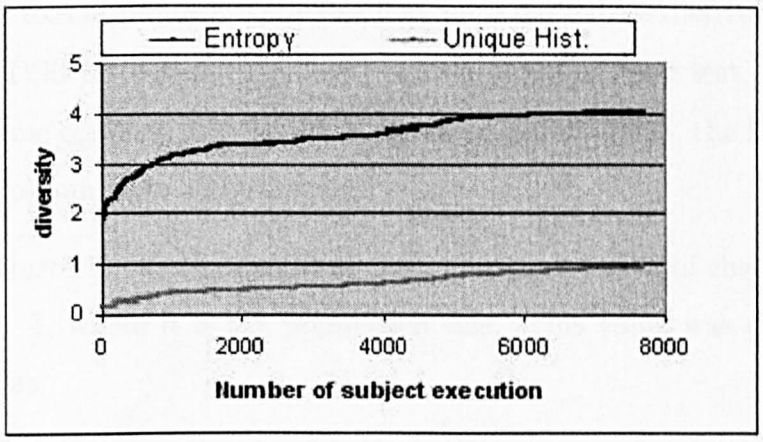


Figure 4.20: The diversity in the program `Orthogonal`. For clarity of presentation, the unique data-state distribution is normalised between 0 and 1.

No figures for the number of executions required to execute the target branch without using data-state scarcity search are given, since no solutions were found after 100,000 executions of the program under test.

Figure 4.19 shows how the data-state scarcity cost descreases during the progress of the search for a single run to find test data for program $\texttt{Log}_{10}$. The branch cost is 1 for all but one point, i.e. when the solution is found.

Figure 4.20 shows the increasing diversity and unique data-state distribution (scaled between 0 and 1 for simplicity of representation) during the progress of the search for a single run to find test data for the `Orthogonal` program in Figure 4.5.

Number of data-state distributions is greater than or equal to the population size

In order to assess the three methods of fitness function definition when the number of histograms is $\geq$ population size, the sample of programs is modified for this purpose, i.e `AllTrue128` means the `AllTrue` program in Figure 4.4 is modified by making the array length equal to 128 instead of 64. This means the number of histograms will be increased to 128. The DSD histogram for the `AllTrue128` program is shown Figure 4.21a, all equivalence class of size 1. The search in this part is random.

After a time of execution, new test cases may have data-state distributions that are already in the DSD histogram. This increases the count of these test cases and these test cases become common and less fit as shown in Figure 4.21b, the DSD histogram of the same program after 520 offspring.

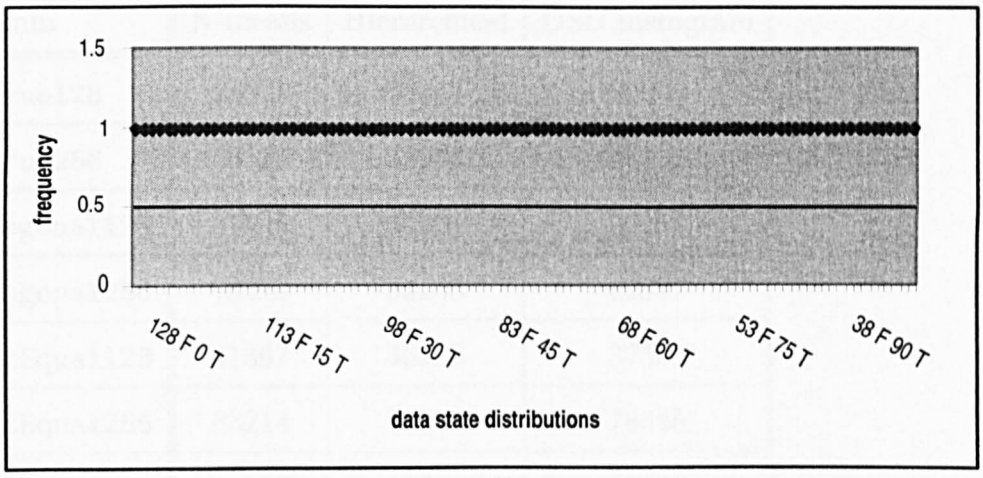
For K-means clustering and hierarchical clustering the number of clusters is selected to be equal to $\frac{n}{2}$, where n is the population size. This value was chosen without detailed analysis.

A problem was identified in hierarchical and K-means clustering algorithms, that when we are building the cluster, sometimes there is more than one closest cluster. Assume we have a cluster A, which has the same distance to both clusters B and C. The algorithm at that point chooses one, arbitrarily. Say the algorithm chooses cluster B, thus forming cluster A'. Now cluster A' has a particular distance to cluster D which may be very different from the distance it would have had if the algorithm had chosen C and A to form A'.

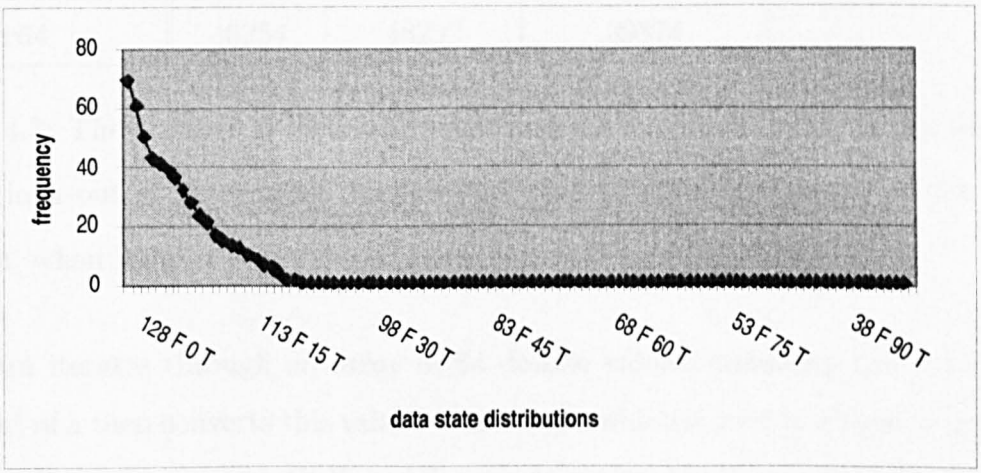
The results in Table 4.3 show the number of program executions required to find input data to achieve branch coverage, averaged over 50 trials. These results provide some evidence that DSD histogram is the most efficient of the three methods and that hierarchical is more efficient than the K-means. The reason for the poorer performance of K-means is related to the previous problem and the selection of initial k cluster.

