University of Plymouth

PEARL **https://pearl.plymouth.ac.uk**

04 University of Plymouth Research Theses 01 Research Theses Main Collection

1996

AN INVESTIGATION INTO ADAPTIVE SEARCH TECHNIQUES FOR THE AUTOMATIC GENERATION OF SOFTWARE TEST DATA

LACHUT WATKINS, ALISON ELIZABETH

http://hdl.handle.net/10026.1/1618

http://dx.doi.org/10.24382/1526
University of Plymouth

All content in PEARL is protected by copyright law. Author manuscripts are made available in accordance with publisher policies. Please cite only the published version using the details provided on the item record or document. In the absence of an open licence (e.g. Creative Commons), permissions for further reuse of content should be sought from the publisher or author.

AN INVESTIGATION INTO ADAPTIVE SEARCH TECHNIQUES FOR THE AUTOMATIC GENERATION OF SOFTWARE TEST DATA

by

ALISON ELIZABETH LACHUT WATKINS

A thesis submitted to the University of Plymouth in partial fulfilment for the degree of

DOCTOR OF PHILOSOPHY

School of Computing

February 1996

90 0274599 7

UNIVERSITY OF PLYMOUTH

Item No.	9002745997
Date	2 0 JUN 1996 T
Class No.	T005.14 WAT
Contl. No.	X703294546

LIBRARY SERVICES

STORE

REFERENCE ONLY

LIBRARY STORE

Abstract

The focus of this thesis is on the use of adaptive search techniques for the automatic generation of software test data. Three adaptive search techniques are used, these are genetic algorithms (GAs), Simulated Annealing and Tabu search. In addition to these, hybrid search methods have been developed and applied to the problem of test data generation. The adaptive search techniques are compared to random generation to ascertain the effectiveness of adaptive search. The results indicate that GAs and Simulated Annealing outperform random generation in all test programs. Tabu search outperformed random generation in most tests, but it lost its effectiveness as the amount of input data increased. The hybrid techniques have given mixed results. The two best methods, GAs and Simulated Annealing are then compared to random generation on a program written to optimise capital budgeting, both perform better than random generation and Simulated Annealing requires less test data than GAs. Further research highlights a need for research into the control parameters of all the adaptive search methods and attaining test data which covers border conditions.

Table of Contents

List of Figures

List of Tables

List of Appendices

ACKNOWLEDGEMENT

I wish to express my thanks and gratitude to:

my supervisors Professor Mike Denham and Professor Patricia Pearce for all their guidance and assistance;

Dr. Charlie Ellis who offered me the original encouragement and enthusiasm to pursue this research;

IPL for their provision of Cantata Software Verification Tools;

the School of Computing and the Plymouth Business School for their support and part-time lecturing;

and David for his patience, attention to detail and knowledge of commas.

AUTHOR'S DECLARATION

At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award.

Presentations and Conferences Attended:

Watkins, A.L. (1995). A tool for the automatic generation of test data using genetic algorithms. In *Proceedings of Software Quality Conference*, Dundee, Scotland.

Watkins, A.L. (1995). Genetic algorithms combined with Tabu search for the automatic generation of test data. In *Proceedings Polymodal 16*, University of Sunderland, UK.

Signed...[illegible]...

Date.........17th May.....1996........

Chapter One

Introduction to Research

1.1. Introduction

Each day both amateur and professional programmers are at work producing software to perform a variety of tasks. Some of these may be perceived as trivial while others can carry universal implications. One application for a piece of software is a payroll system. The development of this should follow a prescribed course from the requirements analysis through module unit testing to maintenance, as illustrated in figure 1. On purchase of the software the expectation of the user is that it will work, that is it will make accurate payments by performing calculations and taking appropriate deductions. This would seem an important detail to anyone who has ever been paid, as an incorrect paycheque can cause great concern to the employer and employee. Because of the level of accuracy required it is therefore of great importance that the developers of

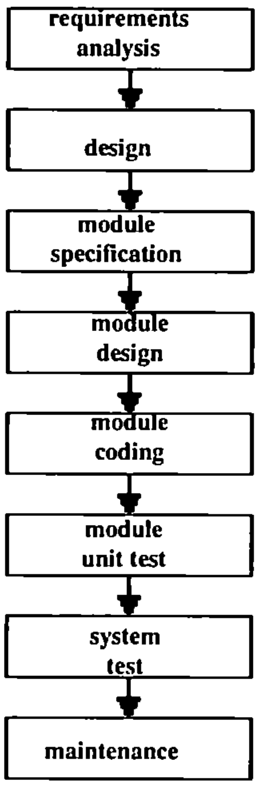


Figure 1 - Method for Developing and Maintaining a System Lifecycle

payroll systems can assure their potential customers that their software will not only work, but work correctly.

If the program contains errors, losses to the user could be significant and the implications to the reputation of the developer could be severe. One such actual error of significant financial ramifications was the NASA probe sent to Venus, which veered off course due to an erroneous FORTRAN repetition statement. This was interpreted as an assignment statement because in this language variables did not have to be declared, had they been, the assignment statement would have raised an error at the compilation stage (Bell *et al,* 1987).

Unfortunately even if a program is thoroughly tested it is no guarantee that it is without error. Bell *at al* (1987) describes a carefully controlled experiment which was carried out in 1978 with fifty-nine people all of whom worked in the computer industry with an average of eleven years experience. They were asked to test a sixty-three line PL/1 program until they thought they had found all the errors (if any). The mean number of errors found was 5.7, the most 9, the least 3. The actual number of errors was 15 and of those, there were four which no one found. A further review of the results indicated that people spent more time testing the normal conditions instead of looking at special cases and invalid input situations.

These examples raise an important issue for software developers, what kind of assurance can be given that a piece of developed software will work accurately? Although most software developers would never admit to errors in their code, errors are a fact of life in software development. Elimination of these errors is very important, and the software testing process, and more specifically the generation of test data for testing, is the main focus of this thesis.

1.2. What is Testing?

According to Myers (1979) the definition of testing causes many of the problems. He suggests some incorrect definitions of testing include "Testing is the process of demonstrating errors do not exist", "The purpose of testing is to show that a program performs its intended functions correctly", and "Testing is the process of establishing confidence that a program does what it is supposed to do." A problem with these definitions is that it is impossible to prove that a piece of software has no errors even if a program performs the task for which it was designed. Myers (1979) feels that the reverse of these definitions is what testing should be, and his definition is:

"Testing is the process of executing a program with the intent of finding errors."

For Myers (1979), testing should be a destructive process of trying to find errors in a program. A successful test case is one that causes a program to fail and the eventual goal of delivering a degree of confidence in the program can only be done by a thorough exploration of errors.

The economies of testing limit how much testing will take place. Exhaustive testing which looks at all possible combinations of test data, both valid inputs and possible inputs, is extremely costly and the amount of test data required borders on infinity (Myers, 1979). Therefore the objective is to minimise the testing investment while maximising the number of errors found by a finite test set (Myers, 1979), but maximising errors resolved and minimising investment on a software product can be mutually exclusive goals. However as the total quality of a deliverable software product is a key issue to software developers and their reputation, a great deal of time, effort and research has to be placed into designing effective testing techniques. There exist numerous developed testing techniques, several of which are discussed in chapter 2.

1.3. Objectives of Research

Many developed testing techniques require test data and the generation of that test data is the main focus of this thesis. The research develops work carried out by Xanthakis *et al* (1992) in the use of genetic algorithms to generate test data for program testing. Their research reports that genetic algorithms have the potential to outperform random test data generation. Similar work has been done independently at the University of Glamorgan and has been reported in Sthammer *et al* (1994). The first objective of this research was to develop a tool which could automatically prepare a function (module of a larger program) for testing by an analysis of the test code, a generation of the flow graph which helps to determine the paths through the code, and finally to measure the coverage level of a function accomplished by running the program with test data.

The goal of this research was to determine the effectiveness of adaptive search techniques for test data generation in comparison to random test data generation. An adaptive search technique developed to work well in most search circumstances may not perform as well when the search space is not normally structured such as with regular peaks and valleys. Some techniques may improve performance and by combining methodologies in a hybrid form even more satisfactory results could be gained. Therefore, presented in this thesis are some new forms of the standard adaptive search techniques which are compared to their original form.

A final goal of this study was to lay a course for the examination of adaptive search techniques and their use for software testing. As a great deal of test data can take a long time both to generate and run through a function under test, it would be advantageous to reduce the time while increasing the test effectiveness. While as Myers (1979) says it can not be proven that no error exists, to find most existing errors in a shorter time period would be of benefit to the software development lifecycle.

1.4. Research Plan

As stated, the goal of this research was to investigate adaptive search techniques for software test data generation. In order to achieve this goal the following research plan was used:

1. An ongoing literature search encompassing all forms of software testing (static and dynamic) was conducted to ascertain methods used and progress on strategies. This confirmed Xanthakis *et al* (1992) idea that using genetic algorithms for test data generation was a viable yet unexplored method for dynamic testing;

2. Research was conducted on the possibility of automating the process of testing from choosing the function to be tested, establishing paths through the function,

the generation of test data, and finally to return anomalies to the user. This resulted in the development of a tool which will perform these steps;

3. This tool was then used for the generation of test data using genetic algorithms. Results gathered using genetic algorithms were favourable when compared to those of random test generation. Research into other adaptive search techniques was performed (Simulated Annealing and Tabu Search), their structure and design is described and the results for these techniques given;

4. Results indicated that adaptive search techniques when combined to form hybrid methods perform better in some circumstances than their predecessors GAs, Simulated Annealing and Tabu search.

This research has developed a tool for the generation of test data and evaluated the use of a number of adaptive search techniques in this role.

1.5. Summary of Chapters

As the purpose of this chapter is to introduce the subject matter, present the objectives of the research and a research plan, what follows is a summary of subsequent chapters in the thesis.

Chapter 2 - Software Testing

This chapter introduces software testing and divides testing into two categories, static and dynamic testing. When further broken down these techniques can be divided into structural or functional methods. A final section of the chapter introduces test effectiveness ratios and how these are used to measure the coverage level of a test.

This chapter builds upon techniques already available in the literature and does not claim any original contribution to the knowledge of the area.

Chapter 3 - A Tool for the Generation of Test Data

The purpose of this chapter is to introduce a tool for software testing, which includes a breakdown of the method used and the data required of a function, in order to produce a flow graph which is a map of all paths through a piece of code. This information is used to distinguish the information on the function, such as branches and linear code sequences and jumps (LCSAJs), which are segments of code from one point, a start or decision point, through to the next decision point in the code. The test data will then attempt to exercise these branches and LCSAJs.

Chapter 4 - Random Testing

This chapter is an introduction on random testing, the most common form of test data generation. The purpose of this chapter is to illustrate how the comparison between random generation and adaptive search techniques will occur.

Chapter 5 - Genetic Algorithms - A Brief Introduction

For the benefit of readers not familiar with adaptive search, such as those involved in software testing, each test technique will begin with an introduction of the technique, a short demonstration in a search environment, before being applied to the test data generation. A chapter has been devoted to an explanation of the many methods and the theory incorporated in Genetic Algorithms. While this chapter does not profess to contribute to the knowledge of Genetic Algorithms it is hoped the reader will get a focused view of Genetic Algorithms with the appropriate references for future research.

Chapter 6 - Automatic Test Data Generation Using Genetic Algorithms

This chapter uses genetic algorithms to generate test data. Two types of testing are attempted, the first looks at the LCSAJs and branches through a piece of code, while the second exercises the paths though a function under test. This chapter represents an original contribution of knowledge to the field of software testing using Genetic Algorithms. While this method of test data generation has been applied before, (Xanthakis, 1992; Sthamer, 1994) the method of data collection, search space recording and the approach necessary to achieve 'black-box' testing are all new.

Chapter 7 - Simulated Annealing

Simulated Annealing has been applied to the task of test data generation using the same sample function under test as used in chapter 6. A comparison has been performed of the acceptance probability rate to determine the most efficient settings. A literature review has not revealed any previous use of Simulated Annealing for the generation of test data, and the use of Simulated Annealing to test data generation constitutes an original contribution to knowledge. Finally, a hybrid GAs-SA is used for the same test program.

Chapter 8 - Tabu Search

This chapter includes a discussion of two related search techniques. First to be introduced is hill-climbing followed by Tabu search which builds on the simple methods involved in hill-climbing. Using Tabu Search for the automatic generation of test data is an original contribution to the field of test data generation. The results of hill-climbing are not as promising as other search techniques, and even random testing produces better results. While the results of hill-climbing were poor, Tabu search performed almost as well as the other adaptive search methods. A hybrid GAs-TS was developed to see if it would improve the results and the philosophy of a Tabu list was used in conjunction with Simulated Annealing. These techniques are an original contribution to knowledge in the field of adaptive search techniques.

Chapter 9 - The Results of Adaptive Techniques on a Suite of Test Functions

After demonstrations of the techniques in the previous chapter a more complete suite of test functions was used. This will determine whether adaptive search techniques can outperform the results of random testing. A final aspect is the introduction of time measurements to the testing procedure. Is time a factor? As faster computers are developed it will be interesting to compare these results on a time basis to determine whether the extra effort involved in designing adaptive techniques is worth the savings in run time. This comparison of techniques is an original contribution to knowledge.

Chapter 10 - A Demonstration of Automatic Test Data Generation on a Program which Optimises Capital Allowances for Company Taxation

The final demonstration in this thesis uses the adaptive search techniques described here to derive test data for a program written to optimise the system of Capital Allowances used in company taxation. This program was written with the intention to optimise the calculations using Genetic Algorithms and Simulated Annealing, but it is good programming practice to test the functions prior to optimisation. As both the size of the search space and the number of branches, LCSAJs and paths through the code is large, this problem is ideal for comparing the test data generation techniques in a 'real-life' environment and is an original contribution to knowledge in the field of adaptive search techniques.

Chapter 11 - Discussion and Future Research

The aim of this chapter is to review the results received and contains a discussion of how robust these methods might be compared to random test data generation. Included is a review of the strengths and weakness of this research and a consideration of future research which needs to be performed.

Chapter 12 - Conclusion

The concluding chapter ties together the ideas presented throughout the thesis, and offers some thoughts on practical implementation.

1.6. Conclusion

This chapter offers an introduction to the subject of this thesis and discusses why the area of software testing and testing tools is important to the software development community. A discussion of the objectives is given along with the research plan that was followed in the comparison of test data generation techniques. In conclusion is a breakdown of each chapter which aims to point out the contribution of knowledge contained within.

Chapter Two

What is Software Testing?

2.1. Introduction

The development of the very first piece of computer software was probably followed by a complaint that it did not work as expected. Total Quality Management and Zero-Defects are important benchmarks and should be applied to computer software and the software development process. Therefore it is necessary to devise methods to test software modules quickly, efficiently and completely, prior to the delivery date. A critical system relies on the software that drives it and the users expect it to contain no errors. Errors are classified by Goodenough and Gerhart, (1975) into two groups, performance or logic. Performance errors are a failure to produce results within a specified or desired time and space limitation, whereas logic errors deliver incorrect results regardless of the time and space required. Erroneous implementation is the most common fault in software according to Goodenough and Gerhart, (1975). Some of these errors are summarised as follows:

- Missing Control Flow Paths: The cause, a failure to test conditions, and the result, an incorrect execution (or non-execution). Failure to test for a zero divisor before a division in Fortran is an example. A program with this type of error will be able to execute all control flow paths without detection.
- Inappropriate Path Selection: An incorrect condition may cause an action to perform or not perform. If the code is written **IF X**, instead of **IF X and Y** an action can occur when X is true and Y is

false. It is possible to execute all branches and statements and not detect this type of error.

- Inappropriate or Missing Action: This results from errors such as an incorrect calculation (W*W instead of W+W), failing to assign a value to a variable or calling a function with a wrong argument list. If all statements are executed most of these errors will be caught, but if errors only exist under certain condition they may not be found.

Goodenough and Gerhart (1975) were the first to establish a sound theoretical basis for testing and their 'Fundamental Theorem of Testing' has been used as a model for formalising testing concepts (White, 1987). Their theorem states that there always exists a finite test set that reliably determines the correctness of a given program over its entire input domain. They further define a 'test selection criterion' which specifies conditions that must be satisfied by a finite test set. For example, if a program specifies that all input variables be integers, it might be specified that all test data sets contain both a positive and negative integer and a zero (White,. 1987). Therefore, two potential test sets can be {-5,0,12} and {-1,0,8}.

It is known therefore that there exists a test data set, but building such a finite set is 'undecidable' as proved by Howden (1976). A problem is described as undecidable or unsolvable if no algorithm can find a solution (White, 1987). The 'halting problem' was influential in defining what is undecidable. It asked whether any given Turing machine will halt given an arbitrary input. To prove the undecidability of a problem, it must be demonstrated that the decidability of the given problem implies the decidability of the halting problem, which would be a contradiction. Undecidability concludes that no computing machine can be designed which given an arbitrary program and input, will always terminate. Of course this problem can be avoided by a study of code loops to determine if they terminate under all conditions (White, 1987), and this will be discussed in later sections.

To further illustrate the issue of decidability in the structure of a computer program, the digraph in figure 2 will be used. This digraph consists of an entry node with no incoming nodes and a terminal node with no arcs leaving it and should consist of a sequence of arcs from the entry node to any specified node through to the terminal node. This, referred to as the directed path, should exist for every node (White, 1987).

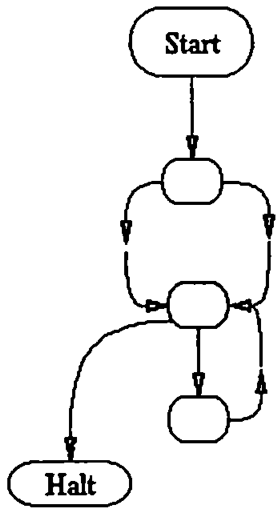


Figure 2 - Sample Directed Graph (Digraph)

A directed path from the entry node to the terminal node on the graph is called a control path. Paths which traverse the same loop for different lengths of time are specified as distinct control paths. The result is that the number of control paths can be infinite. However, not all control paths are executable. If there is input data to satisfy a path condition then the path is executable. If no test data exists then the path is considered infeasible, and not available for testing (White, 1987). Therefore to determine in advance which paths are executable and which are infeasible is undecidable as a test data set can not be built which will satisfy all paths.

What is possible is to search for test data which will execute the feasible paths of a program and while there can be no specific algorithm developed for this purpose, many techniques have been developed using heuristic or ad hoc methodologies.

2.2. Software Testing

There are a number of strategies for designing testing processes which can be combined in a 'pick'n mix' fashion to develop a methodology. Very broadly these strategies can be distinguished as static, dynamic, functional or structural. Coward (1988) offers this distinction between the strategies:

> 'A functional strategy uses only the requirements defined in the specification as the basis for testing, whereas a structural strategy is based on the detailed design. A dynamic approach executes the software and assesses the performance, while a static approach analyses the software without recourse to its execution'.

A more detailed definition is given in the following subsections.

2.2.1. Static Analysis

In static analysis a program is not executed, the review is performed on the requirements analysis and design documents. This is a manual or automatic process which searches for errors in syntax or structural properties. Types of errors discovered can be in language syntax, misspellings, punctuation, line sequencing or specification elements (Andriole, 1986). The general form of a static analysis tool is shown in figure 3. These techniques can be applied to all stages of product development from the requirements statement to the user manual. Manual inspection which can consist of desk checking, inspection and walkthroughs have advantages over automatic techniques in that more than one perspective can be addressed while the program is being examined, such as a review of both high level and detailed properties as well as allowing the analyst to apply various heuristics or subjective judgements. Unfortunately this sort of inspection can be dull and time consuming and as the size of a piece of software grows the inclination to compromise quality increases (Andriole, 1986).

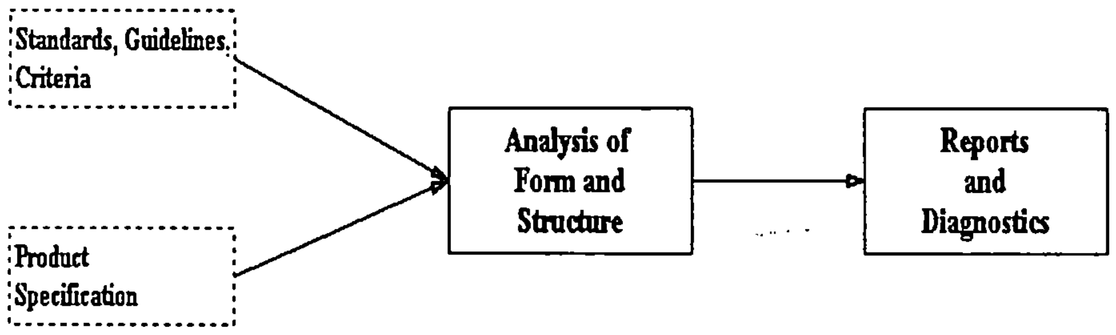

Standards, Guidelines, Criteria
Product Specification
Analysis of Form and Structure
Reports and Diagnostics

Figure 3 - General Form of Static Analysis

Automated static analysis tools operate on both the source code and the requirements and design specifications. There are two kinds of automated tool, the first gathers and reports information about a program but does not usually search for a particular type of error. The second tool detects specific classes of error or anomalies in a program. Examples include parsers which determine the adherence of a program to the language syntax; analysis techniques to test consistency of parameter interfaces; consistency checking of variables to their declaration; and reviewing code for incorrect sequencing such as trying to read from a file before it is opened (Andriole, 1986).

Static testing techniques include symbolic execution, program proving and anomaly analysis.

2.2.2. Dynamic Analysis

Dynamic analysis requires the software to be executed. While it is possible through static analysis to determine if a directed path exists between two nodes of the control flow graph, it is not possible to tell if this path is executable. Analysis routines inserted into a program will record the paths executed, thus keeping a record of exercised portions of the code (Korel, 1990). The record lists execution of program statements, branches or code jumps as well as identifying particular areas of code which may be unreachable. Figure 4 is the general form of this analysis tool which illustrates how the functional analysis of the code through path selection and testing algorithm joins with the specification to compare and analyse the program.

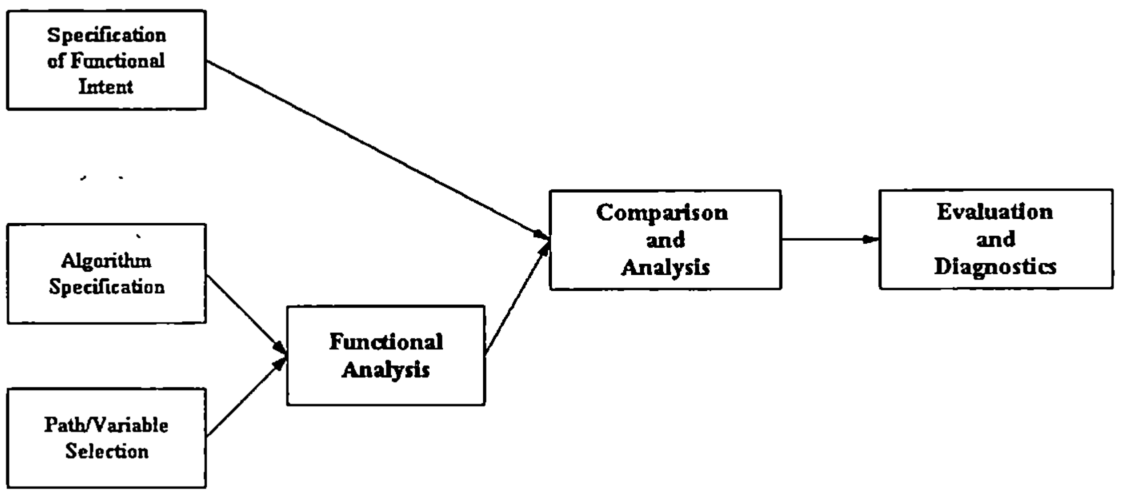


Figure 4 - General Form of Dynamic Analysis Testing Tool

Dynamic analysis can act as a link between functional and structural testing. A set of test cases are generated through functional testing, the execution of which may be monitored by dynamic analysis. The program can then be examined structurally to determine test cases for areas which may not have been exercised. This results in the knowledge that the whole program is being tested and aims to reduce unexpected or redundant code.

Dynamic testing techniques include random testing, domain testing, cause-effect graphing, adaptive perturbation testing, computation testing, domain testing, mutation analysis and automatic path-based test data generation.

2.2.3. Functional Testing

Selection methods which attempt to derive test data to confirm that a function, as determined by the specification, is correctly implemented, are known as functional testing methods. Most are black-box methods, as there is no concern for the structure of the program in contrast to white-box testing. White (1987) states that this method has two problems, firstly a program can contain functions which were not in the original specification which needed to be tested, and secondly there are no formally identified methods for performing these tests, therefore the results of testing can only be of limited use.

Howden (1980;1981a; 1981b;1985), however, has developed an underlying theory of functional testing which attempts to overcome the problems described above. Howden feels that in addition to testing the functions specified in the requirements, an attempt should be made to mimic the development process of the program by testing the simple functions and routines before the more complex procedures. His theory consists of two elements, functional synthesis and testability. He suggests that these four types of synthesis should be tested in addition to those functions defined in the specification and requirements.

- Algebraic synthesis - Algebraic expressions in variable reference either numerical or Boolean values;
- Conditional synthesis - **IF-THEN-ELSE** constructs, built from algebraic expressions;

- Iterative synthesis - Loop iteration such as FOR-NEXT or WHILE, the predicate determines termination and the body of the loop makes an additional function;

- Control synthesis - State transitions models are used to give descriptions of states which can not be quantified such as "improper input data" or "termination".

To test a program can be quite simple, but to be testable a test must be able to control its input and observe its output. Unknown input produces untraceable output. The following is a definition of testability from a US software engineering standard:

> Testability. (1) The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. (2) The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met.
>
> IEEE 610.12

One definition of the testability of a program is its sensitivity to faults when inputs are chosen at random from a specific set of input variables. Its sensitivity to faults indicates testability, therefore an exceptionally sensitive program will be exceptionally testable.

According to Howden (1980), a functional test is one which reviews both the functional synthesis of the programmer and is testable. The functions to be tested should be determined from the following sources:

- specifications - if the specification is casual or unofficial then the verbs used will indicate functions to test, formal specifications will indicate the functions in assertions, tables or formulas;

- programs - from elementary program statements, subroutines and subpaths;

- design information and documents - useful to determine functions and to map design functions to specific aspects of code.

Identified functions should be reviewed for a variable range which covers the entire scope of possible input and output, including both extreme values just outside the permitted range and those just inside the range. Additional illegal values need to be applied to review error-catching procedures, as well as tests on arrays or vectors.

Of the dynamic testing techniques listed in section 2.2.2, functional techniques include random testing, domain testing, cause-effect graphing and adaptive perturbation testing.

2.2.4. Structural Testing

A structural test executes the program under test to attain a coverage level of the code, and this coverage level consists of various tests of code which will suggest reliability. Coverage tests can check whether all statements or branches in a program are exercised, or that all linear code sequence and jumps (LCSAJs) have been executed at least once. An LCSAJ is a segment of code beginning at a decision or loop and concluding when a transfer of control has been made.

In an ideal world the best case testing scenario would be an exhaustive search with all possible paths through the code tested. Problems arise as the number of paths increase. All combinations and conditions must be considered and this number increases if iterations are not constant but rely on input data. This results in combinatorial explosion, therefore limits are required for loops to restrict the size of the search space.

A second obstacle to exhaustive testing is infeasible paths, as it is impossible to ascertain in advance how many paths can be exercised. It is simple, under certain restrictions, to determine the number of paths through a program using the following notations for the nodes of the digraph (Paige and Holthouse, 1977):

•	sequence operator
+	selection operator
*	iteration operator

Loops are restricted to activating zero and once. To illustrate this procedure, the program shown in figure 5, and its respective flow chart in figure 6, will be used.

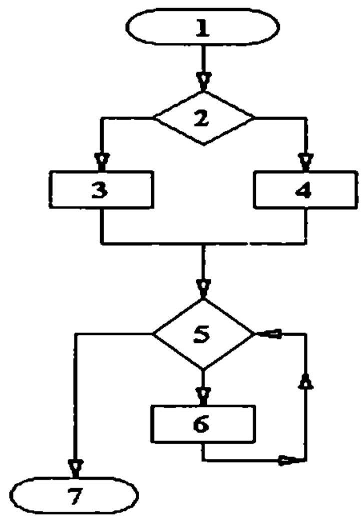


Figure 6 - Flow Chart for Figure 5

```
1.    read_in(int a, int b)
2.    if (a≥b) then
3.      print a
4.    else print b
5.    while a ≤ b
6.      a = a+1
7.    end
```

Figure 5 - Sample Code

The flowchart in figure 6 can be rewritten as the following,

$$1 \cdot 2 \cdot (3+4) \cdot 5 \cdot (6 \cdot 5)^* \cdot 7,$$

where each number represents the statement number in the code. The '+' indicates an option, as in choosing either statement 3 'print a' or statement 4 'else print b', these statements are mutually exclusive.

The $(x)^*$, which represents the loop, statements 5 and 6 in figure 5, is replaced with $(x+\varnothing)$. This new expression will accommodate the loop to be exercised twice. The first time it will make sure the loop is not activated, that is 'a>b', and the second time it is activated at least once, that is 'a≤b'. The new expression is as follows,

$$1\cdot 2\cdot (3+4)\cdot 5\cdot ((6\cdot 5)+\emptyset)\cdot 7.$$

The final stage is to replace all the statement labels in the above expression with a 1, and then sum the equation, as shown below,

$$1\cdot 1\cdot (1+1)\cdot 1\cdot ((1\cdot 1)+1)\cdot 1 = 4.$$

The resulting value is the total number of paths through the code and these four paths are listed in table 1.

Paths
1) 1-2-4-5-6-5-7
2) 1-2-3-5-6-5-7
3) 1-2-3-5-7
4) 1-2-4-5-7

Table 1 - Potential Paths (Feasible And Infeasible) through Code

Unfortunately one of these paths is infeasible, path 4 (if a≥b is false in line 2-4 then a<b has to be true in line 5-6). While it is easy to count the number of potential paths through a piece of code it is impossible to determine in advance which paths are feasible.

Testing methods which do not give as much coverage as exhaustive testing are available, with the advantage of speed. Of the static testing techniques listed in section 2.2.1, structural methods include symbolic execution, program proving and anomaly analysis, and of the dynamic testing techniques listed in section 2.2.2, structural methods include computation testing, domain testing, mutation analysis and automatic path-based test data generation.

2.3. Testing Techniques

2.3.1. Introduction

A complete breakdown of the techniques available for testing and their respective categories is given in table 2. The individual techniques are then discussed in the following sections.

	Structural	**Functional**
Static	Symbolic Execution Program Proving Anomaly Analysis	
Dynamic	Computation Testing Domain Testing Mutation Analysis Automatic Path-Based Test Data Generation	Random Testing Domain Testing Cause-effect Graphing Adaptive Perturbation Testing

Table 2 - Breakdown of Functional and Structural Techniques as Static or Dynamic Testing

2.3.2. Symbolic Execution

In symbolic execution, each input variable is replaced by symbolic values and the output is displayed using these symbols (Clarke, 1976). This output is then examined to determine whether the function tested has been used. In the example in figure 7, the price of the product decreases as the quantity purchased increases, provided a certain margin of (cost-price) is met. While this program looks complicated, Price, Quantity and TotalCost can be assigned the algebraic value of P, Q and TC. If the execution of line one is the goal, the expected output is shown as a relationship between the input variables so that the symbolic values P,Q and TC become TC/0.90Q, TC/0.90P and 0.90PQ respectively.

```
        CalculateCost(Quantity, Price,Cost)
1       if (Quantity ≤ 1 and Price-Cost  > 1.50)
1.1         TotalCost = (Price- (Price *0.10))*Quantity
1.2     else if (Quantity > 1 and Quantity < 4 and cost-price > 1.00)
1.3         TotalCost = (Price - (Price*0.15)) * Quantity
1.4     else if (Quantity ≥ 4 and cost-price > 0.50)
1.5       TotalCost = (Price - (Price*0.20))*Quantity
1.6     else
1.7        TotalCost = Quantity*Price
2       end
```

Figure 7 - Sample Code to Demonstrate Static Analysis

When there are a number of paths through the code as there are in figure 7, choices must be made as to which path is to be tested. A selected control path will use its path conditions as constraints expressed in terms of the symbolic input variables. Path feasibility is determined if there exists an input and an output point to the code. Symbolic execution can illustrate problems effectively as long as the expressions are not too complex. As with all methodologies there is a difficulty in the handling of loops or iterations but the accepted method is for three executions, once when there is no execution of the loop, again for a single execution of the loop, and finally for two executions of the loop. It has been determined that symbolic evaluation can assist in the testing of branches, statements or paths in a function. Clarke and Richardson (1981) describe three types of symbolic evaluators; symbolic execution which is a path analysis technique; dynamic symbolic evaluation which relies on data to represent the program; and global symbolic evaluation, a method of representing a program symbolically.

2.3.3. Program Proving

Program proving also involves an examination of the source code without input data. Floyd (1967) developed the most widely reported method called "inductive assertion verification". This method involves placing assertions at the beginning and end of a selected procedure, each describing the function of the procedure mathematically. A correct procedure will demonstrate that the output procedure is true given a true input procedure. Program proving attempts to provide a proof that accounts for every possible iteration of a loop. If the output is incorrect it must be assumed that errors have occurred in either the program or the proof, and these must be rechecked.

DeMillo *et al* (1980) argue that procedures can not be thought of as 'correct' but as 'acceptable'. If a program was found to be 'acceptable' after being checked by a large user group then there is confidence in the program, the larger the group the greater the confidence.

2.3.4. Anomaly Analysis

Anomaly analysis begins with a review of the programming language syntax before the code is searched for errors possible within the language. Anomaly analysis systems such as Dave (1976), Faces (1974) and Toolpack (Osterweil, 1983) determine the following irregularities:

- unexecutable or island code
- exceeded array boundaries
- uninitialised variables
- redundant variables
- incorrect loop conclusion

Anomaly analysis is performed by producing a flow graph, a scan of this graph will indicate any anomalies, such as no route leaving a particular node or a loop which is never accessed. It will not however, show infeasible paths, as this is not part of its structure.

Data flow analysis is one type of anomaly analysis. The flow of data from input to output is analysed and should indicate misspellings, confusion in variable names, or incorrect parameter passing. Although it is possible for these errors to exist in an otherwise 'correct' program it is good programming practice to ensure compliance with the functional requirements.

2.3.5. Mutation Analysis

Mutation testing is an error based testing technique (Demillo *et al*, 1980; Budd, 1981; Howden, 1982; Geist, 1992; Holmes *et al*, 1993). Errors are added to copies of the code by a mutation of the original code with the intent of exposing the errors through testing. Given a program P which runs successfully under a data set, all mutated versions of P, the incorrect programs, should fail on at least one aspect of the test case. If a test program which has failed is very similar in structure and design to the original then either the program under test is error-free, or the error has not been detected in the mutation process. In either case the test data set is very good.

The mutation of the sample code from figure 5 is shown in figure 8. In the correct program, P is to be tested by some data D, a set of programs which vary from P by a single error chosen from a list of potential errors called M(P).

Original Code P	Mutated Code M(P),
1. read_in(int a, int b)	1. read_in(int a, int b)
2. if A ≥B then	2. if A > B then
3. do A	3. do A
4. else do B	4. else do B
5. while a ≤ b	5. while a ≤ b
6. a = a+1	6. a = a+1
7. end	7. end

Figure 8 - Mutation of Sample Code from Figure 5

```
M(P)_1
read_in(int a, int b)
if A ≥B then
    do A
else do B
while a ≤ b
    a = a+1
return
```

Figure 9 - Functionally Equivalent Program to Original Code

Some of the mutant programs in M(P) will be functionally equivalent, such as shown in figure 9. The error which replaces the **end** statement with a **return**, has no effect on the logic of the subroutine and would be an equivalent mutant E(P) (DeMillo *et al*, 1987).

The programs in M(P) are executed on the data sets, D. The results which are different from the results from P on the same data, D, become the set of mutants DM(P,D), the ones with the same results form E(P). A mutation score is the fraction of the number of non-equivalent mutants of P which are determined by using the test set D. If m,e, and dm are the number of elements in M(P), E(P) and DM(P) respectively then the mutation score of D and P is defined by DeMillo *et al* (1987) as follows,

$$\mathbf{ms(P,D) = dm(m\text{-}e).}$$

A high score indicates that D is very close to being adequate for P relative to the set of mutants for P. A low score illustrates a weakness in the test set D so that P is not distinguishable from P', the flawed program (DeMillo *et al*, 1987). Once a method for establishing M(P) has been developed the test data results can be calculated automatically, which substantially reduces the time required for this testing process.

According to Budd (1981), the most common criticism is that mutation analysis requires a large number of alternative programs generated. In addition to this problem, some programs and their mutants may be 'recursively unsolvable', that is their mutation is difficult to detect and therefore are considered equivalent to the original code. Fosdick and Osterweil (1976) argue however that 70% of the equivalent mutants can be determined through basic automatic processes and that most of the remaining 30% can be eliminated through human detection. It is felt that less than 3% require a deep understanding of the program.

2.3.6. Random Testing

Random testing is the random selection of some subsets of all possible input values. Myers (1979) argues that this is probably the poorest methodology for test case design, although test results do indicate that random testing can be cost-effective for many programs including real-time software (Ince, 1987). Xanthakis *et al* (1992) claim that random testing should be considered as a minimum achievable by a technique for the automatic generation of test data, and as such may be taken as a baseline against which to compare and evaluate the efficacy of alternative techniques.

In random testing a program is executed for a subset of test data and errors are detected by the failure of expected behaviour. Although only a subset, the amount of test data required to execute a segment of code can be quite large, especially if very specific conditional statements exist in the program. If in the sample code from figure 5, the variables a and b were restricted to the range [0..20] and coverage of the path {1-2-3-5-6-5-7} was required, the number of variable combinations out of a total of 441 which satisfy this path would be 21 or 5% of the entire search space. As the variable range increases the combinations which satisfy this path may reduce proportionally.

One problem associated with random testing is the amount of human input needed to examine the test outcomes to determine if they are the expected outcome. Fortunately, not all outcomes will need to be examined, those which finish due to runtime errors will be self explanatory. One benefit of random testing, is that to perform this sort of testing one only needs a random number generator and a means of monitoring the structural program under test (Ince, 1987).

2.3.7. Computational and Domain Testing

Computational and domain testing are additional strategies for selecting test cases, both of which use the structure of the program to select paths. When an error in the flow of the program occurs it is a domain error, whereas when the test data follows the correct path but an assignment error causes the flow of control to go to an incorrect statement it is a computational error (White and Cohen, 1980). Domain testing, as discussed by White and Cohen (1980) and White *et al* (1981), illustrates that it is possible to construct test data for a set of programs which will detect a specific type of error, and as a by-product, uncover computational errors. Test data is selected on the basis of whether it is on or near the boundaries of each path domain, as it is believed that those points close to the boundary but still satisfy the condition are most sensitive to domain errors (Coward, 1988). Clarke *et al* (1982) argue however that large domain errors may remain undetected by the White and Cohen method and recommend additional strategies, V x V and N x N, which require more test points on the boundary points.

The limitations which exist for all testing strategies affect domain and computational testing. The first, coincidental correctness, can occur when a specific test point follows an incorrect path, but the output variables match those that would result from following a correct path. The second limitation that can occur is its inability to indicate that a path is missing, as no path oriented strategy can perform this remarkable feat.

2.3.8. Partition Analysis

Partition analysis, related to domain testing, looks at both the specification and the code. Its purpose is to reveal computational and domain errors. The first step is to use the specification, figure 10, to perform a symbolic evaluation, figure 11, where D represents the domains, C the associated computations and S the specification, and then perform another symbolic evaluation of the code, P, as in figure 12 (Roper, 1994). The program code used to demonstrate this process was given in figure 7. From each of these symbolic evaluations a domain graph is created, figure 13 and figure 14, to illustrate the variable constraints.

1. The customer is allowed discounts on purchases if the product has a certain margin (price-cost).
2. If purchase 1 or less item margin must be 1.50, discount is 10%
3. If purchasing 2 or 3 items margin must be 1.00 and discount is 15%
4. If 4 or greater items margin must be 0.50 and discount is 20%
5. If none of the discounts apply the full price is charged

Figure 10 - Program Specification

$D[S_1]$	Quantity ≤ 1 and margin > 1.50
$C[S_1]$	TotalCost = (Price-(Price*10%)) * Quantity
$D[S_2]$	Quantity ≤1 and margin < 1.50
$C[S_2]$	TotalCost = (Price * Quantity)
$D[S_3]$	(Quantity >1) and (Quantity < 4) and margin > 1.00
$C[S_3]$	TotalCost = (Price - (Price*15%)) * Quantity
$D[S_4]$	(Quantity >1) and (Quantity < 4) and margin < 1.00
$C[S_4]$	TotalCost = (Price* Quantity)
$D[S_5]$	Quantity > 4 and margin > 0.50
$C[S_5]$	TotalCost = (Price - (Price*20%)) * Quantity
$D[S_6]$	Quantity > 4 and margin < 0.50
$C[S_6]$	TotalCost = Price * Quantity

Figure 11 - Symbolic Evaluation of the Specification

$D[P_1]$	Quantity ≤ 1 and (Cost-Price) > 1.50
$C[P_1]$	TotalCost = (Price - (Price*10%)) * Quantity
$D[P_2]$	≠ D[P1] and Quantity > 1 and Quantity < 4 and (cost-price) > 1.50
$C[P_2]$	TotalCost = (Price - (Price*15%)) * Quantity
$D[P_3]$	≠ D[P1] and ≠D[P2] and Quantity > 3 and (Cost-Price) > 0.50
$C[P_3]$	TotalCost = (Price - (Price*20%)) * Quantity
$D[P_4]$	≠ D[P1] and ≠D[P2] and ≠D[P3]
$C[P_4]$	TotalCost = Price * Quantity
$D[P_5]$	output TotalCost to customer

Figure 12 - Symbolic Evaluation of Program Code

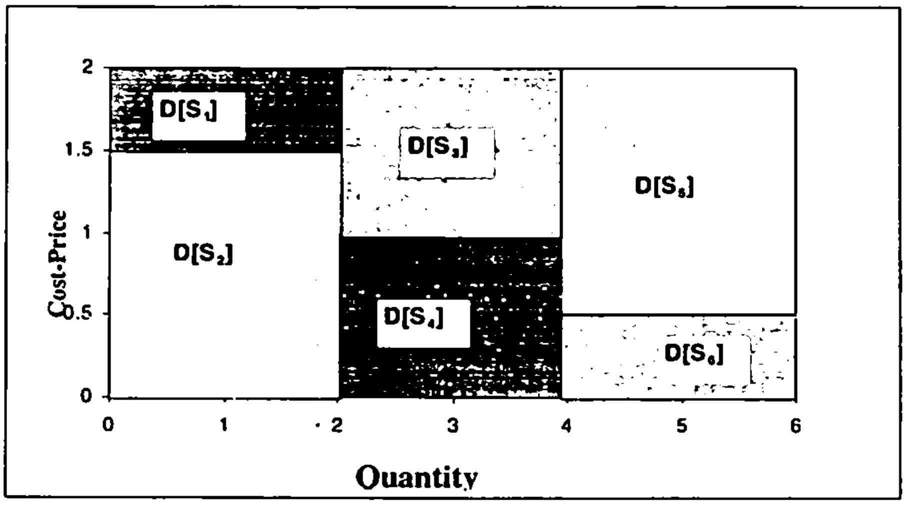


Figure 13 - Domains Created by Symbolic Evaluation of Cost Program Specification

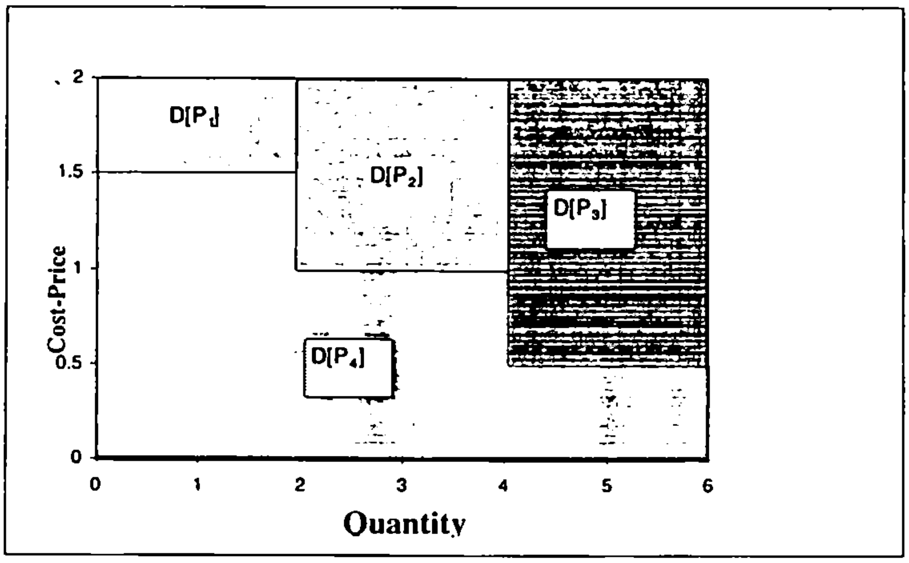


Figure 14 - Domains Created by Symbolic Evaluation of Cost Program Implementation

The two graphs are then matched to determine if the specification and implemented program agree. The implementation may trap some out of range data which is not defined in the specification. This matching is shown in figure 15, the numbers within the brackets such as [23] indicate the corresponding line number from the symbolic evaluation in the program code [2] and the specification [3].

Finally test data is established from the domain boundaries as in figure 16. These are used to test the program. As will be discussed in chapter 4, Duran (1982;1984) used the method in a comparison with random testing, but found no added benefit.

$D[_{11}]$ Quantity ≤ 1 and (Cost-Price) > 1.50
$C[_{11}]$ TotalCost = (Price - (Price*10%)) * Quantity
$D[_{23}]$ ≠ D[P1] and Quantity > 1 and Quantity < 4 and (cost-price) > 1.50
$C[_{23}]$ TotalCost = (Price - (Price*15%)) * Quantity
$D[_{35}]$ ≠ D[P1] and ≠D[P2] and Quantity > 3 and (Cost-Price) > 0.50
$C[_{35}]$ TotalCost = (Price - (Price*20%)) * Quantity
$D[_{46}]$ ≠ D[P1] and ≠D[P2] and ≠D[P3]
$C[_{46}]$ TotalCost = Price * Quantity
$D[_{50}]$ output TotalCost to customer

Figure 15 - Matching of Both Evaluations (Symbolic and Implementation)

Procedure Partition	**Input**		**Expected Output**
	cost - price	Quantity	
D11	1.51	1	10% off
	1.49	1	no discount
	1.51	0	10% off
	1.49	0	no discount
D23	1.01	2	15% discount
	0.99	2	no discount
	1.01	3	15% discount
	0.99	3	no discount
D35	0.51	4	20% discount
	0.49	5	no discount

Figure 16 - Domain Boundaries

2.3.9. Cause-Effect Graphing

High-level specifications of system characteristics are used to develop test cases for cause-effect graphing (Myers, 1979). Its strength is its ability to explore input combinations. The graph is a combinatorial logic network, making use of only the

Boolean operators 'AND', 'OR' and 'NOT'. Myers (1979) suggests a number of steps to determine cases using cause-effect graphing:

- Divide the specification into workable pieces - this might be a specification for an individual transaction as a cause-effect graph for the whole system would be too large;
- Identify cause and effects - a cause is an input, such as a command typed at a terminal, an effect is an output;
- Construct a graph to link cause and effect to represent semantics;
- Annotate graph to demonstrate impossible combinations of causes and illogical effects;
- Convert graph to limited entry decision table, where conditions represent the causes, actions represent the effects and rules represent test cases.

Cause-effect graphing is a systematic method of generating test cases representing combinations of conditions. According to Myers (1979), since cause-effect graphing requires the translation of a specification into a Boolean logic network, it gives a different perspective on the specification and is a good way to uncover ambiguities and incompleteness in specification. A further advantage is that as many aspects can be automated, it is attractive for functional testing (Andriole, 1986).

2.3.10. Adaptive Perturbation Testing

Adaptive perturbation testing is the first to introduce heuristics to the testing methodology. Test data is selected automatically using a 'Parameter Perturbation Algorithm' which could include a gradient, probabilistic or heuristic search. Variable inputs are manipulated until the boundary of input variables is determined. Gradient techniques work best for smooth, continuous unimodal search spaces while probabilistic search is immune to plateaux, discontinuities and the highly granular search spaces.

According to Cooper (1976), heuristic search offers the best chance of success. Heuristics are established by the test engineer, and one method is to relate the data selection method to the system's performance. After application of a heuristic, a check is performed to determine the adequacy of the test set, the heuristic is then modified or a new heuristic chosen and run again. A good heuristic is added to the heuristic set, which is re-ordered and reused to attempt more successful searches.

