School of Physical Sciences and Engineering
King's College London
MSc in Advanced Software Engineering

Search Algorithms for Regression Test Suite Minimisation

By
Benjamin Cook

Supervised by
Prof. Mark Harman

1st September 2006

Abstract

Regression testing is a costly, but necessary process. Unfortunately, there may be insufficient resources to allow for the re-execution of all test cases in the test suite for regression testing. Under these circumstances, test suite minimisation techniques aim to improve the effectiveness of regression testing by removing as many test cases as possible, whilst retaining the same level of program coverage.

Previous work has focused on the regression test case prioritisation problem, using a range of algorithms to order test cases so that the most beneficial are executed first. However, very little work has been carried out on regression test suite minimisation. None of this work has utilised Meta-Heuristic Algorithms, which utilise the automated discovery of heuristics in order to find solutions to problems for which there is no problem-specific algorithm, and Evolutionary Search Algorithms, which are forms of Meta-Heuristic search that use mechanisms inspired by biological evolution to guide the search.

The paper presents the results from an empirical study of the application of several Greedy, Meta-Heuristic and Evolutionary Search Algorithms to five programs, ranging from 374 to 11,148 lines of code for three choices of fitness metric. The paper addresses the problems of choice of fitness metric, characterisation of search landscape and determination of the most suitable search technique to apply.

The empirical results show the nature of the regression test suite minimisation search landscape, indicating that it has many local optima that are almost equivalent to the global optima. The results also show that Greedy Algorithms consistently perform best.

Acknowledgements

I would like to thank all the people who have supported me whilst writing this paper. Firstly, I would like to express my gratitude to my supervisor, Prof. Mark Harman, for his constant support, advice, and patience when no work was initially forthcoming I would like to thank Zheng Li for providing me with the five programs and range of test suites investigated within this paper. I'd also like to thank my Mum, Dad and girlfriend, Jenni, who have endured me through the many weeks I have spent working on this paper!

Table of Contents

Table of Figures

Table of Figures

1. Introduction

Regression Testing

Regression testing is applied to modified software to demonstrate that modified code behaves as intended, and has not been adversely affected by changes. Software engineers regularly save the test suites they have developed for their software so that they can reuse those same test suites to test future software. This test suite reuse, in the form of regression testing, is common in the software industry [10] and, in addition to other regression testing tasks, has been estimated to account for up to half of the total cost of software maintenance [1, 7]. Executing all of the test cases in a test suite, however, can require a large amount of effort. There may not be sufficient resources available to run all test cases during regression testing. For these reasons, this paper has been written in an attempt to reduce the cost of regression testing, specifically by using test suite minimisation techniques. Test suite minimisation aims to reduce the size of a test suite, by removing redundant test cases, whilst still being able to test the entire program, according to some coverage criterion. It can be used to reduce the cost of a testing objective.

Test Suite Minimisation

Test suite minimisation can address a variety of testing objectives. For example, many software testing standards require 100% statement or branch coverage. Although it may be argued that 100% coverage does not equal complete, or even adequate, testing, it is not the purpose of this paper to enter into a discussion of the effectiveness of branch, statement and block coverage as a testing measure. The fact is that these standards do exist and must be adhered to in the software industry. An effective test suite minimisation technique could reduce the size of the test suite, thus reducing the cost in effort of running the original test suite. Whilst this is of little benefit for small programs and test suites, the benefits do, however, become far more apparent when a large program with a large test suite needs to be tested. In such circumstances, this activity could cost several days worth of computational effort. Test suite minimisation is a technique that reduces the computational effort of running an entire test suite, whilst still offering the same level of testing.

In previous works, the effect of test set minimisation on fault detection ability was investigated by Wong et al. [18]. They carried out an empirical study to determine whether it was the size of the test suite or the coverage of the test suite on the program that determined the fault detection effectiveness. Wong et al. found that when the size of a test suite is reduced while the coverage is kept constant, there is little or no reduction in its fault detection effectiveness.