4.9 Combining Program-specific Operators with Data Search

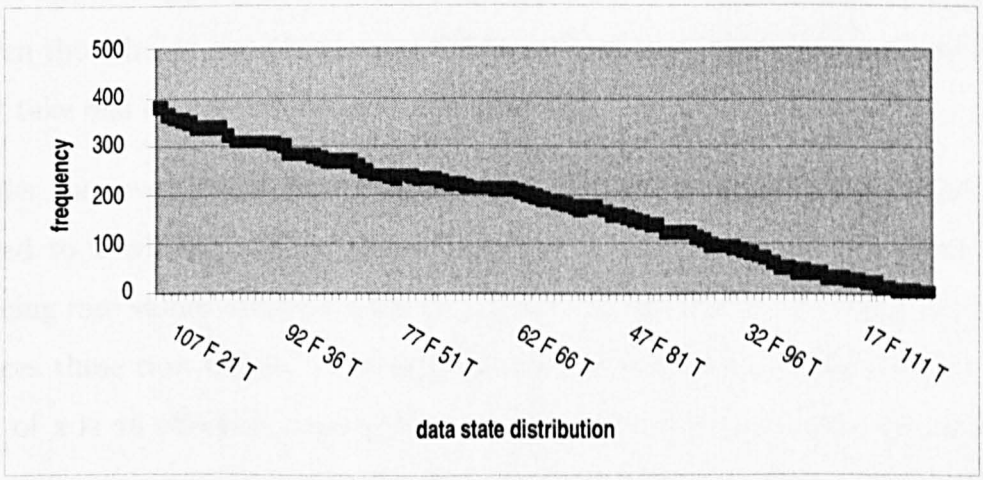
When the fitness landscape becomes locally flat for large areas of the input domain, data-state scarcity will be introduced to solve the problem but this does not exclude the use of program-specific search operators. In this section data-state scarcity is used with program-specific search operators in order to increase the performance of the search. This can be illustrated by example in Figure 4.22 (`FlagAvoid`). This



(a)



(b)



(c)

Figure 4.21: (a) DSD histogram for AllTrue128 program is constructed when the population size is equal to the data-state distribution size, (b) DSD histogram after 520 program executions, (c) DSD histogram after 11000 program executions for `true` value of $InCurrentPop$.

Program	K-means	Hierarchical	DSD histogram
`AllTrue128`	28325	23847	20005
`AllTrue256`	66135	58964	48632
`Orthogonal128`	37945	33264	30987
`Orthogonal256`	73218	69258	62847
`CountEqual128`	41367	40988	37564
`CountEqual256`	83214	82586	78645
`Mask64`	11387	12631	10456
`Error64`	46254	48299	39874

Table 4.3: The number of successful trials and the average number of test program executions out of 50 required to find test data to achieve coverage of the target branch, when number of data-state distribution $\geq$ population size.

program iterates through an array of 64 double values, computes the *sin* of each element of `a` then converts this value to an integer which is used in a branch predicate that leads to the target. All flag assignments within the loop have to be avoided, and the target could be covered only when all of the elements of array `a` are equal to $\frac{\Pi}{2}$. Even though `x` is not a boolean variable, a "flag" variable problem arises because `x` may take one of only two integer values (0, 1).

Consider, however, the values assigned to `x` within the loop. Most of the values assigned to `x` will be zeros. The inputs that assign 1 to `x` may be identified as producing rare values and hence the search may be directed to the input region that produces these rare values. Directing the search towards input that produces rare values of `x` is an effective strategy for changing the final value when the maximum number of ones is assigned. By using the program-specific search operators, although none of the three integer values 0, 1, and 64 that occur in the program are input values that execute the target branch (to execute the target branch all the elements of the array must be equal to $\frac{\Pi}{2}$), they do provide reasonable starting points for a guided search. In particular, to set the variable `a[i]` $= \frac{\Pi}{2}$, it is possible to

```
void  FlagAvoid(double[] a){
    int x = 0;
    double y = 0.0;
    int i;
    for(i = 0; i< 64; i++){
        y = Math.sin (a[i]);  //y in  [-1, 1]
        y = Math.abs(y);      //y in  [0, 1]
        x = Math.floor(y);    // x = 0 or 1 only
        if (x != 1){
             break;
        }
    }
    if (x == 1){//EXECUTED ONLY WHEN ALL VALUES OF a EQUAL TO 90.
        //target executed
    }
}
```

Figure 4.22: A difficult to execute branch in a program **FlagAvoid**

invert the trigonometric function $sin(a[i])$ with parameter value equal to 1 which is $sin^{-1}(1)$ then `a[i]` $= \frac{\Pi}{2}$. Figure 4.23 shows different paths to a solution to test data generation problem shown in Figure 4.22.

Experiments performed using different programs listed in Table 4.3 in addition to the **FlagAvoid** program were used to evaluate the performance of program-specific search operators with data-state scarcity. The test tool collected program literals and mathematical operators and determined the mutation operators during a traversal of the program abstract syntax tree. Each program contained a specific statement that could not be easily covered by GAs, due to a specific low probability statement sequence required to be followed before the target is reached. The aim was