Holmes *et al* (1993) has used an adaptive test data generator which uses heuristics applied to historical test information to predict new test data. The aim was to produce test data which eliminated mutated versions of the code. Holmes *et al* (1993) describes five different heuristics tried, the direct assignment heuristic, the alternating variable heuristic, the effectiveness of test data generated by the direct assignment and the alternating variable heuristics, linear predictor heuristic and the domain boundary heuristic. According to Holmes *et al* (1993), the domain boundary follower heuristic has proved successful in devising test data which exercise thoroughly a piece of software. This heuristic uses some of the principles of linear predictor heuristic, which is a linear extrapolation on each of the input variables. Once this is completed, points on the boundary are applied until searching is completed (Holmes *et al*, 1993). The problem with this approach is the large amount of computation time required, partially solved by the direct selection of starting points.

2.3.11. Automatic Path-Based Test Data Generation

Automatic path-based test data generation is used when a program is to be executed with the intention of achieving a particular level of coverage. Coverage refers to the amount of actual code which has been executed by the test and there are four types of coverage. Statement testing which requires that all program statements be executed at least once, is the simplest. In the sample code, figure 5, and the resulting flowchart, figure 17, the aim is to determine the effect of all executable code and to specify any code which is unreachable. In this example the seven statements must be exercised. This method may at the outset appear efficient in determining the effects of all executable code and to specify any code which is unreachable. However only existing code will be tested, the *else* portion of an *if* statement will not be forced to execute unless it is explicitly written into the code.

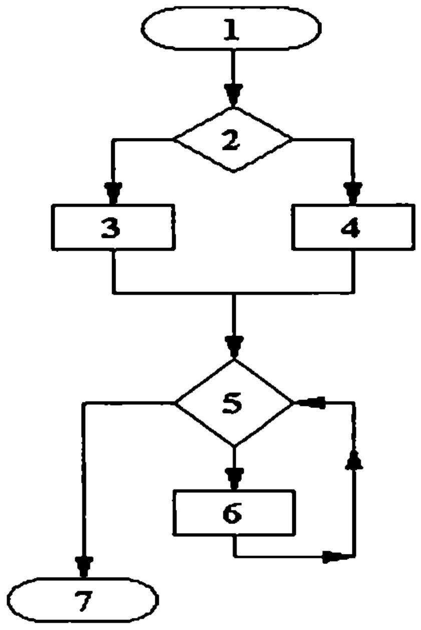


Figure 17 - Flow Chart of figure 5

Branch testing rectifies this problem as test data is generated which will attempt to access all outcomes of program decision points. Therefore any *if* statement must be exercised for both true and false and any loop for looping zero, one or two times. Another testing mechanism introduced by Hedley and Hennell (1984) takes the measurement of branches and loops one step further by reviewing linear code sequences and jumps (LCSAJs). One interpretation of LCSAJs is to review segments of code from one point, the start point or a decision point, through the next sequence of code or decision point, activating the segment of code from starting point to goal. The LCSAJs in the sample program are listed in table 3.

LCSAJs	Notes
1-2-3-5	True branch exercised
1-2-4-5	False branch exercised
3-5-7	loop NOT activated (True)
4-5-7	loop NOT activated (False)
3-5-6-5-7	Loop activated 1 time (True)
4-5-6-5-7	Loop activated 1 time (False)
3-5-6-5-6	Loop activated 2+ times (True)
4-5-6-5-6	Loop activated 2+ times (False)

Table 3 - LCSAJs through Sample Program listed in Figure 5

Woodward *et al* (1980) and Hedley and Hennell (1984) have developed test effectiveness ratios for all three testing methodologies, shown in figure 18. Test data is required which gives a value closest to one, for all three Test Effectiveness Ratios (TERs).

$$TER_1 = \frac{number-of-statements-exercised-at-least-once}{total-number-of-executable-statements}$$

$$TER_2 = \frac{number-of-branches-exercised-at-least-once}{total-number-of-branches}$$

$$TER_3 = \frac{number-of-LCSAJs-exercised-at-least-once}{total-number-of-LCSAJs}$$

Figure 18 - Test Effectiveness Ratios

In addition to these methods is also total path coverage, which as discussed in section 2.2.4, has the problem of infeasible paths. While the sample program only contains one infeasible path out of the four potential paths, the simple Trityp program which is described in chapter 9 contains only 10 feasible paths out of a possible 121. This makes it very difficult to statistically confirm coverage.

The first step in path-based testing is to establish a program control flow graph as in figure 17, from which the paths, both feasible and infeasible, through the program are then determined, and these were given in table 1. After the selection of a testing criterion, such as the near minimal set of paths to exercise, the final step is test data generation. Input data which will execute each test path is established, usually through symbolic execution which generates path constraints consisting of a set of equalities and inequalities for the input variables of the program, all of which need to be satisfied in order for the path to be traversed.

Contrasting this symbolic approach, Korel (1990;1992) introduced a method of dynamic test data generation. Test data is developed using actual values of input variables. As the program is activated the flow of execution is monitored and the input variables responsible for an undesirable flow are noted and used to correct the program. Once the code is amended, it can be rerun on the input data to confirm the flow of execution is now correct. Chapter 3 demonstrates a tool which will determine the paths upon which coverage should be attempted.

2.4. Conclusion

This chapter has included a general discussion of software testing techniques in use and the emphasis has been on the wide variety of testing tools, methods, and results and their individual application to a specific aspect of software testing. This variety of testing techniques itself indicates that there is a definite need and desire for software testing tools and that no one method has been deemed perfect. This allows more and more types of testing techniques. One thing each dynamic testing medium requires though, is test data. Normal test data generation procedures have been performed by random generation, or by looking at the minimum and maximum border values in each variable range. Is there a better method?

Test data once generated is applied to the function that is being tested. A measurement of how the test data performs on the function needs to be taken, and TERs are used to measure the coverage of test code and are applicable to branch, statement, LCSAJs and path testing. What is introduced in future chapters, are adaptive search techniques for the generation of test data to effectively attain the specified TERs for a dynamic test. These adaptive search techniques aim to measure the structural coverage of a function under test, while keeping the amount of disturbance to it to a minimum. This means that only a small amount of recoding should be performed on the function under test so the test is on the original code, not what has been added as analysis routines. Therefore the testing will be performed as black-box testing with slight structural modifications.

Chapter Three

A Tool for the Generation of Test Data

3.1. Introduction

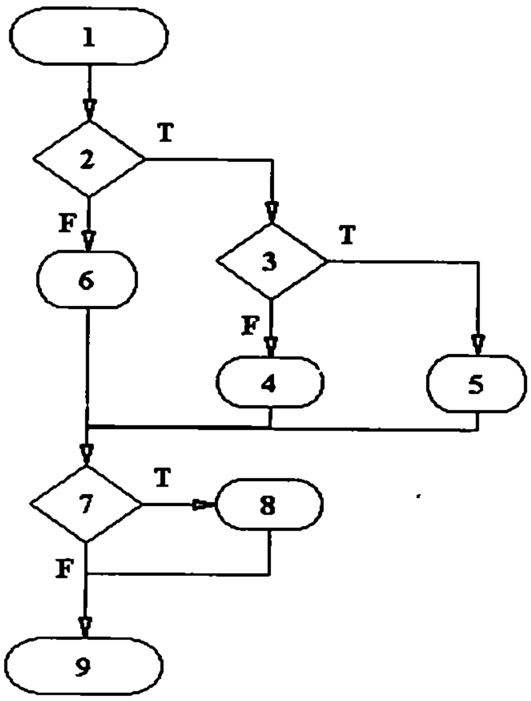
Xanthakis *et al* (1992) suggests that a tool for the automatic generation of test data would be of inestimable value to the software development community, as it will permit the almost total automation of the review process. In order to ascertain what test data is required and what duty it is to perform, a thorough analysis of the code under test needs to be accomplished. This testing tool begins with an analysis of the code and concludes with a manual review of results of a test. The algorithm for this technique is in figure 19.

	Steps for Testing Tool
1	Analyse the test code and determine function to be tested
2	parse the function to determine the movement of conditional statements
3	determine the nearest neighbours for each step of the code
4	determine all potential paths using the nearest neighbour algorithm
5	establish the testing metric to satisfy
6	while desired coverage metric has not been attained
6.1	generate test data using test data generation methods (chapters 6 - 8)
7	the results are evaluated and exceptions are viewed manually

Figure 19 - Algorithm for Testing Tool using Test Data Generation

3.2. Steps of the Technique

In the following subsections each of the steps will be discussed with the exception of steps 6 and 6.1 which will be featured in chapter 6 through 8. The demonstration of this method will be performed using the simple flowchart in figure 20 and program listed in figure 21.



```
1 input (int x, int y, int z)
2   if (x+y ≤ z)
3      if (x<y)
4        x =  z - y
5      else y = z - x
6   else x = z+y
7   if (x < y)
8      x = z
9 end program
```

Figure 20 - Flow Chart of Figure 21 Figure 21 - Function Under Test

3.2.1. Analyse the Test Code

The first step is to analyse the code. A program which requires examination usually consists of many individual functions. Each function should be tested separately, and therefore an analysis of the code needs to be performed to determine how many functions there are, their name, and corresponding input variables. The results of the search are printed out to the user, and could appear as in figure 22.

```
                    Functions available to test

            1. input       (requires 3 variables (x,y,z))
            2. calculate   (requires 3 variables (x,y,z))
            3. output      (requires 2 variables (x,y))
```

Figure 22 - Demonstration Printout after Initial Analysis of the Potential Code

The user selects the function to examine, each of which should be dealt with individually. On completion of the individual test the entire program can be scrutinised with the same data to determine the final status of the program under examination.

3.2.2. Parse the Function Under Test

Once a function is chosen the user is then required to give the range of the variables, or these values could be read from a file. In this sample program there exist three input variables x,y, and z and the chosen range of each variable for this example will be [0..20]. This function consists of two 'if then/else' statements, one which is a nested 'if then/else', and one solo 'if' statement. To illustrate the program design, as in figure 21, knowledge is required of the level of each conditional statement, for example line 2, in figure 21, is at the top level or level 1, whereas line 3 is at level 2. This can be done through a parsing of the code. The parser analyses each statement and attempts to separate out each portion of code. A conditional statement in 'C' does not require brackets, but in order to determine the level of such statements these brackets must be entered. To ascertain the level of each statement all that is required is a counting of brackets. Figure 23 is a bracketed version of figure 21.

```
1         void input (int x, int y, int z)
          {①
2           if (x+y ≤ z)
            {②
3             if (x<y)
              {③
4               x = z-y;
              }
5             else
              {④
6               y = z-x;
              }
            }
7           else
            {⑤
8             x=z+y;
            }
9           if (x<y)
            {⑥
10            x=z;
            }
          }
```

Figure 23 - Figure 21 with Added Brackets (**① etc. is the number of opening brackets used**)

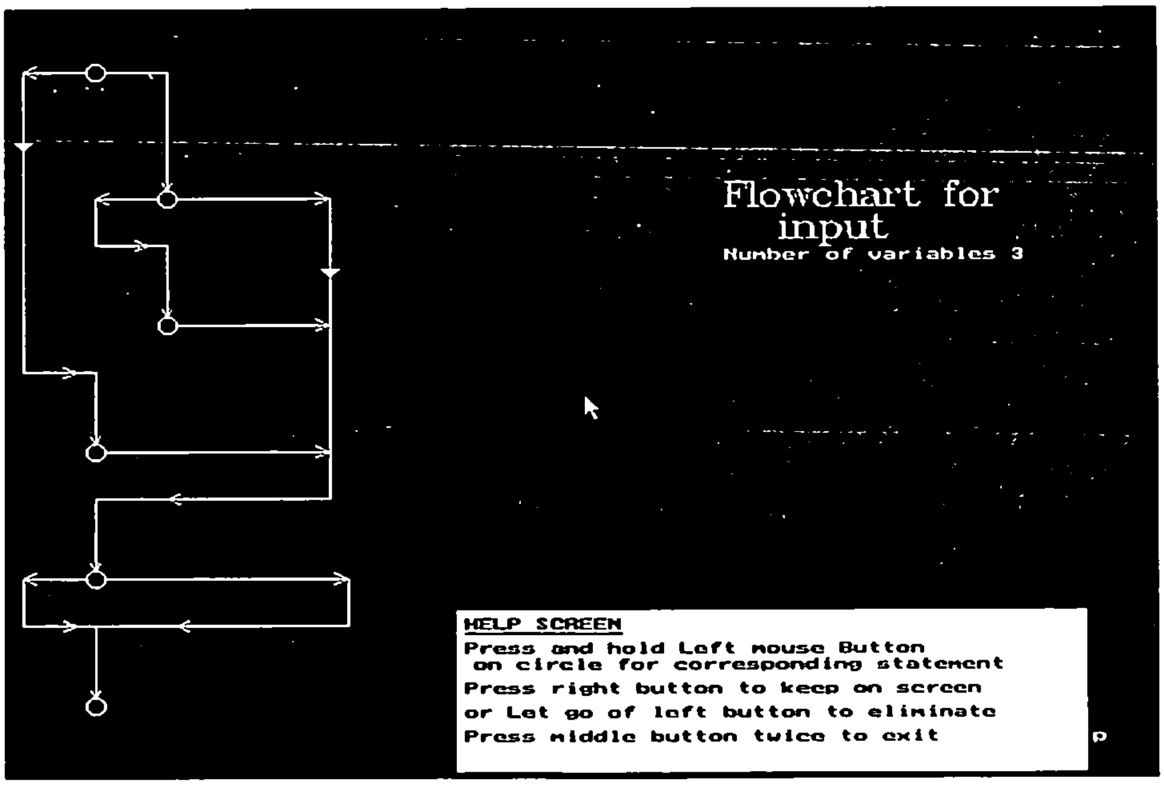
Therefore in this function there are six opening and closing brackets. If on reading the code the ③ is reached, and there have been no closing brackets for ②, this would indicate to the parser that this conditional statement is nested within the previous opening brackets. This division is necessary because it is possible to have a number of nested 'ifs' all on the same line, which would be difficult to count. The results of this activity will help in building a numerical representation of the function which can be used to determine the paths which need to be exercised.

The information in table 4 can be obtained using the parser. Column one is the line number from figure 23, column two is the consecutive number of each line, Column three is the conditional statement label, pairing true and false when appropriate. Column four is a value label derived from column 3. Column five is the conditional statement level, a 1 is a top level statement, a 2 is a single level statement, *etc.* Column six indicates if a node is nested and which statement is it nested within from column two, number 2 and 3 are nested under 1. Finally column seven shows whether the statement is an 'if' statement (1), an 'else' statement (2) or an 'else if' (3). From this information it is possible to build a flowchart, figure 24, automatically, similar to the type drawn manually in figure 20.

Column						
1	2	3	4	5	6	7
Line number	**number**	**label**	**value label**	**level**	**sub-node of:**	**type of statement**
line 2	1	1T	1	1	0	1
line 3	2	2T	2	2	1	1
line 5	3	2F	2	2	1	2
line 7	4	1F	1	1	0	2
line 9	5	3T	3	1	0	1

Table 4 - Conditional Statement Information

The input variables to the function under test are specified and the range of these variables is requested of the user, or read from a file. This gives quick access to the variable range which is used to determine the size of the search space.



HELP SCREEN
Press and hold Left mouse Button
on circle for corresponding statement
Press right button to keep on screen
or Let go of left button to eliminate
Press middle button twice to exit

Figure 24 - Screen-Dump of Automatically Produced Flow Chart

3.2.3. Nearest Neighbours

With the information in table 5, using columns two, three, four, and five, it is possible to determine which statements are accessible from a given statement, such that 1T can reach 2T and 2F, but not 1F or 3T. Each statement's possible neighbours are calculated, a 'Z' indicates that the path could finish after this statement. Column three determines whether a given statement can be the initial statement in a path, a level 0 can start a path, but greater than 0 can not.

Column		
1	2	3
Statement	Nearest Neighbours	Level of Start
1T	2T,2F	0
2T	3T,Z	1
2F	3T,Z	1
1F	3T,Z	0
3T	Z	1

Table 5 - Nearest Neighbours of Each Statement

3.2.4. Determine Potential Paths

It is now possible to determine the potential paths through the program. These are of course not only the feasible paths but also the infeasible ones. Using the method described in chapter 2 it is possible to determine how many paths there will be as shown below

$$1 \cdot 2 \cdot (3 \cdot (4+5)+6) \cdot 7 \cdot (8+9).$$

If all the statements are substituted by 1's, the expression is as follows,

$$1 \cdot 1 \cdot (1 \cdot (1+1)+1) \cdot 1 \cdot (1+1) = 6 \text{ paths.}$$

To determine the details of these paths, the data given in table 5 is combined to list the 6 paths. Table 6 lists the resulting paths.

Path Number	Path
1	1T-2T-3T
2	1T-2T
3	1T-2F-3T
4	1T-2F
5	1F-3T
6	1F-3F

Table 6 - Paths through Function Shown in Figure 21

3.2.5. Testing Metric to be Used

There are a number of methods available for measuring coverage of a function, all of which have been discussed in chapter 2. To reiterate they are path, branch, LCSAJs and statement testing. Once the paths have been established in the previous section it is possible to begin path testing, that is to run the program and check to see which paths the test data can cover from the given list of paths. As an initial step this procedure is fine, but most programs have a lot of paths, some of which will never be covered. If this were to be the sole means of testing a great deal of manual review would need to be performed to determine if the remaining paths are infeasible, and therefore additional coverage measures must be used.

In the function in figure 21 there are 9 statements, including the end statement, and each statement should be coded as in figure 25 to indicate whether it has been accessed or not. To perform statement testing, an accumulator line called 'strcat' in C code is inserted in the parsing stage after each statement to collect the statement numbers as they are accessed. If the randomly selected input variables {1,3,5} were used as variables x, y, and z respectively the resulting value of the string 'states' would be

states = "s1s2s3s4s7s8s9".

From this information of the statements exercised, above, it can be determined that the remaining statements which need to be exercised are "s5" and "s6". A random selection of test data indicates that the following test data sets {6,3,5} and {3,1,5} will exercise these remaining statements. All the statements have now been exercised.

```
1  input (int x, int y, int z)
   {
    char states[20];              // establish string called states which contains information on the
                                  //statements covered in function
    strcpy(states,"s1");          // add s1 to string states as statement 1 has been exercised by start of
                                  // function.
2   if (x+y ≤ z)
    {
        strcat(states, "s2");     // statement 2 if statement has been exercised
3       if (x<y)
        {
           strcat(states, "s3");  // statement 3 if statement has been exercised
4            x = z - y;
           strcat(states, "s4");  // statement 4  with in statement 3 has been exercised
        }
5        else
        {
           strcat(states,"s5");   // statement 5 matching else to statement 3 has been exercised
             y = z - x;
        }
    }
6   else
    {
        strcat(states, "s6");     // statement 6 matching else to statement 2 has been exercised
        x = z+y;
    }
7   if (x < y)
    {
        strcat(states, "s7");     // statement 7 if statement has been exercised
8       x = z;
        strcat(states, "s8");     // statement 8 within statement 7 has been exercised
    }
9   strcat(states, "s9");         // statement 9 - program end
    printf("%s",states);          // print out the statements which have been exercised
   }
```

Figure 25 - Amended Code from Figure 21 for Statement Coverage

```
input (int x, int y, int z)
   {
     char tested[20];              // establish string called tested which contains information on the
                                   //branches covered in function
     if (x+y ≤ z)
     {
          strcat(tested, "1T");    //Branch 1 has been activated for true
          if (x<y)
          {
             strcat(tested, "2T"); // Branch 2 has been activated for true
             x =  z - y;
          }
          else
          {
             strcat(tested,"2F");  // Branch 2 has been activated for false
             y = z - x;
          }
     }
      else
      {
          strcat(tested, "1F");    // Branch 1 has been activated for false
          x = z+y;
      }
     if (x < y)
     {
          strcat(tested, "3T");    // Branch 3 has been activated for true
          x = z;
     }
     else
     {
          strcat(tested, "3F");    // Branch 3 has been activated for false
     }
   printf("%s",tested);            // print out the statements which have been exercised
   }
          x = z+y;
      }
7    if (x < y)
     {
          strcat(states, "s7");    // statement 7 if statement has been exercised
8         x = z;
          strcat(states, "s8");    // statement 8 within statement 7 has been exercised
     }
9    strcat(states, "s9");         // statement 9 - program end
     printf("%s",states);          // print out the statements which have been exercised
   }
```

Figure 26 - Amended Code of Figure 21 for Branches and LCSAJs

Branch testing and LCSAJs can be tested simultaneously by amending the original code of figure 21 as shown in figure 26. Changes are the same as those for path testing. Each path, as found by the nearest neighbour sequence can be broken down into individual branches or LCSAJs. Success will occur when all the branches have been exercised. Using the test data{1,3,5}, the branches covered are below:

tested = "1T2T3T"

The remaining branches to be tested are 1F, 2F, 3F which can be exercised with the following randomly selected test data sets {6,3,5} and {3,1,5}.

LCSAJs, linear code sequence and jumps, are segments of code which follow from the flow of one decision point through to the next decision point, such that 1T2T is one sequence which should occur in exercising the code while 1F3T is another. Table 7 is a listing of all the possible LCSAJs in the function in figure 26, along with the corresponding amount of search space which will satisfy the LCSAJ and the corresponding percent of the total search space. A test data set may satisfy more than one LCSAJ, therefore the total percentage of search space which satisfies each LCSAJ is greater than 100%. The previous test data set {1,3,5} resulting in path "1T2T3T" satisfies LCSAJs number 1 and 3, the test data set {6,3,5} satisfies LCSAJs 8 and the test data set {3,1,5} satisfies the LCSAJs 2 and 6. To exercise the remaining LCSAJs, 4,5, and 7, more test data sets are required. Set {1,2,10} will satisfy LCSAJ 4 and set {1,1,3} will satisfy number 5. Finally there is no data set to satisfy LCSAJ 7 and it is infeasible. Therefore the best TERs which can be achieved is 88%, that is seven of the eight possible LCSAJs that can be exercised and all of the feasible LCSAJs, listed in table 7, can be satisfied with the generation of a minimum of five data sets.

Number	LCSAJs	Amount which Satisfies LCSAJs	% of Search Space
1	1T2T	825	8.9%
2	1T2F	946	10.2%
3	2T3T	440	4.75%
4	2T3F	385	4.16%
5	2F3T	440	4.75%
6	2F3F	506	5.46%
7	1F3T	0	0.0%
8	1F3F	7490	80.88%

Table 7 - List of LCSAJs through Sample Code Shown in Figure 26

This example illustrates that although statement and branch coverage may exercise all portions of the code, testing these pieces of code together as LCSAJs determines whether a sequence of statements is accessible. This offers a stricter test than branch or statement testing, while avoiding the great volume of paths generated for path testing.

3.2.6. Manual Review of Results

If, after completion of the testing procedure, there remains any statements, branches or LCSAJs which are inaccessible, the system should print out a list such as in figure 27 which gives information regarding the status of the test.

Test Complete	
All Statements	exercised
All branches listed	exercised
LCSAJs	not exercised
	7
Number of input variable: 3	
range of variables:	
	x: [0..20]
	y: [0..20]
	z: [0..20]

Figure 27 - Status Printout after Test

3.2.7. The Handling of Loops in Code

Thus far the functions to be parsed have not included loops. When a loop is activated its run-time can be infinite and many testing authorities, e.g. Roper (1994), suggest that the best plan is to activate each loop so it is only tested 0, 1 and 2 times, Zero times indicates that the loop is not activated. Although a loop can last a long time, the number of loops recorded will only be the first two, therefore a loop that runs two times will register the same LCSAJs as one which activates 2+ times. The code, from figure 21, is amended as in figure 28 to include a loop as well as measure the level of path coverage.

```
loop = 0;                      // added to count how many times loop is activates
while (y>2)
{
   y= sqr(y)                   // actual loop activity in function (sqr = squareroot)
   loop += 1;                  // counting times of loop activation
   if (loop == 1)              // checking if loop has been activated 1 time
      strcat(tested,"L1");
   else if (loop == 2)         // checking if loop has been activated twice
      strcat(tested,"L2");
}
if (loop ==0)                  // if  loop has not been activated
   strcat(tested,"L0");
```

Figure 28 - Testing for Activation of Loops

Loops can be tested as well, and figure 29 is the new code with the accompanying flowchart of the code in figure 30. The number of LCSAJs has now increased from eight to fourteen and the additional six are listed in table 8 with corresponding test data which satisfies these new LCSAJs. The number of data sets which satisfies each of the LCSAJs add up to the total search space size, 9261, as each test data set will have to satisfy one of these LCSAJs which involve the activation or non-activation of the loop. The minimum number of test data sets required to be generated before it is possible for all the LCSAJs to be exercised, is now seven. The number of actual paths through the function under test has now increased from 6 to 24, 15 of which are feasible and 9 are infeasible.

```
1  input (int x, int y, int z)
   {
    char tested[20];              // establish string called tested which contains information on the
                                  //branches covered in function
2   if (x+y ≤ z)
    {
        strcat(tested, "1T");     // branch 1 has been exercised
3       if (x<y)
        {
           strcat(tested, "2T");  // branch 2 (true) has been exercised
4            x =  z - y;
        }
5        else
        {
           strcat(tested,"2F");   // branch 2 (false) has been exercised
             y = z - x;
        }
    }
6   else
    {
        strcat(tested, "1F");     // branch 1 (false) has been exercised
        x = z+y;
    }
7   if (x < y)
    {
        strcat(tested, "3T");     // branch 3 (true) has been exercised
8       x = z;
    }
    else strcat(tested,"3F");     // branch 3 (false) has been exercised.  Added code to show
                                  // sequence of steps
    loop = 0;                     // to count how many times loop is activated
a    while (y>2)                  // original function code
    {
b      y = sqr(y);                // original function code
       loop += 1;                 // increment loop each time activated
       if (loop == 1)             // loop activated 1 time
          strcat(tested, "L1");
       else if                    // loop activated 2 times
          strcat(tested, "L2");
    }
    if (loop == 0)                // loop not activated
        strcat(tested, "L0");
9    printf("%s",tested);         // print out the statements which have been exercised
   }
```

Figure 29 - New Function Code Based on Figure 28 and Figure 21

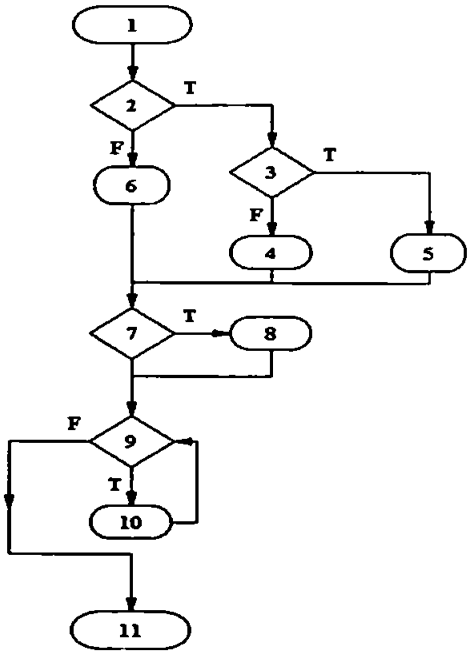


Figure 30 - Flow Chart of Figure 29

Number	LCSAJs	A Data Set which Covers LCSAJ	Amount which Satisfies LCSAJs	% of search space
9	3TL0	{1,1,3}	8	.09%
10	3TL1	{1,3,5}	232	2.5%
11	3TL1L2	{0,0,9}	640	6.9%
12	3FL0	{1,2,10}	855	9.2%
13	3FL1	{6,3,5}	2520	27.2%
14	3FL1L2	{0,9,0}	5006	54.11%
			9261	100%

Table 8 - Additional LCSAJs from Figure 28

When a loop is to be activated and checked to see if it only functions once, then the path searches for an occurrence when L1 is not followed by an L2. All loops can be handled in this way, and while this will not detect if an error occurs after 2+ runs or whether the loop will activate the required number of times, it is an effective way of limiting the number of paths and hence LCSAJs though a piece of code. More complicated loops will be demonstrated in chapter 9 and 10.

Finally listed in table 9 are all the branches in the function which need to be exercised along with the amount of test data which will cover each of the branches and the corresponding percentage of the search space. The size of the search space is 9261, which is three variables each of the range [0..20]. All of the branches can be exercised. The branch which is exercised by the smallest amount of test data is L0, which is when the loop is not activated. Only 9.32% of the search space exercises this branch, that is 863 test data sets. The branch which is exercised the most is L1 which is exercised by 90.68% of the search space, or 8398 test data sets. These figures add up to more than 100% as most test data sets will satisfy more than one branch. All branches can be satisfied with the generation of just three test data sets, but to satisfy both branches and LCSAJs, a minimum of seven test data sets must be generated. The maximum number of test data sets which could be generated prior to complete coverage is 9254. This value represents the fact that only eight data sets satisfy LCSAJ 9 {3TL0} and it is possible for 9254 data sets of the total number 9261 to be generated before this LCSAJ is exercised.

Branch	Amount which Satisfies LCSAJs	Percentage of Search Space
1T	1771	19.12%
1F	7490	80.88%
2T	825	89.08%
2F	946	10.21%
3T	880	9.50%
3F	8381	90.50%
L0	863	9.32%
L1	8398	90.68%
L2	5646	60.97%

Table 9 - Branches and the Corresponding Percentage of the Search Space which Satisfies These Branches

3.3. Conclusion

This chapter illustrates a tool for analysing program code to determine their structure for use in the testing procedure. What has been left out of this procedure, is how test data will be generated and this is discussed in chapter 6 through 8. With guidance from this chapter it is possible to determine a function to be tested, analyse that function to find out the conditional statements, and produce a flow chart. From this information it is possible to ascertain all paths through the code. As this may include a lot of infeasible paths, its purpose is to test the statements, branches and LCSAJs that have been established from the path data. The final coverage information is used to manually review any areas of code which have been unobtainable. While loop testing is more difficult and limited, it can be performed in a similar manner and this method is demonstrated in this chapter.

It has been established that the complete test program, shown in figure 29, has 24 paths through the code, 15 of which are feasible. Additionally there are 14 LCSAJs and 9 branches to be exercised, a successful test will achieve a coverage ratio of 95% as LCSAJ 7 is infeasible. Data generation to cover the paths, branches and LCSAJs will be performed for this test program using random generation and compared to the results gained using GAs in chapter 6. These methods will then be compared to additional adaptive search techniques, Simulated Annealing and Tabu search in future chapters. The next chapter, however, gives an illustrative example of random testing and test data generation.

Chapter Four

What is Random Testing?

4.1. Introduction

Random testing is the most common and accessible method for software testing. One procedure would be to randomly generate test data within the acceptable range of the input variables and then apply it to the piece of code under test. If the test fails, *i.e.* a run-time error, there is an error in the code which needs to be corrected. Once the error is corrected the same test data is used to confirm the 'correctness' of the code by establishing if another run-time error occurs. Other versions of random testing include the use of randomly generated test data for statement, branch coverage, LCSAJs or path testing.

Ince (1987) suggests that random code generation is both inexpensive and timely in comparison to tools which attempt to derive test data from program code, design specifications and functional specifications. Duran (1984) compared random testing to partition testing, see chapter 2, and determined that for 100 simulated random tests and 50 simulated partition test cases, random testing was superior and less expensive. Ince (1984) compared random generation to adaptive techniques and determined little difference in the coverage of the code. He did discover, though, that in measuring the number of runs required before a satisfactory coverage percentage was attained, random generation was superior. Additionally Ince (1984) feels that restrictions should be placed on the input domains, as he believes that a small subset of the variable range will perform equally as well as a larger grouping.

4.2. A Random Number Generator

As Ince (1984) states, all that is required is a random number generator and some way of measuring the level of coverage of the code. In chapter 3 a method for determining coverage level has been developed. Random numbers can now be generated. To illustrate the sort of search space in which the random number generator will operate, the program used in chapter 2 to demonstrate testing techniques will again be used, as shown in figure 31. The reason this function is used is that there are only two input variables, which allows the actual search space to be shown. The input variables x and y each will have the range {0..20}. For this example the goal will be to exercise all the paths through the function. As discussed, there are four paths through this function, shown in figure 32, and the search space divided by the path coverage is given in figure 10. The size of the search space is 21^2 or 441 unique combinations of input variables, and while calculations did determine that there are a possible four paths through the code, the fourth one is infeasible.

```
read_in(int a, int b)
if (a≥b) then
  print a
else print b
while a ≤ b
  a = a+1
end
```

Figure 31 - Sample Two Variable Input Function used to Illustrate the Search Space

Path Number	Path	Number of Data Sets	Share of Search Space
$Path_0$	1-2-3-5-6-5-7	21	4%
$Path_1$	1-2-3-5-7	210	48%
$Path_2$	1-2-4-5-6-5-7	210	48%
Infeasible Path	1-2-4-5-7	0	0%

Table 10 - Paths through Program Listed in Figure 31 and Their Corresponding Share of Search Space shown Graphically in Figure 32

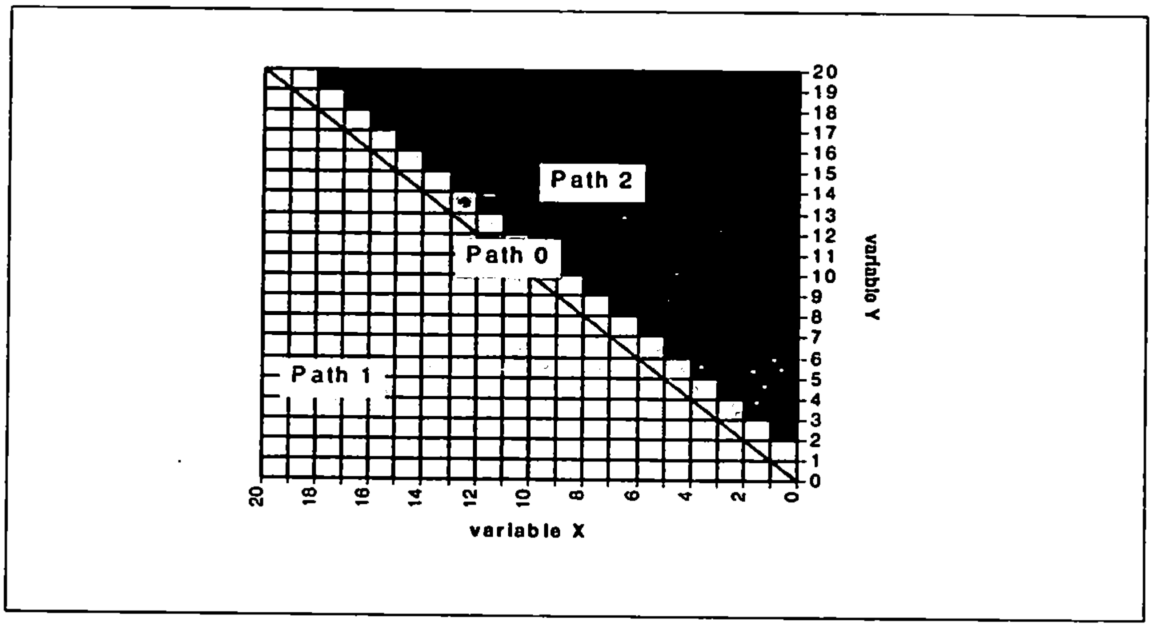


Figure 32 - Graphical Representation of Search Space, the Area Covered by Each of the Three Paths is Indicated

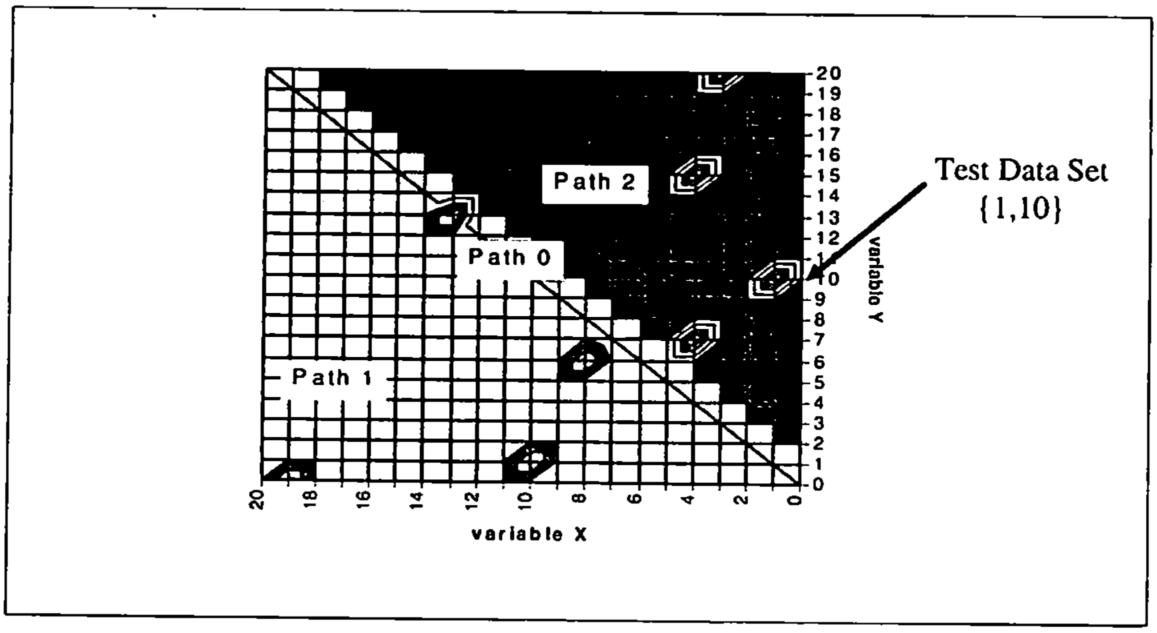


Figure 33 - Graphical Representation of Position of Randomly Selected Test Data Sets within the Specified Search Space

In this sample code, four percent of the search space satisfies $path_0$, while the rest of the search space is equally divided between $path_1$ and $path_2$. The random test data was generated. An example of the location in the search space which was randomly generated eight times is shown in figure 33. These eight randomly generated test data sets exercise all three paths, and follow the same pattern as the complete search space percentages, hence $Path_1$ and $Path_2$ had the most test data generated. The next step is to measure the effectiveness of random generation.

4.3. How is a Measurement Taken?

There are two methods available to measure the abilities of random test data generation. The first method is time. If random testing proved to be quicker in the generation of test data that would be an asset. Actual time, however, might not be the only method of measurement. If a function is quite large it may take a great deal of time before it is exercised and if the same test data is generated again and again this increases the run time of a function. Therefore, to generate a small amount of test data which exercises all the required areas of the function would be beneficial. For this reason the measurement for random testing will be performed on the amount of unique test data which needs to be generated prior to coverage of the code. Therefore the amount of data sets generated is a key factor in the comparison of testing techniques.

In this sample program the search space consists of 441 possible combinations of variables. The random generator was run 1000 times to give the average amount of test data generated before all paths were exercised. The average amount of new test data generated per run was 20.269. Table 11 shows the average amount of test data for each path which was required over 1000 runs.

Path	Average Unique Pairs Generated	Percentage of Space Searched
$path_0$	1.089	5%
$path_1$	9.660	48%
$path_2$	9.520	47%
Total	20.269	100%

Table 11 - The Paths through the Function and the Average Amount of Test Data Required before Coverage was Attained over 1000 Runs

The results of the random test data generation are very close to the percentages generated from the exhaustive search with $path_0$ at 5%, $path_1$ at 48% and $path_2$ at 47%. Random searches, whilst effective, spend too much time looking at areas which have already been searched, by finding the same path, again and again.

4.4. Conclusion

This chapter describes how random test data is used to exercise a function under test. The purpose of this chapter was to illustrate the type of search space that may be encountered and how test data is generated to exercise the specified path. In later chapters a more strenuous test function will be given and compared to other test data generation methods. While random generation is simple and requires few resources, will the amount of test data generated still make it a viable method?

Chapter Five

Genetic Algorithms - A Brief Introduction

5.1. Introduction to GAs

For small search spaces, classical exhaustive methods usually suffice, but as the search space grows search methods need to be devised which can minimise the size of the search. Genetic Algorithms (GAs) are one such search technique. The first influential work was produced by Holland in 1975 although he had worked in the area since the 1960's. GAs are based on the premise that computer algorithms can mimic natural evolution, but what is natural evolution? In natural evolution there exist chromosomes which consist of genes, these chromosomes determine such things as sex, personal characteristics, or hair and eye colour. The chromosomes from each parent are passed on to their children. Some of these chromosomes may replicate a chromosome from one parent, they may be a cross of both parents or a chromosome may mutate during the transfer phase. The resulting offspring is a combination of characteristics and traits from both parents. While the battle of the chromosomes takes place in the offspring, external environmental conditions combine with the received chromosomes for the battle of survival. If the offspring has received particularly sturdy chromosomes to do battle against illness, famine and strife that offspring will have a greater chance of surviving to the next generation, on the other hand if the chromosomes are poor the chance for survival lessens. Hence future generations should consist of many more sturdy offspring with fewer and fewer weaker members.

The established features of evolution are summarised as follows (Davis, 1991):

- Evolution occurs on chromosomes not the living beings encoded;
- Selection is the process in which certain chromosomes are chosen to reproduce more often;
- Reproduction is when evolution takes place. Mutations may cause the chromosomes of the offspring to differ from their parents. Recombination may cause quite different chromosomes in the offspring by combining material from the chromosomes of two parents;
- Evolution depends on the chromosomes of the parent and the structure of the chromosome decoders.

Using the concept that evolution takes place on chromosomes, Holland (1975) created an algorithm which manipulated strings of binary digits by carrying out simulated evolution on populations of chromosomes. The only information available to assist reproduction would be the fitness of each chromosome, a figure relating to how 'well' the chromosome performs on the function under test.

These simple algorithms, known collectively as genetic algorithms and using the simple procedure of reproduction, crossover and mutation, have demonstrated complicated behaviour and solved many problems deemed NP-complete. Areas of research have included scheduling (Davis, 1987; Syswerda, 1989; 1991), game playing (Chi *et al*, 1988), music compositions (Horner and Goldberg, 1991), and transportation (Vignaux and Michalewicz, 1989).

5.2. How do GAs Function?

To illustrate GAs, they will be used to optimise the simple function below,

$$f_z = 1 + \cos\sqrt{\frac{(x^2 + y^2)}{5}}$$

The goal is to maximise f_z. The variable range for x and y is the integer set [-5..5] and the search space is illustrated in figure 34.

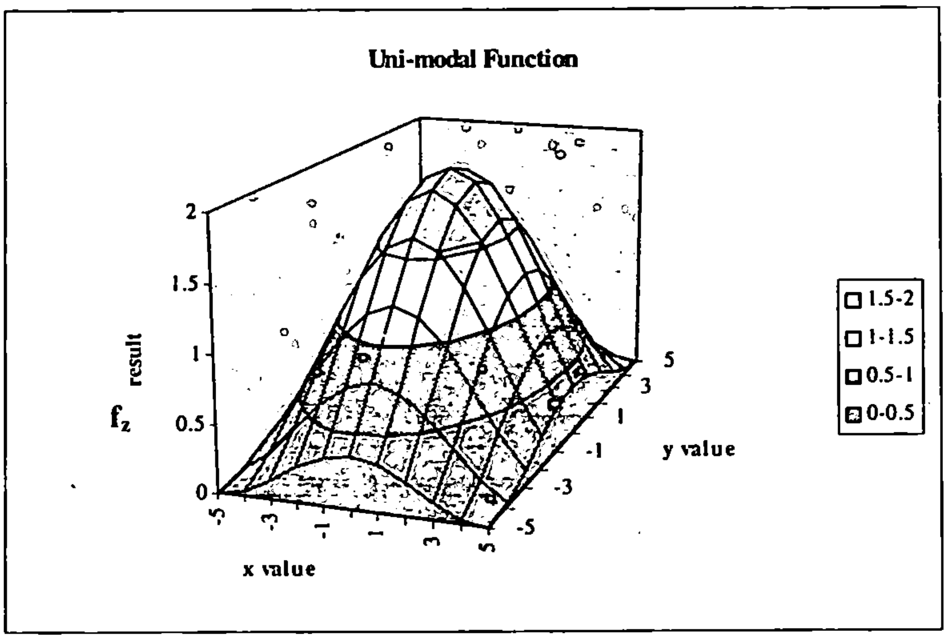


Figure 34 - Surface Graph of Search Space

A chromosome of binary digits is used to represent the integers x and y. The length of the vector is determined by the precision desired and the limitations of binary digits. For this case each variable is represented by a four digit binary string which gives a chromosome length of eight as shown below,

$$2^3 < 10 \le 2^4 .$$

Therefore, a chromosome of length three would evaluate to the value of 8 or less, whereas chromosomes of length four can be evaluated to 16 or less. Using a four digit binary string will thus satisfy the variable range requirement of [-5..5]. The chromosomes (00000000) and (11111111) represent the boundaries of the range [-5,-5] and [5,5], respectively. To illustrate, the chromosome (01010110) encodes the x,y co-ordinates

$$x = ((\text{binaryInt}/15)*10) - 5$$
$$y = ((\text{binaryInt}/15)*10) - 5,$$

which translates to the actual co-ordinates as follows,

$$x_1 = (0101) = ((5/15)*10)-5 = -2$$
$$y_1= (0110) = ((14/15)*10)-5 = 4.$$

5.2.1. Initial Population

An initial random population of ten chromosomes is created, each consists of 8 genes which are composed of 1s and 0s. For this test the population in table 12 is generated. ,

Number	Chromosome
1	01101110
2	11001101
3	01001010
4	11011011
5	00111100
6	10010011
7	00111001
8	10101000
9	00101010
10	11101101

Table 12 - Initial Population Selected Randomly

5.2.2. Fitness Function

The fitness is calculated for each member of the population. The results are given in table 13. The values in the column entitled 'Share of Total Fitness' give the portion of the total fitness for each chromosome. For example, chromosome 1 with its fitness of 0.7303 has a 0.087 or 8.7% share of the sum of the fitness for the entire population which is 8.39. Chromosome 2 , with a fitness of 0.3827, has a 0.046 or 4.6% share of the sum of the population, this gives a unique position in the population for the purposes of establishing the next population, as demonstrated in section 5.2.3, which are all the randomly selected values between 0.087 and 0.133.

Chromosome Number	x	y	f(x,y)	Position of Member in Total Fitness of Population
1	-1	4	0.7303	0.087
2	3	4	0.3827	0.133
3	-2	2	1.3011	0.288
4	4	2	0.5939	0.359
5	-3	3	0.6792	0.440
6	1	-3	1.1559	0.578
7	-3	1	1.1559	0.716
8	2	0	1.6260	0.910
9	-4	2	0.5839	0.979
10	4	4	0.1814	1.000

Table 13 - Integer Values with Associated Fitness from Population of Chromosomes

5.2.3. A New Population

As discussed in section 5.2.2, the fitness of each chromosome is calculated as the percentage of the total fitness of the population. To represent this, figure 35 has been created to illustrate each member's share of the total population fitness of 8.39.

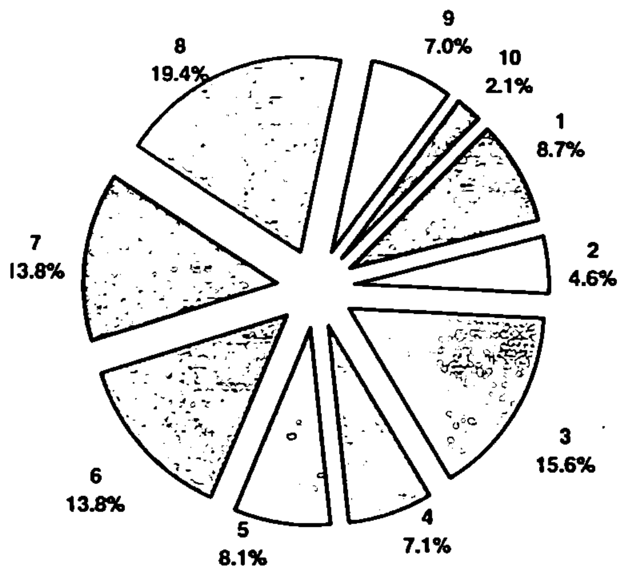


Figure 35 - Each Population Member's Share of the Population Fitness

Before generating the next population it must be noted that it is possible through the process of reproduction, mutation and crossover that the best member of the population could be eliminated. This could be acceptable, as it commonly occurs in nature, however it could have a negative effect on the search population. To avoid this the best member of the previous population is added to the new population, and this can be done in a number of ways, as an additional population member if it does not exist in the new population, as an additional member regardless of whether it exists in the population, or it can replace an existing population member to keep the population size constant. The member of the population to replace can be chosen at random or to replace the least fit member. The replacement style applied here is to remove a random member of the population and replace with the 'best' member, regardless of whether that member already exists in the population. This replacement procedure takes place after mutation and crossover. Therefore chromosome number 8 will be reserved a place in the next population.

It is now time to randomly generate the next population. Ten times a random number between 0 and 1 is generated. This number is then compared to column 5 of table 13. If for example, the random number falls between 0.00 and 0.087 then member number 1 proceeds to the next generation. Table 14 is a random drawing of ten new population members with their corresponding chromosomes and fitness values.

The total fitness of the new population is now 11.909 an improvement of 3.519 over the previous population. This new population is not complete, but before the two operators crossover and mutation are described, a brief discussion of other reproduction methods follows.