Related Work

Techniques for a similar regression testing tasks have also been proposed. One of these techniques, test case prioritisation [8, 12], is used to ensure that the maximum possible coverage is achieved by some pre-determined cut-off point. There has been much investigation into improving test case prioritisation, which orders test cases so that those test cases with highest priority, according to some criterion (a ‘fitness metric’), are executed first.

Most of the proposed test case minimisation techniques were code-based, relying on information relating test cases to coverage of code elements. In [12], Rothermel et al. investigated several prioritising techniques such as total statement (or branch) coverage prioritisation and additional statement (or branch) coverage prioritisation, which can improve the rate of fault detection. In [17], Wong et al. prioritised test cases according to the criterion of 'increasing cost per additional coverage'.

The use of Greedy algorithms for test case prioritisation has been widely studied. Greedy algorithms incrementally add test cases to an initially empty sequence. The algorithms select a test case based on which one achieves the maximum fitness value for the desired metric (e.g. some measure of coverage). However, as Rothermel et al. [12] point out, the greedy algorithms may not always choose the optimal test case ordering.

In an attempt to find a technique more effective than the greedy algorithm, Li and Harman [8] introduced meta-heuristic and evolutionary algorithms to the test case prioritisation problem.

Meta-heuristic search techniques [11] are high-level frameworks that utilise the automated discovery of heuristics in order to find solutions to problems for which there is no satisfactory problem-specific algorithm. Evolutionary algorithms, of which Genetic Algorithms are a subclass, are a form of meta-heuristic search that uses mechanisms inspired by biological evolution to guide the search.

Li and Harman [8] found that Genetic Algorithms performed well in selecting an optimal test case ordering, although Greedy approaches were surprisingly effective, given the multi-modal nature of the search landscape.

Summary

No previous work has investigated the use of the search algorithms described previously in the test suite minimisation problem. This is thus the first paper to study basic search, meta-heuristic and evolutionary algorithms empirically for test suite minimisation for regression testing.

In this paper, five search algorithms are investigated and tested as test suite minimisation techniques: two Meta-Heuristic search algorithms (Hill-Climbing and Genetic Algorithm) and three Greedy algorithms (Basic Greedy, Additional Greedy and 2-Optimal Greedy). Of these five algorithms, only Greedy and Additional Greedy have been studied previously for test suite minimisation. The paper presents results from an empirical study that compared the performance of the five search algorithms applied to five programs.

The rest of this paper is organised as follows: Section 2 reviews the literature on the background to this project and discusses the algorithms used, Section 3 explains the rationale behind the design for the empirical study, Section 4 presents the results of the empirical study and evaluates them, Section 5 suggests future related studies that may be carried out and Section 6 concludes.

2. Literature Review

2.1 Test Case Prioritisation Techniques

Although little previous work has been done using basic search, meta-heuristic, and evolutionary algorithms to minimise test suites, a substantial amount of work has been carried out investigating the use of these algorithms in test case prioritisation. The test suite minimisation problem and test case prioritisation problems are very similar; so much of this related work has been incorporated into this paper.

In [12], Rothermel et al. formally defined the test case prioritisation problem and empirically investigated six prioritisation techniques. Four of these techniques were based on the coverage of either statements or branches of a program and the other two were based on the estimated ability to reveal faults. Several experiments compared these with the use of no test case prioritisation (untreated), random test case prioritisation and optimal prioritisation (manually selecting those test cases that expose the known program faults). The experimental results showed that the prioritisation techniques can improve the rate of fault detection of test suites. These experiments applied a Greedy Algorithm and an Additional Greedy Algorithm based on code coverage.

In [17], Wong et al. presented a way to combine test suite minimisation and prioritisation to select test cases, according to the criterion of 'increasing cost per additional coverage'. Greedy algorithms were also used and were implemented in a too named ATAC.