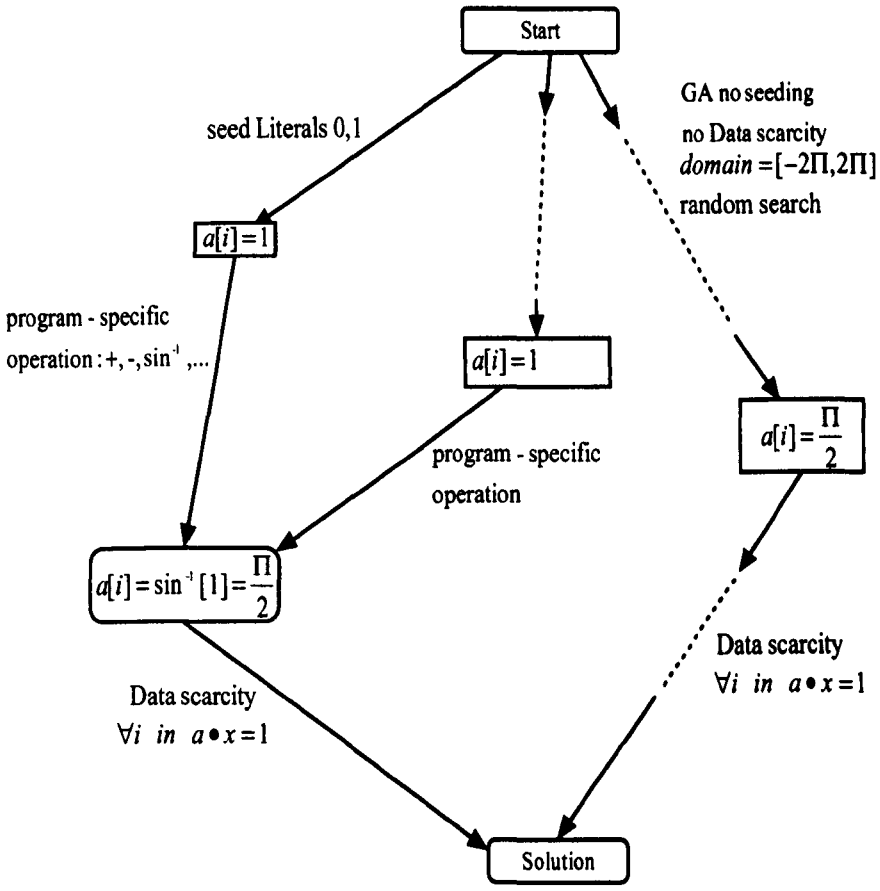


Figure 4.23: The different paths to a solution to the test data generation problem shown in Figure 4.22.

program Name	Execution
`Orthogonal`	55
`Log10`	0
`Mask`	586
`CountEqual`	618
`Error`	11243
`FloatRegEx`	5751
`FlagAvoid`	3971

Table 4.4: Number of subject program executions to cover all branches, average of 50 trials was used, equivalence class was used for population member ranking.

to find input data to execute all the branches in each programs. For each program 50 trials were done. The average number of program executions required to achieve branch coverage over 50 trials is shown in Table 4.4. These results show a significant improvement in performance compared to the results without using program-specific search operators, as shown in Figure 4.24, which compares the results of the data-state scarcity search with and without using program-specific search operators when equivalence class was used for population member ranking. The results in Table 4.5 show the average number of program executions required to achieve branch coverage over 50 trials, when number of data-state distributions $\geq$ population size when DSD histograms was used.

The usefulness of using program data constants by itself without program functions and operators has not been investigated here. Similarly we have not investigated program operators and functions without data constants. There seems to be little advantage in using each of these by itself, i.e the implementation cost is not significantly reduced.

In example, Figure 4.17 (`Error`) the inclusion of literals from the program in the integer domain will bias the search towards arrays that contain a greater than av-

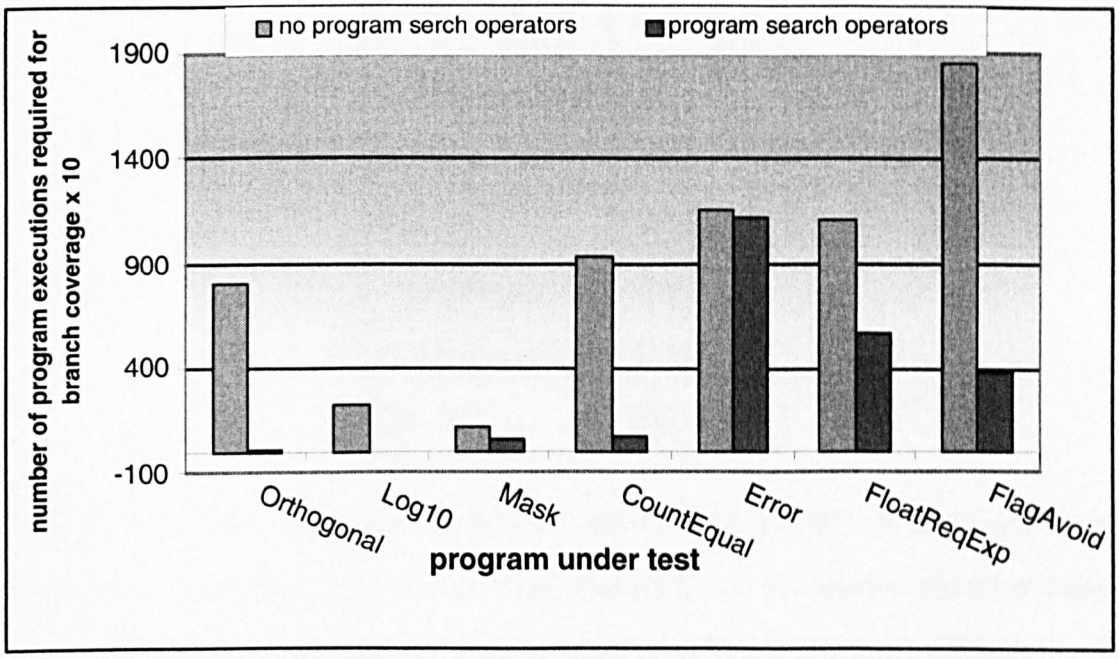


Figure 4.24: A comparison of the number of executions of the program under test required to find test data to achieve branch coverage (averaged over 50 trials) with and without program-specific search operators.

erage proportion of the literals 0 , 1, 4, 16. These values are not helpful in finding the solution array which is [0, ±1, ± 2, ± 3, ± 4, ..., ± 15]. In fact the bias towards the literals 0, 1, 4, 16 is counterproductive, as the bias towards literals impedes the search, but using the program mutation operators ("-" and "+") may make a little improvement.

In general, the use of program-specific search operators leads to about a threefold improvement in search efficiency. In the case of $\texttt{Log}_{10}$, no guided search was required. The solution of the target already exists in the initial population generated from the program literals.

4.10 Summary

Programs that contain flag variables or otherwise generate almost locally flat cost functions pose a problem for heuristic search algorithms that seek to minimise a cost

program Name	Execution
`Orthogonal128`	81
`Orthogonal256`	362
`Mask64`	1746
`CountEqual128`	6107
`CountEqual256`	16984
`Error64`	38798

Table 4.5: Number of subject program executions to satisfy all branches, average of 50 trials was used, equivalence class was used for population member ranking combined with program-specific search operators. When number of data-state distributions $\geq$ population size, DSD histogram is used.

function. Existing methods based on program transformations and data flow search are ineffective for many programs and some examples have been given to illustrate the problem. A new approach of data-state scarcity search is shown to overcome the problem. The solution is most appropriate for programs that have little scope to exploit control flow diversity. Such programs contain few branches and may be "data-driven" of which table-driven finite-state machines are an example. Two fitness functions were investigated, one based on the grouping of equal data-state distributions and another based on the distance between data-state distributions. A limitation emerged when using grouping of equal data-state distributions and using population distance (Pd) when the number of data-state distribution $\geq$ population size. A possible solution is to record the histogram sets of data-state values (DSD histogram). Fitness function depends on the size of the histogram set frequency count in the DSD histogram rather than the size of the class in the population. There is a potential problem with the use of the DSD histogram is that it can become very large. An alternative method to the DSD histogram is to cluster the individuals in a population once the individuals are all in equivalence classes of

size 1. The rank of an individual in the population depends on the size of the cluster to which an individual belongs, instead of the size of the equivalence class to which it belongs. The most significant improvement in performance, however, was obtained by using program-specific search operators. In the empirical investigation, the use of program-specific search operators was shown to give a threefold increase in performance.