Number	Random Value	Original Number	Chromosome	Fitness
1	0.617	7	00111001	1.1559
2	0.104	2	11001101	0.3827
3	0.081	1	01101110	0.7303
4	0.580	7	00111001	1.1559
5	0.720	8	10101000	1.6260
6	0.733	8	10101000	1.6260
7	0.874	8	10101000	1.6260
8	0.222	3	01001010	1.3011
9	0.429	5	00111100	0.6792
10	0.881	8	10101000	1.6260

Table 14 - Selection of Next Generation

5.2.4. Steady State Reproduction

The previously discussed method of reproduction, proportional, has some drawbacks, one of which is that many of the best individuals may not reproduce at all. If they have reproduced, some of their best characteristics may be destroyed by mutation or crossover. One solution, known as steady state, is to replace one or two members of the population at a time rather than the entire population. The replaced members can be the worse members of the population or a random selection of members. The algorithm for steady state (Goldberg, 1989) is given below:

```
Steady State Algorithm
1. Create n children through reproduction
2. Delete n members of the population to make room for the children
3. Insert children into the population
```

Using Steady State reproduction, good members of a population are protected from deletion while poor members are most likely to be deleted (Syswerda, 1989).

5.2.5. Tournament Selection

Another method which attempts to reduce problems associated with proportional reproduction is to use a tournament methodology. Tournament selection, discussed by Brindle (1981), involves choosing some number of population members at random. The best member of this grouping will go through to the next generation. In the example, if chromosome 1 was to be compared to chromosome 3, member 1 with a fitness of 1.1559 would be replicated in the next generation and at this stage member 3 would not. Should member 3 later be compared in a tournament with, for instance member 9, it will then be placed in the next generation.

The average fitness of the next generation should be higher than the previous population. Creating a population which consists of the best members of the original population is referred to as 'selection pressure' (Miller and Goldberg, 1995). To increase the selection pressure the size of the tournament is enlarged. Therefore, the winners, which become the next generation, will have a higher average fitness than the previous population. According to Miller and Goldberg (1995), tournament selection is a good selection mechanism as it is simple to code, easy to implement, robust in the presence of noise and has an adjustable selection pressure.

5.2.6. Fitness Scaling

Fitness scaling is another method used to combat the problems associated with proportional reproduction. One example is linear scaling, a ranking selection described by Baker (1985). The population shown in table 13, is sorted from best to worse as in table 15. This ranking indicates that chromosome 8 is ranked at number one with a fitness of 1.6260, chromosomes 6 and 7 are both in second place with a fitness of 1.1559 and in last place is chromosome 10. This ranking is used to produce a roulette wheel, shown in figure 36, and can be compared with the roulette wheel determined from proportional representation in figure 37.

Chromosome Number	f(x,y)	ranking
1	0.7303	4
2	0.3827	7
3	1.3011	3
4	0.5939	6
5	0.6792	5
6	1.1559	2
7	1.1559	2
8	1.6260	1
9	0.5839	6
10	0.1814	8

Table 15 - Sample Next Generation using Fitness Scaling

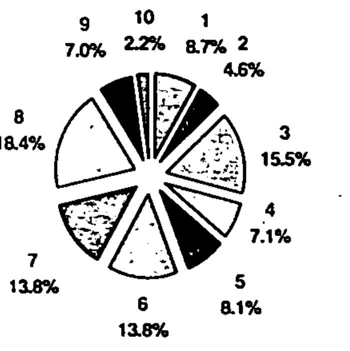


Figure 36 - Fitness Scaling - Population Distribution of Total

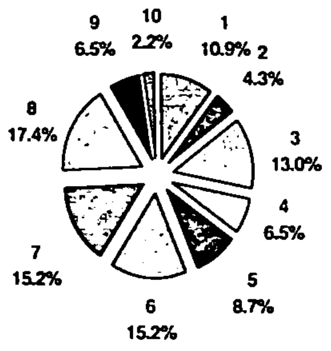


Figure 37 - Fitness of Each Member as a Share of Sum of Population Fitness

If a comparison is made of the share of the population each chromosome receives using fitness scaling (figure 36) to that received using the proportional method (figure 37), it is noticeable that chromosome 1 gained a 2.2% share of the population as it had an 8.7% share of the sum of the fitness of the population, but when using fitness scaling its rank of number 4 gave it a 10.9% share of the population. While this method is criticised for disassociating the fitness function from the underlying objective function, Goldberg (1989) feels that the required link is not based on evolutionary theory and this method does provide a consistent method for offspring selection.

5.2.7. Additional Reproduction Strategies

Additional reproduction strategies are suggested by Bäck *et al* (1992), these are extinction and preservative. In extinctive selection some chromosomes are not allowed to create offspring if they have a zero selection probability, while in preservative selection each population member is guaranteed a probability to produce an offspring. While results of the extinction method are better than those using preservative in a unimodal search space, no improvement is shown in multimodal searches (Bäck *et al*, 1992). Fogarty (1993) has experimented with using a chromosome's age and fitness to determine whether it should reproduce or not as opposed to its rank and fitness. The results suggest in a noisy environment this method outperforms the conventional selection methods.

5.2.8. Crossover

In crossover, pairs of chromosomes exchange genes. From this procedure two new members of the population are produced who replace their parents. It is possible that some members of the population could be crossed more than once, resulting in a new population member very different from its parent. To keep the change to a minimum each member of the population should only be crossed once.

There are a number of ways to apply crossover including one-point, two-point and uniform. In two-point crossover two positions are generated, these are two random numbers between 1 and the length of the chromosome, in this example between one and eight. The genes between these two points are exchanged creating new offspring. In figure 38, chromosome 3 and 4 have been randomly selected to cross at points 3 and 7 and the fitness of the new chromosomes is displayed. The average fitness of these two chromosomes has declined from 0.9331 to 0.9037 a difference of 0.0394, however the fitness of chromosome 3 has improved from 0.7303 to 1.6260, an improvement of 0.8957.

	1	2	3	4	5	6	7	8	
3	0	1	1	0	1	1	1	0	fitness = 0.7303
4	0	0	1	1	1	0	0	1	fitness = 1.1559
3'	0	1	1	1	1	0	1	0	fitness' = 1.6260
4'	0	0	1	0	1	1	0	1	fitness' = 0.1814

Figure 38 - Two-Point Crossover Demonstration

This process will continue until no more than 50% of the population has been crossed. Therefore, with a population of ten, four members of the population will be crossed. Schaffer *et al* (1989) find that two-point crossover gives slightly better results than one-point crossover. If using one-point crossover, only one point is selected to cross, the end of the chromosome will be treated as the second point, so there would be two exchanges between the chromosomes, as opposed to the three in two-point crossover. The example in figure 39, demonstrates one-point crossover, a single position has been selected on the chromosome, three, and the genes are exchanged between that point and the end of the chromosome. In this case the average fitness has increased from 0.9431 to 1.70275 an improvement of 0.7596.

	1	2	3	4	5	6	7	8	
3	0	1	1	0	1	1	1	0	fitness = 0.7303
4	0	0	1	1	1	0	0	1	fitness = 1.1559
3'	0	1	1	1	1	0	0	1	fitness' = 1.98007
4'	0	0	1	0	1	1	1	0	fitness' = 1.42542

Figure 39 - One-Point Crossover Demonstration

A greater distance between the cross points creates offspring that are dissimilar as there has been more disturbance to the chromosome. However the disturbance to the parents' chromosome is even greater when using uniform crossover. In this case, the chromosomes to be crossed are chosen at random from the population but instead of selecting points to cross between, a template is created and this template is a random selection of 1s and 0s of the length of the chromosome. The first offspring is then created using the template. If the template gene for position one is a 0, the gene in that position on parent number 1 is selected for offspring number 1, offspring number 2 gets the gene from parent 2. If the template gene is a 1, the gene in that position on parent number 2 is selected for offspring number 2, offspring number 1 gets the gene from parent 1. This procedure is repeated for each of the remaining genes. Normally the crossover method used remains constant through a GAs run, but to illustrate uniform crossover the remaining cross of the demonstration population will be done using population members 1 and 7 and their offspring will be returned to the population as in figure 40.

1	0 0 1 1 1 0 0 1	fitness = 1.559
7	1 0 1 0 1 0 0 0	fitness = 1.626
Template	1 1 1 0 1 1 0 1	
1'	1 0 1 1 1 0 0 0	fitness' = 1.6260
7'	0 0 1 0 1 0 0 1	fitness' = 0.7837

Figure 40 - Uniform Crossover Example

This sort of random replacement can create better or worse children and the results are all in the 'luck of the draw'.

Sywersda (1989) compared Uniform Crossover with Two and One Point Crossover. The results indicated that Uniform Crossover usually worked best and has empirically been shown as more effective on a number of functional optimisation problems, One Max, Sparse One Max, Contiguous Bits or Lock and Tumbler problem, Exponentially Decreasing Sine, Shekel's Foxholes and the Travelling Salesperson.

5.2.9. Mutation

Mutation works on a single gene of the population and in the given population there are 80 genes. While the rate of mutation applied can vary greatly, 10% will be used, therefore 8 genes have the potential to be mutated. Once a gene is chosen to be mutated it does not automatically switch from 0 to 1 or vice versa, rather a new bit is generated randomly. This means that a mutation can take place without actually altering the string.

The eight randomly chosen genes are {2,11,15,24,45,66,68,75} and the process is demonstrated in table 16.

Gene	Location	Original	Change	New Chromosome	Fitness
2	chromosome 1 gene 2	10111000	1	11111000	0.3827
11	chromosome 2 gene 3	11001101	1	11101101	0.1814
15	chromosome 2 gene 7	11001101	0	11001101	0.3827
24	chromosome 3 gene 8	01111010	1	01111011	1.9017
45	chromosome 6 gene 5	10010011	1	10011011	1.5403
66	chromosome 9 gene 2	00101010	0	00101010	0.257
68	chromosome 9 gene 4	00101010	1	00111010	0.9584
75	chromosome 10 gene 3	11101101	0	11001101	0.3827

Table 16 - Mutation Process

Using both crossover and mutation as a strategy can increase the speed of evolution when compared to using mutation alone (Schaffer and Eshelman. 1993). However crossover will disrupt the schemata more than if mutation only was applied.

5.2.10. Replace Best

The last step is to add the 'best' chromosome of the previous generation to the new population. This is a matter of the users preference as there is always the possibility that any of the previous operators may cause the best member of the previous population to be eliminated. The best could be added to the population to increase the population size, the worse member of the population could be replaced or a randomly selected member could be chosen and replaced. In this example, a randomly choosen chromosome is replaced by the 'best' of the previous generation. The chromosome randomly chosen is chromosome number 5. The new population is shown in table 17.

The total fitness of the population is now 9.441, an improvement of 12.5%. The process will continue generation after generation until the maximum fitness value is reached, which in this case is 2.00.

Number	Chromosome	Fitness
1	11111000	0.3827
2	11001101	0.3827
3	01111011	1.9017
4	00101101	0.1814
5	10101000	1.6260
6	10011011	1.5403
7	00101000	0.7837
8	01001010	1.3011
9	00111010	0.9584
10	11001101	0.3827

Table 17 - New Population for the Next Generation

5.3. Schema Theory

Why do GAs do what they do? To explain this, a few theories have been established, the best known is the Schema Theorem as discussed by Holland (1975) and Goldberg (1989). The basis of this theorem is that the foundation of GAs relies on a binary chromosome representation of solutions and on the concept of schema. Identical segments of genes are evident in some of the chromosomes, and this is called a schema. A schema is a template of genes of a chromosomes which match in all positions other than those marked in the schema by *. For example, if in the sample problem the schema (00111***) is defined there exists two members of the population which match:

5) (00111100)
7) (00111001).

If the schema to be matched is (**101***) then there are four members of the population which match:

1)	(01101110)
8)	(10101000)
9)	(00101010)
10)	(11101101).

The schema (01101110) however represents only one chromosome and the schema (********) represents all chromosomes of length 8 (that is all 2^8 such strings). Every schema matches exactly 2^r possible chromosomes where r is the number of * symbols in a schema. Finally any given chromosome of length x is matched by 2^x schemata. For strings of length x there are in total 3^x possible schemata. In a population of size n, between 2^x and $n*2^x$ different schemata may be represented.

The schema theorem, attempts to illustrate that while the operators, mutation and crossover can disrupt a chromosome, they are not as significant to short and low order schema, and is stated as:

> **Theorem 1 (Schema Theorem)** Short, low-order, above average schemata receive exponentially increasing trials in subsequent generations of GAs (Holland, 1975).

From this, the Building Block Hypothesis suggests that GAs explore their search space by low-order schema which are exchanged during the crossover process.

> **Hypothesis 1 (Building Block Hypothesis)** A Genetic Algorithm seeks near-optimal performance through the juxtaposition of short, low-order, high-performance schemata, called building blocks (Holland, 1975).

In the original population chromosome number 1 (01001010) had a fitness of 0.7373. Through reproduction, this became chromosome 3 in the next generation. It was then chosen for crossover with chromosome number 4 and the resulting offspring improved the fitness of chromosome 3 from 0.7373 to 1.626. The same chromosome was then mutated at gene number 8 to give a final chromosome in the next generation of (01111011) with a fitness of 1.9017, the highest fitness in its generation. Below the original chromosome is compared to its offspring.

Chromosome 1 (Original population).......	(01001010)
Offspring of Chromosome 1.......................	(01111011)

The results indicate that the building block is (01**101*) and this schema apparently contributed towards the development of the best member of the population.

A great deal of research has been performed into aspects of schema theory and how schemata interact and combine during evolution. Two approaches have been discussed in White and Flockton (1995). The first is to develop problems which are difficult for GAs to solve, these are called 'deceptive' as they violate the building block hypothesis in that short low-order building blocks are designed to lead to sub-optimal, longer, higher-order building blocks. A violation of the hypothesis means that GAs will not be able to converge on the optimal solution. While a great deal of effort has been expended to develop these deceptive problems GAs still succeed while many non-deceptive problems can be difficult to solve.

The second approach was to develop problems where the GAs performed as expected (Forrest and Mitchell, 1992; 1993). These, called *Royal Road* functions, add features to the fitness landscape to lead the GAs directly to the global optimum. According to White and Flockton (1995) the Royal Road functions contain a number of building blocks and intermediate 'stepping stones' that result from lower-order schemata. The

Building Block Hypothesis might lead to the assumption that GAs would perform better on functions where there is a clear path via crossover from low order schemata to the optimum (*Ibid.*). Simulations, however, have indicated that GAs may perform worse on these functions *(Ibid.)*. Forrest and Mitchell (1992) followed the generational evolution of a Royal Road function and found what they termed, *genetic hitchhiking*. This occurs when the intermediate 'stepping stones' are so much fitter than the parents that premature convergence may occur, hence slowing down the search (*Ibid.*).

The Walsh-Schema theory is another attempt to analyse schemata. This is a method for describing the dynamics of GAs fitness function. This theory says that as the fitness of F improved, it biased towards partitions with higher order schema as the population evolves (White and Flockton, 1995). Walsh analysis has been used to characterise functions which are either easy or hard for GAs to optimise, but as with Royal Road function, the results weren't always those expected (Mitchell and Forrest, 1992).

Prügel-Bennet *et al* (1994) have applied Statistical Mechanics Formulation to analyse the behaviour of GAs. This method predicts the distribution of energies in the population at each generation. The model developed by Prügel-Bennet *et al* (1994) for a Boltzmann selection mechanism shows that the statistics generated compare favourably with those from a GAs simulation. Unfortunately the probability distribution needs to be recalculated after each crossover and mutation procedure which makes a time consuming task (White and Flockton, 1995).

Additional theoretical research has applied convergence analysis (Ankenbrandt, 1990; Louis and Rawlins, 1992; Qi and Palmieri, 1994), Breeder Genetic Algorithm (Muhlenbein and Schlierkam-Voosen, 1993a; 1993b), and Nonuniform Walsh-schema transform (Goldberg, 1989). Finally Markov Chain Analysis (Goldberg *et al*, 1987; Davis and Principe, 1993) claims that the dependence of each population on its predecessors in the sequence is completely determined by its conditional dependence upon its immediate predecessor population.

5.4. Conclusion

The previous sections discuss GAs, their history, the operators involved and the philosophy of schema theory. Included is an illustrative example of the search process of GAs.

The next chapter will discuss using GAs for the generation of test data. This method of data generation will be compared to random test data generation as discussed in chapter 3 and other adaptive search techniques, Simulated Annealing in chapter 7 and Tabu search in chapter 8.

Chapter Six

Automatic Test Data Generation Using Genetic Algorithms

6.1. Introduction - Previous Research

The use of genetic algorithms to automatically generate test data is not unique. Xanthakis *et al* (1992) used GAs in conjunction with a process entitled maturation to develop a prototype software testing tool, TAGGER. The steps of TAGGER are first to produce a qualitative data flow influence graph which represents the variable in the program, then the generation of test data using GAs, and finally a relaxation process is applied to the test data to access paths which have not been exercised. The procedures involved in TAGGER require a great deal of structural amendment of the code, as well as an element of symbolic execution in the establishment of the elementary path functions required for the relaxation process (Xanthakis *et al*, 1992). However the results of the research, a successful coverage metric of 100% on a small sample PASCAL program, is encouraging for the use of GAs in test data generation.

The automatic generation of test data for ADA programs (Sthammer *et al*, 1992; Jones *et al*, 1995) also used GAs. The emphasis of research was on coverage of the branches in a program and on achieving coverage at the boundary of each subdomain. It was discovered that using the Hamming distance as the fitness function was very effective in attaining coverage. In comparison with random testing, GAs were more successful. GAs required 100 fewer generations. Jones *et al* (1995) used GAs to produce test data sets derived from the structure of the code and its formal specification in Z. This application was applied to the triangle or Trityp problem where GAs proved to be successful in deriving test data sets. According to Jones *et al* (1995), one drawback to this method is the computational effort required.

Schultz *et al* (1995) used GAs to test and evaluate complex software controllers and determined that GAs performed well and that the gain on speed was worth the decline in quality over using an exhaustive testing procedure. Additional research has been performed by Roper *et al* (1995) into using GAs for test data generation, each test data set is accorded a fitness value which measures how many branches are covered by a single test data set. Therefore a test data set which covers the most branches is the goal, however as this alone will not achieve coverage of the code the eventual goal is to achieve a population of chromosomes which together cover the program.

Finally, Chang *et al* (1992) used heuristic rules to create an intelligent test data generator for branch coverage of a test function. The framework of their system consists of a parser/scanner to instrument the input source code and produce information on the program structure; a test case generator that produces test cases to target specific branches from the code structure; a test case analyser runs the test data sets and records coverage. These last two stages loop until a satisfactory coverage level is attained, and finally a report generator which gives the user statistics, coverage metrics and test data sets for the function under test.

Chang *et al* (1992) used four heuristics were used for the generation of test data. The first was fixed percentage modification, which changed each variable by a percentage from the previous generation's best test case. The second method was random modification, which used a random percentage of change from the best test case. The third method, is entitled 'modification based on condition constants used', the constants referred to are those which appeared in the conditional statement of the code which is being covered. This method showed an improvement over random modification (Chang *et al*, 1992). The remaining approach used was boundary computation. An attempt was made to establish the boundary that separates the true and false values of the condition. The best test data set was then modified to find test data through symbolic evaluation which straddles the line of true and false. Of the four methods the last heuristic, which uses symbolic evaluation, is the most successful (Chang *et al*, 1992).

The research described in this chapter aims to generate test data to satisfy the coverage metrics for branch, statement, LCSAJs and path testing. The goal is to measure the structural coverage of a function under test, while keeping the amount of disturbance to it to a minimum. This means that only a small amount of recoding should be performed on the function under test so the test is on the original code, not what has been added as analysis routines. Therefore the testing will be performed as black-box testing with slight structural modifications and a fitness function is required which will help to achieve this goal.

6.2. The Fitness Function

There are a number of ways this can be done and a brief review of the literature reveals methods which have been used for similar problems. Sthammer *et al* (1992) took as a measurement the distance a test data set was from a conditional border and rewarded these test data sets, looking at one branch at a time, however this requires more reworking of the actual test code and moves away from black-box testing. Roper *et al* (1995) used the length of the path achieved through the code and rewarded those test data sets which covered more branches, this method however encouraged finding longer paths and discouraged shorter and equally as important paths. Schultz *et al* (1995) discusses four methods for judging the fitness of test data sets for software controllers, the first was border condition similar to that method used by Sthamer *et al* (1992), dismissed for its high computational costs; the second method was to measure the performance against actual performance, but this method requires a detailed explanation of expected responses, the third method was based on the likelihood and severity of the fault, this requires probability estimates of the fault modes and many of the faults may be of the same probability but of equal importance. The final method which was then applied to the test problem by Schultz *et al* (1995) was to search for scenarios which were interesting, interesting referred to test data sets which produced failures or faults which did not cause a failure, using this method they felt that classes of weaknesses were discovered as opposed to single weaknesses which would be patched specifically.

In an attempt to remain close to black-box testing the fitness function required in the tests performed for this thesis needed to measure fitness of a test data set while encouraging coverage of unexplored areas of the test code. A visualisation of the fitness landcape of the search space would indicate a flat plateau, as no one test data sets has a greater fitness than any other as there is no global optimum. (Complete coverage of the code is the global optimum). As a path is found it is rewarded for achieving coverage, however subsequent test data sets which find the same path should be penalised as once found there is no need for more test data which will cover that path. Therefore a test data set which finds an already covered path should receive a reduced fitness.

Using this idea, a visualisation of the search space might show large peaks rising out of the plateau which then diminish in size over time, until very little plateau exist between hills of varying sizes. When no plateau exist then the code has been covered (global optimum has been reached). A numerical representation of this concept is to count the number of times a path is accessed, using the inverse of this will give a greater fitness to new paths which will decrease over time as the path is found again.

$$FITNESS = \frac{1.0}{Count + 1.0} * 100000.00 \qquad \textbf{(1)}$$

This proportional reduction in fitness is shown in figure 41. The first time a path has been found its fitness is 50000, however when it has been found 100 times its fitness has reduced to 990. This process will continue, the fitness of the path and the test data set which accesses that path, continues to decline until the end of the search.

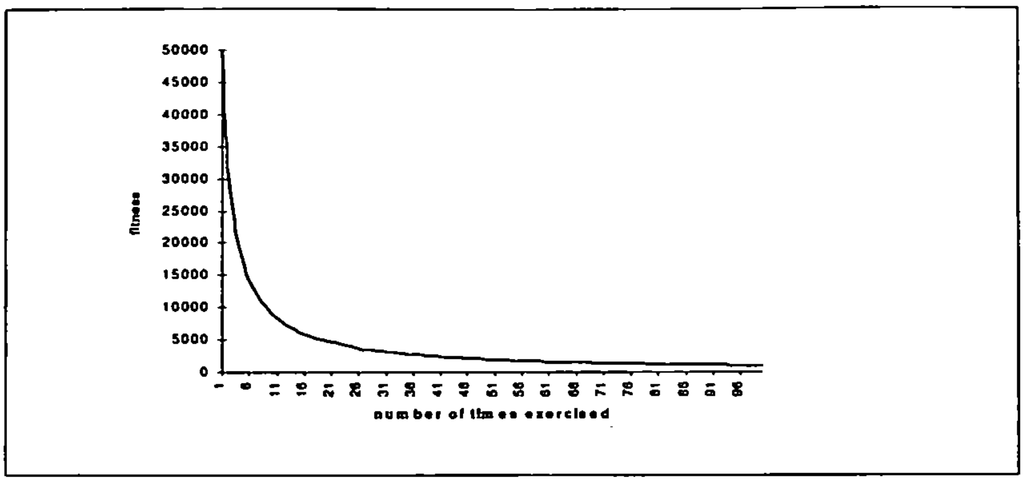


Figure 41 - The Fitness of an Exercised Path, as it is Accessed Subsequent Times, This Demonstrates How the Fitness Declines

This same concept needs to be applied to the coverage of branches and LCSAJs. A single path through the code returns many branches and another path may differ from the first by only one branch. In this case the search will stop when each of the individual branches and LCSAJs have been covered. Therefore when the test data is generated to achieve coverage of branches or LCSAJs a different fitness strategy is required. There are two options for measuring this fitness, the first is to add up the number of times each of the branches and LCSAJs have been exercised and then use this value as *Count* as in (1); or to take the smallest value, *Smallest_Count*, for all the counts (all the branches access times), as in (2).

$$FITNESS = \frac{1.0}{Smallest\_Count + 1.0} * 100000.00 \qquad (2)$$

Using method (2) a test data set is rewarded for finding at least one new branch or LCSAJ. This method will be used for test data generation for coverage of branches and LCSAJs, while (1) will be used for path testing.

6.3. An Illustrative Example of Determining Fitness

In order to determine the fitness, the parsed function, as established from chapter 3, will be attached as the code file to the GAs. This function is given a generic name, TestCode, which can be applied to any function under test. One additional adjustment to the parsed function is the return from code under test of a string containing the path exercised information (*i.e.* 1T2T3F), this is called **'tested'**. This is used by the fitness function, as established in section 6.2, to establish the coverage and fitness value. Therefore the first line of the function under test will read as follows:

TestCode(int x, int y, int z, char tested[20]).

The character array **'tested'** will contain the string of paths exercised within the test function. This string is returned to the fitness calculation function which analyses it to determine which path/branches/LCSAJs have been exercised. In the case of branch testing, one by one each available branch is compared as a substring to the path, **'tested'**, as follows:

if (strstr("1T",tested) == 0).

If the equation evaluates to zero then that substring exists in the exercised path. The branch to be tested has a corresponding integer value (*Count*) which indicates how many times that particular branch has been found. If *Count* is zero then this branch has been exercised for the first time, if it is greater than zero then that branch has already been exercised. As each branch is tested against the string, *Count* of some branches will be higher than others due to previous test sets. A low value of *Count* shows that a particular test data set has exercised a new area of the function under test and should be duly rewarded in its fitness as in (2).

The value of *Smallest_Count* is the smallest value of the set of *Count* established when checking the branches and LCSAJs exercised against the available branches and LCSAJs in the string **'tested'**. Therefore, if a test data set exercises five different branches, the value of *Smallest_Count* will be the value of *Count* for the branch that has been exercised the least. Therefore a test data set which finds an untested branch will have a fitness of 50000.0, while a test data set with values of *Count* ranging from 5 to 20 will have a fitness of 16666.67.

6.4. Remembering Unique Data Sets

In the sample code shown in chapter 3, the search space is of size 9,261, that is there are 9,261 different combinations possible of the three variables x,y and z. These three variables each are within the range [0..20].

To keep run-time to a minimum, a function should not be tested with the same test data. Therefore a record needs to be kept of each test data set and the path it has exercised. To save memory the three variable combination is converted to a single integer, such that the variables {0,0,0} is position 0 in the array titled 'unique', and {20,20,20} is in position 9,260. The information contained in the array is the path string returned using that test data, which is then used to increment the branch coverage levels. In figure 42 is an illustration of the type of array and the information contained in the array. The path used in this example is the actual path exercised by the corresponding test data.

char unique[9261][20]

variable	value	data set
unique[0]	"1T2F3FL0"	{0,0,0}
unique[740]	"1F3FL1L2"	{1,14,5}
unique[9260]	"1F3FL1L2"	{20,20,20}

Figure 42 - Declaration of Array 'Unique' and the Information Contained in Array

Therefore if a test data set {1,14,5} was generated the corresponding array position is 740.

'x' value is '1'	=	$1 * 21^2$	=	441
'y' value is '14'	=	$14 * 21$	=	294
'z' value is '5'	=	$5 * 1$	=	5
			=	740

As this is the first time this test data set has been generated the array position 740 is blank and the test data needs to be run on the test code and **'tested'** (the path exercised) will have to be determined. The value of **'tested'** is the path "1F3FL1L2" is placed in the array position 740 and also used to determine the fitness of the test data set. If this same test data set is again generated the array position will be accessed and upon finding the string "1F3FL1L2" it will not be necessary to run the test code as the string contained in the array position is used as **'tested'**. This reduces the amount of run time as the test code is not required to be run on the same data again and again, which is of great benefit when the test code contains complicated and time consuming calculations. To reduce the test time while achieving a high test effectiveness is what it is hoped will be attained.

6.5. The GAs Process

The algorithm shown in figure 43 and figure 44 is an addition to the algorithm shown in chapter 3. This algorithm demonstrates the complete steps taken by GAs to generate test data for branch coverage. The submodule fitness illustrates the steps taken following the discussion in sections 6.2 through 6.4. This shows the steps to be taken if the test data set has been generated previously or if it is a fresh test data set. Additionally it demonstrates the simple process involved in determining *Smallest_Count*. This value of *Smallest_Count* is used in the fitness function (2) and this value is used as the fitness of the test data set for the GA process.

```
                The GAs Process
1        Generate a random population of binary digits
2        calculate population fitness using submodule FITNESS
3        while (branch coverage < total branches) or (generation < Some_Value) do
3.1          reproduce population
3.2          cross population members
3.3          mutate population members
3.4          calculate population fitness using submodule FITNESS
3.5          increment generation by 1
4        end loop
5        record branch coverage for manual review
```

Figure 43 - Algorithm for GAs for Test Data Generation

```
         submodule FITNESS
1          convert the binary strings to integers
2          convert test data set value to single integer
3          if unique[integer] = empty string
3.1            send test data set to function under test
3.2            get path exercised 'tested'
3.3            set unique[integer] to 'tested'
3.4        else
3.5            get 'tested' in unique[integer]
3.6        end if
4          loop while comparing path exercised with branch data
4.1            if substring of path = branch/LCSAJ
4.1.1              update count
4.1.2              if count < Smallest_Count
4.1.2.1                set Smallest_Count to count
4.1.3              end if
4.2            end if
5          end loop
6          return fitness as value of Smallest_Count using formula (2)
         end module
```

Figure 44 - Algorithm for Fitness Function to GAs, Figure 43, for Test Data Generation

The following sections will review the processes specified in figure 43 and figure 44 in detail.

6.5.1. Random Population Generation

The first step is to generate a random population of ten members of length 15, such that

$$2^4 < 20 \leq 2^5,$$

and these are listed in table 18. Each of the three input variable is represented by five binary digits.

1	101011000010011
2	000111010000110
3	110011110010000
4	111000001011101
5	001100011010001
6	011110101001110
7	001001011110110
8	001011100111110
9	011100010101111
10	111000011011001

Table 18 - Random Binary Population

6.5.2. Calculate Fitness Using Submodule FITNESS

The first step is to convert these binary digits to their respective integer values as follows in table 19. In turn each combination is converted to a single value, position, as in column 5 of table 19. Each position in the array 'unique' is then checked to see whether it contains an empty string or the string containing the path exercised in its test through the code. In this first generation unique[6396] will be an empty string, the test data {14,10,12} is then applied to the test code and the associated path exercised is entered in column 6 of table 19. Population member number 8, which is a duplicate of number 7, activates the else portion (step 3.3) of the algorithm in figure 43 and the test data is not sent to the test code, but the path exercised is automatically generated from the array position.

	x	y	z	Array Value	Path Exercised
1	14	10	12	6396	1F3FL1L2
2	2	13	4	1159	1F3FL1L2
3	16	18	10	7444	1F3FL1L2
4	18	2	19	7999	1F3FL0
5	4	4	11	1859	1T2F3TL1
6	10	6	8	4544	1F3FL1
7	3	15	14	1652	1F3FL1L2
8	3	15	14	1652	1F3FL1L2
9	9	3	10	4042	1F3FL1
10	18	4	16	8038	1F3FL1

Table 19 - Integer Values of Binary Digits and Array Value

Each path is then compared to the branches and LCSAJs specified for this test code as determined in the parsing stage, described in chapter 3. These are listed in table 20 (a) and (b) respectively.

	Branches	times exercised
1	1T	1
2	1F	9
3	2T	
4	2F	1
5	3T	1
6	3F	9
7	L0	
8	L1	9
9	L2	5

(a)

	LCSAJs	times exercised
1	3TL0	
2	3TL1	1
3	3TL1L2	
4	3FL0	1
5	3FL1	3
6	3FL1L2	5
7	1T2T	
8	1T2F	1
9	2T3T	
10	2T3F	
11	2F3T	1
12	2F3F	
13	1F3T	
14	1F3F	9

(b)

Table 20 - Branches and LCSAJs to be Exercised in Test Code and Number of Times Exercised Using First Generation (Random Population)

As each path exercised is compared to the branches and LCSAJs requiring testing, the number of times each has been found by a piece of test data is recorded in column 3 of each table, 20(a) and (b), such that for chromosome one with path exercised {1F3FL1L2} each branch {1F, 3F, L1, L2} is incremented by 1 and LCSAJ 6 and 14 are incremented as well. Note that the LCSAJ 5 is not incremented, as this input exercises both L1 and L2, not L1 singularly. The fitness for chromosome 1 will be the value of the lowest count of found times. As this is the first time testing the code the value of each branch or LCSAJs found will be 1, therefore the returned fitness of chromosome 1 is 50,000. The fitness for the entire population is shown in table 21.

	Fitness Value
1	50000
2	33333
3	25000
4	50000
5	50000
6	50000
7	20000
8	16666
9	33333
10	25000

Table 21 - Fitness of Population

In this first generation it can be noted that there are four best members of population. In successive generations the fitness of the best members shown above begins to decline as more and more test data sets exercise these same areas of code.

6.5.3. The While Loop

The number of branches and LCSAJs successfully exercised by the initial population is recorded (14) and compared to the total number to be exercised(23). It is possible that some of the branches or LCSAJs are unobtainable and the GA search could continue indefinitely. To avoid this the search will stop when either all LCSAJs and branches have been exercised or when the generation number equals 1000. In this example all possible LCSAJs can not be found, therefore the total number of LCSAJs and branches to be exercised is 22. Once the search is activated the next generation is generated by the process described in section 5.2.3. The roulette wheel for the current population looks as in figure 45. The fitness of the population is 353332 and shown in table 22 is their portion of the total fitness of the population which is used to calculate the next population.

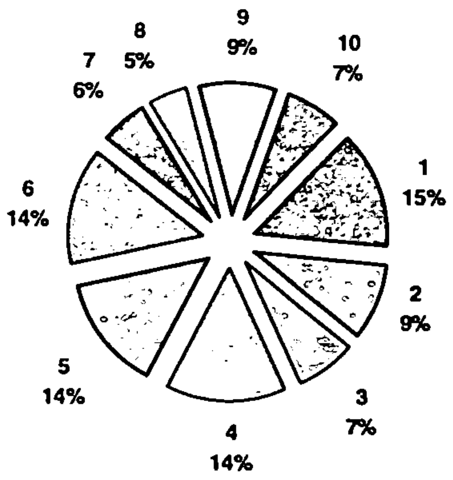


Figure 45 - Roulette Wheel of Population Fitness

Member	Portion to
1	53000
2	83333
3	108333
4	158333
5	208333
6	258333
7	278333
8	294999
9	328332
10	353332

Table 22 - Portion of Populations Fitness Assigned to Each Member

The next population is determined by a random generation of 10 numbers between 0 and 353332 and these numbers are {157326, 128001, 16853, 232967, 214178, 129981, 29186, 160070, 151688, 59008}. This gives the new population as shown in table 23.

	Population Generation Two
1	111000001011101
2	111000001011101
3	101011000010011
4	011110101001110
5	011110101001110
6	111000001011101
7	101011000010011
8	001100011010001
9	111000001011101
10	000111010000110

Table 23 - New Population (Generation 2)

6.5.4. Crossover and Mutation

At this stage 50% of this new population is crossed and 1% of the genes mutated on a random basis. The resulting population is as follows in table 24.

1	111000001011001
2	110110111011101
3	101011100010011
4	010000001001110
5	011110111001110
6	111000100000110
7	101011000000011
8	000000001000001
9	111100001011101
10	001011011011101

Table 24 - New Population (Generation 2) after Crossover and Mutation

None of the new population, when converted to integers as shown in table 25, has already been tested and the average fitness of the population is now 224683. The coverage of all branches and LCSAJs has now increased to 18 leaving only 4 more to be exercised as displayed in table 26 (a and b), as LCSAJ 7 is infeasible.

	X	Y	Z	Array Value	Path Exercised	Fitness
1	14	1	16	6211	1T2F3FL0	50000
2	17	9	19	7705	1F3FL1L2	14286
3	14	15	12	6501	1F3FL1L2	12500
4	5	1	9	2235	1T2F3FL1	33333
5	10	9	9	4608	1F3FL1L2	11111
6	18	5	4	8047	1F3FL1	16667
7	14	10	2	6386	1F3FL1L2	10000
8	0	1	1	22	1T2F3TL0	50000
9	19	3	19	8461	1F3FL1	14286
10	3	14	19	1636	1T2F3TL1L2	12500

Table 25 - Integer Values of Binary Digits and Array Value (2nd Generation)

	Branches	times exercised
1	1T	5
2	1F	15
3	2T	
4	2F	5
5	3T	3
6	3F	17
7	L0	2
8	L1	17
9	L2	11

(a)

	LCSAJs	times exercised
1	3TL0	1
2	3TL1	1
3	3TL1L2	1
4	3FL0	2
5	3FL1	6
6	3FL1L2	9
7	1T2T	
8	1T2F	5
9	2T3T	
10	2T3F	
11	2F3T	3
12	2F3F	2
13	1F3T	
14	1F3F	15

(b)

Table 26 - Branches And LCSAJs to be Exercised in Test Code and Number of Times Exercised for Second Generation

6.5.5. LoopEnd

At the conclusion of the loop a listing is made of all the statements and LCSAJs exercised for a manual review as specified in the algorithm in chapter 3, section 1. To calculate an average number for the amount of unique test data required, the process was run 1000 times. The program stopped when a satisfactory TER of 95% was attained. The average number of unique data sets which were generated by the runs was 341.62 or 3.7% of the total search space, and these were generated over an average number of generations, 685.20, an average of 0.50 new data sets each generation. The minimum number of data sets required by a run was 14, the lowest possible number of data sets generated prior to coverage is 7. The maximum number of data sets required by a run was 1595, the maximum possible number is 9254 as discussed in chapter 3. Table 27 summarises the results.

Average unique data sets 1000 runs GAs	341.62
Standard deviation over 1000 runs	291.59
Average generations required	685.20
Minimum unique data sets required	14
Maximum unique data sets required	1595
Average % of search space searched	3.7%
Average new data sets/generation	0.50

Table 27 - Results of Function Under Test Using GAs Over 1000 Runs For LCSAJs and Branch Testing

In figure 46 is a frequency distribution for the 1000 runs which shows the amount of unique test data required for each run. The peak of the data is between 150-200 unique data sets with 123 runs completing within this range, while the average number of data sets is 341. This graph will be used to compare search techniques and to illustrate where the results for each search technique falls.

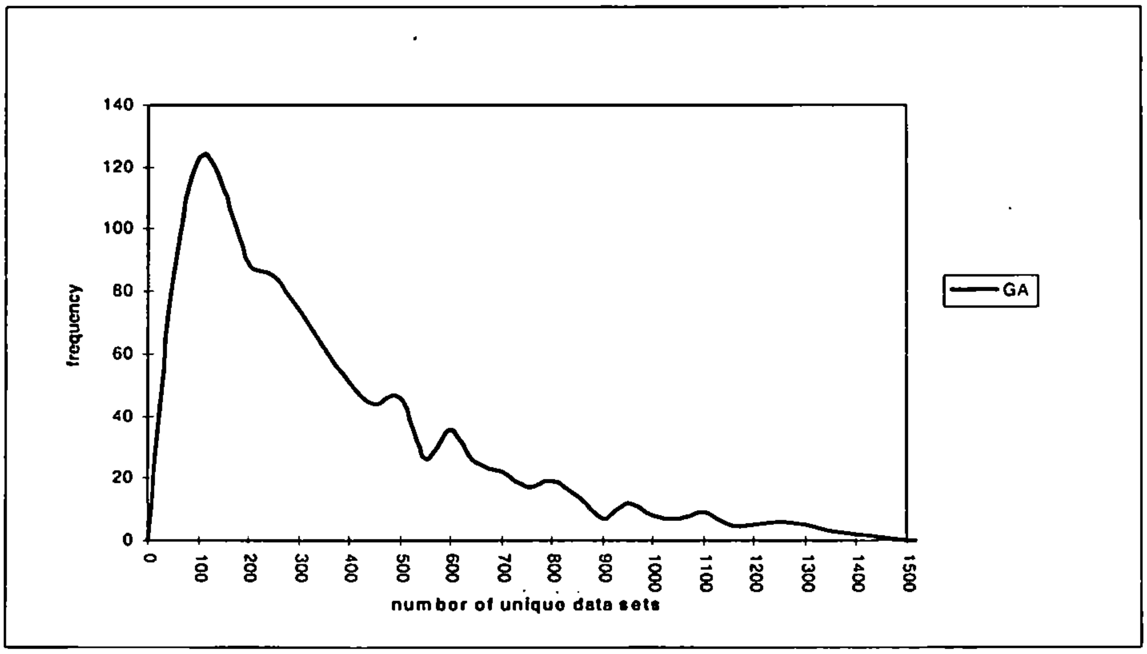


Figure 46 - Frequency Distribution of GAs over 1000 Runs For LCSAJs And Branch Testing

6.6. Comparison to Random Testing

Test data was generated randomly for this same simple function under test. In random testing, data is again generated until for all branches and LCSAJs a satisfactory TER is reached. The same type of count was maintained of the number of unique data sets required and the results are in table 28. These results show that an average of 994.58 unique data sets were required in comparison to the 341.62 by GAs. There was an 89% chance that a newly generated test data set had not been previously applied, compared to a 50% chance for GAs. The minimum amount of test data generated before a successful completion of a run was 14, the same as with GAs, but the maximum was 4958 as opposed to 1595 for GAs. A frequency distribution is shown in figure 47, the highest concentration again fell between 100-150 data sets with a total of 59 runs completing, compared to 123 runs completing in this range for GAs.

Unique data sets 1000 runs Random generation	994.58
Standard deviation	881.13
Average generations required	1113.0
Minimum unique data sets required	14
Maximum unique data sets required	4958
Average % of search space searched	10.7%
Average data sets/generation	0.89

Table 28 - Results of Function Under Test Using Random Generation over 1000 Runs for LCSAJs and Branch Testing

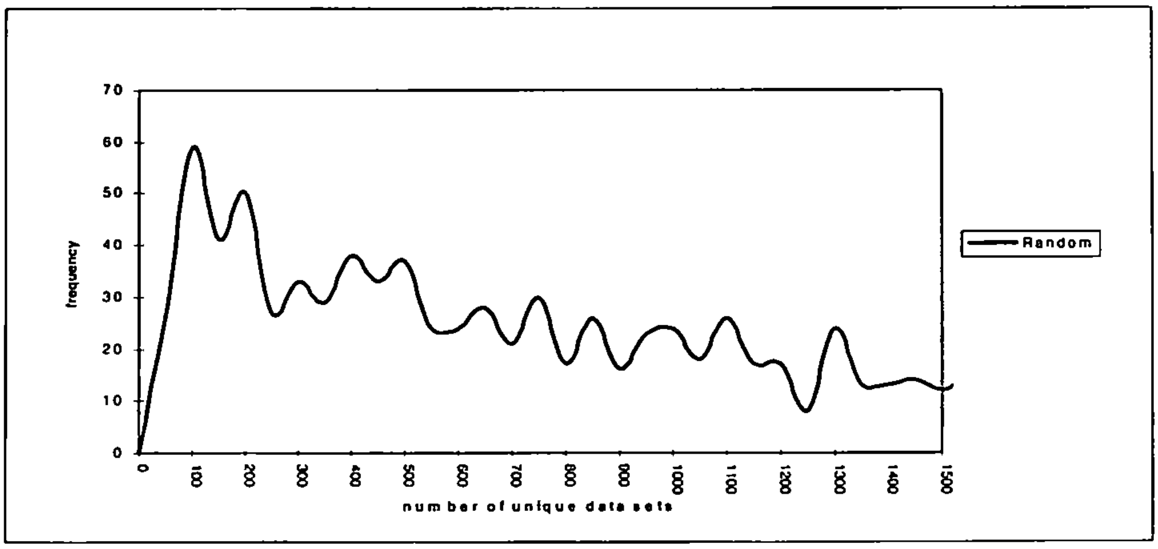


Figure 47 - Frequency Distribution of Random Generation for LCSAJs and Branch Testing

Finally in figure 48 is a graph showing the frequency distribution for both GAs and random generation. This graph show that while many of the random runs satisfy the TER quite early, there are more runs which take more test data than GAs. The GAs however, have a very quick high peak with a gradual slope as there are fewer runs with higher amounts of test data sets, until finally the line finishes at the highest run amount of 1595.

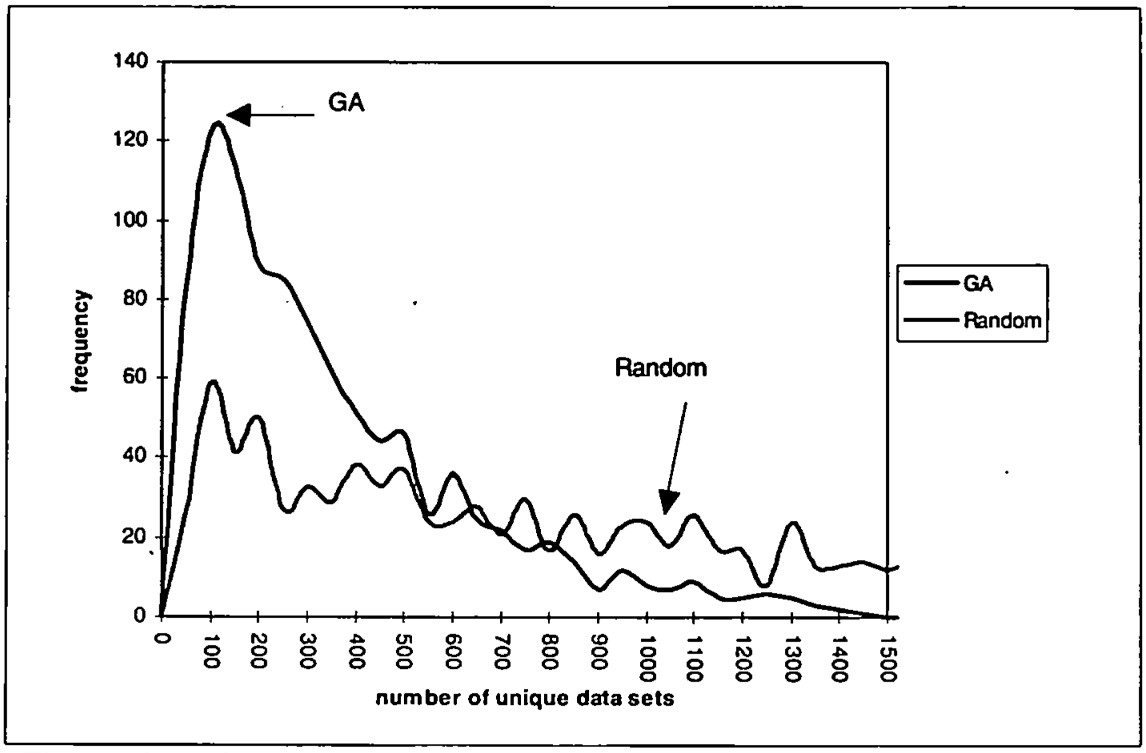


Figure 48 - A Comparison of Frequency Between GAs and Random Test Data Generation for Sample Function under Test for LCSAJs and Branch Testing

6.7. Path testing

It has been stated that path testing, where test data is used to attempt to exercise every path in the function in its entirety, can be a very time consuming practice. It may however, be interesting to note how GAs and random testing perform in testing all the paths of this sample function. There are as stated in chapter 3, 15 feasible paths through the function. Path testing aims to find all those which are listed in table 29 with the corresponding number of combinations in the search space which exercise the path, and its percentage of the total search space. For all 15 paths to be exercised at least 15 test data sets must be generated. The maximum number of data sets possible before coverage is 9258 as there are some paths which can only be exercised by a very small pool of test data.

Path	Times	%
1T2F3FL0	110	1.19%
1T2F3TL0	4	0.04%
1T2F3TL1	116	1.25%
1T2F3TL1L2	320	3.46%
1F3FL0	692	7.47%
1T2T3TL0	4	0.04%
1T2T3FL0	53	0.57%
1F3FL1	1870	20.19%
1T2T3TL1	116	1.25%
1T2T3FL1	295	3.19%
1F3FL1L2	4928	53.21%
1T2T3TL1L2	320	3.46%
1T2T3FL1L2	37	0.40%
1T2F3FL1	355	3.83%
1T2F3FL1L2	41	0.44%
Total	9261	100.00%

Table 29 - Feasible Paths through Function Under Test, Number of Occurrences of Each Path within Search Space and Percentage of Search Space

In table 30 is the comparison of GAs to random test data generation for the generation of test data for path testing. The results indicate that Random Testing requires 2718.92 unique data sets, and examined, as with branch and LCSAJ testing, 27% of the search space and GAs require 529.18 and a search space viewed of 5.7%. The minimum amount of test data generated by a run was 78 for GAs and 199 by random generation. These amounts are above the minimum possible of 15. Random testing however had one run which didn't achieve coverage until it had generated 7135 new test data sets, very close to the figure 9258, which is the maximum which could be generated prior to coverage. In comparison the run of GAs which required the most test data required 1952 data sets or 21% of the possible amount. Figure 49 is the frequency distribution for both random generation and GAs. The peak for GAs is very pronounced and steep between 350-400 data sets with 94 runs completing at this point. Random generation has a number of peaks of height 18, at 1400-1450, 1900-1950 and 2300-2350, the highest point, 19 runs is in the range 3350 to 3400. This illustrates that GAs terminate with a small amount of unique test data required, while the random generation amount required fluctuates over a larger area.