In general, the problem of almost locally flat cost functions is such that no single approach can be expected to solve the different cases in which it may occur. A variety of techniques are required. This chapter describes one such technique.

Chapter 5

Conclusions and Future Work

5.1 Summary of Achievements

The original overall aims and objectives of this thesis were as follows:

1. Generation of string test data automatically.
2. Generation of test data for some programs that exhibit an almost constant cost function at the test goal.
3. Demonstrating the effectiveness of program-specific search operators for many types of test program.

5.1.1 Generating string test data

The examination of the SSCLI source code showed that about 6% of the predicate expressions is a string predicate expression and yet work on test data generation has so far been largely limited to numeric test data . For string equality predicates, the following cost functions were investigated:

1. an adaptation of the binary Hamming distance (HD).

2. character distance(CD), defined as the sum of the absolute difference between the ordinal character values of corresponding character pairs.

3. ordinal edit distance (OED), a version of edit distance costs with fine-grained costs based on the difference in character ordinal values and defined as $OED(s:a, t:b) = min(OED(s:a,t)+k, OED(s,t:b)+k, OED(s,t)+|a-b|)$ where $s:a$, $t:b$ are character strings, each consisting of a possibly empty string s, t, followed by the character a and b, k is the insertion or deletion cost and a, b in $|a-b|$ are interpreted as ordinal values.

An ordinal value ordering in which the string is considered as a number with base equal to the cardinality of the character set is unsuitable as a cost function for string equality since it treats mismatches differently according to their location in the string.
Ordinal edit distance was found the most effective in an empirical study. Three basic kinds of mutation operators, deletion, insertion and substitution were used, initially with equal probability but a progressive increase in the probability with which the character substitution is applied and the standard deviation of the Gaussian substitution operator was reduced has been shown to improve the performance of the search. This was done because later in the search the candidate strings tend to have the same length as the required string. Two functions for string ordering were investigated, Ordinal value ordering and Single character pair ordering, but there was no significant difference in their performance in the empirical investigation although one is easier to implement.

5.1.2 Data-state scarcity search as a solution for flag problem

The computations performed by programs can result in a degree of "information loss" when computing the branch distance measure, producing coarse or flat objective function landscapes for structures within the program. This in turn results in the search receiving little guidance to the required test data, and it typically fails.

Existing methods are ineffective for many programs and some examples have been given.

In an attempt to tackle this problem, Chapter 4 presents a new technique for directing the search. The new technique depends on introducing program data-state scarcity as an additional search goal. The search is guided by a new evaluation (cost) function made up of two parts, one depending on the conventional instrumentation of the test goal, the other depending on the diversity of the data-states produced during execution of the program under test. Two fitness functions were investigated, ranking of candidate solutions by grouping of equal data state distribution and ranking by distance between data state distributions.

Using equivalence classes of data state distribution is ineffective when the number of data state distributions equals the population size. A possible solution is to record the histogram sets of data state values in a DSD histogram that is not limited in size. Fitness function now depends on the size of the histogram set frequency count in the DSD histogram rather than the size of the class in the population. The fitness cost for an individual with a data state distribution that has a frequency count of f is

$$branchCost + f - 1$$

This value is zero when the first solution is found.

A potential problem with the use of the DSD histogram is that it can become very large. An alternative method to the DSD histogram is to cluster the individuals in a population once the individuals are all in equivalence classes of size 1. The rank of an individual in the population depends on the size of the cluster to which an individual belongs, instead of the size of the equivalence class to which it belongs.

Full branch coverage was obtained in all experiments by using DSD histograms and clustering histograms. The results provides some evidence that using DSD histograms is the more efficient than other methods.

5.1.3 Using program-specific search operators

Program-specific search operators aim to exploit the structure and behaviour of the computation in the region in the program from the input variable to the test goal rather than the test goal by itself. The structure of the computation can be used in the search by using the functions available in the program under test as the basis of search operators and constants to seed the search. This idea was applied first on the string data problem. The examination of the SSCLI code showed that about 65% of string predicate expressions contains a string literal.

By exploiting the presence of string literals in programs that process string data, a very significant improvement in performance was obtained. The program-dependent string search operators that focus the search in the region of string literals were presented in Chapter 3, and in the empirical investigation, the use of these operators was shown to give a fivefold increase in performance. The program-specific search operators have been demonstrated for strings but this technique can be generalised to other data types.

In the case of numerical types, additional genetic operators were introduced to increase the performance of search by analysing the program under test and extracting arithmetic operators and trigonometric function presented in the program under test and then using these functions and their inverse as additional mutation operators.

More generally, the proposed approach is to exploit the structure and behaviour of the computation from the input `x` to the test goal, the usual instrumentation point. Assume this computation sequence consists of the sequence of statements of the form $s = f(\acute{s})$ where s and $\acute{s}$ are expressions that reference the data-state and f is a function. The proposed approach is illustrated in Figure 5.1.

The structure of the sequence can be used in the search by using the functions `f1`, ..., `fn` as the basis of search operators and constants to seed the search. The behaviour of the sequence can be used in the search by examining the intermediate store values `s1`, `s2`, ... to provide additional guidance to the search.

Program-specific search operators were combined with data-state scarcity search for

```
...
s1 = f1(x);
s2 = f2(s1);
...
y = fn(sn -1);
if(y == a) { // instrumentation here only, cost = abs(y – a)
}
```

(a)

```
...
s1 = f1(x);
s2 = f2(s1);
...
y = fn(sn -1);
if(y == a) {
}
```

// data-state instrumentation

// test goal instrumentation, cost = abs(y – a)

(b)

Figure 5.1: (a) Existing approach to test data search depends on cost function (b) New approach exploits the structure of the program to use the constants and specific-search operators and intermediate data state values.

the flag problem in Chapter 4 and the empirical investigation of the use of program-specific search operators was shown to give a threefold increase in performance. However, applying program-specific search operators with conventional genetic algorithms increases performance more than 25 times.