	GAs	Random
Average unique data sets 1000 runs	529.18	2718.92
Standard deviation over 1000 runs	279.30	1471.34
Average generations required	1072.35	3501.47
Minimum unique data sets required	78	199
Maximum unique data sets required	1952	7135
Average % of search space searched	5.7%	29.36%
Average new data sets/generation	0.49	0.78

Table 30 - Comparison of 1000 Runs of GAs Vs. Random Generation for Path Testing of Sample Function Under Test

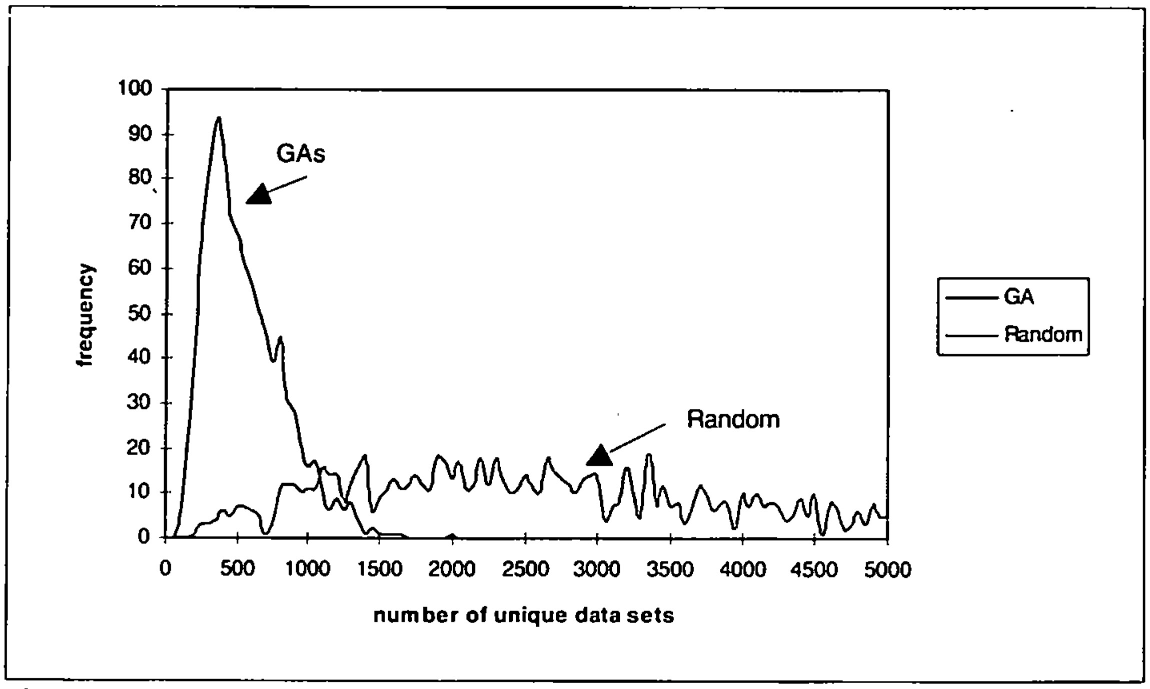


Figure 49 - A Comparison of Frequency between GAs and Random Generation for Sample Function Under Test for Path Testing

6.8. Conclusion

This chapter demonsytrates the use of GAs for the generation of test data. Details are given on the method used to both generate and apply the data to the module under test. Included is an explanation of ho set unique[integer] to 'tested'w rerunning the test code with identical test data can be avoided, which is necessary due to the potential for long run-times. There is a demonstration of how GAs perform over 1000 runs in comparison to random test data generation for the requirement of unique test data sets. This illustrates that GAs on average require less new test data than random generation for this small sample function. Results from GAs show a 66% improvement over random generation for LCSAJs and branch testing. This chapter concludes with a comparison of path testing for random generation and GAs. Even though the sample function under test is a small program, results indicate that GAs continue to outperform random generation by 81%.

In the chapters 7 though 10, GAs and random testing will be compared to other adaptive search techniques. GAs will again be used to generate test data for a suite of general test problems in chapter 9 and in chapter 10 a program which optimises tax payable for companies is used to test the methods in a 'real' environment. These results will be compared to those obtained using Simulated Annealing discussed in chapter 7, Tabu search, chapter 8, and random test data generation.

Chapter Seven

Simulated Annealing

7.1. What is Simulated Annealing?

Many local search algorithms terminate at local optimum and it is difficult to determine how far these results may be from the global optimum (Johnson *et al*, 1985). The quality of the local optimum usually depends on the initial choice of the starting point but there is no specification for choosing this. However, local search algorithms are generally applicable and flexible, requiring only a search space, the fitness of a given solution and a direction in which to search. There are a number of methods which will alleviate problems associated with local optima, the first is to use a large number of starting points and the second is to expand the local search space by introducing a more complex neighbourhood structure. A third is to amend the method used for allowing good neighbours, by accepting an increase or a limited decrease in the fitness. Simulated Annealing uses this third alternative. It has also been referred to as Monte Carlo annealing (Jepsen and Gelatt, 1983), probabilistic hill climbing (Romeo and Sangiovanni-Vincentelli, 1985), statistical cooling (Aarts and Van Laarhoven, 1985; Storer *et al*, 1985) and stochastic relaxation (German and German, 1984).

In physical terms, annealing is the process of toughening (glass or metal) by heating to high temperatures quickly and then cooling slowly (Aarts and Korst, 1990). This process consists of two steps, as described by Kirkpatrick, Gelatt and Vecchi (1982;1983).

- Increase the temperature of the heat bath to a maximum value at which the solid melts.
- Decrease carefully the temperature of the heat bath until the particles arrange themselves in the ground state of the solid.

At the liquid phase the particles arrange themselves randomly, while at the ground state the particles are arranged in a highly structured lattice. This ground state however, is only reached if the maximum temperature is suitably high and the cooling is done at a regulated speed. Otherwise what results is a meta-stable state (Aarts and Korst, 1990).

7.2. Annealing to Simulated Annealing

Simulated Annealing was independently introduced by Kirkpatrick, Gelatt and Vecchi (1982:1983) and Cerny (1985) to mimic this thermal process for obtaining low energy states of a solid in a heat bath. Aarts and Korst (1990) describe the states of the particles as the solutions in the search space, and the energy required to produce these particles as the cost of the solutions. The temperature to which the particles are heated and to which they are cooled is defined as a control parameter. These concepts, combined with a cooling schedule, are Simulated Annealing (Aarts and Korst, 1990). A cooling schedule should identify the following information:

- a sequence of values of the control parameters
    - an initial value of the control parameter C_0
    - a decrement function for decreasing the value of the control parameter
    - a final value of the control parameter specified by a stop criterion
- a finite number of transitions, L_0, at each value of the control parameter, C_0

The aim of Simulated Annealing is to produce an optimum by changing the initial starting point over a fixed period of time. Each new solution is then selected in turn to be annealed based on its fitness. If its fitness is better than the previous solution it is automatically accepted, but if it is worse it may be selected according to the probability distribution:

$$P_c\{accept(S_{new})\} = \exp^{\frac{f(S_{new})-f(S_{old})}{C_0}} .$$

According to Dowsland (1993) the period of time at the middle of the cooling schedule produces the best results. At the beginning the temperature is so high that most of the new solutions are accepted, which could give results no better than random search. If the temperature were to be lower at the start little change would be seen. Therefore, most of the results are determined in the middle range of the cooling schedule. This high rate will allow acceptance of a large number of new solutions, this figure will reduce over time by 5% this reduction figure is referred to as the cooling schedule. The rate of C_0 would eventually coverage to zero, although it is not necessary to reduce this figure to zero according to Dowsland (1993) as C_0 lowers the probability of accepting any uphill move will be indistinguishable from zero. In section 7.5 there is a comparison of the probability acceptance rates to determine how a higher or lower value of C_0 will affect the results.

The mechanics of Simulated Annealing can be modelled using Markov chains, as the conditional dependence on the sequence history of each new solution in the sequence is equal to its conditional dependence upon its immediate predecessor (Davis and Principe, 1993).

7.3. The Simulated Annealing Algorithm

This section will discuss each step of Simulated Annealing in respect to the sample search used in chapter 5, which demonstrated the use of a GA to maximise the fitness function:

$$fz = 1 + \cos\sqrt{\frac{(x^2 + y^2)}{5}}$$

In figure 50, the algorithm for Simulated Annealing is given.

Simulated Annealing Algorithm

```
1           create initial solution ($S_{old}$)
2           initialise $C_0$, $L_0$
3           while (not stop_criterion)
3.1             for (loop=0;loop<$L_0$; loop++)
3.1.1               Generate $S_{new}$ from $S_{old}$
3.1.2               if (fitness($S_{new}$)$\geq$ fitness($S_{old}$))
3.1.2.1                 then $S_{old} = S_{new}$
3.1.3               else
3.1.3.1                 if ( $\exp^{\frac{f(S_{new})-f(S_{old})}{C_0}} > random[0..1]$ )
3.1.3.1.2                   then $S_{old} = S_{new}$
3.2             end for loop
3.3             $C_0 = C_0 * 0.95$
3.4             $L_0 = L_0 + 1$
4           end while loop
```

Figure 50 - Simulated Annealing Algorithm

For this problem, the initial starting point S_{old} is a single binary string of length 8. L_0 will therefore be the value 8, which is the length of the string of binary digits which represents the integers, x and y. The value of C_0 is the equation

$$C_0 = \frac{-d}{t}.$$

The value of C_0 can be described as the drop in fitness (-d) acceptable over time (t) using the logarithmic scale such that

$$\frac{-d}{t} = \frac{-10}{\ln(0.50)} = 14.427$$

This will give a sufficiently high temperature to begin, allowing a greater chance of acceptance of those strings who are worse than S_{old}. This rate will decline over time by 5% each cycle as discussed in the previous section.

S_{new} is created from S_{old}, from S_{old} by mutating a number of bits in the binary string, in this case mutation will be two bits each time. By mutating two bits of the original string a new string is created as shown below,

$$\mathbf{10010010} \rightarrow \mathbf{10010100}$$

S_{old} evaluates to the integer digits {1,-4} and has a fitness of 1.68, S_{new} evaluates to {4,-2} with a fitness of 1.62. In some search methods S_{new} would be rejected and the process would continue with S_{old}. However in Simulated Annealing there is a chance that S_{new} may still replace S_{old} using the calculation specified:

$$\exp^{\frac{1.62-1.68}{14.427}} = 0.996$$

A real number between 0 and 1 is then generated, if that number is less than 0.996 then S_{old} is replaced by S_{new} , and the process continues until the loop is completed. This means that there is a chance the optimum could be reached and then lost before the loop is completed. At the end of the loop the value of C_0 (temperature) is reduced by 5%, as discussed above, and the time value L_0 is increased by 1, L_0 stops incrementing when a selected value, in this case 80, is reached. The reduction to C_0 means that as the number of generations increase, there is less of a chance for an S_{new} with a fitness less than S_{old} to be accepted. The value of S_{old} is then checked against the stopping_criterion which is a maximum fitness of 2.0. If $S_{old} = 2.0$ the process stops, if S_{old} is less than 2.00 the process continues.

7.4. Simulated Annealing for Test Data Generation

How will Simulated Annealing compare to using GAs for the generation of test data? There has been no literature on the use of this technique for test data generation, but there has been some work done on the comparison of GAs and Simulated Annealing. Thornton (1994) compared the two adaptive search techniques in determining feasible engineering designs. Four designs were attempted, these are an aero engine, mobile arm support, a bearing and a spring. In three out of the four designs, Simulated Annealing outperformed GAs and Thornton (1994) accredited this to the ability to represent constraints in the annealing process. Park and Carter (1995) compared the two techniques on MAX-CLIQUE, the problem of finding the size of a maximum clique in a graph. MAX-CLIQUE is an NP-Complete problem which has also been proved as NP-hard. Results indicated that there was no difference in time taken between GAs and Simulated Annealing in a simple version of this problem.

The reason Simulated Annealing was attempted is the high computational costs of maintaining a population in GAs and in the larger problem this indicates that the cost per interaction is much less for Simulated Annealing.

Simulated Annealing is therefore compared to GAs and random testing for the generation of test data. The starting point is one binary string with a length of 15. For the function under test the same fitness formula is used and the procedure will conclude when an acceptable TER has been reached. In the first example, the TER is 95% for branches and LCSAJs, and in the second example for path testing it is 58%.

7.4.1. Test Data Generation for LCSAJs and Branches

This is the same problem used in chapter 6. The goal is to achieve a satisfactory TER for branches and LCSAJs with a small amount of unique test data. The results are in table 31. Simulated Annealing has achieved coverage after an average generation of 348.26 unique test data sets, this compares closely to the results from GAs, 341.62. Simulated Annealing required 3.8% of the search space to be searched while GAs required only 3.7%, a very small difference. In comparison to random test data generation which was required to search 10.74% of the entire search space, Simulated Annealing has offered an improvement. On average there is a 70% chance that a new data set has not been previously found. The minimum amount of test data required in a single run was 16, this figure is close to the lowest possible number 7, only slightly worse than the lowest figure of both GAs and random generation. The maximum number of new data sets required by a single run was 1694, approximately 100 data sets more than GAs where a run required 1595 data sets.

	GAs	Random	SA
Average unique data sets 1000 runs	341.62	994.58	348.26
Standard deviation over 1000 runs	291.59	881.13	294.56
Average generations required	685.20	1113.07	500.79
Minimum unique data sets required	14	14	16
Maximum unique data sets required	1595	4968	1694
Average % of search space searched	3.7%	10.74%	3.8%
Average new data sets/generation	0.50	0.89	0.70

Table 31 - Results of Function Under Test Using Simulated Annealing Over 1000 Runs for LCSAJs and Branch Testing Compared to GAs and Random Generation

The graph in figure 51 is a frequency graph comparing Simulated Annealing to those results ascertained in chapter 6 for GAs and random generation. The peak of Simulated Annealing is between 150-200 with 127 runs completing within this range, as opposed to 123 runs in this range for GAs. The longest run required 1694 unique data sets which is less than the maximum number required by random generation, 4958, but greater than that required by GAs, 1595, these results are outside the scope of the graph.

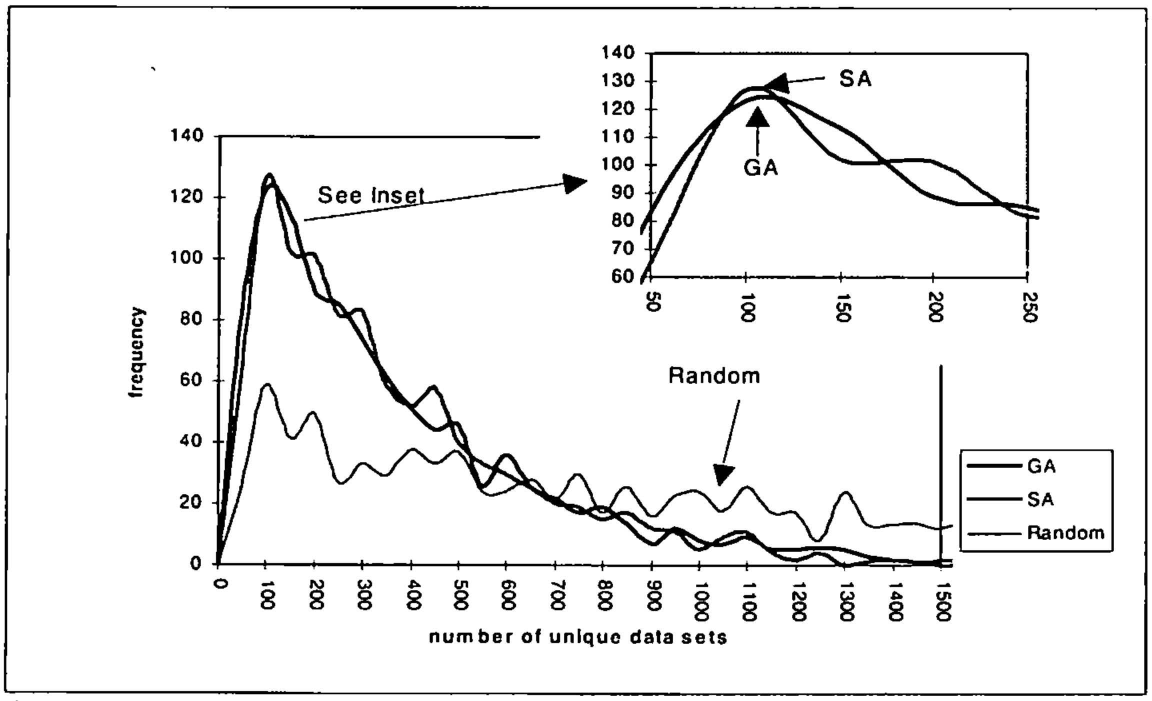


Figure 51 - A Comparison of Frequency between GAs, Simulated Annealing and Random Generation for Sample Function Under Test for LCSAJs and Branch Testing

7.4.2. Test Data Generation for Function Paths

Having accomplished coverage for branches and LCSAJs, concentration will now be placed on path testing. The results are in table 32. Simulated Annealing achieved an average of 507.76 new data sets per run, this figure is just lower than the results of GAs at 529.18 and much better than random generation at 2718.92. The lowest amount of unique test data required was 89 which is more than required by the best run of GAs but less than random generation, the lowest possible figure is 15 as there are 15 paths. The number of generations required was 773.12, which means there was a 66% chance that a newly generated data set was in fact unique. These results indicate that Simulated Annealing required on average less unique data sets than GAs by a small margin, and also reached the coverage ratio in fewer generations.

	GAs	Random	SA
Average unique data sets 1000 runs	529.18	2718.92	507.76
Standard deviation over 1000 runs	279.30	1471.34	268.94
Average generations required	1072.35	3501.47	773.12
Minimum unique data sets required	78	199	89
Maximum unique data sets required	1952	7135	1694
Average % of search space searched	5.7%	29.36%	5.5%
Average new data sets/generation	0.50	0.78	0.66

Table 32 - Results of Function Under Test Using Simulated Annealing Compared weith GAs and Random Generation over 1000 Runs for Path Testing

The graph in figure 52 is a frequency chart comparing Simulated Annealing to those results ascertained in chapter 6 for GAs and random generation. The peak for Simulated Annealing falls between 400-450 with 103 runs finishing within this range. This compares with GAs with 94 runs completed between 350-400 data sets and 3350-3400 for random generation with 19 runs completing in this range. In the resulting graph, the style of the line created by Simulated Annealing is very similar to that achieved by GAs, albeit a bit delayed. There is more similarity between GAs and Simulated Annealing results than with random generation.

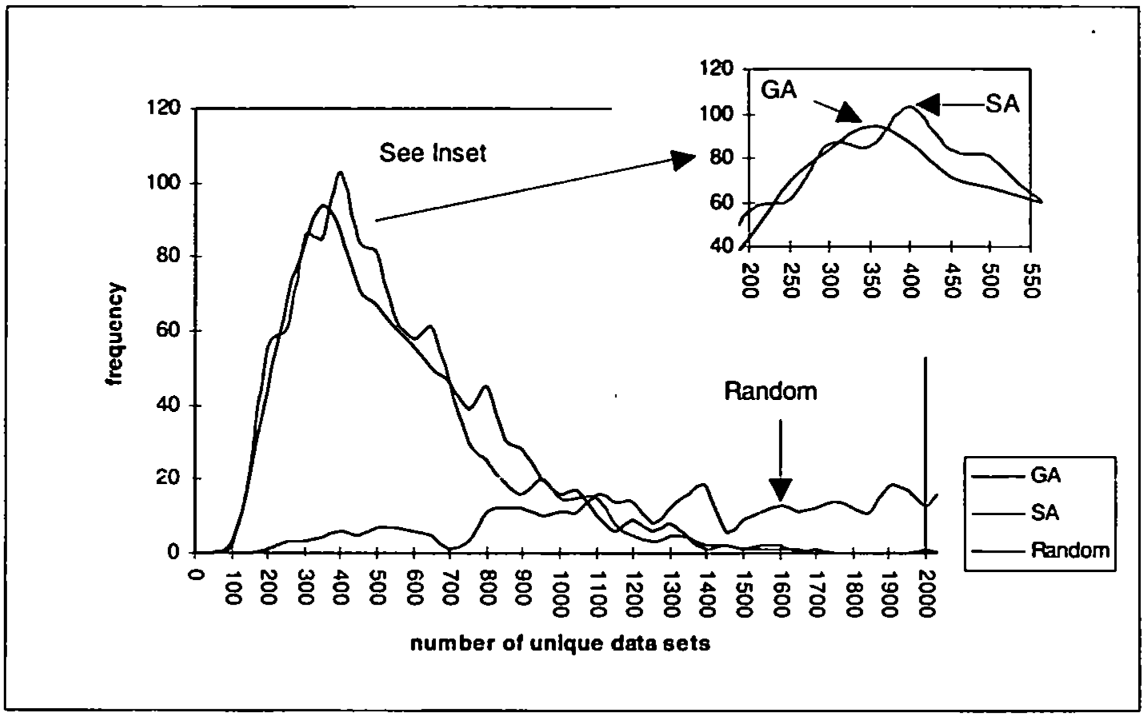


Figure 52 - A Comparison of Frequency between GAs, Simulated Annealing And Random Generation For Sample Function Under Test for Path Testing

7.5. A Comparison of Control Variables

For the previous tests of Simulated Annealing a 50% probability of acceptance was used. Table 33 shows the results when this acceptance rate is changed first to 25% and then to 75% probability of acceptance.

	Acceptance Probability		
Simulated Annealing	25%	50%	75%
Average unique data sets 1000 runs	334.80	348.26	345.79
Standard deviation over 1000 runs	292.49	294.56	297.42
Average generations required	481	500.79	497.46
Minimum unique data sets required	11	16	16
Maximum unique data sets required	2034	1694	1790
Average % of search space searched	3.6%	3.8%	3.7%
Average new data sets/generation	0.70	0.70	0.70

Table 33 - A Comparison of Acceptance Probability of Function Under Test Using Simulated Annealing Over 1000 Runs For LCSAJs and Branch Testing

This indicates that for this sample function under test there is no great difference in the results when the acceptance probability rate is changed. A probability rate of 25%, means that there is a 25% chance that a new solution with a poorer fitness will be accepted. Using this rate, a lower average unique data set is required but this difference is negligible, however this rate managed for one run to achieve coverage with just 11 data sets, the closest to the minimum 7 so far. It also had one run which required 2034 data sets, the most required for Simulated Annealing. The rate of 75% also achieved better results than using 50%, but again the difference is too small to be influential. Therefore, for this test, results indicate that the acceptance rate has little effect on test data generation.

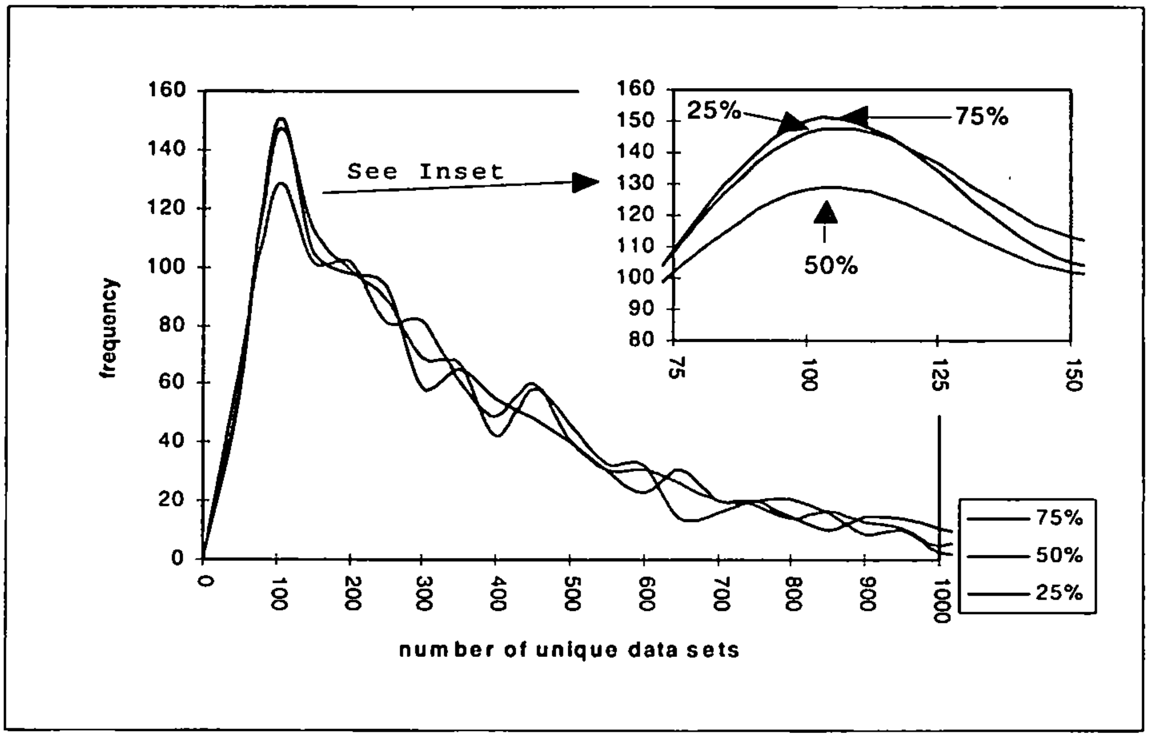


Figure 53 - A Comparison of Frequency for Acceptance Probability of Function Under Test Using Simulated Annealing Over 1000 Runs for LCSAJs and Branch Testing

Figure 53, is the frequency distribution of these three tests, 25%, 50% and 75%, the peak for each probability falls between 100-150 unique data sets. The 25%

probability had 146 runs fall within this range, 50% had 127 within this range and there were 150 runs within the range for a probability of 75%.

7.6. A Hybridisation of GAs and Simulated Annealing

Dowsland (1993) suggests that the capabilities of Simulated Annealing can be enhanced by combining them with other search methods. This can consist of either pre or post processing before beginning the annealing process. GAs could perform as a pre-processing method, to concentrate the search onto a good starting position from which to begin the annealing process. According to Dowsland (*ibid.*) the starting temperature must be lower than normal to avoid destroying the characteristics of that 'good' solution. With this in mind, GAs were combined with the Simulated Annealing process. The global search will be performed by GAs before a localised search is performed by Simulated Annealing on the best member of the final population. The GA was run 50 generations before the best member was used as the starting point for Simulated Annealing. A variety of cooling temperature were again used to determine if the temperature affected the results. The technique was first attempted for LCSAJs and branch testing and the results follow in table 34.

	Acceptance Probability		
Simulated Annealing	25%	50%	75%
Average unique data sets 1000 runs	334.53	330.56	329.68
Standard deviation over 1000 runs	305.52	301.34	287.50
Average generations required	434.23	428.13	424.77
Minimum unique data sets required	17	14	13
Maximum unique data sets required	2368	1914	2159
Average % of search space searched	3.6	3.56%	3.56%
Average new data sets/generation	0.77	0.77	0.77

Table 34 - Hybrid GAs-SA Comparison for a Range of Acceptance Probabilities of Function Under Test Over 1000 Runs For LCSAJs and Branch Testing

These results demonstrate that using a higher acceptance probability does generate a better average result, however as there is only one data set between the best and second best the probability rate seems to have little effect on the amount of test data generated. In table 35, is a comparison between the best of the hybrid method of GAs-SA, and its predecessors, GAs and Simulated Annealing. The Simulated Annealing results are those when using a probability acceptance rate of 50%.

While the hybrid GAs-SA achieves the best average result 329.68 when compared to GAs 341.62, and Simulated Annealing of 348.26, the gap is not very large. The GAs-SA did manage to complete a run with the smallest amount of test data, 13, but it also had a run with the most for these three techniques, 2159.

	GAs	SA	GAs-SA
Average unique data sets 1000 runs	341.62	348.26	329.68
Standard deviation over 1000 runs	291.59	294.56	287.50
Average generations required	685.20	500.79	424.77
Minimum unique data sets required	14	16	13
Maximum unique data sets required	1595	1694	2159
Average % of search space searched	3.7%	3.8%	3.56%
Average new data sets/generation	0.50	0.70	0.77

Table 35 - Results of Function Under Test Comparing GAs, Simulated Annealing, and the Hybrid GAs-SA Over 1000 Runs for LCSAJs and Branch Testing

Figure 54 is a frequency chart for the GAs-SA (all three probability rates), GAs and Simulated Annealing. More runs completed within the range 100-150 unique test data sets with the hybrid technique, than with the conventional GAs and Simulated Annealing runs. When a probability rate of 25% was used 152 runs completed, 50% 132 completed and 136 finished in this range for 75%. These figures compared to 123 for GAs and 127 for Simulated Annealing.

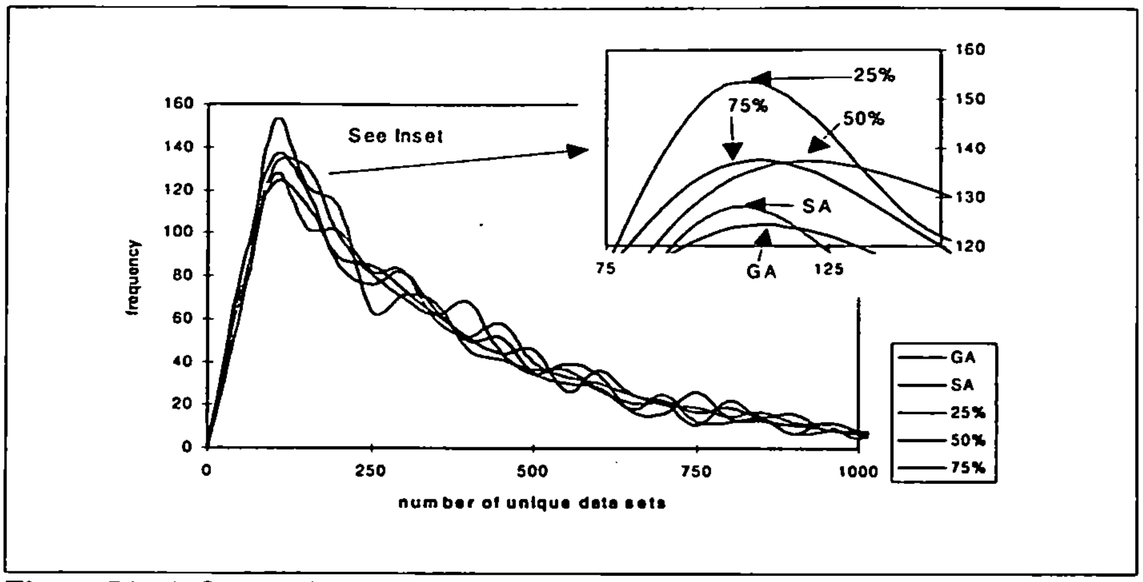


Figure 54 - A Comparison of Frequency between GAs, Simulated Annealing and the Hybrid GAs-SA (for all Three Probability Rates) for the Sample Function Under Test for LCSAJs and Branch Testing

The same experiment was attempted with path testing to ascertain the impact of GAs-SA, again 50 generations were run for GAs before the best member of the population was used as the starting point for Simulated Annealing, the results are in table 36.

	Acceptance Probability		
Simulated Annealing	25%	50%	75%
Average unique data sets 1000 runs	501.05	484.81	468.58
Standard deviation over 1000 runs	254.17	253.97	234.90
Average generations required	721.50	694.74	666
Minimum unique data sets required	90	84	57
Maximum unique data sets required	1676	1819	1328
Average % of search space searched	5.41%	5.23%	5.06%
Average new data sets/generation	0.69	0.70	0.70

Table 36 - Hybrid GAs-SA Comparison for a range of Acceptance Probabilities of Function Under Test Over 1000 Runs For Path Testing

These results again indicate that the 75% probability rate achieves the best average amount of test data, 468.58 or 5.06% of the search space, compared to 484.81, 5.23%, for 50% and 501.05, 5.41%, for 25%. The 75% probability also managed to achieve coverage in one run with the smallest amount of test data so far of 57, and also offered the smallest spread between minimum and maximum required. The lowest possible number of unique test data sets is 15, the smallest amount of test data by GAs was 78, and 89 by Simulated Annealing. The results of this comparison are in table 37. The GAs-SA achieved coverage using only 5.06% of the search space compared to 5.48% for Simulated Annealing and 5.71% for GAs.

	GAs	SA	GAs-SA
Average unique data sets 1000 runs	529.18	507.76	468.58
Standard deviation over 1000 runs	279.30	268.94	234.90
Average generations required	1072.35	773.12	666
Minimum unique data sets required	78	89	57
Maximum unique data sets required	1952	1694	1328
Average % of search space searched	5.71%	5.48%	5.06%
Average new data sets/generation	0.50	0.66	0.70

Table 37 - Results of Function Under Test Comparing GAs, Simulated Annealing, and the Hybrid GAs-SA over 1000 Runs for Path Testing

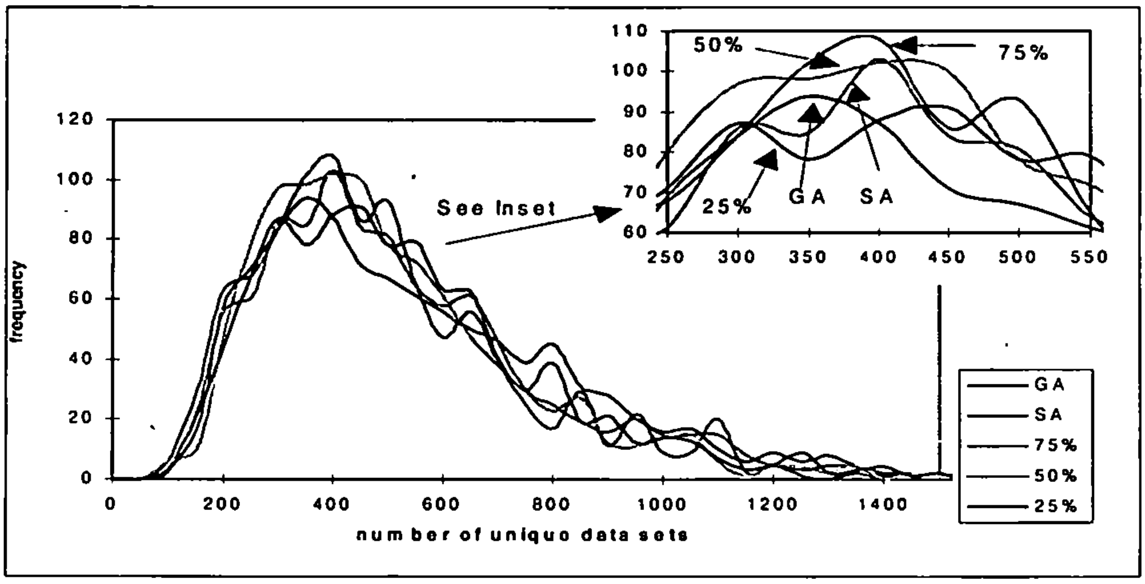


Figure 55 - A Comparison of Frequency between GAs, Simulated Annealing and the Hybrid GAs-SA (for all Three Probability Rates) for the Sample Function Under Test for Path Testing

Figure 55 shows the frequency chart for GAs, Simulated Annealing and GAs-SA. The peak range for the 75% probability is between 400-450 unique data sets with 108 runs, the 50% probability has 102 runs within this range compared to Simulated Annealing with 103 runs. The range 450-500 has the greatest number of runs completing, 91 for 25% probability. Finally GAs have a maximum 94 completing in the range of 350-400 unique data sets. In this case, the GAs-SA option offers the best average, an improvement of 11.45% over GAs and 7.7% over Simulated Annealing.

7.7. Conclusion

This chapter describes Simulated Annealing and discusses the process involved in the generation of test data using Simulated Annealing. Included is the method with which a new solution is selected over the old solution, even if that new solution has a fitness worse than the original. The final part of this chapter is a comparison of Simulated Annealing to the use of GAs and random test generation as applied in chapter 6. This comparison reveals that while Simulated Annealing improves on random generation for this sample function under test, the results are not as successful as those from using GAs. The similarity of the results however agree with those obtained by Thornton (1994) on a simple problem. More difficult test functions will be introduced in Chapters 9 and 10.

The chapter concludes with results attained when the probability acceptance criteria has been adjusted, using an initial value of 50% which was used for the previous tests, this rate was adjusted by 25% in each direction. The results indicate that the probability of acceptance has little effect for this sample function. The final section in this chapter looks at a hybrid GAs-SA which applies the global searching capabilities of GAs and then uses the Simulated Annealing to approximate a local search of the best member of the population. This hybrid technique was attempted for a range of acceptance probabilities. In testing for LCSAJs and branches there was little difference in the results, 5.35% over Simulated Annealing and 3.50% over GAs.

When path testing, the GAs-SA offers a 7.7% improvement over Simulated Annealing and 11.45% over GAs. This greater improvement for path testing could be due to the fact that for this test program, path testing is a more difficult problem as there are two paths which each have only 4% of the search space, as opposed to the smallest search space for one of the LCSAJ of 9%. The next chapter will introduce Tabu search, another adaptive search technique.

Chapter Eight

Tabu Search and Its Use for the Generation of Test Data

8.1. Introduction

Tabu search is a heuristic algorithm that uses memory to find a good solution to a search problem. The Tabu search as discussed here derives from the work of Glover (1989;1990;1994) and Glover *et al* (1993). According to Glover (1990) Tabu search is defined as

> "...a higher level heuristic procedure for solving optimisation problems, designed to guide other methods (or their component processes) to escape the trap of local optimality"

Tabu search is a neighbourhood search and can be equated to hill-climbing. Unfortunately hill-climbing is limited by local optimum as the search will conclude at reaching the local optimum. Tabu search attempts to go beyond termination at the local optimum by allowing moves to be made from one result to another even if that new result is not the best of the neighbourhood. This means that there is the possibility of looping within the search (Nurmela, 1995). To avoid this, Tabu search uses flexible memory to record a history of a search. This memory can structure the history list by four dimensions, recency, frequency, aspiration and influence (Glover and Laguna, 1993).

By maintaining a selective history of all states encountered, a restricted area can be created with 'no-go' sections which are the best results of previous searches. These are areas of a neighbourhood which may have been previously searched or one that may cycle a search back toward a previous result. While the search is not banned from these ‘no-go’ areas, it is strongly influenced against it by penalising the results. Membership of the Tabu list expires and becomes ‘tabu-inactive’ after a specified amount of time which can be either static or dynamic. Glover and Laguna (1993) suggest the type of list is problem specific. Occasionally it may be necessary to ‘bend the rules’ by either releasing a restriction placed on a result’s attributes (attribute aspiration) or a restriction placed on the move (move aspiration) by treating a result as ‘tabu-inactive’. This may be necessary for the good of the search as a particular move may have a 'sphere of influence' far greater than the restrictions placed upon it, and if this influence is considered greater than the restriction placed on it, the move can be made. According to Glover and Laguna (1993), the objective is to stimulate the discovery of new high quality solutions.

Tabu search is relatively new and most of the applications have only been attempted since 1989 (Glover, 1993). They have however been successful in scheduling (Laguna and Glover, 1992), which applied Tabu search to single machine scheduling problems. Dammeyer and Voss (1993) used a Tabu search type method to solve the knapsack problem, which determines the maximum number of items which can be packed in a knapsack. Over a series of 57 problems, they compared the Tabu search method to Simulated Annealing and determined they take comparable time, but that Tabu search finds the optimal solution for 50% more problems. They also determined that Simulated Annealing had a greater dependence on the initial selection of control parameters than Tabu.

One problem however with using Tabu search for the generation of test data is that a lot of test data sets will be generated as the neighbourhood is searched, and the results achieved may therefore not be satisfactory.

The algorithm for Tabu search, based on the description by Glover (1989), is as follows in figure 56

Tabu Search Algorithm

```
1       select s ∈ S  //S = search space
2       initialise tabulist and set to empty list
3       initialise TabuListSize = SIZE
4       initialise TabuCounter = 0
5       while fitness(s) < stopping criterion
5.1         generate neighbourhood(N) of s
5.2         calculate fitness of each  member using submodule Fitness
5.3         select member(N) with best fitness and make it $s_{new}$
5.4         add s to tabulist and increment TabuCounter by 1
5.5         if TabuCounter = SIZE
5.5.1           set TabuCounter = 0  //start at beginning of list to eliminate old members
5.6         $s=s_{new}$
6       end while loop
```

SubModule Fitness

```
1       calculate fitness of member
2       if member is member(tabulist) by some_attribute
2.1         reduce fitness by some_amount
3       return fitness
```

Figure 56 - Algorithm for Tabu Search including Submodule to Calculate Fitness

8.2. Tabu Search in Action

The following is a demonstration of Tabu search using the maximisation problem discussed in Chapter 5 for GAs and Chapter 6 for Simulated Annealing. The search space used in the previous examples will be extended from the range of the variables x and y of [-5..5] to a range of [−10..10] to demonstrate the effect of local optimum on the search. The graphical representation of the new search space is shown in figure 57, the representation of the peak 2.0 is in the range 2.0-2.5 to illustrate that this is the optimum peak in the search space. As there are similarities between hill-climbing and Tabu search a short demonstration of hill climbing will begin this example.

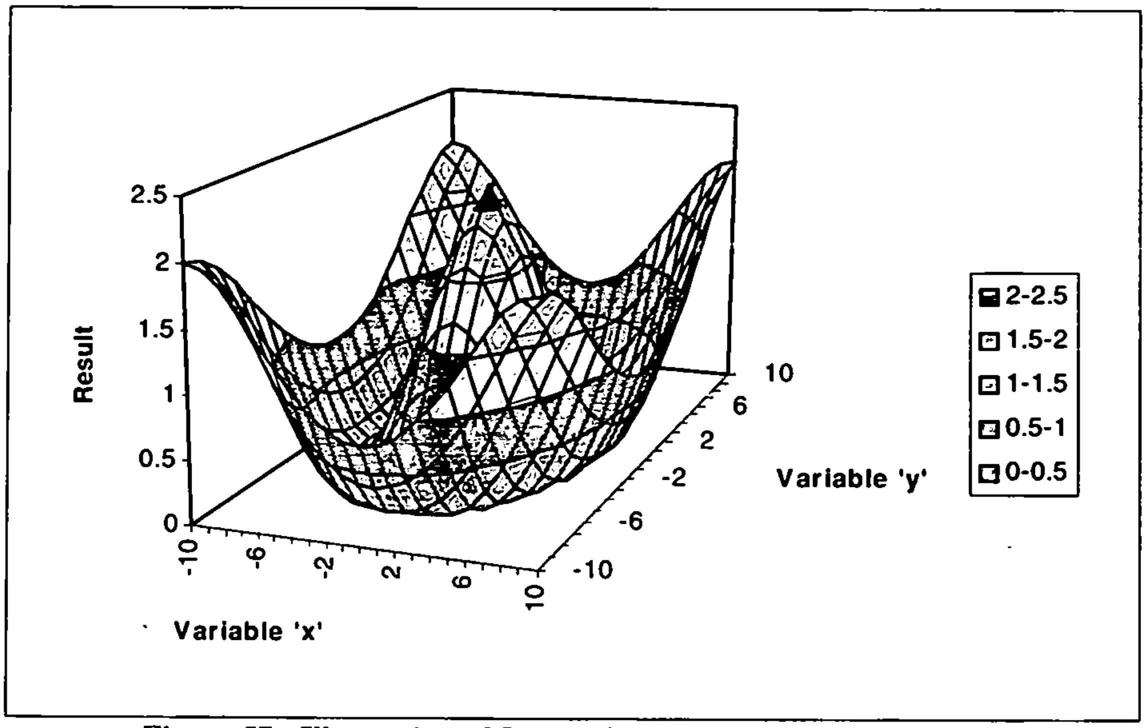


Figure 57 - Illustration of Search Space for Sample Function

8.2.1. Hill-Climbing

As Tabu search builds upon the ideas of hill-climbing, the algorithm for this from Winston (1984) slightly amended, is summarised in figure 58.

```
Hill-Climbing Algorithm
1       Establish a random initial start point called Initial_Solution and determine
        its fitness
2       while goal has not been reached or stop ≠ true
2.1         determine the neighbourhood of the Initial_Solution
2.2         sort the neighbourhood by their fitness
2.3         establish Best_Member of neighbourhood
2.4         if (fitness(Best_Member) > fitness(Initial_Solution))
2.4.1           Initial_Solution = Best_Member
2.5         else
2.5.1           stop = true
3       end while loop
```

Figure 58 - Hill-Climbing Algorithm

Therefore using a random initial start point of {5,-5}, its corresponding fitness is 0.002. At step 2.2 the neighbourhood of {5,-5} are sorted as follows in table 38.

	Member	Fitness
1	{4,6}	0.0035
2	{4,5}	0.0384
3	{4,4}	0.1814
4	{5,6}	0.0611
5	{5,4}	0.0384
6	{6,6}	0.2058
7	{6,5}	0.0611
8	{6,4}	0.0035

Table 38 - Neighbourhood of Initial Start Point

The member of the neighbourhood with the highest fitness is the move to position {6,-6}. That solution then becomes the Best_Member of the queue and its neighbourhood are sorted. This process continues and in table 39 the search path is shown.

Loop Time	Best Member of Neighbourhood	Fitness
1	{6,6}	0.2058
2	{7,7}	0.7187
3	{8,8}	1.3403
4	{9,9}	1.8303
5	{10,10}	1.9991

Table 39 - Path Hill-Climbing Takes by Accepting Best Member of Neighbourhood until Maximum Solution (Local Optimum) is Reached

After loop number five the local optimum solution of {10,-10} has been found and the boundary of the search space has been reached. The hill-climber stops and returns {10,-10} as the best solution in the search space. Figure 59 illustrates the search path again the centre is the peak of 2.0, the range of the four corners in actuality are in the range 1.75 to 1.9999. This range was again used to illustrate the single global maximum.

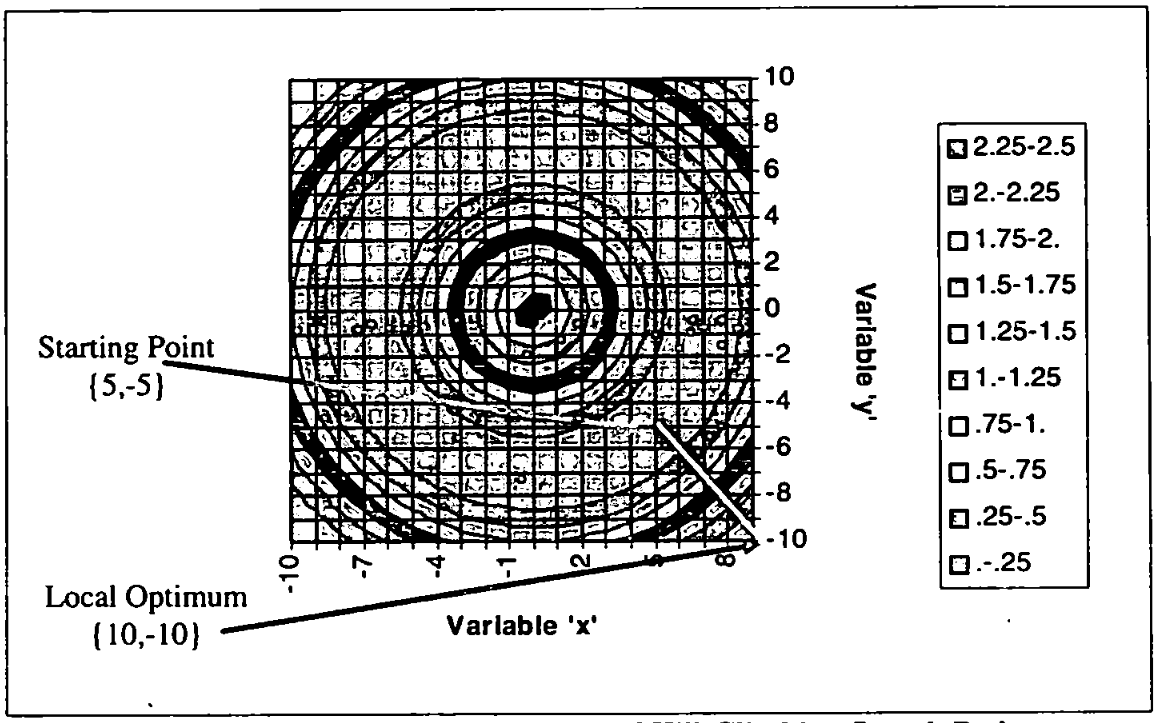


Figure 59 - Graphical Representation of Hill-Climbing Search Path

Hill-climbing therefore has not found the best solution but merely a local optimum. What will be attempted now is to use the same example to illustrate how a Tabu search may aid in finding the global optimum.

8.2.2. Tabu Search for Global Optimum

Tabu search uses the same initial starting point {5,-5}. Once its neighbourhood has been searched this initial combination will become the first member of the Tabu List. The Tabu list is to record the actual moves made at each stage. A solution from the neighbourhood is penalised if there is a chance that that solution will backtrack over that part of the search space which has already been searched. If it matches a previous move it will be penalised more harshly than if it matches only one variable of a previous move. Therefore, if in this example move {5,-5} is a member of a new neighbourhood its fitness will be (fitness*penalty), using a penalty figure for matching both variables of 0.00001 the fitness of that combination is (0.0002 *0.00001). If a member of the new neighbourhood is {5,-6} and therefore matches one variable then the fitness of that combination will be fitness*0.001. Using this penalty routine a combination is penalised more severely for backtracking over the same search space than for generating a new combination which matches only one of the previous Best_Member combinations. Other penalty values and size of Tabu list were attempted, but this combination achieved the best result, in some cases the global optimum was not attained.

The fitness of the initial point {5,-5} is 0.002. Identical to hill-climbing {6,-6} is chosen as the Best_Member of the neighbourhood and {5,-5} is added to the Tabu List, the size of the tabu list is static at 200, half the search space. When the list becomes full new additions to the list replace the oldest members of the list, and this means that if any of these combination are members of a new neighbourhood they will no longer be penalised. Table 40 is the neighbourhood of {6,-6} with their corresponding fitness which for some combinations includes their penalty.

			Penalty Value		
	Variables	Original Fitness	Matching Two	Matching One	New Fitness
1	{5,7}	0.2387		0.001	0.000239
2	{5,6}	0.0611		0.001	0.0000611
3	{5,5}	0.0002	0.0001		0.00000002
4	{6,7}	0.4442		0.001	0.4441
5	{6,5}	0.0611			0.0000611
6	{7,7}	0.7187			0.7187
7	{7,6}	0.4442			0.4442
8	{7,5}	0.2387		0.001	0.000239

Table 40 - Ranking of Neighbourhood from Table 38 using Tabu List Restrictions

In this neighbourhood the combination {5,-5} is penalised the harshest as this matches a member of the Tabu List, some of the other combinations only match one variable {{5,-7},{5,-6},{6,-5},{7,-5}} and are penalised less harshly. The Tabu search does find an answer after 263 moves, and the area of the search space which has been looked at is shown in figure 60. The line shows the general search path while the arcs and circles represent the areas of the search space where intensive examination took place.

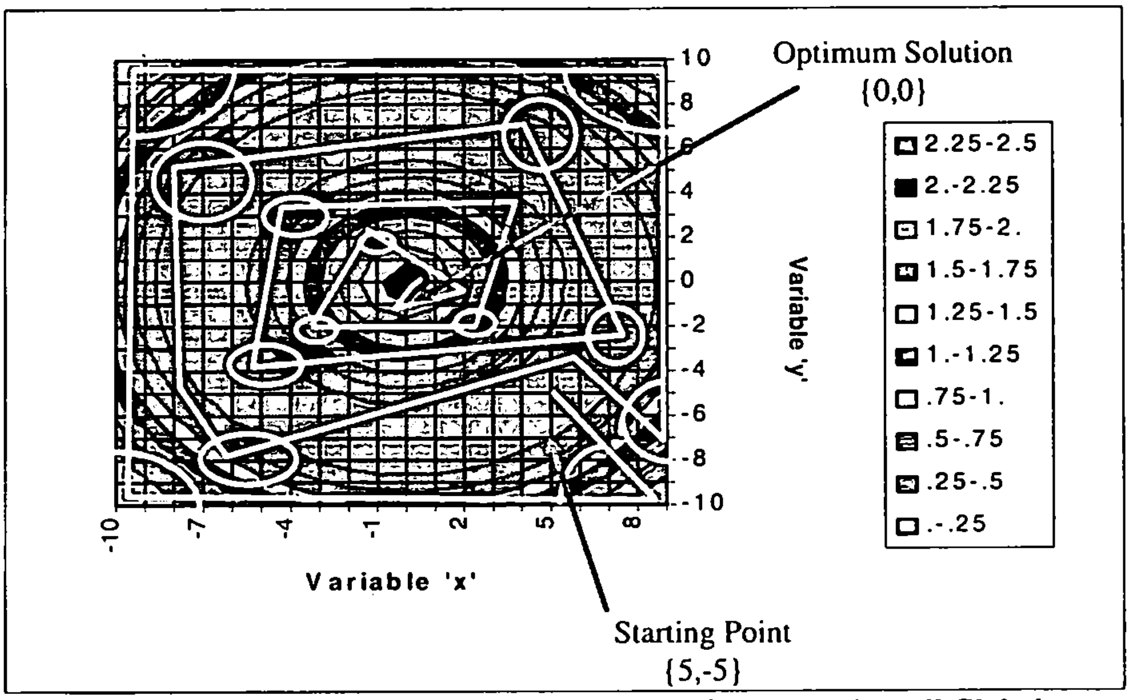


Figure 60 - Graphical Representation of Tabu Search Path until Global Optimum is Reached

8.3. Hill-Climbing for the Generation of Test Data

Even in the small example used, a great deal of the search space is being examined to determine its fitness in relation to its neighbours. This can be a definite deterrent from using hill-climbing to generate test data as a lot of test data sets will be required in order to determine if one has a better fitness than its neighbour. Additionally the structure of the fitness function may restrict the search for global optimum, as there is no one global optimum in this type of search as is expressed in the fitness function. As a path is found it has a high fitness, but for every successive 'discovery' of this path the fitness decreases. Every time the search gets caught in a local optimum it will be necessary to begin the search again from another random initial solution. In table 41 is a breakdown of unique data sets generated for 1000 runs of the hill-climber before all LCSAJs and branches were found for the sample function under test as described in 3. The hill-climber viewed on average 2379.41 unique test data sets prior to complete coverage of the code, more that one-fourth of the search space. Random generation required 994.58. The maximum amount of test data required by a run was 7969, this number is close to the maximum amount required before coverage of 9254. These results indicate that hill-climbing views a lot of test data in its search.

Hill-Climbing	
Average unique data sets 1000 runs	2379.41
Standard deviation over 1000 runs	1766.75
Average number of generations	144.51
Minimum unique data sets required	52
Maximum unique data sets required	7969
Average % of search space searched	25.69%
Average new data sets/generation	16.46

Table 41 - The Results of Hill-Climbing on Sample Function for LCSAJs and Branches Over 1000 runs

Figure 61 is a comparison frequency chart between GAs, random generation, Simulated Annealing and hill-climbing, 23 runs of the hill-climber completed in the range of 750-800 unique test data sets.

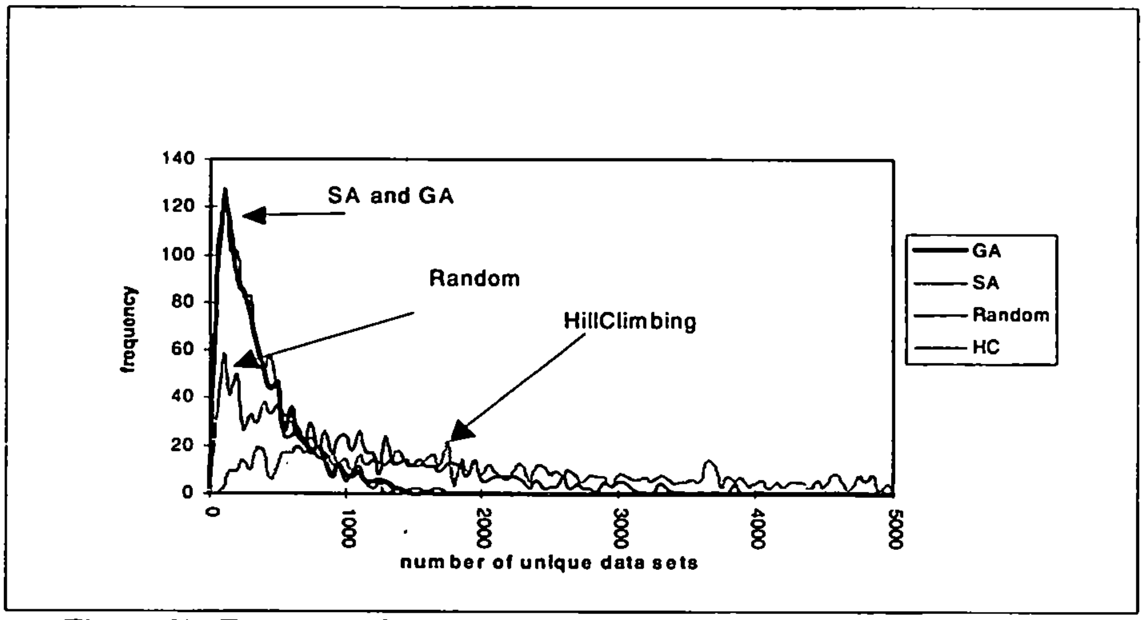


Figure 61 - Frequency Chart Comparing Hill-Climbing to Other Search Techniques, GAs, Random Generation, and Simulated Annealing for LCSAJs and Branch Testing

As seen from these results hill-climbing produces very poor results for which there are a number of reasons, the most obvious being that the entire neighbourhood must be generated to choose the fittest, and this produces a lot of test data. By the very nature of hill-climbing the search is up a slope, and while very good for some search problems this particular space appears to benefit from a more varied search pattern. Is it possible that Tabu search can improve on the results from hill-climbing?

8.4. Using Tabu Search for the Automatic Generation of Test Data

In Tabu search each test data set is compared to the Tabu list and is penalised for matching a member of the list. This is a minimisation problem, as opposed to Simulated Annealing and GAs which are performed as maximisation problems, and use the inverse of the fitness. This makes it easier in GAs to establish the best of a population and in Simulated Annealing make the spread between strings smaller. This should have no bearing on the result. If a test data set matches a set on the list then it is penalised by having its fitness multiplied by 100, the size of the Tabu list is also 100. This method is very similar to the penalty method used in the example in section 7.2.2. One change has been made to the way a neighbourhood is searched. Normally the process would always begin in the same corner or the neighbourhood. If however, all the members of the neighbourhood return the same path the first member generated will be deemed the best, as the fitness value for that path has worsened each subsequent time found. Therefore to avoid directing the search by virtue of first in the neighbourhood, the starting position is changed every 20 generations. The initial member of the neighbourhood is derived from the south-west corner as opposed to the north-east. The results follow in table 42.

	Tabu Search	GAs	Simulated Annealing	GAs-SA	Random Testing
Average unique data sets 1000 runs	366.19	341.62	348.26	329.68	994.58
Standard deviation over 1000 runs	136.24	291.59	294.56	287.50	881.13
Average generations required	97.31	685.20	500.79	424.77	1113.07
Minimum unique data sets required	44	14	16	13	14
Maximum unique data sets required	711	1595	1694	2159	4968
Average % of search space searched	3.95%	3.70%	3.80%	3.56%	10.74%
Average new data sets/generation	3.76	0.50	0.70	0.77	0.89

Table 42 - The Results of Tabu Search on Sample Function Under Test for LCSAJs and Branches

The frequency distribution in figure 62 demonstrates that the structure of Tabu search, while not as effective as GAs or Simulated Annealing, does give results which are very close, as the average is 366.19 unique data sets. This is not too many more than GAs with 341.62 and Simulated Annealing with 348.66, and a great improvement over hill-climbing which required 2379.41. One note of contrast is the minimum new data sets required by a run which was 44 for Tabu search, which is well above the 14 required by GAs and 10 by Simulated Annealing, but the maximum required is 711 which gives a smaller range of unique data.

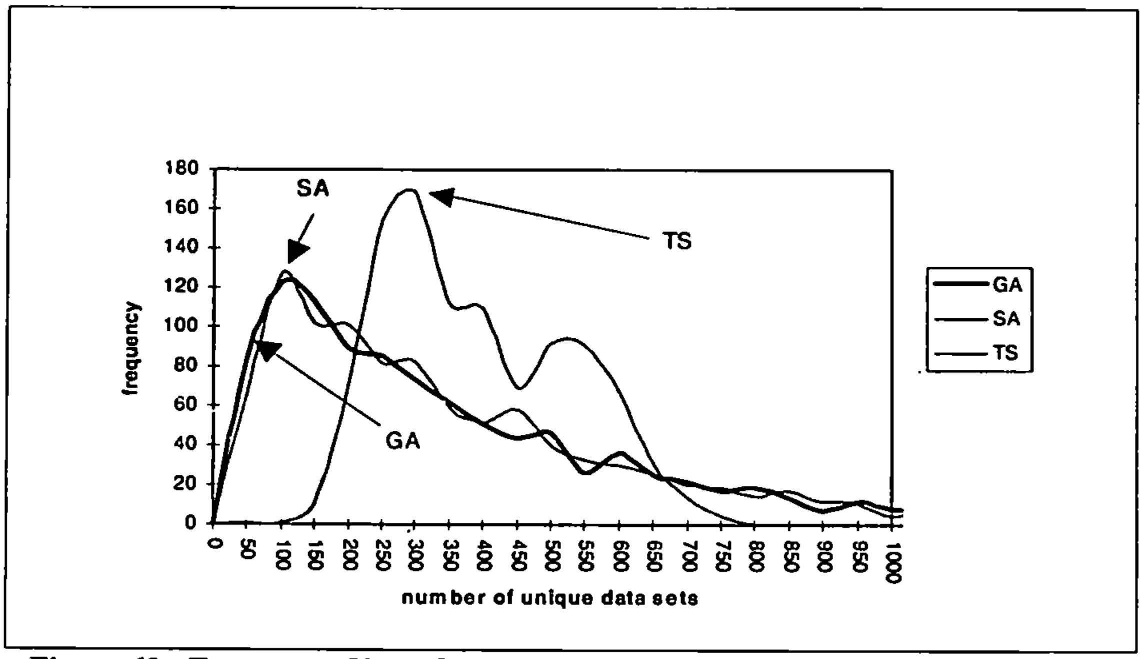


Figure 62 - Frequency Chart Comparing Tabu Search to GAs and Simulated Annealing over 1000 Runs for LCSAJs and Branch Testing

The Tabu search method was also used for path testing for this demonstration function. The results for Tabu search are compared to those achieved by GAs, Simulated Annealing, random generation and GAs-SA in table 43. Tabu Search performs 68.5% better than random generation, however the results are worse than those when using GAs, 80.5%, Simulated Annealing, 81.3%, and the hybrid GAs-SA, 82.7%. The minimum run for Tabu search also required the most unique test data, 249, higher than even random generation at 199, although the range between minimum and maximum is again the smallest for any of the techniques used so far.

	Tabu Search	GAs	Simulated Annealing	GAs-SA	Random Testing
Average unique data sets 1000 runs	855.19	529.18	507.76	468.58	2718.92
Standard deviation over 1000 runs	141.47	279.30	268.94	234.90	1471.34
Average generations required	307.05	1072.35	773.12	666	3501.47
Minimum unique data sets required	249	78	89	57	199
Maximum unique data sets required	1152	1952	1694	1328	7135
Average % of search space searched	9.23%	5.71%	5.48%	5.06%	29.36%
Average new data sets/generation	2.79	0.50	0.66	0.70	0.78

Table 43 - The Results of Tabu Search on Sample Function Under Test for Path Testing Compared to GAs, Simulated Annealing, the hybrid GAs-SA and Random Generations

Figure 63 is the frequency chart for all five methods. The peak for Tabu search is between 1000-1050 unique data sets, 197 runs completed in this range. The most runs, 94, completed for GAs between 350-400 data sets, for Simulated Annealing, 103 completed in the range 400-450 and for GAs-SA, 108 completed in this same range.

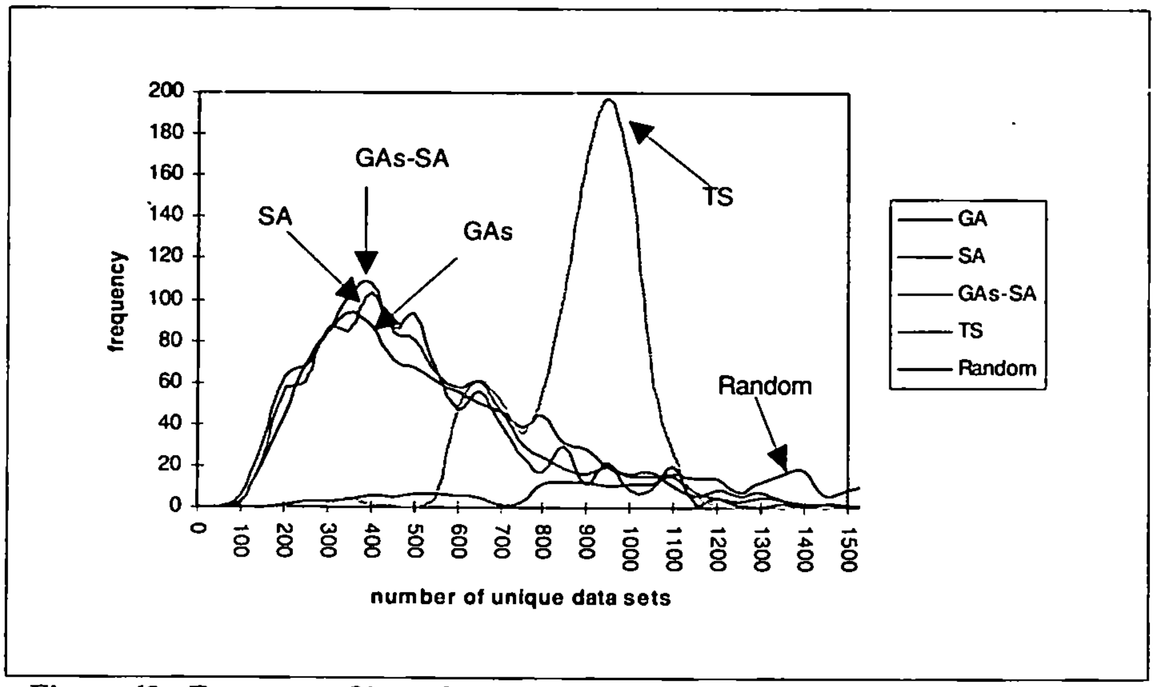


Figure 63 - Frequency Chart Comparing Tabu Search to GAs and Simulated Annealing over 1000 Runs for Path Testing

8.5. Tabu Search Assisting Other Adaptive Search Techniques

The results achieved by Tabu search were poorer than those achieved by GAs and Simulated Annealing for this sample program. However Tabu search like Simulated Annealing can be used as a local search mechanism when attached to GAs. Glover (1994) suggests this idea. Rayward-Smith and Debuse (1994) and Kido *et al* (1993) suggest the combination of all three techniques, GAs for the global search and Simulated Annealing and Tabu search for local search. Kido *et al* (1993) state that for the TSP problem, GA+SA+TS achieve better results than GA+SA and GA+TS, although GA+TS was a close second. In the following section GAs will perform the global search while Tabu performs the local search.

8.5.1. Using GAs with Tabu Search

The GAs were run for 50 generations before Tabu search took over using the best member of the GAs as the initial value. Results for this technique, compared to using Tabu search and GAs alone, are shown in table 44. The GAs-TS performed much better than its predecessors and required only 213.15 unique data sets as opposed to 366.19 by Tabu search and 341.62 by GAs. The range between minimum and maximum was also much smaller, 595, when compared to Tabu search 667, and GAs 1581. Although the best run of GAs-TS was slightly worse by two data sets than GAs it was a great improvement over the minimum run of 44 achieved by Tabu search.

	GAs-TS	Tabu Search	GAs
Average unique data sets 1000 runs	213.15	366.19	341.62
Standard deviation over 1000 runs	94.67	136.24	291.59
Average generations required	22.25	97.31	685.20
Minimum unique data sets required	16	44	14
Maximum unique data sets required	595	711	1595
Average % of search space searched	2.3%	3.95%	3.7%
Average new data sets/generation	9.58	3.76	0.50

Table 44 - The Results of the Hybrid GAs-TS compared to Tabu Search and GAs on Sample Function Under Test for LCSAJs and Branches

Figure 64 is the frequency distribution of GAs-TS compared to the other techniques used so far. The peak for GAs-TS is in the range 250 data sets with 230 runs completing. While this peak is in a higher range than the other methods it does still have the best average performance of the adaptive search methods applied. The next closest result is GAs-SA which averaged 329.68 unique test data sets but had a minimum-maximum range of 2146 compared to 579 for GAs-TS. The overall improvement of GAs-TS from the results attained from GAs was 38%, and 42% improvement over Tabu search.

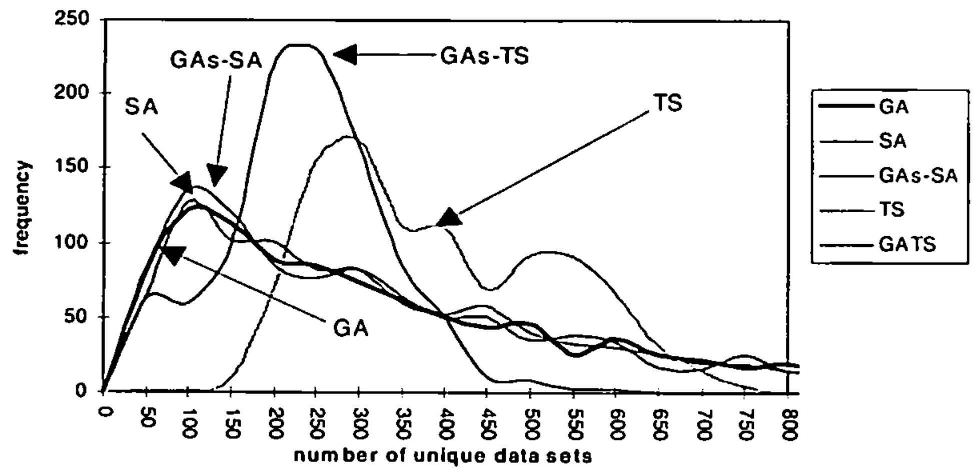


Figure 64 - Frequency Chart Comparing the Hybrid GAs-TS to the Other Adaptive Search Techniques over 1000 Runs for LCSAJs and Branch Testing

The hybrid GAs-TS was also used for path testing, the results are in table 45. While GAs-TS with an average number of unique data sets of 770 offer an improvement over Tabu search, 855.19, the average is less than that achieved when using GAs alone, 529.18.

	GAs-TS	Tabu Search	GAs
Average unique data sets 1000 runs	770	855.19	529.18
Standard deviation over 1000 runs	259	141.47	279.30
Average generations required	259	307.05	1072.35
Minimum unique data sets required	132	249	78
Maximum unique data sets required	1332	1152	1951
Average % of search space searched	8.31%	9.23%	5.71%
Average new data sets/generation	2.97	2.79	0.50

Table 45 - The Results of the Hybrid GAs-TS compared to Tabu Search and GAs on Sample Function Under Test for Path Testing

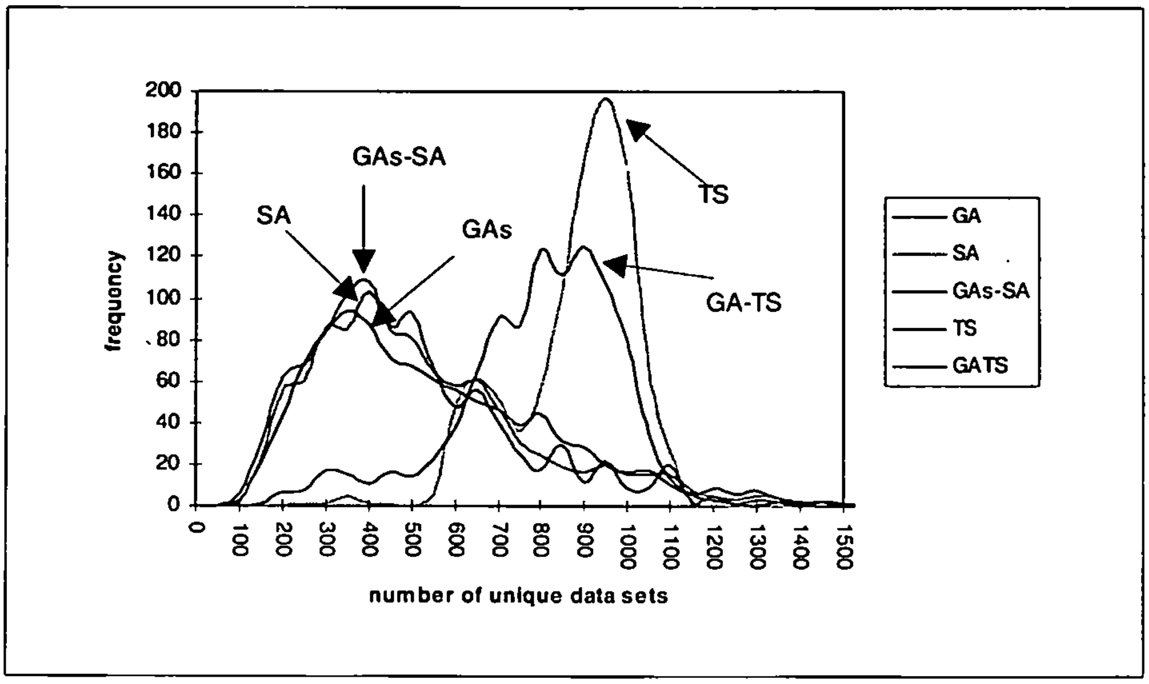


Figure 65 - Frequency Chart Comparing the Hybrid GAs-TS to the Results from Other Adaptive Search Techniques over 1000 Runs for Path Testing

The frequency chart for all the adaptive search method attempted so far for path testing is in figure 65. The frequency distribution for the hybrid GAs-TS method has two peaks, the first is between 900-950 unique data sets with 125 runs completing in this range, and the second is in the range 800-850 data sets with 123 runs completing.

The results for path testing are not as convincing as those received when testing LCSAJs and branches, as with the hybrid GAs-SA this could be due to the difficulty of the path testing search space when compared to that of branch and LCSAJs testing.

8.5.2. Simulated Annealing and Tabu Search

To attach the memory capabilities from Tabu search to Simulated Annealing may encourage the Simulated Annealing search to concentrate on new areas of the search space and to avoid backtracking over previous positions. When a new solution is generated it is checked against the Tabu list. If it is a member of the list the solution is discarded, otherwise the process continues as normal (as described in chapter 7). Two fixed sizes of Tabu list were tried, a size 10 and 100, as used in previous examples in this chapter. If a new solution has proceeded to the Simulated Annealing process and has been over the previous solution, it is the added to the Tabu list. When the list reaches its maximum size new solutions are entered at the beginning of the list replacing older members. The first stage was to discard any new solution which matched a solution on the list. The second stage was to discard a solution if it matched just 2 members of the list. The process was attempted to determine the effectiveness of this procedure for both LCSAJs and branch testing and path testing. These results were compared to those received when using Simulated Annealing on its own and are shown in table 46 for branch and LCSAJs. The new technique which performed the best was a Tabu list of size 10 and a new solution was rejected if it matched a member of the list, this method gave a result of 343.0, very close to the result given by Simulated Annealing alone of 348.26. The other techniques were very poor performers, especially when the list size grew to 100. Therefore, it appears that a Tabu list has little affect on the results for LCSAJs and branch testing.

	Simulated Annealing	Tabu List Size 100		Tabu List Size 10	
		match 2	match 3	match 2	match 3
Average unique data sets 1000 runs	348.26	1092.44	648.87	424.14	343.0
Standard deviation over 1000 runs	294.56	849.92	483.19	350.93	291.36
Average generations required	500.79	8241.10	1858.61	686.98	492.39
Minimum unique data sets required	16	16	17	17	13
Maximum unique data sets required	1694	4990	2998	3500	1873
Average % of search space searched	3.76%	11.80%	7.01%	4.58%	3.70%
Average new data sets/generation	0.70	0.13	0.35	0.62	0.70

Table 46 - Comparison Of Results For Simulated Annealing Combined with a Tabu List which Rejects a Variable Combination if it Matches a Member of the List for LCSAJs and Branch Testing

The results for path testing are in table 47. Again the best performance is by a Tabu list size of 10 when all three match a member of the list, this result 515.12 is not however an improvement over Simulated Annealing which gave a result of 507.76. The worse technique was using a Tabu list of 100 and two match the list, the result was 2292.0, 3.5 times the result given when using Simulated Annealing alone. While attaching a Tabu list to the Simulated Annealing process does not seem very successful in this example it is possible with a different program this procedure may have more success.

	Simulated Annealing	Tabu List Size 100		Tabu List Size 10	
		match 2	match 3	match 2	match 3
Average unique data sets 1000 runs	507.76	2292.0	1315.29	773.97	512.12
Standard deviation over 1000 runs	268.94	1407.32	702.19	417.96	268.12
Average generations required	773.12	20405.73	4524.47	1427.55	786.92
Minimum unique data sets required	89	131	120	106	71
Maximum unique data sets required	1694	8151	4978	2483	2047
Average % of search space searched	5.48%	24.75%	14.20%	8.36%	5.56%
Average new data sets/generation	0.66	0.11	0.29	0.54	0.65

Table 47 - Comparison Of Results For Simulated Annealing Combined with a Tabu List which Rejects a Variable Combination if it Matches a Member of the List for Path Testing

8.6. Conclusion

This chapter introduced the adaptive search technique called Tabu search. Tabu search, first discussed by Glover (1989), is based on hill-climbing with restrictions placed on returning to previous search positions within a given period, 100 moves in this example. Unfortunately hill-climbing did not prove successful in test data generation, while Tabu search did much better than random test data generation and the results were very similar to other adaptive search techniques for branch and LCSAJs testing. When a hybrid GAs-TS was created the results for branch and LCSAJs testing were the best of all the adaptive search techniques, a 37.6% improvement over GAs used on their own and a 41.7% improvement over Tabu search alone. The closest result to GAs-TS was by the hybrid GAs-SA which achieved coverage with an average unique data set of 329.68, when compared to the result from GAs-TS, 213.15, a 35.3% improvement for GAs-TS. The results for the hybrid GAs-TS were not as successful for path testing as those achieved by GAs, Simulated Annealing and GAs-SA, although they did improve the results achieved by Tabu search alone by 10%. If this is accredited to the difficulty of the search space, then it would appear that Tabu search works well in easier, less specific, search spaces. However more tests will need to be performed to validate these results.

The philosophy of Tabu search was then applied to Simulated Annealing which used a Tabu list to assist in guiding the search, unfortunately this does not improve the search, and in fact in all but one case the results were worse than those achieved by Simulated Annealing alone. The next chapter will introduce a wider variety of test functions to assist in determining which method of test data generation performs the best.

Chapter Nine

The Results of Test Data Generation Using Adaptive Search Techniques for a Range of Test Functions

9.1. Introduction

There exist many software functions more complicated than that which have been used so far to illustrate test data generation, and DeMillo and Offut (1988) have specified a number in their work with mutation analysis. Using these test functions as a guideline for designing additional ones, this chapter introduces a collection of difficult functions with which to test the capabilities of random test data generation and adaptive search techniques.

These test functions include two versions of the classic Trityp problem and the Find program, all of which will be described in their respective sections in this chapter, and finally the sample function which has been used throughout this thesis with a much expanded search space. The functions will be compared to random test data generation for both path testing and the testing of LCSAJs and branches. The largest problem with these tests is the measurement of unique test data. A great deal of the tests were performed on an IBM386 compatible in the programming language, C. The size of the search space created problems as it is almost impossible to measure unique data sets when the search space exceeds 100,000 points. There have been adaptations, however, made to C to allow much larger arrays and the use of this larger array will be demonstrated for a function.

9.2. The Trityp Problems

The Trityp problem is concerned with determining the type of triangle created by three given variables. In the first problem, Trityp (easy), the program is first required to determine if the three variables can form a triangle and if so, is it an equilateral, isosceles or scalene triangle. The second program, Trityp (hard), has the additional task of determining whether it is a right-angled triangle. The types of triangle are displayed in figure 66.

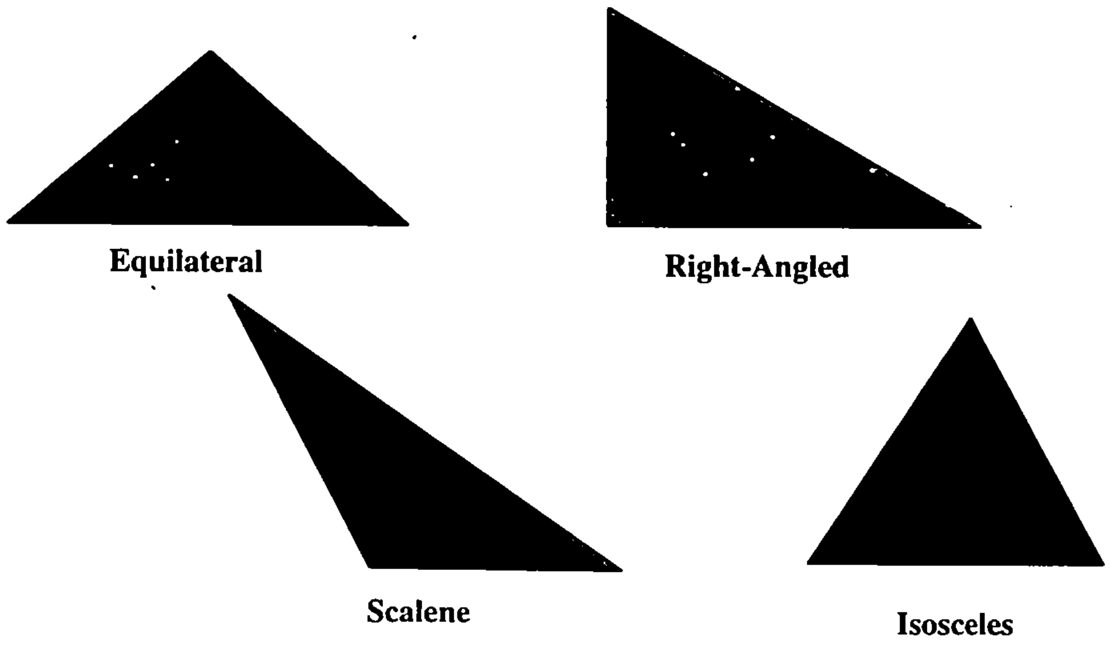


Figure 66 - Types of Triangles Distinguished by Trityp (Easy) and Trityp (Hard)

9.2.1. Trityp (Easy)

The Trityp easy program begins simply by determining if the three variables (x,y,z) involved will make a legal triangle, that is if all sides are greater than 0. The three variables are tested as follows

if (x > 0 || y > 0 || z > 0).

If the triangle is legal, the next step is to test how many sides of the triangle are of the same length. This is performed as follows:

```
type = 0;
if ( x == y )
{
        type += 1;
}
if ( x == z)
{
        type += 2;
}
if ( y == z)
{
        type += 3;
}
```

The value of **type** is then used to determine if the triangle is equilateral, scalene, or isosceles as follows:

```
if ( type == 0 )
{

     if ( (x + y <= z || (y + z) <= x || (x + z) <= y )
      {
            type = 4 ;          //illegal triangle
      }
      else
      {
            type = 1 ;          //scalene
      }
}
if ( type > 3 )
{
      type = 3 ;                // equilateral
}
else if ( type == 1 && (x + y)> z )
{
      type = 2 ;                // isosceles
}
else if ( type == 2 && (x + z)> y)
{
      type = 2 ;                // isosceles
}
```

```
else if ( answer == 3 && (y + z) > x)
{
        type = 2 ;              //isosceles
}
else
{
        type = 4 ;              //illegal triangle
}
```

The original code is in appendix A and flow chart in appendix B. Through the code there are a possible 121 paths but only 10 of these paths are feasible. The code can be broken down to 17 LCSAJs and 13 branches as shown in Appendix C. The size of the search space is 41^3 or 68921 and each variable is of the range [-20..20]. The results of the test runs for LCSAJs and branches are shown in table 48.

1000 Runs	GAs	SA	Tabu	GAs-SA	GAs-TS	Random Testing
Average unique data sets	320.34	349.37	676.6	643.28	476	3487.93
Standard deviation	202.19	183.35	171.1	492.19	274.01	2831.74
Average generations required	682.39	478.03	48.61	768.87	30	3647.48
Minimum unique data sets required	42	52	163	50	35	162
Maximum unique data sets required	1418	1560	1780	3064	1590	20341
Average % of search space searched	0.46%	0.51%	0.98%	0.93%	0.69%	5.06%
Average new data sets/generation	0.47	0.73	13.92	0.84	15.87	0.96

Table 48 - Comparison of Adaptive Search Techniques against Random Test Data Generation for Trityp (Easy) Over Search Space of 68921 for LCSAJs and Branch Testing

These results indicate that GAs in requiring on average 320.34 unique data sets per run, performed the best for the Trityp(easy) program for LCSAJs and branch testing. The method with the second best results was Simulated Annealing which achieved coverage with an average 349.37 data sets. The hybrid GAs-SA did not perform as well as its predecessors, requiring on average 643.28 unique data sets. Random performed the worse, as it required 5.06% of the population to be viewed prior to coverage as compared to 0.46% for GAs. The improvement of GAs over random generation was 91%, and 90% for Simulated Annealing. All the adaptive search techniques achieved coverage through exploring less than 1% of the search space while Tabu search required the most at 0.98% of the search space.

The frequency chart is in figure 67. The peak of the GAs is within the range 200-250 with 136 runs completing, for Simulated Annealing 143 runs completed in the range 300-350. The hybrid GAs-TS, 142 runs completed in the range 300-350 unique data sets, the second smaller peak for GAs-TS is in the range 800-850 with 78 data sets. The other hybrid technique, GAs-SA achieved 120 within the range 150-200, and Tabu search achieved coverage for the most runs, 143 in the range 650-700. Finally random generation, whose maximum range is the greatest at 20341 and outside the scope of the graph, had a minor peak between 1100 and 1150 data sets of 19.

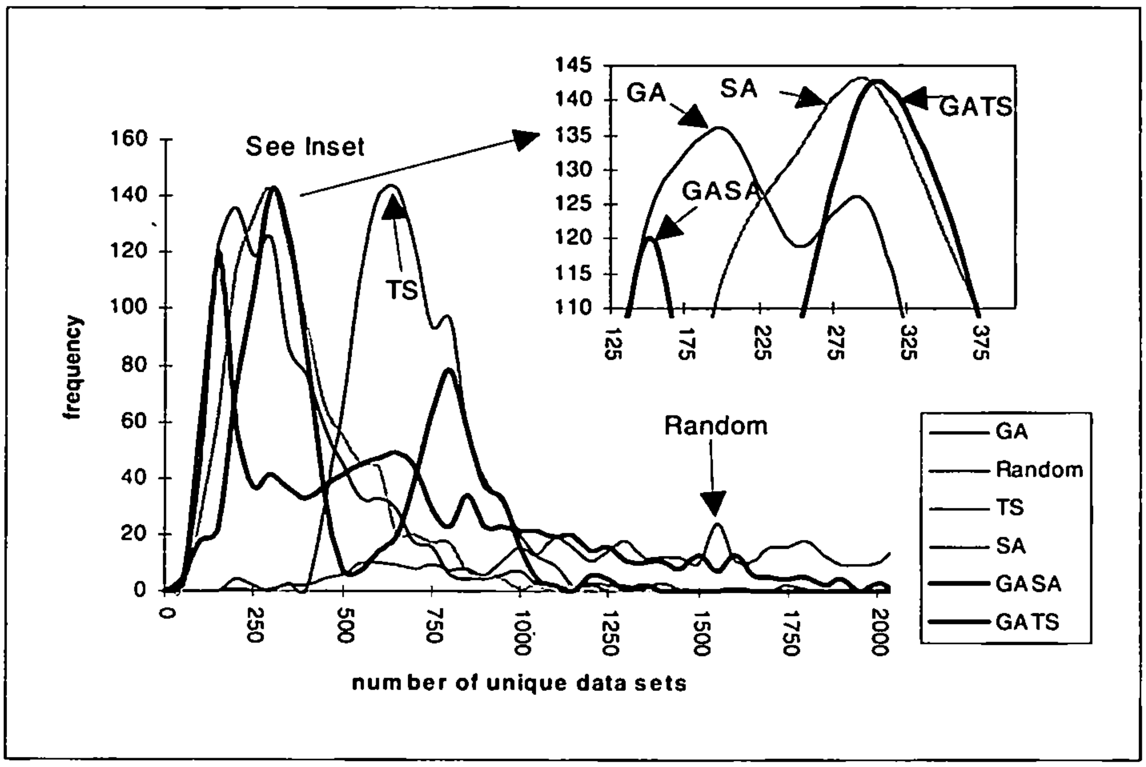


Figure 67 - Frequency Comparison of Adaptive Search Techniques Against Random Test Data Generation for Trityp (Easy) Over Search Space of 68921 for LCSAJs and Branch Testing

In path testing for the Trityp (easy) program, a run should find all 10 paths, these paths are listed in table 49 with their corresponding percentage of search space. Path number two will be the most difficult path to cover as there are only 20 data sets which test this path. The results for path testing are in table 50.

	Path	% of Search Space
1	1T	88.39%
2	1F2T3T4T7T	0.03%
3	1F2T7F	0.14%
4	1F3T7F	0.14%
5	1F4T7C	0.41%
6	1F5T6T7T	5.35%
7	1F4T7F	0.14%
8	1F3T7B	0.41%
9	1F2T7B	0.41%
10	1F5T6F7A	4.57%
	Total	100%

Table 49 - Paths Through Trityp(Easy) Program and their Respective Amounts of Search Space

1000 Runs	GAs	SA	Tabu	GAs-SA	GAs-TS	Random Testing
Average unique data sets	230.04	337.38	721.0	663.95	456.75	3580.45
Standard deviation	187.05	191.68	318.3	624.50	261.57	2968.09
Average generations required	459.55	489.34	173.5	1236.78	29.08	3752.87
Minimum unique data sets	26	38	101	30	49	174
Maximum unique data sets	1381	1294	3309	4053	1501	21158
Average % of search space	0.33%	0.49%	1.05%	0.96%	0.66%	5.20%
Average new data sets/generation	0.50	0.69	4.16	0.53	15.71	0.95

Table 50 - A Comparison of Techniques for Path Testing on Trityp (Easy) Over Search Space of 68921.

In path testing there is an improvement over random testing of 94% for GAs, 91% for Simulated Annealing and 87% for the hybrid GAs-TS. While the results of Tabu search and the hybrid GAs-SA are better than those with random generation, these two search methods do not appear to be as effective as GAs and Simulated Annealing. Of the minimum amount of test data possible before complete coverage of 10, GAs was the closest as one run required only 26 unique data sets to be generated, in second place was the hybrid GAs-SA which required only 30. Tabu search which performed the worse of the adaptive search techniques had a minimum data set requirement of 101 which is nearly four times the amount for GAs.

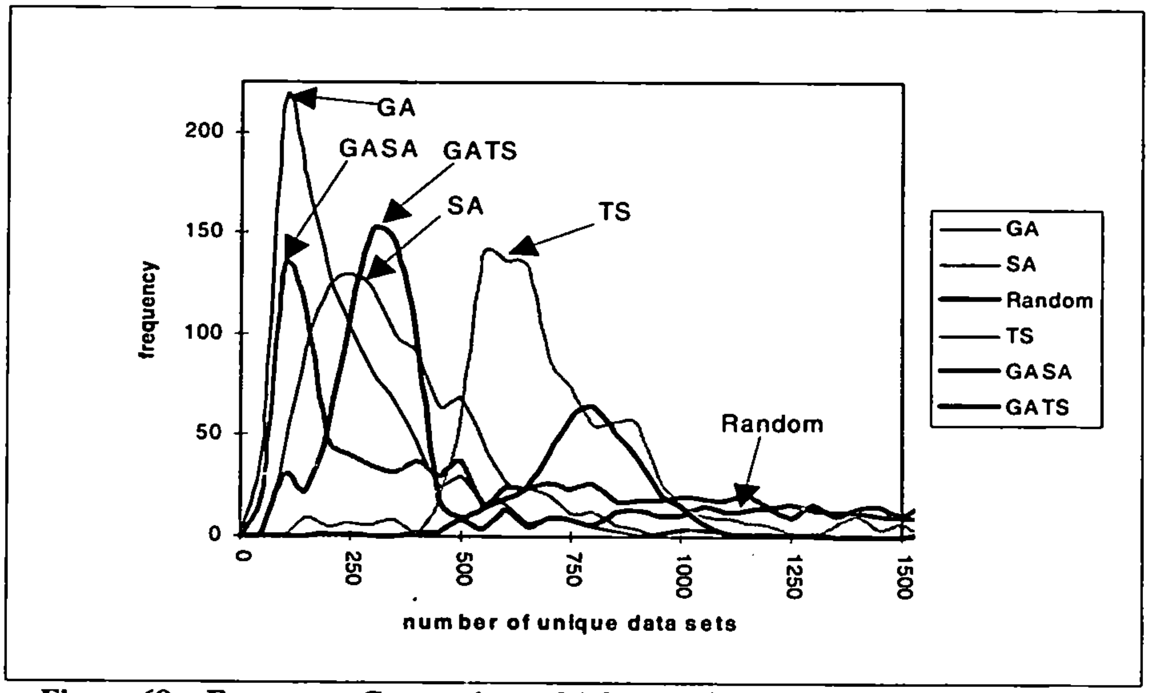


Figure 68 - Frequency Comparison of Adaptive Search Techniques against Random Test Data Generation for Trityp (Easy) Over Search Space of 68921 for LCSAJs and Branch Testing

Figure 68 is a frequency chart for all the methods, the peak for GAs is in the range 100 to 150 with 215 runs completing, GAs-SA had 133 runs in the range 150-200, Simulated Annealing had 130 runs finishing in the range 250-300. Additionally there were 152 runs of GAs-TS in the 300 to 350 range, Tabu search finished 141 runs between 550 and 600 and finally random generation had a peak of 15 runs in the range 1050-1100. GAs have outperformed the other adaptive search techniques in the Trityp(easy) problem. Surprisingly the hybrid GAs-SA gave average results much worse than GAs and Simulated Annealing, in fact the results with GAs was 65% better than the hybrid method and Simulated Annealing was 49% better. This is quite a bit better than the improvement of 11% over GAs and 8% over Simulated Annealing demonstrated in the test program used in previous chapters.

9.2.2. Trityp (Hard)

The Trityp (hard) problem is more difficult than the Trityp (easy) in that the program also determines whether a given triangle is a right-angled triangle. This is done by solving Pythagorus's theorem by determining if the square root of the two smaller sides, squared, are equal to the third side, *i.e.*

$$\sqrt{x^2 + y^2} = z$$

This function uses the same search space as in Trityp (easy) and within the space there are 42 variable combinations which solve this equation, that is 14 for each form. This makes the search much more restrictive. There are 18 LCSAJs and 23 branches to be exercised, listed in appendix D. The results are shown in table 51, these indicate that the best performer is the hybrid GAs-TS which required 1006.60, the second best is Tabu search which required 1381.23 and the third is the other hybrid method, GAs-SA. These offered a 90%, 86% and 61% improvement, respectively over random generation, Simulated Annealing and GAs offer a 44% and 42% improvement over random generation. The frequency chart is in figure 69.

1000 Runs	GAs	SA	Tabu	GAs-SA	GAs-TS	Random Testing
Average unique data sets	5640.34	5436.94	1381.23	3723.42	1006.6	9655.66
Standard deviation	2929.41	3057.76	571.08	1992.88	373.17	5405.47
Average generations required	15359.0	10008.0	102.75	3960.97	46.11	10719.5
Minimum unique data sets	430	343	205	311	309	883
Maximum unique data sets	21501	26808	3832	12151	3296	37335
Average % of search space	8.18%	7.89%	2.00%	5.40%	1.46%	14.01%
Average new data sets/generation	0.36	0.54	13.44	0.94	21.83	0.90

Table 51 - Comparison of Adaptive Search Techniques against Random Test Data Generation for Trityp (Hard) Over Search Space of 68921 for LCSAJs and Branch Testing

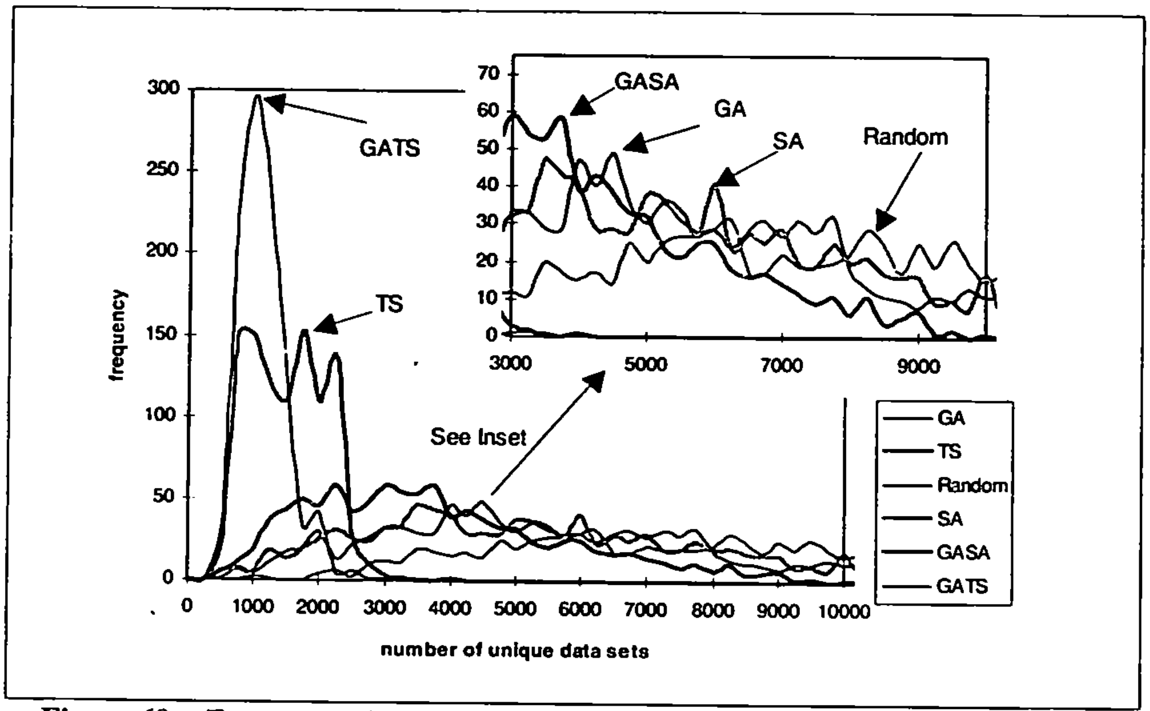


Figure 69 - Frequency Distribution of Adaptive Search Techniques against Random Test Data Generation for Trityp (Hard) Over Search Space of 68921 for LCSAJs and Branch Testing

The peak for GAs-TS is between 1000 and 1250 data sets with 297 runs completing in this range, Tabu search had two major peaks, one in the range 750 to 1000 with 152 data sets and the other between 1750 and 2000 with 153 data sets. The other search methods were not as effective, the GAs-SA had a mild peak between 3750 and 4000 of 59, GAs completed the most runs in the range 4500 - 4750 with 49 and Simulated Annealing in the range 3500 and 3750 had 47. Random generation completed 31 runs in the range 6250 to 6500.