On the whole, program-specific search operators can be combined with other techniques, e.g. transformation (Harman et al., 2004), data-state scarcity and conventional genetic algorithms to increase the speed of search.

5.2 Limitations and Future Work

The data-state scarcity search method, as it currently stands, has some limitations with respect to the type of programs that can be handled:

1. A program with a loop may generate a large number of different values for a

particular variable assignment which lead to a large number of histograms. In this work, the number of classes in the histogram is limited until the number of assignment $\times$ k equals a constant s (set to 1000 for this work), where k is the maximum number of values stored in one histogram. It is not clear how this heuristic performs for programs that have large numbers of loop iterations.

2. The array is not instrumented in this work. It is not clear how large data structure should be instrumented to collect data-state values.

There is also a potential problem in that the DSD histogram may become very large.

These limitations could be addressed in future work.

Appendix A

Test Programs

A.1 Calc

```
double  Calc(String op, double arg1, double arg2){
    op = op.ToLower();
    double  result   = 0.0
    if ("pi" == op) { //CONSTANT OPERATOR
        result = System.Math.PI;
    }
    else if ("e" == op) {
      result = System.Math.E;
    }       //UNARY OPERATOR
    else if ("sqrt" == op) {
      result = System.Math.Sqrt(arg1);
    }
    else if ("log" == op) {
      result = System.Math.Log(arg1);
    }
```

```
    else if ("sine" == op) {
      result = System.Math.Sin(arg1);
    }
    else if ("cosine" == op) {
      result = System.Math.Cos(arg1);
    }
    else if ("tangent" == op) {
      result = System.Math.Tan(arg1);
    }      //BINARY OPERATOR
    else if ("plus" == op) {
        result = arg1 + arg2;
      }
      else if ("subtract" == op) {
        result = arg1 - arg2;
      }
      else if ("multiply" == op) {
        result = arg1 * arg2;
      }
      else if ("divide" == op) {
          result = arg1 / arg2;
      }
      return result;
}
```

A.2 Cookie

```
int Cookie(String name, String val, String site){
   name = name.ToLower();
   val = val.ToLower();
   site = site.ToLower();
   int  result  = 0;
       if ("userid" == name) {
         if (val.Length > 6) {
           if ("user" == val.Substring(0, 4)) {
         result = 1;
       }
       }
     }
     else if ("session" == name) {
         if ("am" == val && "abc.com" == site) {
         result = 1;
     }
     else {
         result = 2;
     }
     }
  return result;
}
```

A.3 DateParse

```
void  DateParse(String dayname, String  monthname){
    var result : int = 0;
    var month : int = -1;
    dayname = dayname.ToLower();
    monthname = monthname.ToLower();
    if ("mon" == dayname ||
        "tue" == dayname ||
        "wed" == dayname ||
        "thur" == dayname ||
        "fri" == dayname ||
        "sat" == dayname ||
        "sun" == dayname) {
      result = 1;
    }
    if ("jan" == monthname) {
      result += 1;
    }
    if ("feb" == monthname) {
      result += 2;
    }
    if ("mar" == monthname) {
      result += 3;
    }
    if ("apr" == monthname) {
      result += 4;
    }
```

```
    if ("may" == monthname) {
      result += 5;
    }
    if ("jun" == monthname) {
      result += 6;
    }
    if ("jul" == monthname) {
      result += 7;
    }
    if ("aug" == monthname) {
      result += 8;
    }
    if ("sep" == monthname) {
      result += 9;
    }
    if ("oct" == monthname) {
      result += 10;
    }
    if ("nov" == monthname) {
      result += 11;
    }
    if ("dec" == monthname) {
      result += 12;
    }
  }
```

A.4 FileSuffix

```
int FileSuffix(String directory, String file){
       //EG pathname = "...WORD/FILE.DOC";
     Object[] files;
     String[] fileparts;
     int lastfile = 0;
     int lastpart = 0;
     String suffix ;
     fileparts = file.Split(".");
     lastpart = fileparts.Length - 1;
   if (lastpart > 0) {
        suffix = fileparts[lastpart];
        if ("text" == directory) {
            if ("txt" == suffix) {
                //print("text");
            }
         }
        if ("acrobat" == directory) {
             if ("pdf" == suffix) {
               //print("acrobat");
             }
         }
         if ("word" == directory) {
             if ("doc" == suffix) {
               //print("word");
             }
         }
```

```
            if ("bin" == directory) {
                if ("exe" == suffix) {
                  //print("bin");
                }
            }
            if ("lib" == directory) {
                if ("dll" == suffix) {
                  //print("lib");
                }
            }
        }
    return 1;
  }
}
//var ct = new CUT();
//ct.Subject("word", "file.doc");
//ct.Subject("text", "file.txt");
//ct.Subject("acrobat", "file.pdf");
//ct.Subject("bin", "file.exe");
//ct.Subject("lib", "file.dll");
```

A.5 Stem

```
String Suffix(String s, int len) :  {
    //SUFFIX OF NO MORE THAN len CHARS
    int  slen = s.Length;
    if (slen > len) {
      return s.Substring(slen - len, len);
    }
    else {
      return s;
    }
  }

  boolean  Consonant(String s, int pos) :  {
   //CONSONANT AT pos
   //CONSONANT EXCLUES VOWELS AND Y PRECEDED BY A CONSONANT, E.G. ...TY
   int  slen  = s.Length;
   if (pos < 0 || pos > slen - 1) {
     return false;
   }
   switch (s[pos]) {
     case 'a':
     case 'e':
     case 'i':
     case 'o':
     case 'u': {
       return false;
     }
```

```
      case 'y': {
        if (0 == pos) {
          return true;
        }
        else {
          return !Consonant(s, pos - 1);
        }
      }
      default: {
        return true;
      }
    }
  }

  boolean  DoubleConsonant(String s, int pos)   {
    //DOUBLE CONSONANT AT pos
    //CONSONANT EXCLUES VOWELS AND Y PRECEDED BY A CONSONANT, E.G. ...TY
     int  slen = s.Length;
     if (pos < 1 || pos > slen - 1) {
       return false;
     }
     if (s[pos - 1] != s[pos]) {
       return false;
     }
     return Consonant(s, pos - 1);
   }

  // does stem end with CVC?
```

```
boolean  EndsWithCVC(String s){
 int slen = s.Length;
 if (slen < 3) {
    return false;
 }
 if (!Consonant(s, slen - 1) || Consonant(s, slen - 2)
                             || !Consonant(s, slen - 3)) {
    return false;
 }
 char c = s[slen - 1];
 return !(c == 'w' || c == 'x' || c == 'y');
}

int  stringMeasure(String s, int len){
 // returns a CVC measure for the string
 //len IS LENGTH OF PREFIX OF s TO BE CONSIDERED
 int  n  = 0;
 int  i  = 0;
 while(true) {
   if (i >= len) {
     return n;
   }
   if (!Consonant(s, i)) {
     break;
   }
   i++;
 }
 i++;
```

```
    while(true) {
      while(true) {
        if (i >= len) {
          return n;
        }
        if (Consonant(s, i)) {
          break;
        }
        i++;
      }
      i++;
      n++;
      while(true) {
        if (i >= len) {
          return n;
        }
        if (!Consonant(s, i)) {
          break;
        }
        i++;
      }
      i++;
    }
  }

  // does string contain a vowel?
boolean  ContainsVowel(String s, int len){
    int  i = 0;
```

```
        while(i < len) {
             if (!Consonant(s, i)) {
               return true;
             }
             i = i + 1;
        }
        return false;
    }

int  Subject(String s){
     char c;
     int i = 0;
     s = s.Trim();
     if (s.Length < 2) {
       return 0;
     }
     //all characters must be LOWERCASE
     s = s.ToLower();
     i = 0;
     while(i < s.Length) {
       if (s[i] > "z" || s[i] < "a"){
         //  return "Invalid term";
         return 0;
       }
       i = i + 1;
     }
        // IES -> I
    if ("zies" == Suffix(s, 4)) {
```

```
        s = s.Substring(0, s.Length - 2);
      }
      // SS -> S
      else if ("ess" == Suffix(s, 3))  {
        s = s.Substring(0, s.Length - 1);
      }
      // S ->
      else if ("sses" == Suffix(s, 4))  {
        s = s.Substring(0, s.Length - 1);
      }
      else {
        s = s;
      }
    // end step1a
    //step1b
    if (s.Length < 3) {
      return 0;
    }
    else // AT -> ATE
      if ("stat" == Suffix(s, 4) ||
          "bibl" == Suffix(s, 4) ||
          "lsiz" == Suffix(s, 4)) {
       s = s + "e";
    }
    return 0;
  }//end Stem
```

A.6 Pat

```
 int Pat(String txt, String pat){
//SEARCH txt FOR FIRST OCCURRENCE OF pat OR REVERSE OF pat
//IF pat (STRING OF LENGTH AT LEAST 3) OCCURS IN txt, RTN 1
//IF REVERSE OF pat OCCURS IN txt, RTN 2
//IF pat AND REVERSE OF pat OCCURS IN txt, RTN 3
//IF PALINDROME CONSISTING OF pat FOLLOWED BY REVERSE pat OCCURS IN txt, RTN
//IF PALINDROME CONSISTING OF REVERSE pat FOLLOWED pat OCCURS IN txt, RTN 5
    int  result = 0;
    int  i  = 0;
    int j  = 0;
    int txtlen  = txt.Length;
    int patlen  = pat.Length;
    String possmatch;
    if (patlen > 2) {
        String patrev  = Reverse(pat);
        for (i = 0; i <= txtlen - patlen; i++) {
            if (txt[i] == pat[0]) {
                possmatch = txt.Substring(i, patlen);
                if (possmatch == pat) {
                    //FOUND pat
                    result = 1;
                    //CHECK IF txt CONTAINS REVERSE pat
                    for (j = i + patlen; j <= txtlen - patlen; j++) {
                        if (txt[j] == patrev[0]) {
                            possmatch = txt.Substring(j, patlen);
                            if (possmatch == patrev) {
```

```
                        if (j == i + patlen) {
                            return 4;
                        }
                        else {
                            return 3;
                        }
                    }
                }
            }
        }
    }
    else if (txt[i] == patrev[0]) {
        possmatch = txt.Substring(i, patlen);
        if (possmatch == patrev) {
            //FOUND pat REVERSE
            result = 2;
            //CHECK IF txt CONTAINS pat
            for (j = i + patlen; j <= txtlen - patlen; j++) {
                if (txt[j] == pat[0]) {
                    possmatch = txt.Substring(j, patlen);
                    if (possmatch == pat) {
                        if (j == i + patlen) {
                            return 5;
                        }
                        else {
                            return 3;
                        }
                    }
```

```
                        }
                    }
                }
            }
        } //pat NOR REVERSE FOUND
    }
    return result;
} }
//  var ct = new CUT();
//  print(ct.Subject("", ""));      //0
//  print(ct.Subject("", "word"));  //0
//  print(ct.Subject("word", ""));  //0
//  print(ct.Subject("word", "or"));   //0
//  print(ct.Subject("word", "wor"));  //1
//  print(ct.Subject("word", "ord"));  //1
//  print(ct.Subject("word", "row"));  //2
//  print(ct.Subject("word", "dro"));  //2
//  print(ct.Subject("worddrow", "dro"));  //5
//  print(ct.Subject("worddrow", "ord"));  //4
//  print(ct.Subject("wordxdrow", "dro")); //3
//  print(ct.Subject("wordydrow", "ord")); //3
```

A.7 Txt

```
void Txt(String word1, String word2, String word3){
     //CONVERT ENGLISH TEXT txt INTO MOBILE TELEPHONE TXT
     //BY SUBSTITUTING ABBREVIATIONS FOR COMMON WORDS
     String result  = "";
     if (word1 == "two") {
          result = "2";
     }
     if (word1 == "for" || word1 == "four") {
          result = "4";
     }
     if (word1 == "you") {
          result = "u";
     }
     if (word1 == "and") {
          result = "n";
     }
     if (word1 == "are") {
          result = "r";
     }
     else if (word1 == "see" && word2 == "you") {
          result = "cu";
     }
     else if (word1 == "by" && word2 == "the" && word3 == "way") {
          result = "btw";
     }
 }
```