The hybrid GAs-TS performed better than GAs alone by 82% and Tabu search alone by 27% for LCSAJs and branch testing. Is this success limited to this problem or will path testing give such impressive results? There exists 14 paths through the Trityp (hard) program which need to be exercised. Each path and its percentage of the population is shown in table 52, these paths correspond with the function code given in appendix E and the flow chart in appendix F. The most difficult paths to cover are numbers 12, 13 and 14, each of which is only exercised by 14 test data sets. Path number 1 is exercised by more than half the data sets, this path determines if the input contains a negative number as a triangle can not have a side of length less than zero.

	PATH	% of Search Space
1	1T	51.2%
2	1A	25.00%
3	1B	12.19%
4	2F3F4F5T6T	0.03%
5	2F3F4T	1.93%
6	2F3T	1.93%
7	2F3F4F5F7F8T	0.41%
8	2T	1.93%
9	2F3F4F5F7T	0.41%
10	2F3F4F5T6F	0.41%
11	2F3F4F5F7F8F9F10F11F	4.5%
12	2F3F4F5F7F8F9T	0.02%
13	2F3F4F5F7F8F9F10F11T	0.02%
14	2F3F4F5F7F8F9F10T	0.02%
	Total	**100%**

Table 52 - Paths Through Trityp (Hard) with the Percentage of Total Population (68921) which Exercises the Path

The amount of unique test data required to satisfy these paths is shown in table 53 for each test data generation technique. Again the hybrid GAs-TS with an average of 1011.97 unique data sets outperformed all other adaptive search techniques as well as random generation, the next best technique was again Tabu search which required 1377.60 data sets. GAs-TS showed an improvement over random generation of 89%

and over Tabu search of 27%. Simulated Annealing and GAs did do better than random generation but required 4.21 and 4.28 times more data sets than the hybrid GAs-TS.

1000 Runs	GAs	SA	Tabu	GAs-SA	GAs-TS	Random Testing
Average unique data sets	4333.52	4266.9	1377.6	3498.31	1011.97	9468.09
Standard deviation	2842.14	2431.8	748.25	2131.34	379.38	5096.93
Average generations required	12454.6	9169.6	215.60	4126.18	45.1	10461.47
Minimum unique data sets	210	294	192	304	423	1332
Maximum unique data sets	17023	15421	7524	11761	3208	34872
Average % of search space	6.29%	6.19%	2.00%	5.08%	1.47%	13.74%
Average new data sets/generation	0.35	0.47	6.39	0.85	22.44	0.91

Table 53 - A Comparison of Techniques for Path Testing on Trityp (Hard) Over Search Space of 68921.

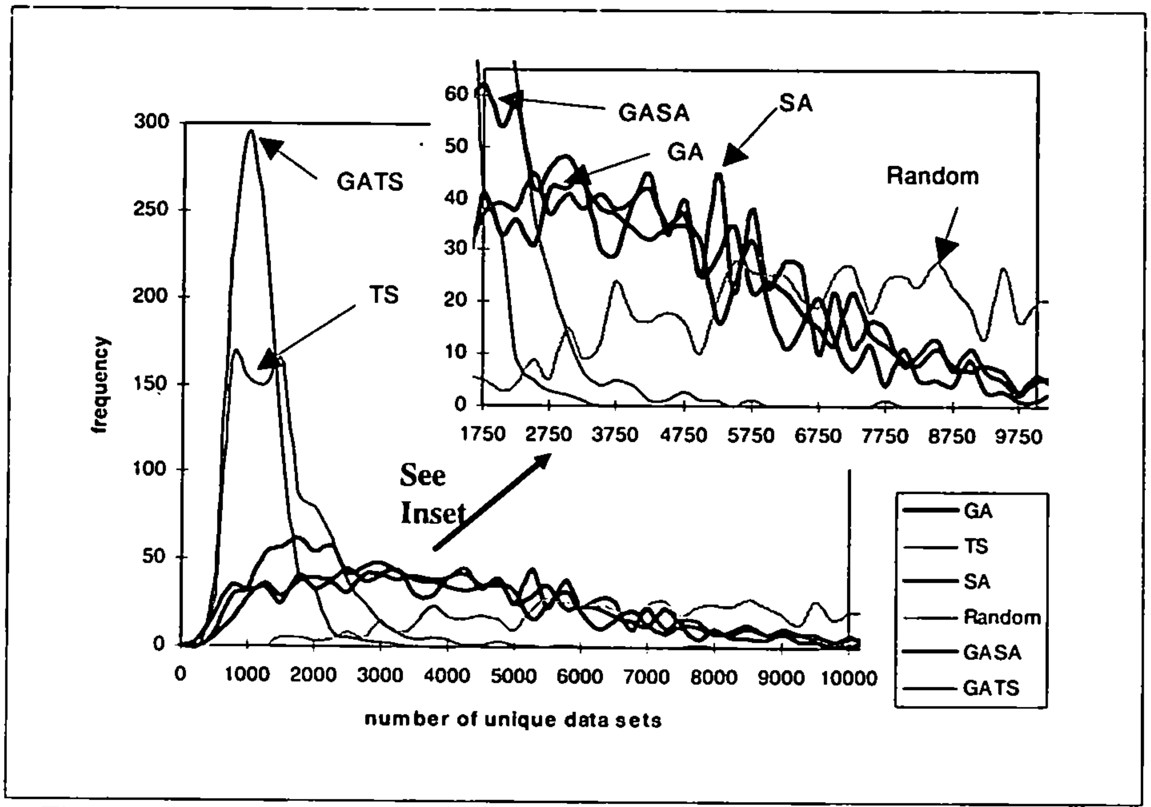


Figure 70 - Frequency Distribution of Adaptive Search Techniques against Random Test Data Generation for Trityp (Hard) Over Search Space of 68921 for Path Testing

The frequency chart in figure 70, the peak for GAs-TS is in the range of 1000 to 1250 data sets with 296 or 30% of the 1000 runs completing, for Tabu search the range is 750 to 1000 with 168 runs, the second peak is between 1500 and 1750 data sets with

163 runs. Simulated Annealing produced 45 runs in the range 2500 to 2750, GAs-SA had 62 runs in the range 1750 to 2000 and GAs had 44 runs in the range 3250 to 3500.

9.2.3. Summary for Trityp Examples

When attempting the Trityp (easy) program the GAs performed the best with Simulated Annealing in second place, however for the Trityp(hard) program the results from using the hybrid GAs-TS are very impressive when compared to the other adaptive search technique. These Trityp problems do not contain loops however, and the next program, Find, will introduce quite complicated loops. Will the hybrid technique perform well here or will GAs or Simulated Annealing?

9.3. The Find Program

The Find program is more complicated than the Trityp in that it involves loops, indeed more loops than were used for the sample function. The purpose of the program is to sort an array of integers. The function consists of an array of values (A) of length N, and index F, so that all values below A(F) in the array are less than or equal to A(F) and all those above are greater than A(F). The Find program consist of 7 input variables, the integer array A which contains 5 variables between 1 and 5, F (the index) which is an integer between 0 and 5, and N, which is the length of A to be used for the sort, an integer between 1 and 5. The total search space is therefore 5^7 or 78,125 possible combinations. The input variables are encoded into a bit string of length 21, 3 bits to each variable. The procedure consists of four loops all of which need to be exercised 0, 1 and 2 times and seven conditional statements (see appendix

G for the flow chart and appendix H for the program code). Therefore there are 19 branches and 39 LCSAJs to test, appendix I. The aim is for 89% coverage for branches and 100% for LCSAJs with a total coverage metric of 95% as some of the branches and LCSAJs are unobtainable. Normally this would not be known in advance, but coverage of 95% is still a very high figure. The comparison of the methods follows in table 54. In this program the best performance is by Simulated Annealing which required on average 466.41 unique data sets, the hybrid GAs-SA was second with 541.99 and GAs was third with 609.53. Tabu search and GAs-TS have actually performed worse than random generation which required only 898.51 data sets compared to 3389.41 for Tabu search and 3335.80 for GAs-TS. Simulated Annealing required on 0.78% of the search space to be generated whereas the worse performer Tabu search required 4.34%.

1000 Runs	GAs	SA	Tabu	GAs-SA	GAs-TS	Random Testing
Average unique data sets	609.53	466.41	3389.41	541.99	3335.80	898.51
Standard deviation	365.69	303.17	1591.85	313.56	1916.73	613.50
Average generations required	903	599.07	88.39	587.03	8.13	1188.47
Minimum unique data sets	88	61	720	58	73	112
Maximum unique data sets	3512	2710	10600	2647	10451	4029
Average % of search space	0.78%	0.59%	4.34%	0.69%	4.27%	1.15%
Average new data sets/generation	0.67	0.78	38.34	0.92	410.31	0.76

Table 54 - A Comparison of Techniques For LCSAJs And Branch Testing For Find Over Search Space Of 78125

The frequency chart is in figure 71, the peak for Simulated Annealing is in the range 500 to 750 with 405 data sets, GAs-SA with 398, GAs with 354, and random testing with 243 are in this range as well. GAs-TS peak in the range 2500 to 2750 with 90 data sets, and Tabu search has a small peak between 2750 and 3000 data sets with 125 runs completing in this range.

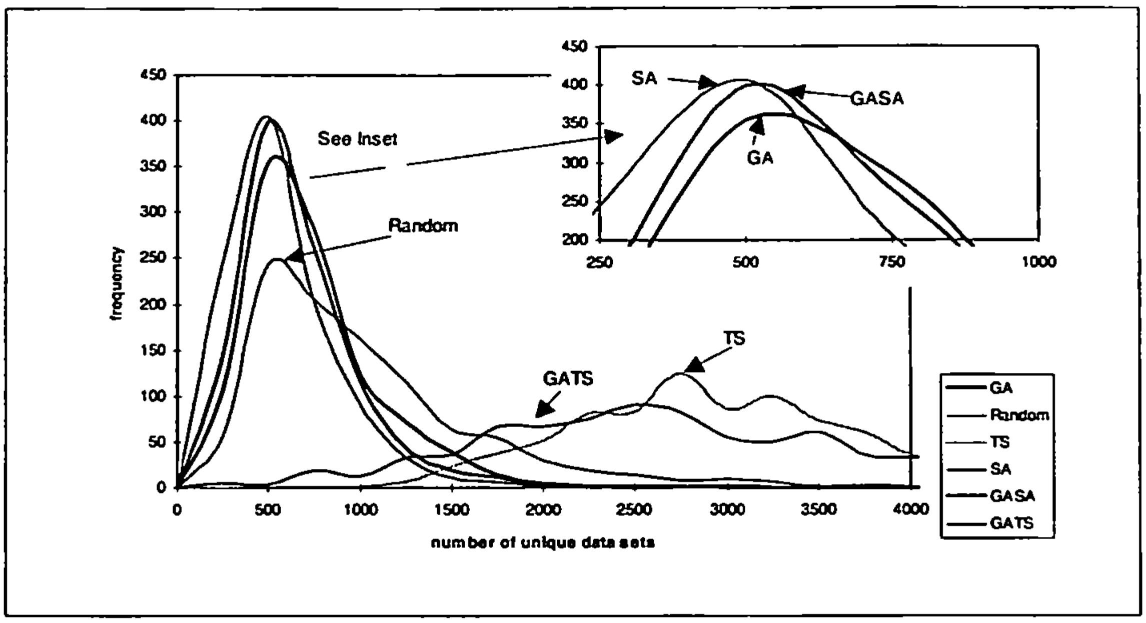


Figure 71 - Frequency Distribution of Adaptive Search Techniques against Random Test Data Generation for Find Over Search Space of 78125 for LCSAJs and Branch Testing

There are many more paths in the Find problem than the test programs used thus far. There are 80 paths which must be covered, listed in appendix J. To test each of these paths is a very time consuming task and the results are given in table 55. The best performance was again by Simulated Annealing which required 1743.58 or 2.23% of the search space before complete coverage, GAs were very close and needed on average 1794.84 unique data sets or 2.30% of the search space. Random testing was once again better at covering all the paths than Tabu search and the hybrid GAs-TS. Random generation required 9.58% of the search space while Tabu search and GAs-TS needed 20.95% and 10.83% respectively. The large amount of test data required by Tabu search and the hybrid technique is probably due to the large size of the neighbourhood, as each neighbourhood consists of 342 members.

1000 Runs	GAs	SA	Tabu	GAs-SA	GAs-TS	Random Testing
Average unique data sets	1794.84	1743.5	16364	1913.87	8461.46	7485.34
Standard deviation	780.74	754.33	5749.3	885.90	4161.89	3072.07
Average generations required	2362.79	2335.3	63.90	2484.10	25.79	7946.08
Minimum unique data sets	597	671	4029	751	2595	2552
Maximum unique data sets	6667	6989	35563	10040	23777	25243
Average % of search space	2.30%	2.23%	20.95%	2.45%	10.83%	9.58%
Average new data sets/generation	0.76	0.75	256.09	0.77	328.09	0.94

Table 55 - A Comparison of Techniques for Path Testing for Find Program Over Search Space of 78125.

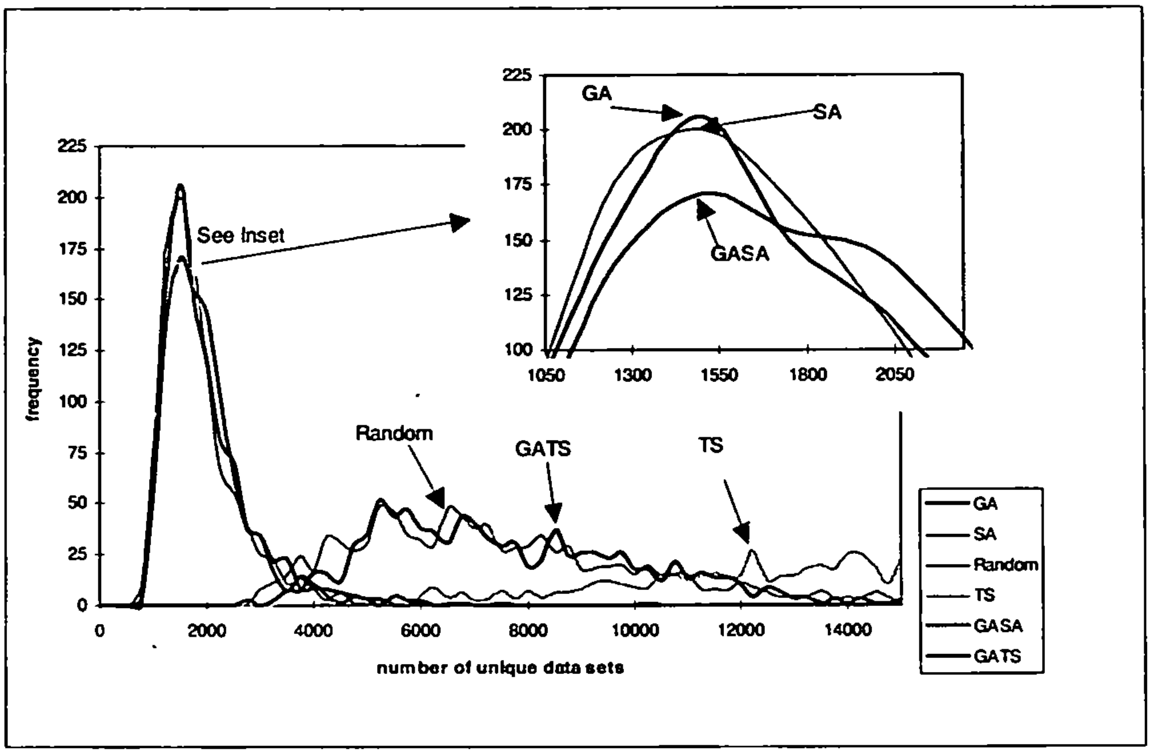


Figure 72 - Frequency Distribution Comparing Adaptive Search Techniques to Random Test Data Generation for Find Over Search Space of 78125 for Path Testing

The frequency chart in figure 72, the peak for Simulated Annealing, GAs and GAs-SA are all in the range 1500 to 1750 unique data sets with 200, 206 and 171 respectively. Random generation and GAs-TS both peaked in the range 5250 to 5500 with 49 and 52 respectively. Finally Tabu search has a small peak in the range 12250 to 12500 with 27 data sets, the longest run for Tabu search looked at 35563 data sets, almost half of the entire search space, whereas the longest run for random generation completed at 25243 data sets, less than a third of the search space. This compares to

Simulated Annealing which only needed to generate a maximum 6989 data sets, approximately one-eleventh of the search space and GAs which need approximately one-twelfth of the search space to be generated.

9.4. Discussion of Test Program Results

Technique	GAs	Simulated Annealing	Tabu Search	GAs-SA	GAs-TS	Random Testing
Trityp (L)(easy)	320.34①	349.37	676.64	643.28	476	3487.93
Trityp (P)(easy)	230.04①	337.38	721.07	663.95	456.75	3580.45
Trityp (L)(hard)	5640.34	5436.94	1381.23	3723.42	1006.6①	9655.66
Trityp (P)(hard)	4333.52	4266.99	1377.6	3498.31	1011.97①	9468.09
Find (L)	609.53	466.41①	3389.41	541.99	3335.8	898.51
Find(P)	1794.84	1743.58①	16364	1913.87	8461.46	7485.34
Average	2154.77	2100.11	3984.99	1830.80①	2458.10	5762.66

Table 56- Overall Results of the Three Test Programs Each for LCSAJs and Branch Testing (L) and Path Testing (P). The Average Figure Given is the Average Amount of Unique Test Data Required for the Method for all Six Tests. (The ① indicates the best result for each test)

The average amount of test data required for each of the techniques is shown in table 56 over all of the six tests. The average amount of test data is calculated for the techniques to give an overall picture of how they fare. Although GAs-SA did not get the least amount of test data in any of the programs it has on average the least amount with Simulated Annealing second and GAs third. Simulated Annealing did perform better than GAs on 4 of the six test programs, but each achieved two first places while the winner of the other two programs, Trityp (hard), was the hybrid, GAs-TS. While Tabu and GAs-TS gave good results for the Trityp(hard) program it did very poorly in the other tests, but on average all the adaptive search techniques did perform better than random test data generation. In the next section, the three best performing techniques GAs, Simulated Annealing and GAs-SA, are applied to a larger search space.

9.5. A Larger Search Space

The generation of unique test data for small search space was originally limited to the capacity of allowable arrays in the programming language C. Fortunately it is now possible to expand this search space through a specially written module which accepts arrays of greater than 100000 points. Therefore it is possible to determine the average unique data sets in a larger search space. Additionally the average time it takes to cover the program will be registered to compare the adaptive techniques and to determine the time difference between the small and large search. The sample function under test which was used in chapters 6,7 and 8 is tested for paths in an enlarged search space of one million. The range size for the three variables is now [-50..50] which is attained using a string of size 21, each seven bit string represents one input variable. There are 15 paths to be exercised, each one and its percentage of search space is shown in table 57. Paths number 2 and 6 should be the most difficult to cover as there are only 4 data sets out of 1,000,000 which exercise these. Most runs should have no trouble covering path 11 as more than 78% of the search space exercises it.

	PATH	Number of Test Data Sets	% of search space
1	1T2F3FL0	584	0.0584%
2	1T2F3TL0	4	0.0004%
3	1T2F3TL1	116	0.0116%
4	1T2F3TL1L2	42805	4.2805%
5	1F3FL0	15149	1.5149%
6	1T2T3TL0	4	0.0004%
7	1T2T3FL0	290	0.0290%
8	1F3FL1	32917	3.2917%
8	1T2T3TL1	116	0.0116%
9	1T2T3FL1	2902	0.2902%
10	1F3FL1L2	780234	78.0234%
11	1T2T3TL1L2	42805	4.2805%
12	1T2T3FL1L2	38458	3.8458%
13	1T2F3FL1	3436	0.3436%
14	1T2F3FL1L2	40180	4.0180%
	Total	1000000	100%

Table 57 - Paths through Sample Function with the Amount of Test Data within Search Space which Satisfies Each Path and Its Percentage of Total Search Space

	Sample Function - Path Testing			
100 runs	GAs	Simulated Annealing	GAs-SA	Random Testing
Average unique data sets	2012.62	2665.70	2961.12	294338.80
Standard deviation	1272.47	1989.80	4966.81	209038.90
Average generations required	2283.09	3188.88	3470.08	388867.40
Minimum unique data sets required	176	535	353	6379
Maximum unique data sets required	6274	12767	49294	732600
Average % of search space searched	0.20%	0.27%	0.30%	29.43%
Average new data sets/generation	0.88	0.83	0.85	0.76
Average Run Time (seconds)	1.21	1.07	1.12	117.88

Table 58 - A Comparison of Techniques for Path Testing for the Sample Function Over Search Space of 1000000

Table 58 is the comparison of the three best testing methods, GAs, Simulated Annealing and GAs-SA for the generation of test data for path testing. Included in these details is the average time required for each run. GAs produced the best results for this large search space and required 2012.62 or 0.20% of the search space before coverage. Simulated Annealing with an average of 2665.70 unique data sets, and GAs-SA with 2961.12, are very close behind. These results show an improvement over random generation of approximately 99% for all three of the methods.

Figure 73 shows the frequency distribution for the three adaptive search methods, the peak for GAs is in the range 1250 to 1500 unique data sets with 16 of the 100 runs completing in this range, for Simulated Annealing the range 1500-1750 has 18 runs completing and for GAs-SA, 9 runs finish in the range 2000-2250.

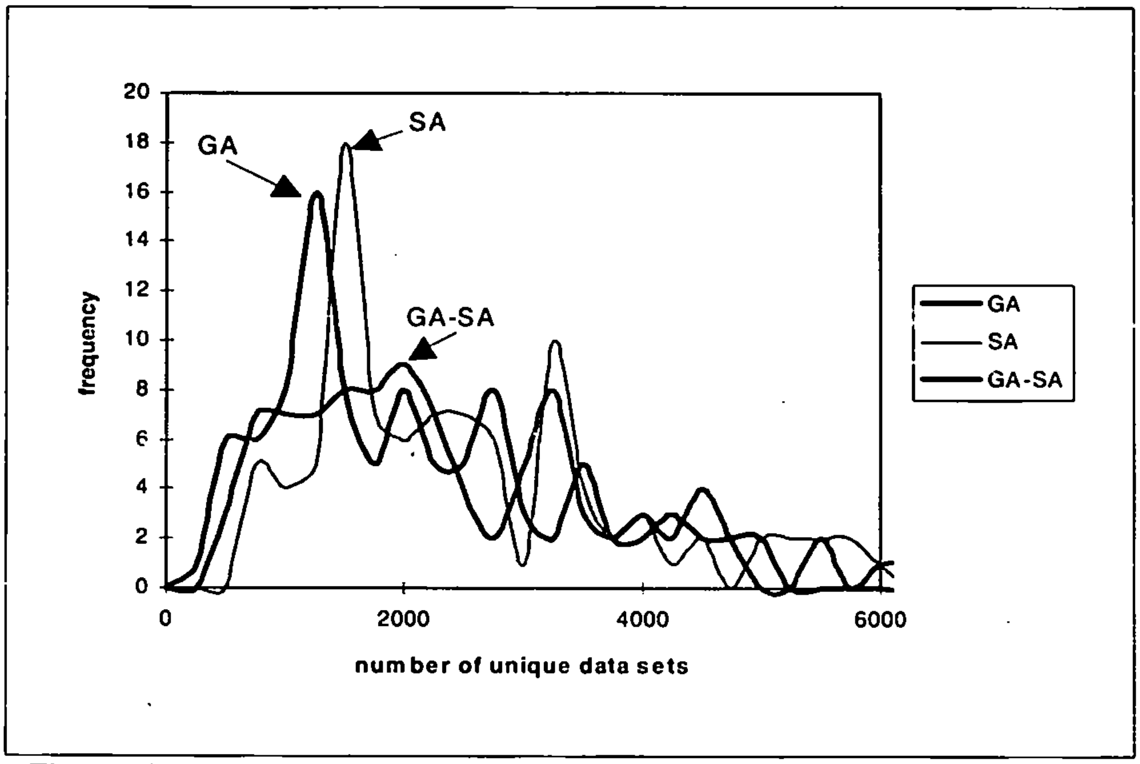


Figure 73 - A Frequency Comparison of Three Adaptive Search Techniques, GAs, Simulated Annealing and GAs-SA, for Path Testing for the Sample Function Over Search Space of 1000000

Shown in table 59 is a time comparison between the same function as in table 58 for the small search space, 9261, and the large search space, 1000000. These results indicate that as the search space size increases, GAs, Simulated Annealing and GAs-SA, continue to perform quickly, while random test data generation's ability to compete deteriorates greatly. Random generation took nearly two minutes per run, while GAs, Simulated Annealing and GAs-SA all took little more than a second. This is most likely due to the fact GAs, Simulated Annealing and GAs-SA, only needed to search between 0.20% and 0.30% of the search space as opposed to random generation which required 29.43%.

	Average Number of Seconds Before All Paths Were Exercised			
Search Space	GAs	Simulated Annealing	GAs-SA	Random Generation
Small (9261)	0.35	0.27	0.29	0.78
Large (1000000)	1.21	1.07	1.12	117.88

Table 59 - Average Time Comparison (in seconds) of Sample Function Using Both Small Search Space and Large Search Space

9.6. Conclusion

This chapter demonstrates the capabilities of adaptive test data generation against that of random testing. The chapter begins with a fairly simple but commonly used test program and then advances to a more complicated version of the same test. The next test program, Find, includes more variables, a slightly larger search space, and complicated loops to be searched. The results of all these tests indicated that adaptive search techniques usually perform better within the search space than random test data generation. This is usually the case except in the final program, Find, where the results of Tabu search and the hybrid GAs-TS were actually worse than those of random generation. This can be explained by the large size of a neighbourhood which needs to be generated at each stage. This is contrasted to the Trityp(hard) program where in path testing, Tabu search and GAs-TS did extremely well, in fact Tabu search showed an improvement over GAs of 75% while GAs-TS had an improvement of 82%.

GAs performed the best on the Trityp(easy) program although only 8% better than Simulated Annealing for LCSAJs and Branch testing, but 32% better in path testing. Simulated Annealing performed the best of the adaptive search techniques on the

Find program, where it averaged the minimum amount of unique test data, 14% better than the nearest result GAs-SA for LCSAJs and branch testing and 2.9% better than GAs in path testing. Although GAs-SA never succeeded in achieving the lowest amount of unique test data for any of the programs, it did have on average the smallest average amount overall. Using the example of path testing for instance, GAs (4333.52) and Simulated Annealing (4266.99), were not as successful as Tabu search (1377.60) and GAs-TS (1011.97), but GAs-SA achieved a figure of 3498.31, improving its overall average performance.

The search space was never very large for these example programs, therefore the three overall best techniques were compared against random generation for a larger search space. In this example, GAs outperformed both Simulated Annealing and GAs-SA in contrast to the results achieved on the same program when using a small search space where GAs-SA performed the best. However, the difference between the techniques is quite small and these adaptive techniques perform much better than random generation. The time comparison also indicated a great saving in time for the adaptive techniques in comparison to random test data generation. The most savings are in Simulated Annealing, which does not have to endure the same memory requirements of GAs which go through the procedure of reproduction, crossover and mutation. The hybrid technique, GAs-SA, only performs the GA process for a short time, 50 generations, and therefore ranks second in time.

The purpose of this chapter was to widen the test suite of programs to which adaptive techniques had been applied. The next chapter will apply the adaptive search techniques of GAs and Simulated Annealing to a test program written for the purpose of optimising capital budgeting.

Chapter Ten

A Demonstration of Automatic Test Data Generation on a Program which Optimises Capital Allowances for Company Taxation

10.1. Introduction

The coverage of program code has thus far been shown on demonstration problems. While all these programs perform routines which are necessary they could still be referred to as notional problems. This chapter introduces coverage testing for a program written not for the purpose of demonstrating testing techniques, but to optimise tax benefits. The program will be used to compare the two more effective methods of adaptive search, GAs and Simulated Annealing, to random test data generation.

10.2. A Description of the Program to be Tested

The purpose of this program is to optimise the net present value of cash flows. The effect of the UK taxation system is to create financial opportunities by the prudent selection of projects which a company can undertake at the most beneficial time and the application of an efficient rule for loss handling. A project is defined as an outgoing capital expenditure, such as on plant and equipment, made by the company with the aim of receiving income. This capital expenditure can be offset against profit at a rate of 25% a year. A reduction in profit will decrease the amount of tax to be paid, see table 60, and therefore increasing the funds available for dividends.

Therefore the objective is to pay less tax, but there are other opportunities available for reducing the tax bill and increasing dividends, these are the rules which govern losses in any given year. If in year one there was a profit of £350,000, tax would be paid at 25% for the portion to £300,000 and at 35% for the remaining £50,000, that is a tax bill of £92,500. In year two there is a loss of £55,000, and there are two options for writing off this loss. Option A is called "carry backward", where the amount of the loss is deducted from previous years' profits, going backward a limit of three years before preceding to the second option, option B, "carry forward" where the loss reduces the profit over the next years. Option B can be undertaken without having exhausted option A. Therefore, if option A was chosen the £55,000 is reduced from the profit and there is a tax rebate of £18,750. The benefit of option A over option B is that the NPV is higher if the tax rebate is realised early on in the calculations.

Tax Band of Profit	Tax Rate
less than £300,000	25%
300,000 to 1,500,000.00	
0 - 300,000	25%
300 - 1,500,000	35%
1,500,000 +	33%

Table 60 - Tax Rate Bands on Profit

The final decision to be made is when or if a project should begin, this can be in any year during the current budgeting lifecycle, a six year period, and can be stopped and restarted at any time. Should the project be stopped, all assets will be sold and must be repurchased before it can be restarted.

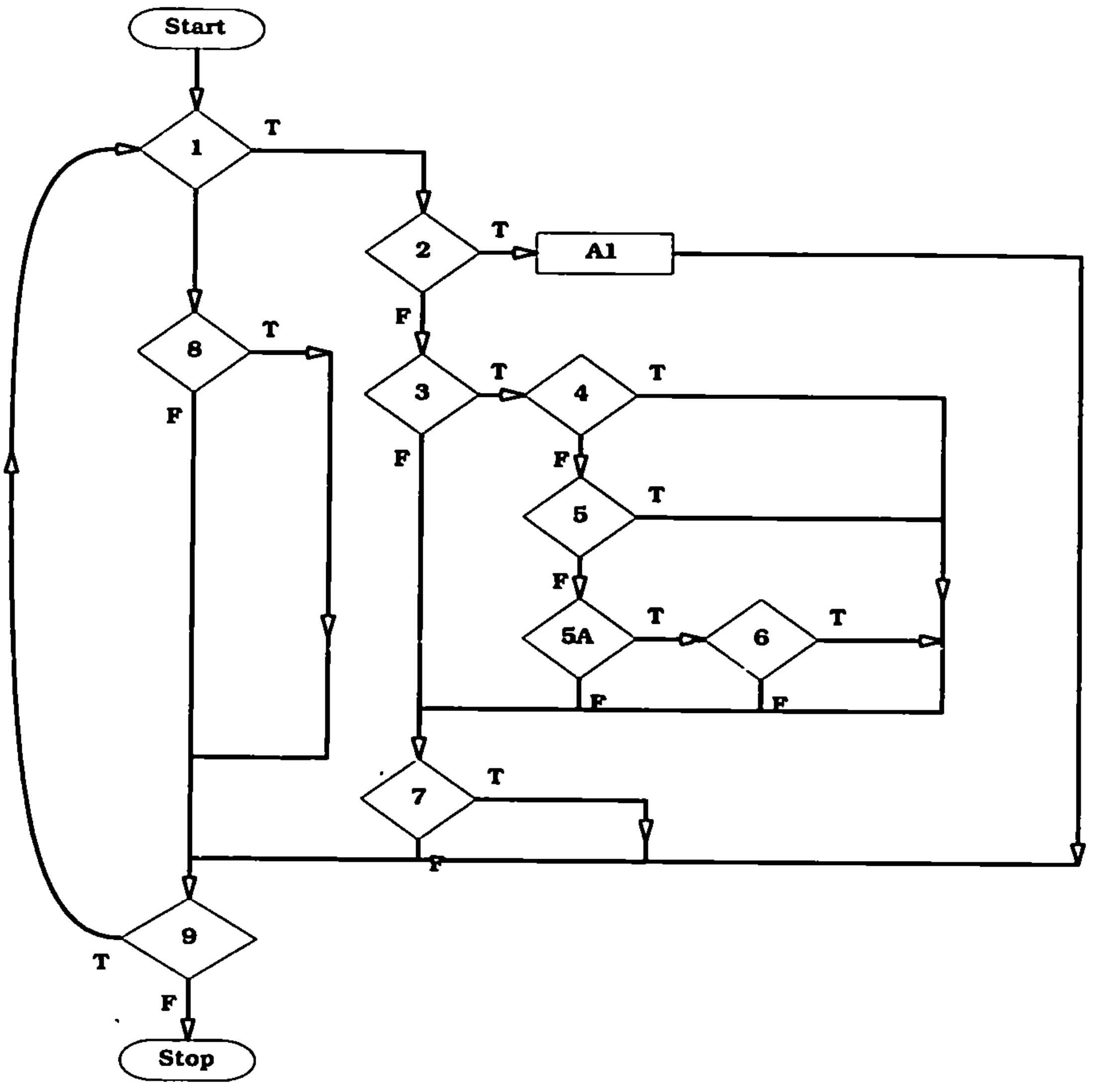


Figure 74 - Flow Chart of Capital Budgeting Program

10.3. Testing the Program

This very complicated capital budgeting problem is itself a candidate for optimisation using GAs (Berry and Smith, 1993;Farrar, 1995), but the program also illustrates the need for testing. If, using the assumption that there are three projects from which to choose, then there are 24 different choices to be made. These, 6 for each project, determine if a project should begin or finish in any given year. The remaining six determine whether a loss should be carried forward or backward in a given year. A listing of the code is in appendix K. The flowchart for the program is in figure 74.

This establishes that there are eight decisional points, seven of which register true and false, and one (5) which includes an if/elseif/else statement. There are over 1000 feasible paths through the program, 500 are listed in appendix L.

In addition to the decisional input variables which guide the program, there is also the current financial situation of the company to consider, and any predictions of profit on selected projects. These consist of 31 input variables, each between the range [0..8,000,000.00] for every 1000 positions. A list of the input variables representing the potential financial situation of the company are listed in table 61.

Financial Information			Time Period								
			t-3	t-2	t-1	t	t+1	t+2	t+3	t+4	t+5
Income						1,500,000					
Capital Allowances						1,000,000					
Taxable Profits			3,000	60,000	100,000						
Forecast Operations Income							1,000,000	3,500,000	5,000	89,000	990,000
Projects											
Project 1	Initial Outlay	2,000,000									
	Estimated Cash Inflows					10,000	20,000	30,000	40,000	50,000	100,000
Project 2	Initial Outlay	100,000									
	Estimated Cash Inflows					5,000	8,000	9,000	12,000	20,000	25,000
Project 3	Initial Outlay	6,000,000									
	Estimated Cash Inflows					100,000	200,000	300,000	400,000	550,000	650,000

Table 61` - Company Financial Information which are the 31 Required Input Variables.

10.4. The Comparison of Testing Techniques on the Test Program

The binary string which represents the input variables consists of the 24 decisions (24 bits) and the 31 input variables each represented by a 13 bit combination giving a total string length of 427.

As has previously been determined there are over 1000 feasible paths through the program, 500 of these are listed in appendix L. To save run-time a run will conclude when it has exercised 500 paths, a TER of approximately 50%. The results are given in table 62 for 500 runs of each of the three methods.

500 Runs	Random Testing	Simulated Annealing	GAs
Average unique test data sets	117361.40	12746.1	18398.06
Minimum required	76864	7327	10270
Maximum required	151829	43612	29236
Range (minimum..maximum)	74965	36285	18966
Average number of generations	123530.83	12746.38	2065.99
Estimated time/run (seconds)	675.48	84.72	199.85

Table 62 - Results of Function Under Test Comparing GAs, Simulated Annealing and Random Test Data Generation for Capital Budgeting Program for Path Testing

GAs again performed much better than random test data generation, however Simulated Annealing gave the best results for this program needing on average only 12746 new data sets before coverage of the 500 paths compared to 18398 by GAs. The minimum amount of test data required for a run was 500 as there are 500 paths, a run of the GA managed to find all the paths using a minimum 10270 data sets, while a run of Simulated Annealing succeeded in only 7327. The range between minimum and maximum was much smaller for GAs at 18966 than Simulated Annealing at 36285, and the maximum amount of test data sets required by a run of Simulated Annealing was much higher at 43612 compared to GAs at 29236. The random generator however required an average of 117361.40 new data sets before coverage, the minimum for a single run was 76864. Simulated Annealing showed an improvement over the result achieved by random generation of 89% and GAs showed an improvement of 84%.

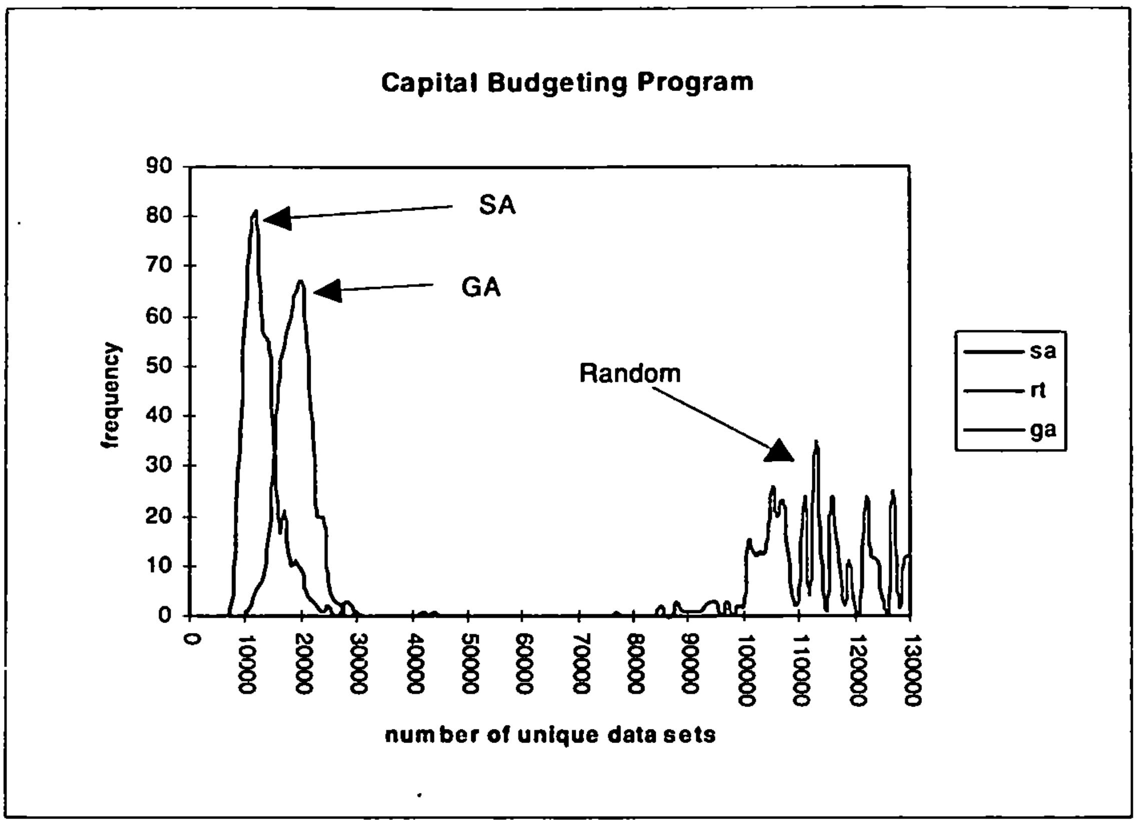


Figure 75 - A Comparison of Frequency between GAs, Simulated Annealing and Random Generation for Capital Budgeting Program for Path Testing

The frequency graph in figure 75 shows the peak for Simulated Annealing between 12000 and 13000 data sets with a maximum number of runs, 81 completing in this range, the peak of the GAs was between 20000 and 21000 data sets with 67 runs. Random generation completed the most runs within the range 113000 and 114000 with 35. The graph does not show the complete distribution for random generation as it exceeds the range, however this graph indicates the results of random testing are much worse than those achieved with Simulated Annealing or GAs.

10.5. Conclusion

In chapter 9, the sample function was run for a large search space, 1000000 and in this chapter the capital budgeting program also had a very large search space. Are the results comparable? In the sample function Simulated Annealing and GAs showed an improvement over random of 99%, however GAs performed 25% better than Simulated Annealing. While the results are similar for the capital budgeting program used in this chapter, Simulated Annealing has achieved better results than GAs.

Simulated Annealing and GAs improved over the result of random generation by 89% and 84%, respectively. However Simulated Annealing showed an improvement over GAs of 31%. Therefore Simulated Annealing was the much better performer in this example. One item of note is the estimated time for each run, GAs taking on average more than twice as long as Simulated Annealing. This is similar to the results achieved in the sample function with a large search space where GAs took longer than Simulated Annealing to achieve a result. The test used in this chapter expected to cover only 500 of the over 1000 paths through the code, a TER of approximately 50%, a further test would need to determine the effectiveness of the methods when complete coverage is expected.

Chapter Eleven
Discussion and Further Research

How do adaptive search techniques perform in the generation of test data? The results shown in this thesis indicate that while not all adaptive techniques will always perform better than randomly generated test data, they usually require fewer amounts of unique test data to be generated. However it might not be cost effective to create a one-off Simulated Annealing tool to generate test data, this could be extremely time consuming and the development phase error prone in itself. What needs to be done is to incorporate an adaptive search technique into a testing tool which will both determine what sort of test data is required and then generate that test data requiring only limited input from the user. This model of a testing tool incorporating adaptive search techniques is what is described in this thesis. The merit of the techniques used and the method in which they are applied and measured are discussed in this section along with suggested future research that might be performed in this area.

This research project began with an attempt to determine how GAs might perform in the generation of test data to follow work performed by Xanthakis *et al* (1992). That research involved the test data generation for a Pascal program, but also included specifications for delving further into the specifications of a test function by investigating the test function by adjusting the code and using GAs and a relaxation method to find suitable test data. The research performed here was an attempt to generate test data with as little knowledge of the test function (black-box testing) as possible, while using white-box testing techniques to determine paths which need to be exercised within the test code. This meant the performance of test data within the function under test was measured only by returning a listing of paths exercised by the test data, therefore keeping the disturbance of the function under test to a minimum. The fitness function received no additional information to assist in its search other than what path had been exercised. The responsibility of this fitness function was to determine whether a path, branch or LCSAJs had been previously exercised and to reduce its fitness each time it had. A path which is newly discovered only returns a good fitness for a short while, as each time it is subsequently exercised its fitness is reduced. This is seen as an original contribution to knowledge.

Of primary concern to any user who requires test data is the amount of time required to generate this test data. Unfortunately time was an issue which could not be addressed until later stages of the research for this thesis due to the availability of facilities to perform the required tests at a speed and size which could illustrate the capabilities of adaptive techniques. However, it was determined that not only the time involved in the generation of test data should be an issue in measuring the effectiveness of a technique. A further issue was the amount of test data which was generated before a required coverage metric was reached. A great deal of test data can be generated randomly, but a lot of this test data duplicates previous paths, branches or LCSAJs. What would be considered beneficial is to reduce the amount of new test data which needs to be generated, this in turn will reduce the amount of time a function under test needs to be run. Requiring the function under test to be run fewer times offers savings, in that while generating test data can be a time consuming task the actual running of a function with complex and time consuming calculations, can be even more expensive.

In later tests however, it was possible to measure the time taken for the generation of the testing techniques for the sample function used in this thesis. The results indicate that for a smaller search space random generation takes twice as long as both GAs and Simulated Annealing. Unfortunately, this difference is a comparison of an average rate of one-third of a second to two-thirds of a second, not a very large difference. However, when the size of the search space is enlarged the adaptive search techniques take little more than a second, while random generation increases to nearly two minutes on a Pentium PC. GAs therefore are 98.97% faster while Simulated Annealing are even quicker by 99.1%.

When one looks at the amount of new test data which had to be generated for coverage on the small search space, adaptive searches require only approximately 50% of that required by random generation. When the size of the search space increases, adaptive techniques require the generation of approximately 0.25% of the entire search space while random generation requires 30%. These results are for a fairly simple function under test, results for a more complex search space would need to be examined before confirmation could be given as to the effectiveness in comparison of time.

An inherent weakness to using adaptive search techniques for the generation of test data is that the search space for a function does not look like a normal search area, there are no hills, peaks or valleys. In fact the search space is non-existent until the testing process begins, and can only be described as a flat plateau of branches, paths or LCSAJs all waiting to be exercised. As each is exercised its fitness rises sharply out of the plateau, however as subsequent test data sets exercise that same area the fitness of that path lowers and all test data which satisfies that area will continue to reduce the fitness of the path. Figure 76 illustrates the fitness of an exercised area as it is exercised more and more.

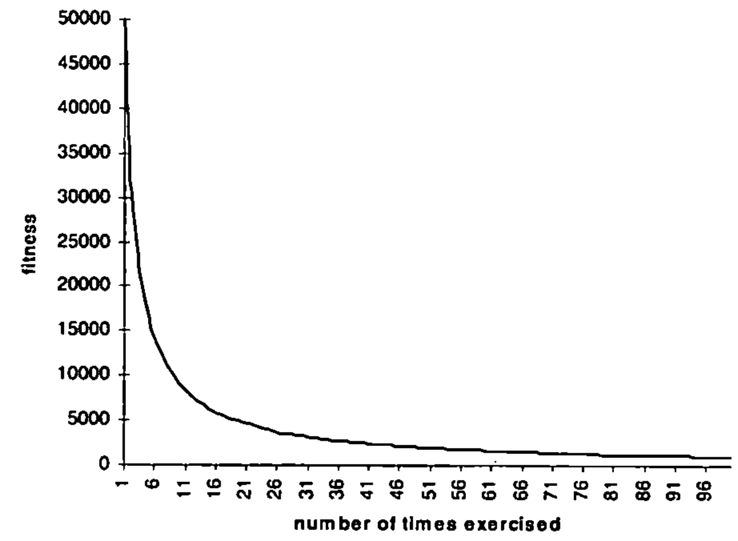


Figure 76 - The Fitness of an Exercised Path, Branch or LCSAJs as it is Accessed Subsequent Times, This Demonstrates How the Fitness Declines Each Successive Time it is Exercised

While the fitness of this exercised path is declining other areas are being exercised as well, and are rising quickly and then declining at different rates. Therefore a test data set which exercises a new area is rewarded, but there is only limited benefit from this reward to encourage searching a new direction. If this fitness were to be set to zero as a path was exercised, there would be no mechanism to compare one data set to another. One data set may be slightly fitter than another as one path had been exercised less times, and usually a reduction in fitness means good portions of the data sets can be maintained for reproduction, crossover and mutation in GAs, for the annealing process in Simulated Annealing, or for a neighbourhood search in Tabu. The information maintained by the fitness function, the list of exercised paths, branches or LCSAJs could be thought of as a Tabu list which restricts and penalises test data which has been used before or that has exercised the same area of the test code.

It is difficult to determine how a test data set can be rewarded for being close to a new area which has yet to be exercised. This would be advantageous as a combination which is close to attaining a new path can be encouraged to move in that direction. This would require more manipulation of the function under test to ascertain how close a test data set is to this boundary. However future research should address these two issues. First, how should the fitness function be structured to accommodate 'closeness' to an unexercised area? Secondly, how can 'closeness' be determined while keeping disturbance to the original function to a minimum? Work has been done in this area by Sthamer *et al* (1994) which rewards a test data set that tests the boundaries of the conditional statements by taking the hamming distance between the test data and the conditional statement requirements. This is a very good idea but it would appear that the function would have to undergo some detailed change to determine the input data and its relationship to the conditional statements would require an additional function added to the original code which moves from a mostly black-box procedure to a white-box procedure of software testing.