A.8 Title

```
int  Title(String sex, String title){
//CHECK PERSONAL TITLE CONSISTENT WITH SEX
  title = title.ToLower();
  int result   = -1;
  if ("male" == sex) {
     if ("mr" == title    ||  "dr" == title ||
         "sir" == title   ||  "rev" == title ||
         "rthon" == title ||  "prof" == title) {
              result = 1;
    }
  }
  else if ("female" == sex) {
       if ("mrs" == title   || "miss" == title ||
           "ms" == title    || "dr" == title ||
           "lady" == title  || "rev" == title ||
           "rthon" == title || "prof" == title){
               result = 0;
       }
  }
  else if ("none" == sex) {
       if ("dr" == title || "rev" == title ||
           "rthon" == title || "prof" == title){
               result = 2;
       }
```

```
    }
    return result;
}
```

A.9 FloatRegEx

```
public int [,] initilize ()   {
   nextstate:int[,] = new int[8, 256];
   String  s    = ".";
   int    temp  = int(s[0]);
   for (int j  = 0 ;j < 256 ; j++){
       nextstate[0,j] = 0;
       if(j >= 48 && j<= 57){
           nextstate[1,j] = 2;
           nextstate[3,j] = 4;
           nextstate[7,j] = 7;
       }
       else{
           nextstate[1,j] = 0;
           nextstate[3,j] = 0;
           nextstate[7,j] = 0;
       }
       s =".";
       temp = int(s[0]);
       if(j == temp)
           nextstate[2,j] = 3;
       else
           nextstate[2,j] = 0;
       s = "e";
       temp = int(s[0]);
       if(j >= 48 && j<= 57)
           nextstate[4,j] = 4;
```

```
        else if( j == temp )
            nextstate[4,j] = 5;
        else
            nextstate[4,j] = 0;
        String  s1  = "+";
        temp1 = int(s1[0]);
        s = "-";
        temp = int(s[0]);
        if(j ==  temp || j == temp1 )
            nextstate[5,j] = 6;
        else if(j >= 48 && j<= 57)
            nextstate[5,j] = 7;
        else
            nextstate[5,j] = 0;
        if(j >= 49 && j<= 57)
            nextstate[6,j] = 7;
        else
            nextstate[6,j] = 0;
    }
    return nextstate;
}

  int FloatRegEx(String  input){
    int[,] nextstate  = new int[8, 256];
    nextstate = initilize();
    int  currentstate    = 1;
    int  i   =0;
    while(currentstate != 7 && i< input.Length){
```

```
        currentstate = nextstate[currentstate,int(input[i])];
        i = i+1;
    }
    if(currentstate == 7){
        //Target executed
    }
    return 0;
}
```

A.10 Polygon shape

```
void Shape(a double []) {
        int length  = a.Length;
        double sinAngle1    =  Math.sin(a[0]);
        double sinAngle2    =  Math.sin(a[1]);
        double sinAngle3    =  Math.sin(a[2]);
        double sinAngle4    =  Math.sin(a[3]);
        double val1    = Math.abs(sinAngle1);
        double val2    = Math.abs(sinAngle2);
        double val3    = Math.abs(sinAngle3);
        double val4    = Math.abs(sinAngle4);
    // The shape is Polygon not Triangle
    if(length == 8){
       if ((val1 - 1.0)<=Double.Epsilon && (val2 - 1.0)<=Double.Epsilon &&
            (val3 - 1.0)<=Double.Epsilon  && (val4 - 1.0)<=Double.Epsilon){
         if(a[4] == a[6] && a[5] == a[7] && a[4] == a[7]){
           //The figure is Square
          }
         else if(a[4] == a[6] && a[5] == a[7]){
          // The  figure is Rectangle
         }
         else{
           //The figure is neither Square nor Rectangle
         }
       }
     }
     //The figure is Triangle
```

```
  else if(length == 6){
    double  TotalAngle    = a[0] + a[1] + a[2];
     if((TotalAngle - Math.PI)<= Double.Epsilon){//Traingle figure
        if((val1 - 1.0)<=Double.Epsilon || (val2 - 1.0)<=Double.Epsilon
        || (val3 - 1.0)<=Double.Epsilon ){
          //Right triangle: Has one 90 degree angle
        }
        else if(a[0] == a[1] && a[1] == a[2] ){
         //Equilateral triangle: All angles are the same (60 degrees)
        }
        else if((a[0] == a[1] || a[1] == a[2] ||a[0] == a[2]) &&
               (a[3] == a[4] || a[4] == a[5] ||a[3] == a[5])){
        //Isosceles triangle: Has two angles the same and two sides the same
        }
        else if((a[0] != a[1] && a[1] != a[2] && a[0] != a[2]) &&
               (a[3] != a[4] && a[4] != a[5] && a[3] != a[5])) {
        //Scalene triangle:Has all three angles and all three sides different
        }
     }
     else {
          //The figure is not Triangle
     }
  }
}
```

References

A. Baresel, J. Wegener, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):41–54, 2001.

A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 108–118, 2004.

B. Beizer. *Software Testing Techniques.* van Nostrand Rheinhold, New York, 2nd edition, 1990. ISBN 0442206720.

L. B. Booker. *intelligent behavior as an adaptation to the task environment.* PhD thesis, The University of Michigan, Ann Arbor, MI, 1982.

L. Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithm. *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2002*, pages 1337–1342, 2002.

L. Bottaci. Predicate expression cost functions to guide evolutionary search for test data. In *Genetic and Evolutionary Computation Conference (GECCO 2003)*, pages 2455–2464, July 2003.

L. Bottaci. Use of branch cost functions to diversify the search for test data. In *Proceedings of the UK Software Testing Workshop (UKTest 2005)*, pages 151–163, University of Sheffield, UK, September 5-6, 2005 2005.

L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2:215–222, 1976.

R. J. Collins and D. R. Jefferson. Selection in massively parallel genetic. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International on Genetic Algorithms*, page 249256, San Mateo, CA. Morgan Kaufmann, 1991.

P. Coward. Symbolic execution and testing. *Information and Software Technique*, 33(1):229–239, 1991.

L. Davis. *Handbook of Genetic Algorithms.* International Thomson Computer Press, 1996.

E. de Jong, R. Watson, and J. Pollack. Reducing bloat and promoting diversity using multi-objective methods. In L. e. a. In Spector, editor, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18, San Francisco, CA. Morgan Kaufmann, 2001.

K. Deb and D. E. Goldberg. An investigation of niche and species formation in genetic function optimization. In *Proceedings of the Third International Conference on Genetic Algorithms*, page 4250, San Mateo, CA. Morgan Kaufmann, 1989.

K. A. DeJong. *An analysis of the behavior of a class of thesis.* PhD thesis, The University of Michigan, Ann Arbor, MI, 1975.