The hybrid techniques incorporating GAs and SA and GAs and Tabu search to the problem of software test data generation is an original contribution to knowledge. Unfortunately these techniques have not proved as successful as their predecessors in all tests, but in some examples they were more successful. In the demonstration function under test, GAs-SA outperformed GAs by 3.50% and Simulated Annealing by 5.35% for LCSAJs and branch testing, and for path testing showed an improvement of 11.45% and 7.7% for GAs and Simulated Annealing, respectively. While for the hybrid GAs-TS the results were only slightly different, for LCSAJs and branch testing GAs-TS improved on GAs results by 38% and Tabu search by 42%, but for path testing while GAs-TS improved over Tabu search by 9.95%, the results were not as good as those achieved using GAs alone. In the test programs the results were more varied, the hybrid GAs-TS were the best on the Trityp (hard) program, much better than GAs or Simulated Annealing, but on the Find program the results were dismal, worse than those achieved with random generation. The hybrid GAs-SA performed best on average for the six tests, although it never achieved the best results for any given test. In these hybrid techniques the GA was run for approximately one fourth of its average required generation before the local search began. Tests with this figure need to be performed to determine if this is indeed the optimum.

Tabu search performed extremely well on the Trityp (hard) program, 75.5% better than GAs and 74.5% better than Simulated Annealing for LCSAJs and branch testing. This was surprising as it performed worse, requiring twice as much test data than GAs and Simulated Annealing for the Trityp (easy) program. The only explanation can be the nature of the search space and when a test data set finds one of the more difficult branches, it has a good chance of finding the others, and more research should be performed on this result. Tabu search unfortunately does generate a lot of test data as the number of input variable increase, such as the Find program. This fact discounted it from use in the larger programs, especially for the capital budgeting program where the number of input variables was great. A larger list size or penalty function may help in restricting the search and there is scope for future research.

The discussion over which is the best technique, GAs or Simulated Annealing continues. These two adaptive search methods always performed better than random test data generation from the lowest improvement of 32% for GAs in the find program when testing for LCSAJs and branch testing to 99% for the large demonstration function. In the smaller program the results for GAs and Simulated Annealing were never far apart, in some GAs performed better (Trityp (Easy)) in others Simulated Annealing did better (Find). In the demonstration function GAs performed better at LCSAJs and branch testing while Simulated Annealing did better on path testing. In the larger program there was again the slight contrast in the results as GAs performed 25% better than Simulated Annealing but in the capital budgeting program Simulated Annealing was 31% better than GAs. Simulated Annealing does have the benefit of being quicker in time trials, this can most likely be explained in that GAs work with a population, while the Simulated Annealing algorithm is using a single string which is then mutated producing one new string at a time. Future research into the GAs would help in determining whether reproduction of the population using the roulette wheel is in fact the best method for this type of problem. Perhaps what is required is a method of reproduction which allows for a greater disparity in new population members. The percentage of mutation and crossover remained static for this research and changes to these, as well as to the rate of change to the string in Simulated Annealing, should be investigated.

With the exception of the capital budgeting program in chapter 10, most of the test functions used in this research have been very limited, and while this is a weakness of the research the test functions used have been applied by many researchers into software testing. Applying these test data generation techniques to an actual piece of code written by another programmer would be the next stage of research. This would demonstrate the abilities of this test data generation tool to adapt to the coding style of other programmers, and illustrate the capabilities of the test data generator when the amount of feasible paths through the program is not known in advance. A further benefit would be to generate test data from outside a user defined range to determine whether the code contains error-handling routines that will determine if a received piece of test data falls within the specified acceptable guidelines.

A program which does not reject incorrect data can be corrected prior to general release. What, however, defines incorrect data? Poor data could exceed a specified range of acceptable values but it could also include such things as a real number instead of an integer (Miller and Spooner, 1976), a character instead of a string, an unacceptable string or an array instead of a pointer. A thorough test must address all these potential problems and must produce test data which examines all constraints within, and external to, their boundaries.

Finally, research should be performed into objected-oriented programming. This is completely removed from the type of functions which have been tested in this research. In testing structured programs it is possible to test function by function, but in object-oriented programming the program must be tested in its entirety, and as objects have no links with one another, so it is not as easy to pre-determine paths which might exist within the code. Therefore the entire program needs to be run and each object be recorded as accessed (Poston, 1994; Jorgensen and Erickson, 1994). This can be a very time consuming process which could possibly be shortened by the quick generation of test data.

This chapter discusses the research performed in this thesis and aims to point out any shortcomings which may exist as well as to suggest further research which could be performed to alleviate these. The chapter concludes with a discussion of the direction which the research may take in the future, such as additional programs or the testing of functions which require test data which is of the type real, characters or strings. The final area of future research is to use this method of test data generation for object-oriented programming.

Chapter Twelve

Conclusion

When this research began very little had been performed into the use of adaptive search techniques for test data generation. In fact the research has covered a period of three years, during which time there have been parallel developments. To keep as up to date as possible, this has necessitated a refocussing of original objectives and consequent conclusions. What existed, in 1992, was a collection of techniques for testing programs which ranged from static to dynamic methods. Since that time, the use of adaptive search techniques for the generation of test data (Xanthakis *et al* (1992); Sthamer *et al* 1994; Roper, 1995) has been attempted. These techniques have been compared to a method which has been used for test data generation and performs quite adequately for small search space. This method is random testing. Determining how random testing compares to adaptive search techniques for coverage of a function under test, has been the focus of this thesis.

The primary goals of the research were as follows:

- develop a tool for the automatic generation of test data;
- measure the effectiveness of adaptive search technique for the generation of test data in comparison with random test data generation;

- develop hybrid adaptive search techniques and compare these with the original methods used;
- lay a course for further examination of test data generation and adaptive search techniques.

The objectives of this research have been met in the following manner:

- a tool for the automatic generation of test data has been developed and described;
- a comparison of adaptive search techniques, GAs, Simulated Annealing and Tabu search to random test data generation has shown that in almost all tests these techniques perform better, and that GAs and Simulated Annealing always perform better than random generation;
- hybrid techniques have been developed which combine the adaptive search techniques GAs and Simulated Annealing and GAs and Tabu search. Results indicate these techniques occasionally perform better than their predecessors;
- these adaptive search techniques have been applied to test programs to illustrate their capabilities. These techniques have also been applied to a larger search problem not written specifically as a demonstration of test data generation.

Areas of future research have been identified, these fall in two categories, adaptive search techniques and test data generation. Future research in adaptive search can concentrate on the following areas:

- adjustment of operators on current techniques (e.g. inclusion of hyper-mutation or random immigrants for changing fitness environment in GAs);
- introduction of gray coding as opposed to binary interpretation of strings in both GAs and Simulated Annealing;
- fine-tuning of hybrid techniques;
- other adaptive search techniques (evolutionary programming, guided local search, population-oriented Simulated Annealing, *etc.*).

Further research should also be performed to improve the test data which has been generated, and two suggested areas of research are:

- boundary testing - the attempt to attain test data which not only exercises a given path but also identifies if it is close to the boundary of a condition. This must be performed while attempting to remain black-box testing. A relaxation algorithm may help with this problem;

- mutation analysis - using mutation analysis to verify the quality of the generated test data to determine if it will kill mutant versions of the code under test.

Finally, additional future research as discussed in chapter eleven, would be into a more complex function written by external sources. This will give the opportunity to determine how adaptive techniques perform when the goals are unknown. It is hoped that these adaptive search techniques, with refinements, will continue to produce the high quality results as shown in this thesis.

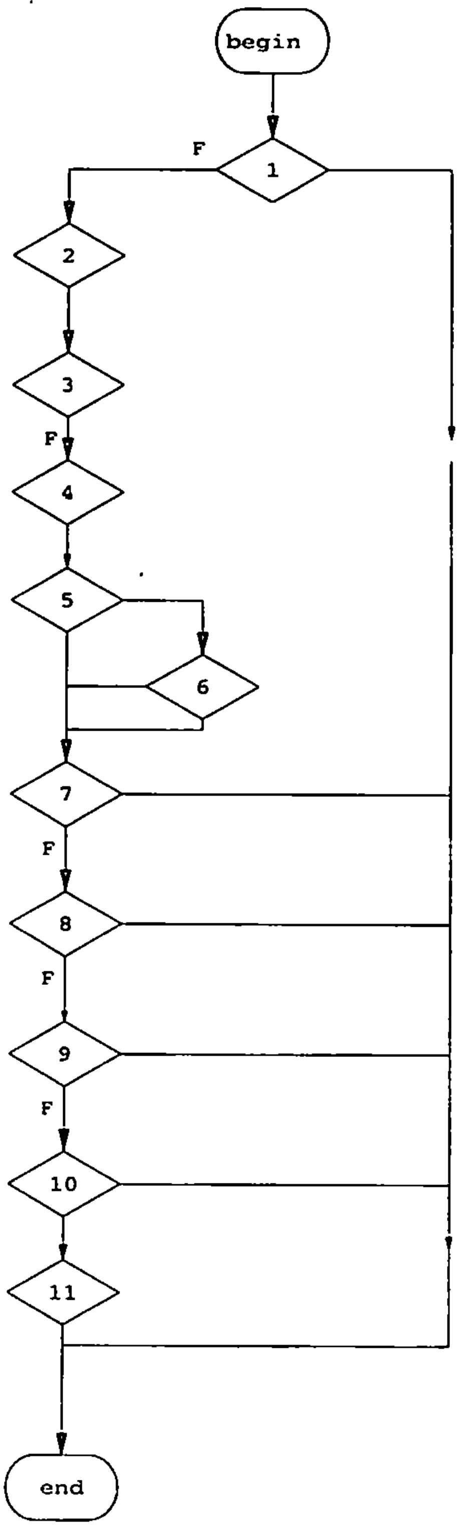
Appendix A

Original Code for Trityp (Easy)

```
void TritypEasy(int x,int y, int z)
{
int type ;
if ( x <= 0 || y <= 0 || z <= 0 )
{
        type = 4 ;  //illegal triangle
}
else
{
        type = 0 ;
        if ( x == y )
        {
                type += 1 ;
        }
        if ( x == z )
        {
                type += 2 ;
        }
        if ( y == z )
        {
                type += 3 ;
        }
        if ( type == 0 )
        {
                if ( x + y <= z || y + z <= x || x + z <= y )
                {
                        type = 4 ;  //illegal triangle
                }
                else
                {
                        type = 1 ;  //scalene
                }
        }
        if ( type > 3 )
        {
                type = 3 ;  //equilateral
        }
        else if ( type == 1 && x + y > z )
        {
                type = 2 ;  //isosceles
        }
        else if ( type == 2 && x + z > y )
        {
                type = 2 ;  //isosceles
        }
        else if ( type == 3 && y + z > x )
        {
                type = 2 ;  //isosceles
        }
        else
        {
                type = 4 ;  //illegal triangle
        }
}
}
```

Appendix B

Flow Chart of Trityp(Easy)



Appendix C

List of LCSAJs and Branches Through Trityp (Easy)

Branches of Trityp(Easy) Program

1) 1T
2) 1F
3) 2T
4) 3T
5) 4T
6) 5T
7) 6T
8) 6F
9) 7T
10) 7F
11) 7A
12) 7B
13) 7C

LCSAJs of Trityp(Easy) Program

1) 1F2T
2) 1F3T
3) 1F4T
4) 1F5T`
5) 2T3T
6) 2T7A
7) 2T7F
8) 3T4T
9) 3T7B
10) 3T7F
11) 4T7C
12) 4T7F
13) 4T7T
14) 5T6F
15) 5T6T
16) 6F7A
17) 6T7T

Appendix D

List of LCSAJs and Branches Through Trityp (Hard)

Branches of Trityp(Hard) Program

1) 1T
2) 1A
3) 1B
4) 2T
5) 2F
6) 3T
7) 3F
8) 4T
9) 4F
10) 5T
11) 5F
12) 6T
13) 6F
14) 7T
15) 7F
16) 8T
17) 8F
18) 9T
19) 9F
20) 10T
21) 10F
22) 11T
23) 11F

LCSAJs of Trityp(Hard) Program

1) 2F3F
2) 2F3T
3) 3F4F
4) 3F4T
5) 4F5F
6) 4F5T
7) 5F7F
8) 5F7T
9) 5T6F
10) 5T6T
11) 7F8F
12) 7F8T
13) 8F9F
14) 8F9T
15) 9F10F
16) 9F10T
17) 10F11F
18) 10F11T

Appendix E

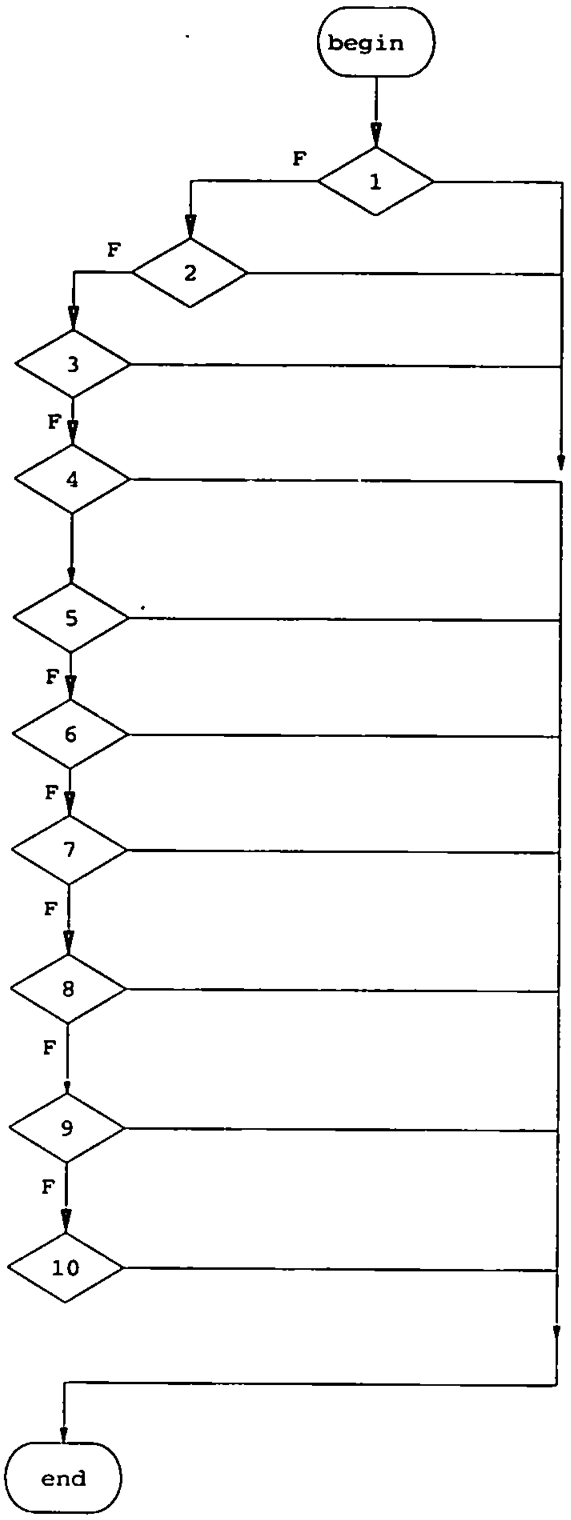
Original Code for Trityp (Hard)

```
int TritypHard(int x, int y, int z)
{
Tot = x+y+z;
if (x<= 0)
{
      return('illegal triangle');
}
else if (y<=0)
{
      return('illegal triangle');
}
else if (z<=0)
{
      return('illegal triangle');
}
if ((x*2)>= Tot)
{
      return('illegal triangle');
}
if ((y*2) >= Tot)
{
      return('illegal triangle');
}
if ((z*2) >= Tot)
{
      return('illegal triangle');
}
if (x == y)
{
      if (y==z)
      {
            return('equilateral');
      }
      return('isosceles');
}
if (x == z)
{
      return('isosceles');
}
if (y == z)
{
      return('isoscoles');
}
if (x*x + y*y = z*z)
{
      return('right-angled');
}
if (y*y+z*z=x*x)
{
      return('right-angled');
}
if (x*x + z*z== y*y)
{
      return('right-angled');
}

                                             }
```

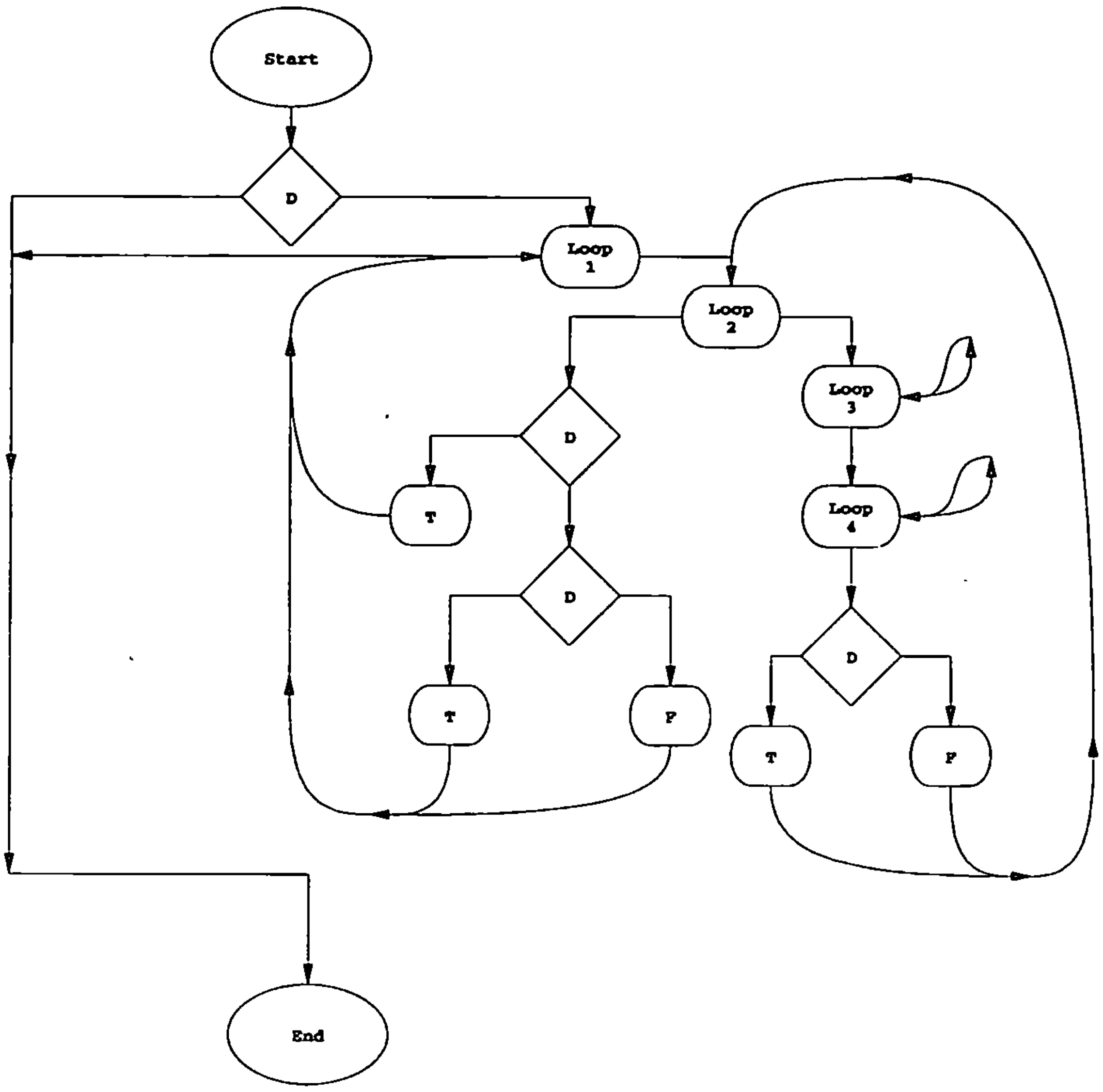
Appendix F

Flow Chart of Trityp(Hard)



Appendix G

Flow Chart of Find Program



Appendix H

Original Code for Find Progam

```
void find(int a[5],int n,int f)
{
int m,ns,r,i,j,w;
if (f>0 && f< n)
{
        m = 0;
        ns = n;
        while (m<ns)
        {
                r = a[f];
                i = m;
                j = ns-1;
                while (i<=j)
                {
                        while (a[i] < r)
                        {
                                i = i+1;
                        }
                        while (r<a[j] )
                        {
                                j. = j-1;
                        }
                        if (i<=j)
                        {
                                w = a[i];
                                a[i] = a[j];
                                a[j] = w;
                                i = i+1;
                                j = j-1;
                        }
                }
                if (f<=j)
                {
                        ns=j+1;
                }
                else if (i<=f)
                {
                        m = i;
                }
        }
}
}
```

Appendix I

List of LCSAJs and Branches Through Find Program

Branches of Find Program

1) 1T
2) 1F
3) 2T
4) 2F
5) 3T
6) 3A
7) 3F
8) A0
9) A1
10) A2
11) B1
12) B2
13) B3
14) C1
15) C2
16) C3

LCSAJs in Find Program

1) 1FA1
2) 1FA1A2
3) 2F3A
4) 2F3F
5) 2F3T
6) 2T3A
7) 2T3T
8) 2TB2
9) 3AA2
10) 3AB1
11) 3AB1B2
12) 3TA2
13) 3TB1
14) 3TB1B2
15) A1B1
16) A1B1B2
17) A2B1
18) A2B1B2
19) B1C0
20) B1C1
21) B1C1C2
22) B2C0
23) B2C1
24) B2C1C2
25) C0D0
26) C0D1
27) C0D1D2
28) C1D0
29) C1D1
30) C1D1D2
31) C2D0
32) C2D1
33) C2D1D2
34) D02F
35) D02T
36) D12F
37) D12T
38) D22F
39) D22T

Appendix J

List of Paths (80) Through Find Program

1) 1FA1B1C0D02T3AA2B1C0D02T3F
2) 1FA1B1C0D02TB2C0D02T3AA2B1C0D02T3AB1C0D0
3) 1FA1B1C0D02TB2C0D02T3AA2B1C0D02T3F
4) 1FA1B1C0D02TB2C0D02T3AA2B1C0D02T3TB1C0D0
5) 1FA1B1C0D02TB2C0D02T3AA2B1C0D12T3F
6) 1FA1B1C0D02TB2C0D02T3AA2B1C1D02T3F
7) 1FA1B1C0D02TB2C0D02T3F
8) 1FA1B1C0D02TB2C0D02T3TA2B1C0D02T3AB1C0D0
9) 1FA1B1C0D02TB2C0D02T3TA2B1C1D02T3F
10) 1FA1B1C0D02TB2C0D12F3AA2B1C0D02T3AB1C0D0
11) 1FA1B1C0D02TB2C0D12F3AA2B1C1D02T3F
12) 1FA1B1C0D02TB2C0D12T3AA2B1C0D02T3AB1C0D0
13) 1FA1B1C0D02TB2C0D12T3AA2B1C1D02T3F
14) 1FA1B1C0D02TB2C0D12T3F
15) 1FA1B1C0D02TB2C0D1D22F3AA2B1C0D02TB2C0D02T3AB1C0D0
16) 1FA1B1C0D02TB2C0D1D22F3AA2B1C0D02TB2C0D12F3AB1C0D0B1C0
D0
17) 1FA1B1C0D02TB2C0D1D22F3AA2B1C0D02TB2C0D12F3AB1C1D0
18) 1FA1B1C0D02TB2C0D1D22F3AA2B1C0D02TB2C1D02F3AB1C0D0
19) 1FA1B1C0D02TB2C0D1D22F3AA2B1C1C2D02T3F
20) 1FA1B1C0D02TB2C0D1D22F3AA2B1C1D02T3AB1C0D0
21) 1FA1B1C0D02TB2C1C2D02F3AA2B1C0D02T3F
22) 1FA1B1C0D02TB2C1D02F3AA2B1C0D02T3F
23) 1FA1B1C0D02TB2C1D02T3AA2B1C0D02T3F
24) 1FA1B1C0D02TB2C1D02T3F
25) 1FA1B1C0D02TB2C1D12F3AA2B1C0D02T3AB1C0D0
26) 1FA1B1C0D02TB2C1D12F3AA2B1C1D02T3F
27) 1FA1B1C0D12T3AA2B1C0D02T3TB1C0D0

28) 1FA1B1C0D12T3AA2B1C0D12T3F

29) 1FA1B1C0D12TB2C0D02T3AA2B1C0D02T3TB1C0D0

30) 1FA1B1C0D12TB2C0D02T3AA2B1C0D12T3F

31) 1FA1B1C0D12TB2C0D02T3F

32) 1FA1B1C0D12TB2C0D12F3AA2B1C0D02TB2C0D02T3F

33) 1FA1B1C0D12TB2C0D12F3AA2B1C0D12T3AB1C0D0B1C0D0

34) 1FA1B1C0D12TB2C0D12F3AA2B1C0D12T3AB1C0D1

35) 1FA1B1C0D12TB2C0D12F3AA2B1C1D02T3TB1C0D0B1C0D0

36) 1FA1B1C0D12TB2C0D12F3AA2B1C1D02T3TB1C1D0

37) 1FA1B1C0D12TB2C0D12F3AA2B1C1D12T3F

38) 1FA1B1C0D12TB2C1D02F3AA2B1C0D02T3TB1C0D0

39) 1FA1B1C0D12TB2C1D02F3AA2B1C0D12T3F

40) 1FA1B1C0D1D22T3AA2B1C0D02TB2C0D02T3TB1C0D0

41) 1FA1B1C0D1D22T3AA2B1C0D02TB2C0D12F3TB1C0D0

42) 1FA1B1C0D1D22T3AA2B1C0D02TB2C1D02F3TB1C0D0B1C0D0

43) 1FA1B1C0D1D22T3AA2B1C0D02TB2C1D02F3TB1C0D1

44) 1FA1B1C0D1D22T3AA2B1C0D12T3TB1C0D0

45) 1FA1B1C0D1D22T3AA2B1C0D1D22T3F

46) 1FA1B1C1C2D02T3AA2B1C0D02T3F

47) 1FA1B1C1C2D02T3F

48) 1FA1B1C1C2D02T3TA2B1C0D02TB2C0D02T3AB1C0D0

49) 1FA1B1C1C2D02T3TA2B1C0D02TB2C0D12F3AB1C0D0B1C0D0

50) 1FA1B1C1C2D02T3TA2B1C0D02TB2C0D12F3AB1C1D0

51) 1FA1B1C1C2D02T3TA2B1C0D02TB2C1D02F3AB1C0D0

52) 1FA1B1C1C2D02T3TA2B1C1C2D02T3F

53) 1FA1B1C1C2D02T3TA2B1C1D02T3AB1C0D0

54) 1FA1B1C1C2D12T3F

55) 1FA1B1C1D02T3AA2B1C0D02T3F

56) 1FA1B1C1D02T3F

57) 1FA1B1C1D02T3TA2B1C0D02T3AB1C0D0

58) 1FA1B1C1D02T3TA2B1C1D02T3F

59) 1FA1B1C1D02TB2C0D02T3AA2B1C0D02T3F

60) 1FA1B1C1D02TB2C0D02T3F

61) 1FA1B1C1D02TB2C0D02T3TA2B1C0D02T3AB1C0D0

62) 1FA1B1C1D02TB2C0D02T3TA2B1C1D02T3F

63) 1FA1B1C1D02TB2C0D12F3AA2B1C0D02T3AB1C0D0

64) 1FA1B1C1D02TB2C0D12F3AA2B1C1D02T3F

65) 1FA1B1C1D02TB2C0D12F3TA2B1C0D02T3AB1C0D0

66) 1FA1B1C1D02TB2C0D12F3TA2B1C1D02T3F

67) 1FA1B1C1D02TB2C1D02F3AA2B1C0D02T3F

68) 1FA1B1C1D02TB2C1D02F3TA2B1C0D02TB2C0D02T3F

69) 1FA1B1C1D02TB2C1D02F3TA2B1C0D12T3AB1C0D0B1C0D0

70) 1FA1B1C1D02TB2C1D02F3TA2B1C0D12T3AB1C0D1

71) 1FA1B1C1D02TB2C1D02F3TA2B1C1D02T3TB1C0D0B1C0D0

72) 1FA1B1C1D02TB2C1D02F3TA2B1C1D02T3TB1C1D0

73) 1FA1B1C1D02TB2C1D02F3TA2B1C1D12T3F

74) 1FA1B1C1D12T3AA2B1C0D02T3TB1C0D0

75) 1FA1B1C1D12T3AA2B1C0D12T3F

76) 1FA1B1C1D12T3F

77) 1FA1B1C1D12T3TA2B1C0D02T3AB1C0D0

78) 1FA1B1C1D12T3TA2B1C1D02T3F

79) 1FA1B1C1D1D22T3F

80) 1T

Appendix K

Code for Capital Budgeting Program

```
#define     LENGTH 427
int         chrom[2][LENGTH]; //binary string
int         OnOff[10];  // project on or off in any year
int         swtch[10];  // project off swtch for deducting capital
                              //allowances
int         TPr = 0;    //number of projects total possible
int         jump = 13;  // binary range of each input variable
int         projs;      // number of projects possible to be done
int         x;          //counter
float       t[6];       //income
float       t1m[3];     //past profit
float       ca[6];      //capital allowances already
float       inOut;
float       proj[10][6];//projects
float       cap[10][6]; // capiatl allowances of projects
float       at1m[9][2];  //new profit (taking away loss)
float       divA[7];    // calculate dividend
float       profit[6];  // profit
float       cForward;   // amount carried forward from year to year
float       taxDue[10]; // tax due for each year
float       taxRebate[10];    //tax returned for each year if any

//The rules of taxations

//determine capital allowances of input financial information
void Inputs(void)
{
for (xa=1;xa<6;xa++)
      ca[xa] = ca[xa-1]-(ca[xa-1]*.25);

for (xa=0;xa<projs;xa++)
{
      cap[xa][ya] = cap[xa][ya-1] - (cap[xa][ya-1] * .25);
}
}
```

```
// confirm no starting balance in these variables
void emptyBalance(void)
{
int n;
for (n=0;n<3;n++)
      at1m[n][0] = t1m[n];
for (n=0;n<3;n++)
{
      taxduein(n);
      swtch[n] = 0;
}
for (n=0;n<10;n++)
{
      taxRebate[n] = 0;
      OnOff[n] = 0;
}
}

// create single binary string from population member
void putin(int member[LENGTH])
{
int y1;
for (y1=0;y1<LENGTH;y1++)
      chrom[0][y1] = member[y1];
}
```

```
// runs program to determine dividend of given inputs
float Profit(int member[LENGTH])
{
int n,m,x;
float fitness = 0;
x = 0;
      putin(member);
      cForward = 0;       // starting balance of carry forward is zero
      makeMoney(x);       // convert binary string to financial
                                //    information
      Inputs();           // create Capiptal Allowance for Each Year
                                //    Based on Financial Information
      emptyBalance();     // set all additional variables to xero
      makeProfit(x);      // calculate profit for each year
      for (n=1;n<7;n++)
      {
            DoYears(n,x,n+17);      // calculate tax due in and
carry                                           // forward/carry
backward
      }
      doDividend(x);                //calculate dividend
      for (n=0;n<7;n++)
            fitness += divA[n];     // fitness = largest dividend
      for(n=0;n<TPr*6;n=n+3)        //deduct for any project
finished    {                             //in given year
            for (m=0;m<TPr;m++)
            {
                  if (chrom[x][m+n] == 1&& swtch[m] == 0)
                  {
                        fitness -=   cap[m][0]/pow(1.06,(n/3));
                        swtch[m] = 1;
                  }
                  else if (chrom[x][m+n] == 0)
                        swtch[m] = 0;
            }
      }
      return(fitness)
}
```

```
//calculate profit for 6 years
void makeProfit(int x)
{
int n;
for (n=0;n<6;n++)
{
      prof(n+1,x);
}
}

// calculate profit based on whether project is running in given
year //(deduct capital allowance from profit
void prof(int year,int x)
{
int n,ProOn;
profit[year-1] = t[year-1] - (ca[year-1]*0.25);  //time t
ProOn = 0;
for (n=(year-1)*TPr;n<((year-1)*TPr)+TPr;n++)
{
      if (chrom[x][n] == 1)
      {
            profit[year-1] += proj[ProOn][year-1];
            profit[year-1] -= cap[ProOn][OnOff[ProOn]]*.25;
            OnOff[ProOn]++;
            ProOn++;
      }
      else
      {
            profit[year-1] -= cap[ProOn][OnOff[ProOn]+1];
            OnOff[ProOn] = 0;
            ProOn++;
      }
}
}
```

```
// Determine if loss should be carried forward (deducted from next
years //profit) of carried backward (deducted from past 3 years
profit before //carried forward).  If Profit calculate tax due in.
void DoYears(int year, int x, int c)
{
if (profit[year-1] < 0)
{
        if (chrom[x][c] == 1)
        {
                forward(year);
        }
        else
        {
                backward(year);
        }
}
else
{
        takeProfit(year);
        at1m[year+2][0] = profit[year-1];
        taxduein(year+2);
}
}

// calculate loss to be carried forward
void forward(int year)
{
                cForward += profit[year-1];
                profit[year-1] = 0;
}
```

```
// deduct loss from last three years profit.  If not enough carry
//forward
void backward(int year)
{
        float taxTemp;
        int j = (year+1);
        taxRebate[year-1] = 0;
        if (profit[year-1] < 0)
        {
                while (profit[year-1] < 0 && j > (year+1)-3)
                {
                        taxTemp = atlm[j][0];
                        if (profit[year-1]*-1 < atlm[j][0])
                        {
                                atlm[j][0] = atlm[j][0] + profit[year-1];
                                profit[year-1] = 0;
                        }
                        else
                        {
                                profit[year-1] = profit[year-1]+atlm[j][0];
                                atlm[j][0] = 0;
                        }
                        if ((atlm[j][1] == 1))
                        {
                                taxRebate[year-1] += (taxTemp -
                                        atlm[j][0])*0.25;
                        }
                        else if (atlm[j][1] ==2)
                        {
                                if(atlm[j][0] >300000.0)
                                {
                                        taxRebate[year-1] += (taxTemp-
                                                atlm[j][0])*0.35;
                                }
                                else
                                {
                                        taxRebate[year-1] += (taxTemp -
                                                300000.00)*0.35;
                                        taxRebate[year-1] += (300000.00-
                                                atlm[j][0])*0.25;
                                }
                        }
                        else
                        {
                                taxRebate[year-1] += (taxTemp - atlm[j][0])
                                        *0.33;
                        }
                        j--;
                }
        }
        if (profit[year-1]<0)
        {
                forward(year);
        }
}

// reduce profit by balance of carry forward
void takeProfit(int year)
{

        if (profit[year-1] > cForward*-1)
        {
                profit[year-1] = profit[year-1] + cForward;
                cForward = 0;
        }
        else
        {
                cForward = cForward + profit[year-1];
                profit[year-1] = 0;
        }
}
```

```
// calculate divedend due
void doDividend(int c)
{
int n,p1;

for (n=0;n<6;n++)
{
      divA[n] = t[n];
      for (p1=n*3;p1<(TPr*n)+3;p1++)
      {
            if (chrom[c][p1] == 1)
                  divA[n] += proj[p1-(n*3)][n];
      }
      divA[n] += (-taxDue[n+2]+taxRebate[n]);
      divA[n] = divA[n]/pow(1.06,n+1);
}
divA[n] = -taxDue[n+2];
divA[n] = divA[n]/pow(1.06,n+1);
}

// calculate tax due
void taxduein(int n)
{
      if (at1m[n][0] <= 300000.00)
      {
            taxDue[n] = at1m[n][0]*0.25;
            at1m[n][1] = 1;
      }
      else if (at1m[n][0] < 1500000.00)
      {
            taxDue[n] = 300000.0*0.25;
            taxDue[n] += (at1m[n][0]-300000.0)*0.35;
            at1m[n][1] = 2;
      }
      else
      {
            taxDue[n] = at1m[n][0]*0.33;
            at1m[n][1] = 3;
      }
}
```

```
// determine initial financial information from Binary String
void makeMoney(int f)
{
int x1,y1;
int bin;  // position in binary string
t[0] = findIt(24,f,0);
ca[0] = findIt(37,f,1);
tlm[0] = findIt(50,f,2);
tlm[1] = findIt(63,f,3);
tlm[2] = findIt(76,f,4);
t[1] = findIt(89,f,5);
t[2] = findIt(102,f,6);
t[3] = findIt(115,f,7);
t[4] = findIt(128,f,8);
t[5] = findIt(141,f,9);
bin = 154;
for (x1=0;x1<3;x1++)
{

      cap[x1][0] = findIt(bin,f,10+(7*x1));
      bin+=jump;
      proj[x1][0] = findIt(bin,f,11+(7*x1));
      bin+=jump;
      proj[x1][1] = findIt(bin,f,12+(7*x1));
      bin+=jump;
      proj[x1][2] = findIt(bin,f,13+(7*x1));
      bin+=jump;
      proj[x1][3] = findIt(bin,f,14+(7*x1));
      bin+=jump;
      proj[x1][4] = findIt(bin,f,15+(7*x1));
      bin+=jump;
      proj[x1][5] = findIt(bin,f,16+(7*x1));
      bin+=jump;
}
}
```

```
// determine integer value
float findIt(int m1, int f,int dig)
{
long bin1;
bin1 = 0;
for (x=m1;x<(m1)+jump;x++)
{

      if (chrom[f][x] == 1)
      {
            bin1<<=1;
            bin1 += 1;
      }
      else
            bin1<<=1;
}
return((float)bin1*1000);
}
```

Appendix L

500 of over 1000 Paths through Capital Budgeting Program

1) 1F8T1F8T
2) 1F8T1T2F3T4F5A6F4F5A6F4F5A6F7TA1
3) 1F8T1T2F3T4F5A6F4F5A6F4F5F7TA1
4) 1F8T1T2F3T4F5A6F4F5A6F4F5T7TA1
5) 1F8T1T2F3T4F5A6F4F5A6F4T5F7F
6) 1F8T1T2F3T4F5A6F4F5A6F4T5T7F
7) 1F8T1T2F3T4F5A6F4F5F4F5A6F7TA1
8) 1F8T1T2F3T4F5A6F4F5F4F5F7TA1
9) 1F8T1T2F3T4F5A6F4F5F4F5T7TA1
10) 1F8T1T2F3T4F5A6F4F5F4T5A6F7F
11) 1F8T1T2F3T4F5A6F4F5F4T5A6T7F
12) 1F8T1T2F3T4F5A6F4F5F4T5F7F
13) 1F8T1T2F3T4F5A6F4F5T4F5A6F7TA1
14) 1F8T1T2F3T4F5A6F4F5T4F5F7TA1
15) 1F8T1T2F3T4F5A6F4F5T4F5T7TA1
16) 1F8T1T2F3T4F5A6F4F5T4T5F7F
17) 1F8T1T2F3T4F5A6F4T5A6F7F
18) 1F8T1T2F3T4F5A6F4T5A6T7F
19) 1F8T1T2F3T4F5A6F4T5F7F
20) 1F8T1T2F3T4F5A6F4T5T7F
21) 1F8T1T2F3T4F5A6F7F
22) 1F8T1T2F3T4F5F4F5A6F4F5A6F7TA1
23) 1F8T1T2F3T4F5F4F5A6F4F5F7TA1
24) 1F8T1T2F3T4F5F4F5A6F4F5T7TA1
25) 1F8T1T2F3T4F5F4F5A6F4T5F7F
26) 1F8T1T2F3T4F5F4F5A6F4T5T7F
27) 1F8T1T2F3T4F5F4F5F4F5A6F7TA1
28) 1F8T1T2F3T4F5F4F5F4F5F7TA1

29) 1F8T1T2F3T4F5F4F5F4F5T7TA1
30) 1F8T1T2F3T4F5F4F5F4T5A6F7F
31) 1F8T1T2F3T4F5F4F5F4T5A6T7F
32) 1F8T1T2F3T4F5F4F5F4T5F7F
33) 1F8T1T2F3T4F5F4F5T4F5A6F7TA1
34) 1F8T1T2F3T4F5F4F5T4F5F7TA1
35) 1F8T1T2F3T4F5F4F5T4F5T7TA1
36) 1F8T1T2F3T4F5F4F5T4T5A6F7F
37) 1F8T1T2F3T4F5F4F5T4T5A6T7F
38) 1F8T1T2F3T4F5F4F5T4T5F7F
39) 1F8T1T2F3T4F5F4T5A6F7F
40) 1F8T1T2F3T4F5F4T5A6T7F
41) 1F8T1T2F3T4F5F4T5F7F
42) 1F8T1T2F3T4F5T4F5A6F4F5A6F7TA1
43) 1F8T1T2F3T4F5T4F5A6F4F5F7TA1
44) 1F8T1T2F3T4F5T4F5A6F4F5T7TA1
45) 1F8T1T2F3T4F5T4F5A6F4T5F7F
46) 1F8T1T2F3T4F5T4F5F4F5A6F7TA1
47) 1F8T1T2F3T4F5T4F5F4F5F7TA1
48) 1F8T1T2F3T4F5T4F5F4F5T7TA1
49) 1F8T1T2F3T4F5T4F5F4T5F7F
50) 1F8T1T2F3T4F5T4F5T4F5F7TA1
51) 1F8T1T2F3T4F5T4F5T4F5T7TA1
52) 1F8T1T2F3T4F5T4F5T4T5F7F
53) 1F8T1T2F3T4F5T4T5A6T7F
54) 1F8T1T2F3T4F5T4T5F7F
55) 1F8T1T2F3T4F5T4T5T7F
56) 1F8T1T2F3T4T5A6F7F
57) 1F8T1T2F3T4T5A6T7F
58) 1F8T1T2F3T4T5F7F
59) 1F8T1T2F3T4T5T7F

60) 1F8T1T2TA1

61) 1T2F3T4F5A6F4F5A6F4F5A6F7TA11F8F

62) 1T2F3T4F5A6F4F5A6F4F5A6F7TA11F8T

63) 1T2F3T4F5A6F4F5A6F4F5A6F7TA11T2F3T4F5A6F4F5A6F4F5A6F7TA1

64) 1T2F3T4F5A6F4F5A6F4F5A6F7TA11T2F3T4F5F4F5A6F4F5A6F7TA1

65) 1T2F3T4F5A6F4F5A6F4F5A6F7TA11T2F3T4T5A6T7F

66) 1T2F3T4F5A6F4F5A6F4F5A6F7TA11T2F3T4T5F7F

67) 1T2F3T4F5A6F4F5A6F4F5A6F7TA11T2TA1

68) 1T2F3T4F5A6F4F5A6F4F5F7TA11F8F

69) 1T2F3T4F5A6F4F5A6F4F5F7TA11F8T

70) 1T2F3T4F5A6F4F5A6F4F5F7TA11T2F3T4F5A6F4F5A6F4F5A6F7TA1

71) 1T2F3T4F5A6F4F5A6F4F5F7TA11T2F3T4F5F4F5A6F4F5A6F7TA1

72) 1T2F3T4F5A6F4F5A6F4F5F7TA11T2TA1

73) 1T2F3T4F5A6F4F5A6F4F5T7TA11F8F

74) 1T2F3T4F5A6F4F5A6F4F5T7TA11F8T

75) 1T2F3T4F5A6F4F5A6F4F5T7TA11T2F3T4F5T4F5A6F4F5A6F7TA1

76) 1T2F3T4F5A6F4F5A6F4F5T7TA11T2TA1

77) 1T2F3T4F5A6F4F5A6F4T5A6F7F1F8T

78) 1T2F3T4F5A6F4F5A6F4T5A6T7F1F8T

79) 1T2F3T4F5A6F4F5A6F4T5A6T7F1T2TA1

80) 1T2F3T4F5A6F4F5A6F4T5F7F1F8T

81) 1T2F3T4F5A6F4F5A6F4T5F7F1T2F3T4F5A6F4F5A6F4F5A6F7TA1

82) 1T2F3T4F5A6F4F5A6F4T5F7F1T2F3T4F5F4F5A6F4F5A6F7TA1

83) 1T2F3T4F5A6F4F5A6F4T5F7F1T2F3T4F5T4F5A6F4F5A6F7TA1

84) 1T2F3T4F5A6F4F5A6F4T5F7F1T2TA1

85) 1T2F3T4F5A6F4F5A6F4T5T7F1F8T

86) 1T2F3T4F5A6F4F5F4F5A6F7TA11F8F

87) 1T2F3T4F5A6F4F5F4F5A6F7TA11F8T

88) 1T2F3T4F5A6F4F5F4F5A6F7TA11T2F3T4F5A6F4F5A6F4F5F7TA1

89) 1T2F3T4F5A6F4F5F4F5A6F7TA11T2F3T4F5F4F5A6F4F5F7TA1

90) 1T2F3T4F5A6F4F5F4F5A6F7TA11T2F3T4F5T4F5A6F4F5F7TA1

91) 1T2F3T4F5A6F4F5F4F5A6F7TA11T2TA1

92) 1T2F3T4F5A6F4F5F4F5F7TA11F8F

93) 1T2F3T4F5A6F4F5F4F5F7TA11F8T

94) 1T2F3T4F5A6F4F5F4F5F7TA11T2F3T4F5A6F4F5A6F4F5F7TA1

95) 1T2F3T4F5A6F4F5F4F5F7TA11T2F3T4F5F4F5A6F4F5F7TA1

96) 1T2F3T4F5A6F4F5F4F5F7TA11T2F3T4F5T4F5A6F4F5F7TA1

97) 1T2F3T4F5A6F4F5F4F5F7TA11T2F3T4T5F7F

98) 1T2F3T4F5A6F4F5F4F5F7TA11T2TA1

99) 1T2F3T4F5A6F4F5F4F5T7TA11F8F

100) 1T2F3T4F5A6F4F5F4F5T7TA11F8T

101) 1T2F3T4F5A6F4F5F4F5T7TA11T2F3T4F5A6F4F5A6F4F5F7TA1

102) 1T2F3T4F5A6F4F5F4F5T7TA11T2F3T4F5F4F5A6F4F5F7TA1

103) 1T2F3T4F5A6F4F5F4F5T7TA11T2F3T4F5T4F5A6F4F5F7TA1

104) 1T2F3T4F5A6F4F5F4F5T7TA11T2TA1

105) 1T2F3T4F5A6F4F5F4T5A6F7F1F8T

106) 1T2F3T4F5A6F4F5F4T5A6F7F1T2F3T4F5A6F4F5A6F4F5F7TA1

107) 1T2F3T4F5A6F4F5F4T5A6F7F1T2F3T4F5F4F5A6F4F5F7TA1

108) 1T2F3T4F5A6F4F5F4T5A6F7F1T2F3T4F5T4F5A6F4F5F7TA1

109) 1T2F3T4F5A6F4F5F4T5A6F7F1T2TA1

110) 1T2F3T4F5A6F4F5F4T5A6T7F1F8T

111) 1T2F3T4F5A6F4F5F4T5A6T7F1T2F3T4F5A6F4F5A6F4F5F7TA1

112) 1T2F3T4F5A6F4F5F4T5A6T7F1T2F3T4F5F4F5A6F4F5F7TA1

113) 1T2F3T4F5A6F4F5F4T5A6T7F1T2F3T4F5T4F5A6F4F5F7TA1

114) 1T2F3T4F5A6F4F5F4T5A6T7F1T2TA1

115) 1T2F3T4F5A6F4F5F4T5F7F1F8T

116) 1T2F3T4F5A6F4F5F4T5F7F1T2F3T4F5A6F4F5A6F4F5F7TA1

117) 1T2F3T4F5A6F4F5F4T5F7F1T2F3T4F5F4F5A6F4F5F7TA1

118) 1T2F3T4F5A6F4F5F4T5F7F1T2F3T4F5T4F5A6F4F5F7TA1

119) 1T2F3T4F5A6F4F5F4T5F7F1T2F3T4T5F7F

120) 1T2F3T4F5A6F4F5F4T5F7F1T2TA1

121) 1T2F3T4F5A6F4F5F4T5T7F1F8T

122) 1T2F3T4F5A6F4F5T4F5A6F7TA11F8F

123) 1T2F3T4F5A6F4F5T4F5A6F7TA11F8T

124) 1T2F3T4F5A6F4F5T4F5A6F7TA11T2F3T4F5A6F4F5A6F4F5T7TA1

125) 1T2F3T4F5A6F4F5T4F5A6F7TA11T2F3T4F5F4F5A6F4F5T7TA1

126) 1T2F3T4F5A6F4F5T4F5A6F7TA11T2F3T4F5T4F5A6F4F5T7TA1

127) 1T2F3T4F5A6F4F5T4F5A6F7TA11T2TA1

128) 1T2F3T4F5A6F4F5T4F5F7TA11F8F

129) 1T2F3T4F5A6F4F5T4F5F7TA11F8T

130) 1T2F3T4F5A6F4F5T4F5F7TA11T2F3T4F5A6F4F5A6F4F5T7TA1

131) 1T2F3T4F5A6F4F5T4F5F7TA11T2F3T4F5F4F5A6F4F5T7TA1

132) 1T2F3T4F5A6F4F5T4F5F7TA11T2F3T4F5T4F5A6F4F5T7TA1

133) 1T2F3T4F5A6F4F5T4F5F7TA11T2TA1

134) 1T2F3T4F5A6F4F5T4F5T7TA11F8T

135) 1T2F3T4F5A6F4F5T4F5T7TA11T2F3T4F5F4F5A6F4F5T7TA1

136) 1T2F3T4F5A6F4F5T4F5T7TA11T2TA1

137) 1T2F3T4F5A6F4F5T4T5A6F7F1F8T

138) 1T2F3T4F5A6F4F5T4T5A6F7F1T2F3T4F5F4F5A6F4F5T7TA1

139) 1T2F3T4F5A6F4F5T4T5A6F7F1T2TA1

140) 1T2F3T4F5A6F4F5T4T5A6T7F1F8T

141) 1T2F3T4F5A6F4F5T4T5A6T7F1T2TA1

142) 1T2F3T4F5A6F4F5T4T5F7F1F8T

143) 1T2F3T4F5A6F4F5T4T5F7F1T2F3T4F5A6F4F5A6F4F5T7TA1

144) 1T2F3T4F5A6F4F5T4T5F7F1T2F3T4F5F4F5A6F4F5T7TA1

145) 1T2F3T4F5A6F4F5T4T5F7F1T2F3T4F5T4F5A6F4F5T7TA1

146) 1T2F3T4F5A6F4F5T4T5F7F1T2TA1

147) 1T2F3T4F5A6F4T5A6F7F1F8T

148) 1T2F3T4F5A6F4T5A6F7F1T2F3T4F5A6F4F5A6F4F5A6F7TA1

149) 1T2F3T4F5A6F4T5A6F7F1T2F3T4F5F4F5A6F4F5A6F7TA1

150) 1T2F3T4F5A6F4T5A6F7F1T2TA1

151) 1T2F3T4F5A6F4T5A6T7F1F8T

152) 1T2F3T4F5A6F4T5A6T7F1T2F3T4F5A6F4F5A6F4F5A6F7TA1

153) 1T2F3T4F5A6F4T5A6T7F1T2F3T4F5F4F5A6F4F5A6F7TA1

154) 1T2F3T4F5A6F4T5A6T7F1T2F3T4F5T4F5A6F4F5A6F7TA1

155) 1T2F3T4F5A6F4T5A6T7F1T2TA1

156) 1T2F3T4F5A6F4T5F7F1F8T

157) 1T2F3T4F5A6F4T5F7F1T2F3T4F5A6F4F5A6F4F5F7TA1

158) 1T2F3T4F5A6F4T5F7F1T2F3T4F5A6F4F5A6F4T5F7F

159) 1T2F3T4F5A6F4T5F7F1T2F3T4F5F4F5A6F4F5F7TA1

160) 1T2F3T4F5A6F4T5F7F1T2F3T4F5F4F5A6F4T5F7F

161) 1T2F3T4F5A6F4T5F7F1T2F3T4F5T4F5A6F4F5F7TA1

162) 1T2F3T4F5A6F4T5F7F1T2F3T4F5T4F5A6F4T5F7F

163) 1T2F3T4F5A6F4T5F7F1T2F3T4T5F7F

164) 1T2F3T4F5A6F4T5F7F1T2TA1

165) 1T2F3T4F5A6F4T5T7F1T2TA1

166) 1T2F3T4F5F4F5A6F4F5A6F7TA11F8F

167) 1T2F3T4F5F4F5A6F4F5A6F7TA11F8T

168) 1T2F3T4F5F4F5A6F4F5A6F7TA11T2F3T4F5A6F4F5F4F5A6F7TA1

169) 1T2F3T4F5F4F5A6F4F5A6F7TA11T2F3T4F5F4F5F4F5A6F7TA1

170) 1T2F3T4F5F4F5A6F4F5A6F7TA11T2TA1

171) 1T2F3T4F5F4F5A6F4F5F7TA11F8F

172) 1T2F3T4F5F4F5A6F4F5F7TA11F8T

173) 1T2F3T4F5F4F5A6F4F5F7TA11T2F3T4F5A6F4F5F4F5A6F7TA1

174) 1T2F3T4F5F4F5A6F4F5F7TA11T2F3T4F5F4F5F4F5A6F7TA1

175) 1T2F3T4F5F4F5A6F4F5F7TA11T2TA1

176) 1T2F3T4F5F4F5A6F4F5T7TA11F8F

177) 1T2F3T4F5F4F5A6F4F5T7TA11F8T

178) 1T2F3T4F5F4F5A6F4F5T7TA11T2F3T4F5F4F5F4F5A6F7TA1

179) 1T2F3T4F5F4F5A6F4F5T7TA11T2F3T4F5T4F5F4F5A6F7TA1

180) 1T2F3T4F5F4F5A6F4F5T7TA11T2TA1

181) 1T2F3T4F5F4F5A6F4T5A6F7F1F8T

182) 1T2F3T4F5F4F5A6F4T5A6F7F1T2F3T4F5A6F4F5F4F5A6F7TA1

183) 1T2F3T4F5F4F5A6F4T5A6F7F1T2F3T4F5F4F5F4F5A6F7TA1

184) 1T2F3T4F5F4F5A6F4T5A6F7F1T2F3T4T5F7F
185) 1T2F3T4F5F4F5A6F4T5A6F7F1T2TA1
186) 1T2F3T4F5F4F5A6F4T5A6T7F1F8T
187) 1T2F3T4F5F4F5A6F4T5A6T7F1T2F3T4F5A6F4F5F4F5A6F7TA1
188) 1T2F3T4F5F4F5A6F4T5A6T7F1T2F3T4F5F4F5F4F5A6F7TA1
189) 1T2F3T4F5F4F5A6F4T5A6T7F1T2F3T4T5F7F
190) 1T2F3T4F5F4F5A6F4T5A6T7F1T2TA1
191) 1T2F3T4F5F4F5A6F4T5F7F1F8T
192) 1T2F3T4F5F4F5A6F4T5F7F1T2F3T4F5A6F4F5F4F5A6F7TA1
193) 1T2F3T4F5F4F5A6F4T5F7F1T2F3T4F5F4F5F4F5A6F7TA1
194) 1T2F3T4F5F4F5A6F4T5F7F1T2TA1
195) 1T2F3T4F5F4F5A6F4T5T7F1F8T
196) 1T2F3T4F5F4F5A6F4T5T7F1T2TA1
197) 1T2F3T4F5F4F5F4F5A6F7TA11F8F
198) 1T2F3T4F5F4F5F4F5A6F7TA11F8T
199) 1T2F3T4F5F4F5F4F5A6F7TA11T2F3T4F5A6F4F5F4F5F7TA1
200) 1T2F3T4F5F4F5F4F5A6F7TA11T2F3T4F5F4F5F4F5F7TA1
201) 1T2F3T4F5F4F5F4F5A6F7TA11T2TA1
202) 1T2F3T4F5F4F5F4F5F7TA11F8F
203) 1T2F3T4F5F4F5F4F5F7TA11F8T
204) 1T2F3T4F5F4F5F4F5F7TA11T2F3T4F5A6F4F5F4F5F7TA1
205) 1T2F3T4F5F4F5F4F5F7TA11T2F3T4F5F4F5F4F5F7TA1
206) 1T2F3T4F5F4F5F4F5F7TA11T2F3T4F5T4F5F4F5F7TA1
207) 1T2F3T4F5F4F5F4F5F7TA11T2F3T4T5F7F
208) 1T2F3T4F5F4F5F4F5F7TA11T2TA1
209) 1T2F3T4F5F4F5F4F5T7TA11F8F
210) 1T2F3T4F5F4F5F4F5T7TA11F8T
211) 1T2F3T4F5F4F5F4F5T7TA11T2F3T4F5A6F4F5F4F5F7TA1
212) 1T2F3T4F5F4F5F4F5T7TA11T2TA1
213) 1T2F3T4F5F4F5F4T5A6F7F1F8T
214) 1T2F3T4F5F4F5F4T5A6F7F1T2F3T4F5A6F4F5F4F5F7TA1

215) 1T2F3T4F5F4F5F4T5A6T7F1F8T

216) 1T2F3T4F5F4F5F4T5A6T7F1T2F3T4F5A6F4F5F4F5F7TA1

217) 1T2F3T4F5F4F5F4T5A6T7F1T2F3T4F5F4F5F4F5F7TA1

218) 1T2F3T4F5F4F5F4T5A6T7F1T2TA1

219) 1T2F3T4F5F4F5F4T5F7F1F8T

220) 1T2F3T4F5F4F5F4T5F7F1T2F3T4F5A6F4F5F4F5F7TA1

221) 1T2F3T4F5F4F5F4T5F7F1T2F3T4F5F4F5F4F5F7TA1

222) 1T2F3T4F5F4F5F4T5F7F1T2F3T4F5T4F5F4F5F7TA1

223) 1T2F3T4F5F4F5F4T5F7F1T2F3T4T5F7F

224) 1T2F3T4F5F4F5F4T5F7F1T2TA1

225) 1T2F3T4F5F4F5T4F5A6F7TA11F8F

226) 1T2F3T4F5F4F5T4F5A6F7TA11F8T

227) 1T2F3T4F5F4F5T4F5A6F7TA11T2F3T4F5A6F4F5F4F5T7TA1

228) 1T2F3T4F5F4F5T4F5A6F7TA11T2F3T4F5F4F5F4F5T7TA1

229) 1T2F3T4F5F4F5T4F5A6F7TA11T2F3T4F5T4F5F4F5T7TA1

230) 1T2F3T4F5F4F5T4F5A6F7TA11T2TA1

231) 1T2F3T4F5F4F5T4F5F7TA11F8F

232) 1T2F3T4F5F4F5T4F5F7TA11F8T

233) 1T2F3T4F5F4F5T4F5F7TA11T2F3T4F5A6F4F5F4F5T7TA1

234) 1T2F3T4F5F4F5T4F5F7TA11T2F3T4F5T4F5F4F5T7TA1

235) 1T2F3T4F5F4F5T4F5F7TA11T2TA1

236) 1T2F3T4F5F4F5T4F5T7TA11F8F

237) 1T2F3T4F5F4F5T4F5T7TA11F8T

238) 1T2F3T4F5F4F5T4F5T7TA11T2F3T4F5A6F4F5F4F5T7TA1

239) 1T2F3T4F5F4F5T4F5T7TA11T2F3T4F5F4F5F4F5T7TA1

240) 1T2F3T4F5F4F5T4F5T7TA11T2F3T4F5T4F5F4F5T7TA1

241) 1T2F3T4F5F4F5T4F5T7TA11T2F3T4T5F7F

242) 1T2F3T4F5F4F5T4F5T7TA11T2TA1

243) 1T2F3T4F5F4F5T4T5A6F7F1F8T

244) 1T2F3T4F5F4F5T4T5A6F7F1T2TA1

245) 1T2F3T4F5F4F5T4T5A6T7F1F8T

246) 1T2F3T4F5F4F5T4T5A6T7F1T2F3T4F5A6F4F5F4F5T7TA1

247) 1T2F3T4F5F4F5T4T5A6T7F1T2TA1

248) 1T2F3T4F5F4F5T4T5F7F1F8T

249) 1T2F3T4F5F4F5T4T5F7F1T2F3T4F5A6F4F5F4F5T7TA1

250) 1T2F3T4F5F4F5T4T5F7F1T2F3T4F5F4F5F4F5T7TA1

251) 1T2F3T4F5F4F5T4T5F7F1T2TA1

252) 1T2F3T4F5F4F5T4T5T7F1F8T

253) 1T2F3T4F5F4F5T4T5T7F1T2TA1

254) 1T2F3T4F5F4T5A6F7F1F8T

255) 1T2F3T4F5F4T5A6F7F1T2F3T4F5A6F4F5F4F5A6F7TA1

256) 1T2F3T4F5F4T5A6F7F1T2F3T4F5F4F5F4F5A6F7TA1

257) 1T2F3T4F5F4T5A6F7F1T2TA1

258) 1T2F3T4F5F4T5A6T7F1F8T

259) 1T2F3T4F5F4T5A6T7F1T2F3T4F5A6F4F5F4F5A6F7TA1

260) 1T2F3T4F5F4T5A6T7F1T2F3T4F5F4F5F4F5A6F7TA1

261) 1T2F3T4F5F4T5A6T7F1T2F3T4F5F4F5F4T5A6T7F

262) 1T2F3T4F5F4T5A6T7F1T2F3T4T5F7F

263) 1T2F3T4F5F4T5A6T7F1T2TA1

264) 1T2F3T4F5F4T5F7F1F8T

265) 1T2F3T4F5F4T5F7F1T2F3T4F5A6F4F5F4F5F7TA1

266) 1T2F3T4F5F4T5F7F1T2F3T4F5A6F4F5F4T5F7F

267) 1T2F3T4F5F4T5F7F1T2F3T4F5F4F5F4F5F7TA1

268) 1T2F3T4F5F4T5F7F1T2F3T4F5F4F5F4T5F7F

269) 1T2F3T4F5F4T5F7F1T2F3T4F5T4F5F4F5F7TA1

270) 1T2F3T4F5F4T5F7F1T2F3T4F5T4F5F4T5F7F

271) 1T2F3T4F5F4T5F7F1T2F3T4T5F7F

272) 1T2F3T4F5F4T5F7F1T2TA1

273) 1T2F3T4F5F4T5T7F1F8T

274) 1T2F3T4F5F4T5T7F1T2F3T4F5F4F5F4T5T7F

275) 1T2F3T4F5F4T5T7F1T2TA1

276) 1T2F3T4F5T4F5A6F4F5A6F7TA11F8F

277) 1T2F3T4F5T4F5A6F4F5A6F7TA11F8T

278) 1T2F3T4F5T4F5A6F4F5A6F7TA11T2F3T4F5A6F4F5T4F5A6F7TA1

279) 1T2F3T4F5T4F5A6F4F5A6F7TA11T2F3T4F5F4F5T4F5A6F7TA1

280) 1T2F3T4F5T4F5A6F4F5A6F7TA11T2F3T4F5T4F5T4F5A6F7TA1

281) 1T2F3T4F5T4F5A6F4F5A6F7TA11T2F3T4T5F7F

282) 1T2F3T4F5T4F5A6F4F5A6F7TA11T2TA1

283) 1T2F3T4F5T4F5A6F4F5F7TA11F8F

284) 1T2F3T4F5T4F5A6F4F5F7TA11F8T

285) 1T2F3T4F5T4F5A6F4F5F7TA11T2F3T4F5A6F4F5T4F5A6F7TA1

286) 1T2F3T4F5T4F5A6F4F5F7TA11T2F3T4F5F4F5T4F5A6F7TA1

287) 1T2F3T4F5T4F5A6F4F5F7TA11T2F3T4F5T4F5T4F5A6F7TA1

288) 1T2F3T4F5T4F5A6F4F5F7TA11T2TA1

289) 1T2F3T4F5T4F5A6F4F5T7TA11F8T

290) 1T2F3T4F5T4F5A6F4F5T7TA11T2TA1

291) 1T2F3T4F5T4F5A6F4T5A6F7F1F8T

292) 1T2F3T4F5T4F5A6F4T5A6F7F1T2F3T4F5A6F4F5T4F5A6F7TA1

293) 1T2F3T4F5T4F5A6F4T5A6F7F1T2F3T4F5T4F5T4F5A6F7TA1

294) 1T2F3T4F5T4F5A6F4T5A6T7F1F8T

295) 1T2F3T4F5T4F5A6F4T5A6T7F1T2TA1

296) 1T2F3T4F5T4F5A6F4T5F7F1F8T

297) 1T2F3T4F5T4F5A6F4T5F7F1T2F3T4F5A6F4F5T4F5A6F7TA1

298) 1T2F3T4F5T4F5A6F4T5F7F1T2F3T4F5F4F5T4F5A6F7TA1

299) 1T2F3T4F5T4F5A6F4T5F7F1T2F3T4F5T4F5T4F5A6F7TA1

300) 1T2F3T4F5T4F5A6F4T5F7F1T2TA1

301) 1T2F3T4F5T4F5F4F5A6F7TA11F8F

302) 1T2F3T4F5T4F5F4F5A6F7TA11F8T

303) 1T2F3T4F5T4F5F4F5A6F7TA11T2F3T4F5A6F4F5T4F5F7TA1

304) 1T2F3T4F5T4F5F4F5A6F7TA11T2F3T4F5F4F5T4F5F7TA1

305) 1T2F3T4F5T4F5F4F5A6F7TA11T2F3T4F5T4F5T4F5F7TA1

306) 1T2F3T4F5T4F5F4F5A6F7TA11T2F3T4T5F7F

307) 1T2F3T4F5T4F5F4F5A6F7TA11T2TA1

308) 1T2F3T4F5T4F5F4F5F7TA11F8F

309) 1T2F3T4F5T4F5F4F5F7TA11F8T

310) 1T2F3T4F5T4F5F4F5F7TA11T2F3T4F5A6F4F5T4F5F7TA1

311) 1T2F3T4F5T4F5F4F5F7TA11T2F3T4F5F4F5T4F5F7TA1

312) 1T2F3T4F5T4F5F4F5F7TA11T2F3T4F5T4F5T4F5F7TA1

313) 1T2F3T4F5T4F5F4F5F7TA11T2TA1

314) 1T2F3T4F5T4F5F4F5T7TA11F8F

315) 1T2F3T4F5T4F5F4F5T7TA11F8T

316) 1T2F3T4F5T4F5F4F5T7TA11T2F3T4F5A6F4F5T4F5F7TA1

317) 1T2F3T4F5T4F5F4F5T7TA11T2F3T4F5T4F5T4F5F7TA1

318) 1T2F3T4F5T4F5F4T5A6F7F1F8T

319) 1T2F3T4F5T4F5F4T5A6F7F1T2F3T4F5T4F5T4F5F7TA1

320) 1T2F3T4F5T4F5F4T5A6F7F1T2TA1

321) 1T2F3T4F5T4F5F4T5A6T7F1F8T

322) 1T2F3T4F5T4F5F4T5A6T7F1T2F3T4F5A6F4F5T4F5F7TA1

323) 1T2F3T4F5T4F5F4T5A6T7F1T2F3T4F5F4F5T4F5F7TA1

324) 1T2F3T4F5T4F5F4T5A6T7F1T2TA1

325) 1T2F3T4F5T4F5F4T5F7F1F8T

326) 1T2F3T4F5T4F5F4T5F7F1T2F3T4F5A6F4F5T4F5F7TA1

327) 1T2F3T4F5T4F5F4T5F7F1T2F3T4F5F4F5T4F5F7TA1

328) 1T2F3T4F5T4F5F4T5F7F1T2F3T4F5T4F5T4F5F7TA1

329) 1T2F3T4F5T4F5F4T5F7F1T2F3T4T5A6F7F

330) 1T2F3T4F5T4F5F4T5F7F1T2TA1

331) 1T2F3T4F5T4F5F4T5T7F1F8T

332) 1T2F3T4F5T4F5F4T5T7F1T2F3T4F5A6F4F5T4F5F7TA1

333) 1T2F3T4F5T4F5F4T5T7F1T2F3T4F5T4F5T4F5F7TA1

334) 1T2F3T4F5T4F5T4F5A6F7TA11F8F

335) 1T2F3T4F5T4F5T4F5A6F7TA11F8T

336) 1T2F3T4F5T4F5T4F5A6F7TA11T2F3T4F5A6F4F5T4F5T7TA1

337) 1T2F3T4F5T4F5T4F5A6F7TA11T2F3T4F5F4F5T4F5T7TA1

338) 1T2F3T4F5T4F5T4F5A6F7TA11T2F3T4F5T4F5T4F5T7TA1

339) 1T2F3T4F5T4F5T4F5A6F7TA11T2TA1

340) 1T2F3T4F5T4F5T4F5F7TA11F8F

341) 1T2F3T4F5T4F5T4F5F7TA11F8T

342) 1T2F3T4F5T4F5T4F5F7TA11T2F3T4F5A6F4F5T4F5T7TA1

343) 1T2F3T4F5T4F5T4F5F7TA11T2F3T4F5F4F5T4F5T7TA1

344) 1T2F3T4F5T4F5T4F5F7TA11T2F3T4F5T4F5T4F5T7TA1

345) 1T2F3T4F5T4F5T4F5F7TA11T2TA1

346) 1T2F3T4F5T4F5T4F5T7TA11F8T

347) 1T2F3T4F5T4F5T4F5T7TA11T2F3T4F5A6F4F5T4F5T7TA1

348) 1T2F3T4F5T4F5T4F5T7TA11T2F3T4F5F4F5T4F5T7TA1

349) 1T2F3T4F5T4F5T4F5T7TA11T2F3T4F5T4F5T4F5T7TA1

350) 1T2F3T4F5T4F5T4F5T7TA11T2TA1

351) 1T2F3T4F5T4F5T4T5A6F7F1F8T

352) 1T2F3T4F5T4F5T4T5A6F7F1T2F3T4F5F4F5T4F5T7TA1

353) 1T2F3T4F5T4F5T4T5A6F7F1T2TA1

354) 1T2F3T4F5T4F5T4T5A6T7F1F8T

355) 1T2F3T4F5T4F5T4T5A6T7F1T2F3T4F5A6F4F5T4F5T7TA1

356) 1T2F3T4F5T4F5T4T5A6T7F1T2F3T4F5T4F5T4F5T7TA1

357) 1T2F3T4F5T4F5T4T5A6T7F1T2TA1

358) 1T2F3T4F5T4F5T4T5F7F1F8T

359) 1T2F3T4F5T4F5T4T5F7F1T2F3T4F5A6F4F5T4F5T7TA1

360) 1T2F3T4F5T4F5T4T5F7F1T2F3T4F5F4F5T4F5T7TA1

361) 1T2F3T4F5T4F5T4T5F7F1T2F3T4F5T4F5T4F5T7TA1

362) 1T2F3T4F5T4F5T4T5F7F1T2TA1

363) 1T2F3T4F5T4T5A6F7F1F8T

364) 1T2F3T4F5T4T5A6F7F1T2TA1

365) 1T2F3T4F5T4T5A6T7F1F8T

366) 1T2F3T4F5T4T5A6T7F1T2F3T4F5F4F5T4F5A6F7TA1

367) 1T2F3T4F5T4T5A6T7F1T2TA1

368) 1T2F3T4F5T4T5F7F1F8T

369) 1T2F3T4F5T4T5F7F1T2F3T4F5A6F4F5T4F5F7TA1

370) 1T2F3T4F5T4T5F7F1T2F3T4F5A6F4F5T4T5F7F
371) 1T2F3T4F5T4T5F7F1T2F3T4F5F4F5T4F5F7TA1
372) 1T2F3T4F5T4T5F7F1T2F3T4F5F4F5T4T5F7F
373) 1T2F3T4F5T4T5F7F1T2F3T4F5T4F5T4F5F7TA1
374) 1T2F3T4F5T4T5F7F1T2F3T4F5T4F5T4T5F7F
375) 1T2F3T4F5T4T5F7F1T2F3T4T5F7F
376) 1T2F3T4F5T4T5F7F1T2TA1
377) 1T2F3T4F5T4T5T7F1F8T
378) 1T2F3T4F5T4T5T7F1T2F3T4F5A6F4F5T4F5T7TA1
379) 1T2F3T4T5A6F7F1F8T
380) 1T2F3T4T5A6F7F1T2F3T4F5A6F4F5A6F4F5A6F7TA1
381) 1T2F3T4T5A6F7F1T2F3T4F5A6F4F5A6F4F5T7TA1
382) 1T2F3T4T5A6F7F1T2F3T4F5A6F4F5A6F4T5F7F
383) 1T2F3T4T5A6F7F1T2F3T4F5F4F5A6F4F5A6F7TA1
384) 1T2F3T4T5A6F7F1T2F3T4F5F4F5A6F4F5F7TA1
385) 1T2F3T4T5A6F7F1T2F3T4F5F4F5A6F4F5T7TA1
386) 1T2F3T4T5A6F7F1T2F3T4F5F4F5A6F4T5F7F
387) 1T2F3T4T5A6F7F1T2F3T4F5T4F5A6F4F5F7TA1
388) 1T2F3T4T5A6F7F1T2F3T4F5T4F5A6F4F5T7TA1
389) 1T2F3T4T5A6F7F1T2TA1
390) 1T2F3T4T5A6T7F1F8T
391) 1T2F3T4T5A6T7F1T2F3T4F5A6F4F5A6F4F5A6F7TA1
392) 1T2F3T4T5A6T7F1T2F3T4F5A6F4F5A6F4F5F7TA1
393) 1T2F3T4T5A6T7F1T2F3T4F5A6F4F5A6F4F5T7TA1
394) 1T2F3T4T5A6T7F1T2F3T4F5A6F4F5A6F4T5F7F
395) 1T2F3T4T5A6T7F1T2F3T4F5A6F4T5A6T7F
396) 1T2F3T4T5A6T7F1T2F3T4F5F4F5A6F4F5A6F7TA1
397) 1T2F3T4T5A6T7F1T2F3T4F5F4F5A6F4F5F7TA1
398) 1T2F3T4T5A6T7F1T2F3T4F5F4F5A6F4F5T7TA1
399) 1T2F3T4T5A6T7F1T2F3T4F5F4F5A6F4T5F7F
400) 1T2F3T4T5A6T7F1T2F3T4F5F4T5A6T7F

401) 1T2F3T4T5A6T7F1T2F3T4F5T4F5A6F4F5A6F7TA1

402) 1T2F3T4T5A6T7F1T2F3T4F5T4F5A6F4F5F7TA1

403) 1T2F3T4T5A6T7F1T2F3T4F5T4F5A6F4F5T7TA1

404) 1T2F3T4T5A6T7F1T2F3T4F5T4F5A6F4T5F7F

405) 1T2F3T4T5A6T7F1T2F3T4T5F7F

406) 1T2F3T4T5A6T7F1T2TA1

407) 1T2F3T4T5F7F1F8T

408) 1T2F3T4T5F7F1T2F3T4F5A6F4F5F4F5A6F7TA1

409) 1T2F3T4T5F7F1T2F3T4F5A6F4F5F4F5F7TA1

410) 1T2F3T4T5F7F1T2F3T4F5A6F4F5F4F5T7TA1

411) 1T2F3T4T5F7F1T2F3T4F5A6F4F5F4T5A6F7F

412) 1T2F3T4T5F7F1T2F3T4F5A6F4F5F4T5A6T7F

413) 1T2F3T4T5F7F1T2F3T4F5A6F4F5F4T5F7F

414) 1T2F3T4T5F7F1T2F3T4F5A6F4T5F7F

415) 1T2F3T4T5F7F1T2F3T4F5F4F5F4F5A6F7TA1

416) 1T2F3T4T5F7F1T2F3T4F5F4F5F4F5F7TA1

417) 1T2F3T4T5F7F1T2F3T4F5F4F5F4F5T7TA1

418) 1T2F3T4T5F7F1T2F3T4F5F4F5F4T5A6F7F

419) 1T2F3T4T5F7F1T2F3T4F5F4F5F4T5A6T7F

420) 1T2F3T4T5F7F1T2F3T4F5F4F5F4T5F7F

421) 1T2F3T4T5F7F1T2F3T4F5F4T5F7F

422) 1T2F3T4T5F7F1T2F3T4F5T4F5F4F5A6F7TA1

423) 1T2F3T4T5F7F1T2F3T4F5T4F5F4F5F7TA1

424) 1T2F3T4T5F7F1T2F3T4F5T4F5F4F5T7TA1

425) 1T2F3T4T5F7F1T2F3T4F5T4F5F4T5A6F7F

426) 1T2F3T4T5F7F1T2F3T4F5T4F5F4T5A6T7F

427) 1T2F3T4T5F7F1T2F3T4F5T4F5F4T5F7F

428) 1T2F3T4T5F7F1T2F3T4F5T4T5F7F

429) 1T2F3T4T5F7F1T2F3T4T5F7F

430) 1T2F3T4T5F7F1T2TA1

431) 1T2F3T4T5T7F1F8T

432) 1T2F3T4T5T7F1T2F3T4F5A6F4F5T4F5A6F7TA1
433) 1T2F3T4T5T7F1T2F3T4F5A6F4F5T4F5T7TA1
434) 1T2F3T4T5T7F1T2F3T4F5F4F5T4T5F7F
435) 1T2F3T4T5T7F1T2F3T4F5T4F5T4T5F7F
436) 1T2F3T4T5T7F1T2F3T4T5F7F
437) 1T2F3T4T5T7F1T2TA1
438) 1T2TA11F8F
439) 1T2TA11F8T
440) 1T2TA11T2F3T4F5A6F4F5A6F4F5A6F7TA1
441) 1T2TA11T2F3T4F5A6F4F5A6F4F5F7TA1
442) 1T2TA11T2F3T4F5A6F4F5A6F4F5T7TA1
443) 1T2TA11T2F3T4F5A6F4F5A6F4T5A6F7F
444) 1T2TA11T2F3T4F5A6F4F5A6F4T5F7F
445) 1T2TA11T2F3T4F5A6F4F5A6F4T5T7F
446) 1T2TA11T2F3T4F5A6F4F5F4F5A6F7TA1
447) 1T2TA11T2F3T4F5A6F4F5F4F5F7TA1
448) 1T2TA11T2F3T4F5A6F4F5F4F5T7TA1
449) 1T2TA11T2F3T4F5A6F4F5F4T5A6F7F
450) 1T2TA11T2F3T4F5A6F4F5F4T5F7F
451) 1T2TA11T2F3T4F5A6F4F5T4F5A6F7TA1
452) 1T2TA11T2F3T4F5A6F4F5T4F5F7TA1
453) 1T2TA11T2F3T4F5A6F4F5T4F5T7TA1
454) 1T2TA11T2F3T4F5A6F4F5T4T5A6F7F
455) 1T2TA11T2F3T4F5A6F4F5T4T5F7F
456) 1T2TA11T2F3T4F5A6F4T5A6F7F
457) 1T2TA11T2F3T4F5A6F4T5A6T7F
458) 1T2TA11T2F3T4F5A6F4T5F7F
459) 1T2TA11T2F3T4F5A6F4T5T7F
460) 1T2TA11T2F3T4F5F4F5A6F4F5A6F7TA1
461) 1T2TA11T2F3T4F5F4F5A6F4F5F7TA1
462) 1T2TA11T2F3T4F5F4F5A6F4F5T7TA1

463) 1T2TA11T2F3T4F5F4F5A6F4T5A6F7F
464) 1T2TA11T2F3T4F5F4F5A6F4T5A6T7F
465) 1T2TA11T2F3T4F5F4F5A6F4T5F7F
466) 1T2TA11T2F3T4F5F4F5A6F4T5T7F
467) 1T2TA11T2F3T4F5F4F5F4F5A6F7TA1
468) 1T2TA11T2F3T4F5F4F5F4F5F7TA1
469) 1T2TA11T2F3T4F5F4F5F4F5T7TA1
470) 1T2TA11T2F3T4F5F4F5F4T5A6T7F
471) 1T2TA11T2F3T4F5F4F5F4T5F7F
472) 1T2TA11T2F3T4F5F4F5T4F5A6F7TA1
473) 1T2TA11T2F3T4F5F4F5T4F5F7TA1
474) 1T2TA11T2F3T4F5F4F5T4F5T7TA1
475) 1T2TA11T2F3T4F5F4F5T4T5F7F
476) 1T2TA11T2F3T4F5F4T5A6F7F
477) 1T2TA11T2F3T4F5F4T5A6T7F
478) 1T2TA11T2F3T4F5F4T5F7F
479) 1T2TA11T2F3T4F5F4T5T7F
480) 1T2TA11T2F3T4F5T4F5A6F4F5A6F7TA1
481) 1T2TA11T2F3T4F5T4F5A6F4F5F7TA1
482) 1T2TA11T2F3T4F5T4F5A6F4F5T7TA1
483) 1T2TA11T2F3T4F5T4F5A6F4T5A6T7F
484) 1T2TA11T2F3T4F5T4F5A6F4T5F7F
485) 1T2TA11T2F3T4F5T4F5A6F4T5T7F
486) 1T2TA11T2F3T4F5T4F5F4F5A6F7TA1
487) 1T2TA11T2F3T4F5T4F5F4F5F7TA1
488) 1T2TA11T2F3T4F5T4F5F4F5T7TA1
489) 1T2TA11T2F3T4F5T4F5F4T5F7F
490) 1T2TA11T2F3T4F5T4F5T4F5A6F7TA1
491) 1T2TA11T2F3T4F5T4F5T4F5F7TA1
492) 1T2TA11T2F3T4F5T4F5T4F5T7TA1
493) 1T2TA11T2F3T4F5T4F5T4T5F7F

494) 1T2TA11T2F3T4F5T4T5A6F7F

495) 1T2TA11T2F3T4F5T4T5A6T7F

496) 1T2TA11T2F3T4F5T4T5F7F

497) 1T2TA11T2F3T4T5A6F7F

498) 1T2TA11T2F3T4T5A6T7F

499) 1T2TA11T2F3T4T5F7F

500) 1T2TA11T2TA1

References

Aarts E. and Van Laarhoven P.J.M. (1985). Statistical Cooling; a general approach to combinatorial optimization problems. *Philips Journal of Research*, 40, pp. 193-226.

Aarts, E. and Korst, J. (1989). *Simulated Annealing and Boltzman Machines*. John Wiley & Sons, Chichester.

Andriole, S.J. (1986). *Software Validation, Verification. Testing and Documentation*, Petrocelli Books.

Ankenbrandt, C.A. (1990). An extension to the theory of convergence and a proof of the time complexity of genetic algorithms. In GJE Rawlins (Ed.), Foundations of Genetic Algorithms. Morgan Kaufmann San Mateo, CA, pp. 53-68.

Bäck, T., Hoffmeister, F., Schwefel H-P. (1991). Extended selection mechanisms in genetic algorithms. In RK Belew and LB Booker (Eds.), *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 92 - 99.

Baker, J.E. (1985). Adaptive selection methods for genetic algorithms. In JJ Grefenstette (Ed.), *Proceedings of an International Conference on Genetic Algorithms and their Application*, (1985) pp. 100-111.

Bell, D., Morrey, I., and Pugh J. (1987). *Software Engineering: A Programming Approach*, Prentice Hall, International (UK).

Berry, R.H. and Smith, G.D. (1993). Using a genetic algorithm to inestigate taxation induced interactions in capital budgeting. In R.F. Albrecht, C.R. Reeves and N.C. Steele, *Artificial Neural Nets and Genetic Algortihms*, Springer-Verlag Wirn, New York.

Brindle, A. (1981). *Genetic Algorithms for Function Optimization* (Technical Report TR81-2) Department of Computer Science, University of Alberta, Edmonton.

Budd, T.A. (1981). Mutation Analysis: ideas, examples, problems and prospects. In B. Chandrasekaran and Radicchi S. (Eds), *Computer Program Testing*. North-Holland Publishing Company. pp. 129 - 148.

Cerny, V. (1985). A thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45, pp. 41-55.

Chang, K-H., Cross, J.H. II, Carlisle, W.H., Brown, D.B. (1992). A framework for intelligent test data generation. *Journal of Intelligent and Robotic Systems* 5, pp. 147-165.

Chi, P.C. and Nau, D.S. (1988) Improving game board evaluator with genetic algorithms. In *Proceedings - 1988 Spring Symposium Series: Computer Game Playing*, pp. 29-30, Stanford, CA.

Clarke, L.A. (1976). A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, vol se-2 no 3 (September) pp. 215 - 222.

Clarke, L.A. and Richardson, D.J. (1981). Symbolic evaluation methods. In B. Chandrasekaran and Radicchi S. (Eds), *Computer Program Testing*. North-Holland Publishing Company. pp. 65 - 102.

Clarke, L.A., Hassell, J., Richardson D.J. (1982). A close look at domain testing. *IEEE Transactions on Software Engineering*, vol se-8 no 4, pp. 380 - 390.

Coward, P.D. (1988). A review of software testing. *Information and Software Technology*. vol 30, no 3 (April) pp. 189-198.

Cooper, D.W. (1976). Adaptive Testing. In *Proceedings of the 2nd International Conference on Software Engineering*, pp. 102 - 105.

Dammeyer, F. and Voss S. (1993). Dynamic tabu list management using the reverse elimination method. *Annals of Operations Research*.

Davis, L. (1985). Job shop scheduling with genetic algorithms. In Grefenstette, JJ (Ed.), *Proceedings of the First International Conference on Genetic Algorithms*. Lawrence Erlbaum Associates, Hilldale, NJ, 1985.

Davis, L. (Ed.) (1987). *Genetic Algorithms and Simulated Annealing*. Morgan Kaufmann Publishers, Inc. Los Altos, CA.

Davis, L. (1991). *Handbook of the Genetic Algorithms*, Van Nostrand Reinhold, New York.

Davis, T.E. and Principe, J.C. (1993). A simulated annealing like convergence theory for the simple genetic algorithm. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pp. 174 - 181.

Davis, T.E. and Principe, J.C. (1993). A Markov chain framework for the simple genetic algorithm. *Evolutionary Computing* 1(3), pp. 269-288.

DeMillo, R. and Systems Research Laboratories, Waukesha, Wisconsin, (1980). Mutation analysis as a tool for software quality assurance. In *Proceedings of Compsac*, pp. 390 - 393.

DeMillo, R.A., McCracken, W.M., Martin R.J., Passafiume J.F. (1987). *Software Testing and Evaluation*, The Benjamin/Cummings Publishing Company, Inc.

DeMillo, R. and Offutt A.J. (1988). Experimental results of automatically generated adequate test sets. In *Proceedings 6th Annual Pacific Northwest Software Quality Conference.* September, Portland, Oregon, USA pp. 209-232.

Dowsland, K.A. (1993). Simulated Annealing. In *Modern Heuristic Techniques for Combinatorial Problems*, Reeves, CR ed. Blackwell Scientific Publications, Oxford.

Duran, J.W. and Ntafos, S.C. (1982). A report on random testing. In *Proceeds of the 5th International Conference on Software Engineering*, pp. 349 - 356.

Duran, J.W. and Ntafos, S.C. (1984). An evaluation of random testing. *IEEE Transactions on Software Engineering*, vol se-10 no 4 (July) pp. 438 - 444.

Farrar, S.V. (1995). *Optimisation models for corporate taxation in capital budgeting*. PhD Thesis, University of Plymouth.

Floyd, R.W. (1967). Assigning meanings to programs. In *Proceedings of the Symposium on Applied Mathematics*. Vol 19, Providence RI: American Mathematical Society, pp. 19-32.

Fogarty, T.C. (1993). Reproduction, ranking, replacement and noisy evaluations: experimental results. *Proceedings of the Fifth International Conference on Genetic Algorithms*, pp. 634.

Forrest, S. and Mitchell, M. (1993). Relative building-block fitness and the building block hypothesis. *Advances in Neural Information Processing* Systems 6. San Mateo, CA. Morgan Kaufmann. pp. 109- 126.

Forrest, S. and Mitchell, M. (1992). What makes a problem hard for a genetic algorithm? Some anomalous results and their explanation. *Machine Learning* 13, pp. 285 - 319.

Fosdick, L.D. and Osterweil, L.J. (1976). Data flow analysis in software reliability. *ACM Computer Surveys* 8(3):305-330. September.

Geist R., Offutt, A.J., Harris, F.C. (1992). Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computers*, vol 41 no 5 (May) pp. 550 - 558.

German S. and German D. (1984). Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-6, pp. 721-741.

Glover, F. (1989). Tabu Search - Part I. *ORSA Journal on Computing*. Vol 1, No 3, pp. 190-206.

Glover, F. (1990). Tabu Search - Part II. *ORSA Journal on Computing*. Vol 2, No 3, pp. 4-32.

Glover, F., Taillard, E., de Werra, D. (1993). A user's guide to tabu search. *Annals of Operations Research* 41, pp. 3-28.

Glover. F. and Laguna M. (1993). Tabu Search. In Reeves, CR (Ed.), *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific Publications, Oxford.

Glover, F. (1994). Tabu search for nonlinear and parametric optimization (with links to genetic algorithms). *Discreet Applied Mathematics*, 49. pp. 231 -235.

Goldberg, D.E. and Segrest, P. (1987). Finite Markov chain analysis of genetic algorithms. In: *Genetic Algorithms and their Application: proceedings of the*

Second International Conference on Genetic Algorithms. Lawrence Earlbaum Associates, Hillsdale, NJ.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison Wesley, Reading, MA.

Goodenough, J.B. and Gerhart, S.L. (1975). Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, vol se-1 no 2 (June) pp. 156 - 173.

Hedley, S. and Hennell, P. (1984). The cause and effect of infeasible paths in computer programs. In *Proceedings of the 8th International Conference on Software Engineering*, IEEE.(August), pp. 259 - 266.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI.

Holmes, S.T., Jones, B.F., Eyres, D.E. (1993). An improved strategy for the automatic generation of test data, *Software Quality Management.*

Horner, A. and Goldberg, D. (1991). Genetic algorithms and computer-assisted music composition. In RK Belew and LB Booker (Eds), *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 437 - 441.

Howden, W.E. (1976). Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering.* vol se-2 no 3 (September) pp. 208 - 215.

Howden, W.E. (1980). Functional program testing. *IEEE Transactions on Software Engineering* vol se-6 no 2 (March) pp. 162 - 169.

Howden, W.E. (1981). Errors, design properties and functional program tests. In B. Chandrasekaran and Radicchi S. (Eds), *Computer Program Testing* North-Holland Publishing Company. pp. 104 - 127.

Howden, W.E. (1981). Completeness criteria for testing elementary program functions. In *Proceedings of the 5th conference of Software Engineering.* pp. 235 - 243.

Howden, W.E. (1982). Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering,* vol se-8 no 4 (July) pp. 371 - 379.

Howden, W.E. (1985). The theory and practice of functional testing. *IEEE Software* (September) pp. 6 - 17.

Ince, D. and Hekmatpour, S. (1984). *An evaluation of some black-box testing methods.* Technical report no 84/7 Computer Discipline, Faculty of Mathematics Open University 1984.

Ince, D. (1987). The automatic generation of test data. *The Computer Journal.* vol 30 no 1, February, pp. 63 - 69.

Jepson, D.W. and Gelatt, C.D. Jr. (1983). Macro placement by Monte Carlo Annealing. In *Proceedings IEEE Conference on Computer Design*, Port Chester, pp 495-498.

Johnson, D.S., Papadimitriou, C.H., Yannakakis, M. (1985). How easy is local search? In *Proceedings Annual Symposium on Foundations of Computer Science*, Los Angeles, pp 39-42.

Jones, B.F., Sthamer, H.H., Eyres, D.E. (1995). Generating test data for ADA procedures using genetic algorithms. *Genetic Algorithms in Engineering Systems: Innovations and Applications.* 12 - 14 September, pp. 65 - 70.

Jones, B.F., Sthamer, H., Yang, X., Eyres, D.E. (1995). The automatic generation of software test data sets using adaptive search techniques. In *Proceedings of Third International Conference on Software Quality Management.* Vol 2, pp. 435-444.

Jorgensen, P.C. and Erickson, C. (1994) Object-Oriented integration testing, *Communications of the ACM* , September p 30 - 38..

Kido, T., Kitano, H., Nakanishi, M (1993). A hybrid search for genetic algorithms: combined genetic algorithms, tabu search, and simulated annealing. *Proceedings of the Fifth International Conference on Genetic Algorithms*, pp. 641.

Kirkpatrick, S., Gellat, C.D., Vecchi M.P. (1983). Optimization by simulated annealing. *Science*, 220 pp. 671 - 680.

Korel, B. (1990). Automated software test data generation. *IEEE Transactions on Software Engineering*, Vol 16 No 8 (August) pp. 870-879.

Korel, B., Wedde,H., Ferguson,R. (1992). Dynamic method of test data generation for distributed software. *Information and Software Technology,* Vol 34 No 8 (August) pp. 523-531.

Louis, S. and Rawlins G.J.E. (1992*). Predicting Convergence Time for Genetic Algorithms.* Technical Report TR 370, Indiana University, Department of Computer

Michalewicz, Z. (1992)., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, Berlin.

Miller, W. and Spooner D.L. (1976). Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, vol se-2 no 3 (September) pp. 223 - 226.

Miller, B.L. and Goldberg, D.E. (1995). *Genetic Algorithms, Tournament Selection and the Effects of Noise*. IlliGAL Report No. 95006.

Mühlenbein, H. and Schlierkamp-Voosen, D. (1993). Predictive models for the breeder genetic algorithm in continuous parameter optimization. *Evolutionary Computation* 1(1), pp. 25 - 49.

Myers, Glenford J.(1979). *The Art of Software Testing*, John Wiley and Sons.

Nurmela, K.J. (1995) *Constructing Spherical Codes by Global Optimization Methods*. Technical Research Report no 32. Helsinki University of technology Digital Systems Laboratory.

Osterweil, L.J. (1983). Toolpack - an experimental software development environment research project. *IEEE Transactions on Software Engineering*, vol se-9 no 6 (November) pp. 673 - 685.

Paige, M.R. and Holthouse, M..A. (1977). On sizing software testing for structured programs. In Proceedings of 7th Annual International Conference on Fault-Tolerant Computing, IEEE 28-30 July, Los Angeles, CA pp 217.

Park, K. and Carter, B. (1994). *On the effectiveness of genetic search in combinatorial optimization*. Boston University Technical Report BU-CS-94-010.

Poston, Robert M. (1994). Automated testing from object models. *Communication of the ACM,* September, pp. 48 - 59.

Prügel-Bennet, A. and Shapiro, J.L. (1994). Analysis of genetic algorithms using statistical mechanics. *Physical Review Letters* 72(9), 1305 - 1309.

Qi, X. and Palmieri, F. (1994). Theoretical analysis of evolutionary algorithms with a infinite population size in continuous space, Part 1: Basic properties of selection and mutation. *IEEE Transactions on Neural Networks: Special Edition on Evolutionary Computation*, 5(1), pp. 102-119.

Rayward-Smith, V.J. and Debuse, J.C.W. (1994). Generalised adaptive search techniques. In *Proceeding of ACEDC'94.* Plymouth, England.

Romeo, F. and Sangiovanni-Vincentelli, A.L. (1985). Probablistic hill climbing algorithms: properties and applications. In *Proceedings Chapel Hill Conference on VLSI*, Chapel Hill, NC pp 393-417.

Roper, M. (1994). *Software Testing*, McGraw-Hill, Inc.

Roper, M., Maclean, I., Brooks, A., Miller, J., Wood, M. (1995). *Genetic Algorithms and the Automatic Generation of Test Data*. Technical Report, University of Strathclyde, Department of Computer Science Technical Reports.

Schaffer, J.D., Carvara, R.A., Eshelman, L.J. and Das, K. (1989). A study of control parameters affecting online performance of genetic algorithms for function optimization. In J.D. Schaffer (Ed.), of the *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Los Altos, CA, pp. 51-60.

Schaffer J.D. and Eshelman, L.J. (1993). On crossover as an evolutionary viable strategy. *Proceedings of the Fifth International Conference on Genetic Algorithms*, pp. 61 - 68.

Sthamer, H.H., Jones B.F., Eyres, D.E. (1994) Generating test data for ADA generic procedures using genetic algorithms, In *Proceedings of the ACEDC 1994*, University of Plymouth, UK pp. 134 - 140.

Storer, J.A., Becker, J., Nicas, A.J. (1985). Uniform circuit placement. In P. Bertolazzi and F. Luccio (Eds.) *Proceeding International Workshop on Parallel*

Computing and VLSI, Amalfi, Elsevier Science Publishers, Amsterdam, pp 255 - 273.

Syswerda, G. (1989). Scheduling optimization using genetic algorithms, In Schaffer, J (Ed), *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Los Altos, CA, pp. 332-349.

Syswerda, G. (1989). Uniform crossover in genetic algorithms. In Schaffer, J (Ed), *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 2 - 9

Syswerda, G. and Palmucci, J. (1991). The application of genetic algorithms to resource scheduling, In Belew, R, and Booker L., (Eds), *Proceedings of the Fourth International Conference on Genetic Algorithms*, Morgan Kaufmann Publishers, Los Altos, CA, 1991, pp. 502 - 508.

Thornton, A.C. (1994). Genetic Algorithms Versus Simulated Annealing: Satisfaction of large sets of algebraic mechanical design constraints. *Artificial Intelligence in Design '94*. pp. 381-398.

Vignaux, G.A. and Michalewicz Z. (1989). Genetic Algorithms for the transportation problem. *Methodologies for Intelligent Systems*, 5, pp. 252-259.

White L.J. and Cohen E.I. (1980). A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, vol se-6 no 3 (May) pp. 247 - 257.

White, L.J., Cohen, E.I., Zeil, S.J. (1981). A domain strategy for computer program testing. In B. Chandrasekaran and Radicchi S. (Eds), *Computer Program Testing.* North-Holland Publishing Company , pp. 103 - 113.

White L.J. (1987). Software testing and verification. *Advances in Computers,* vol 26, pp. 335-394.

White, M.S. and Flockton, S.J. (1995). Modeling the behaviour of the genetic algorithm. *Genetic Algorithms in Engineering Systems: Innovations and Applications.* 12 - 14 September, pp. 349 - 356.

Wiston, P.H. (1984). *Artificial Intelligence.* Adison-Wesley Publishing Company.

Woodward, M.R., Hedley, D., Hennell, M.A. (1980). Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering*, vol se-6 no 3 (May) pp. 278 - 287.

Xanthakis,S., Ellis,C., Skourlas,C., LeGall, A., Katsikas,S. (1992). Application of genetic algorithms to software testing. In *Proceedings 5th International Conference on Software Engineering*, Toulouse, France, December.