R. DeMillo and A. Offutt. experimental results of automatically generated adequate test sets. *in proc. 6th Ann. Pacific Northwest Software Quality Conf.*, pages 209–232, Sept. 1988.

R. DeMillo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–909, 1991.

J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–443, 1984.

D. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In *In Proceedings of the second international conference on genetic algorithms*, page 148154, Morgan Kaufmann, 1987.

D. E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning.* Addison Wesley, 1989. ISBN 0-201-15767-5.

M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. *In Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1359–1366, 2002.

M. Harman, L. Hu, and etl. Testability transformation. *IEEE Transaction on software Engineering.*, 30(1):73–81, 2004.

B. Jones, R. H. Sthame, and D. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

D. J. Kasik and H. G. George. Toward automatic generation of novice user test scripts. *Proceedings 1996 Confernce on Human factors in Computing Systems.*, pages 244–251, April 1996.

M. Keijzer. *Advances in Genetic Programming 2 chapter 13, pages 259-278.* MIT Press, MA, USA, 1996.

J. King. A new approach to program testing. *In Proceedings of the International Conference on Reliable Software*, pages 228–233, 1975.

J. King. Symbolic execution and program testing. *Communications of the ACM*, 19 (7):385–394, 1976.

B. Korel. Dynamic method for software test data generation. software testing. *Verification and Reliability*, 2(4):203–213, 1990.

B. Korel. Dynamic method for software test data generation. *Software Testing, verification and Reliablity*, 2:203–213, 1992.

B. Korel and R. Ferguson. The chaining approach for software test data generation. *IEEE Transactions on Software Engineering*, 5(1), January 1996.

B. Korel, M. Harman, and etl. Data dependence based testability transformation in automated test generation. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 245–254, 2005.

J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

W. Langdon. Data structures and genetic programming: Genetic programming + data structures = automatic programming! *Genetic Programming*, 1, 1999.

J. S. Leon. Frequency of character pairs in english language text (expected no of occurences per 463 characters). *http://tigger.uic.edu/jleon/mcs425-s05/index.html [OnLine 20/6/2005]*, 2002.

J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, University of California Press, 1967.

P. Martin. The porter stemming algorithm. *http://www.tartarus.org/ martin/index.html [on line] 07/09/2005*, 2005.

C. Mattiussi, M. Waibel, and D. Floreano. Measures of diversity for populations and distances between individuals with highly reorganizable genomes. *Evolutionary Computation*, 12(4):495–515, 2004.

P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

P. McMinn, D. Binkley, and M. Harman. Testability transformation for efficient automated test data search in the presence of nesting. In *Proceedings of the Third UK Software Testing Workshop (UKTest 2005)*, pages 165–182, University of Sheffield, UK, September 2005.

C. Michael, G. McGraw, M. Schatz, and C. Walton. *Genetic algorithms for dynamic test data generation: Technical Report RSTR-003-97-11.* RST Corporation, Suite 250, 21515 Ridgetop Circle, Sterling VA 20166, 1997.

W. Miller and Spooner. Automatic generation of floating-point test data. *IEEE Transaction on Software Engineering*, 2(3):223–226, 1976.

J. Myers, Glenford. *The art of software testing.* New York , Wiley, 1979. ISBN 0471043281.

G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, March 2001.

A. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction provedure for test data generation. *Software practice and Experience*, 29(2):167–193, 1997.

U. M. O'Reilly. Using a distance metric on genetic programs to understand genetic operators. *In IEEE International Conference on Systems, Man, and Cybernetics, Computational Cybernetics and Simulation*, 5:4092–4097, 1997.

R. Pargas, M. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

M. Pei, D. Erik, Z. Goodman, and Z. Kaixang. Automated software test data generation using a genetic algorithm. *Technical report*, 1994.

C. V. Ramamoorthy and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, December 1976.

C. R. Reeves. *Moderm Heuristic Techniques for Combinatorial Problems.* 1995.

J. Rosca. Entropy-driven adaptive representation. *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, 1995.

H. P. Schwefel. *Evolution and Optima Seeking.* Wiley, New York, 1995.

J. E. Smith and T. C. Fogarty. Evolving software test data - ga's learn self expression. *Evolutionary Computing.*, 1996.

H. Sthamer. *The Automatic Generation of Test Data Using Genetic Algorithms.* PhD thesis, University off Galmorgan, Pontypridd, Wales, 1995.

D. Stutz, T. Neward, and G. Shilling. *Shared Source CLI Essentials.* O'Reilly, first edition, March 2003. ISBN 0-596-00351-x.

N. Tracey. A search-based automated test-data generation framework for safety critical software. *PhD Thesis, University of York*, 2000.

J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, May 1995.

A. Watkins. The automatic generation of test data using genetic algorithms. *In Proceedings of the Fourth Software Quality Conference*, page 300309, 1995.

J. Wegener and etl. Systematic testing of real-time systems. *proceddings of the 4th Europen Conference on Software Testing Analysis and Review (EuroStar 1996),Amsterdam, Netherlands*, 1996.

J. Wegener and M. Grochtman. Verifying timing constraints of real-time systems by means of evoultionary testing. *Real-Time Systems*, 15:275–298, 1998.

J. Wegener, G. K., M. Grochtmann, H. Sthamer, and B. Jones. Testing temproral corrctness of real-time systems by means of genetic algorithms. *proceddings of the 10th International Software Quality Week, San Francisco, USA*, May 1997.

J. Wegner, H. Pohlheim, and H. Sthmar. Testing the temporal behaviour of real-time tasks using extended evolutionary algorithms. *proceeding Of the 7th Europen Conference on Software Testing Analysis and Review, Barcelona, Spain*, December 1999.

J. Wegner, R. Pitschinz, and H. Sthmar. Automated testing of real-time tasks. *Proceedings of the 1st Intrenational Workshop on Automated program Analysis, Testing and Verification, Limerick, Ireland*, June 2000.

D. Whitley. The genitor algorithm and selective pressure: Why rank-based allocation of reproductive trials is best. *Proceedings of the Third International Conference on Genetic Algorithms (ICGA-89)*, pages 116–121, 1989.

S. Xanthakis, C. Ellis, and C. Skourlas. Application of genetic algorithms to software testing. *In 5th International Conference on Software Engineering and its Applications*, pages 625–636, 1992.

R. Zhao. Character string predicate based automatic software test data generation. In *Third International Conference On Quality Software*, pages 255–263, 2003.