In [15], Srivastava and Thiagarajan studied a prioritisation technique based on the changes that have been made to the program being tested. Their technique orders test cases to most thoroughly cover the updated parts of the program so that defects are likely to be found quickly and inexpensively. A test case prioritisation system, named Echelon, was built, based on this technique. It too uses a Greedy Algorithm. Echelon is currently being integrated into the Microsoft software development process. It has been tested on large Microsoft binaries and it has proved to be effective in ordering tests based on changes between two program versions.

In [8], Li and Harman investigated the use of meta-heuristic and evolutionary algorithms for test case prioritisation. An empirical study was carried out on the application of several greedy, meta-heuristic and evolutionary search algorithms to six programs, ranging from 374 to 11,148 lines of code for three choices of fitness metric. The empirical results replicated previous results concerning Greedy Algorithms. They recognize the nature of the regression testing search space and indicate that it is multi-modal. The results also show that Genetic Algorithms perform well, although Greedy approaches are surprisingly effective, given the multi-modal nature of the landscape.

2.2 Search Algorithms

A search algorithm is an algorithm that takes a problem as input and returns a solution to the problem, usually after evaluating a number of possible solutions. The set of all possible solutions to a problem is called the search space. Brute-force search or uninformed search algorithms use

the simplest, most intuitive method of searching through the search space, whereas informed search algorithms use heuristics to apply knowledge about the structure of the search space to try to reduce the amount of time spent searching.

In an informed search, a heuristic that is specific to the problem is used as a guide. A good heuristic will make an informed search dramatically out-perform any uninformed search.

2.3 Meta-Heuristic Search

A Meta-Heuristic Search is a Heuristic method for solving a very general class of problem by combining user given procedures — usually heuristics themselves — in a hopefully efficient way. Meta-Heuristics Searches are generally applied to problems for which there is no satisfactory problem-specific algorithm or heuristic; or when it is not practical to implement such a method. The most commonly used Meta-Heuristics are targeted to optimisation problems but they can, in theory, tackle any problem that can be recast in that form.

2.4 Evolutionary Algorithms

An Evolutionary Algorithm is a subset of evolutionary computation, a generic population-based Meta-Heuristic optimisation algorithm. An Evolutionary Algorithm uses some mechanisms inspired by biological evolution: reproduction, mutation, recombination, natural selection and survival of the fittest. Candidate solutions to the optimisation problem play the role of individuals in a population, and the fitness function determines the environment within which the solutions "live". Evolution of the population then takes place after the repeated application of the above operators.

Evolutionary Algorithms perform consistently well approximating solutions to all types of problems because they do not make any assumption about the underlying fitness landscape.

2.5 Fitness Function

A fitness function is a particular type of objective function that quantifies the optimality of an individual in a Genetic Algorithm so that that particular individual may be ranked against all the other individuals. Optimal individuals, or at least individuals which are more optimal, are allowed to breed and mix their datasets by any of several techniques, producing a new generation that will (hopefully) be even better.

Another way of looking at fitness functions is in terms of a fitness landscape, which shows the fitness for each possible individual.

An ideal fitness function correlates closely with the algorithm's goal, and yet may be computed quickly. Speed of execution is very important, as a typical genetic algorithm must be iterated many times in order to produce a useable solution to a problem.

2.6 Greedy Algorithm

A Greedy Algorithm is an implementation of the 'next best' search strategy. It makes the locally optimum choice at each stage with the hope of finding the global optimum. It works on the principle that the element with maximum fitness is selected first, followed by the element with second highest fitness and so on until a complete, but possibly sub-optimal, solution has been created. Greedy aims to minimise the estimated cost to reach some goal.

Greedy algorithms mostly, but not always, fail to find the globally optimal solution because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later. For example, all known greedy algorithms for NP-complete problems do not consistently find optimum solutions. However, they are useful because they are quick to implement, require low computational effort and often give good approximations to the optimum.

A simple example, based on statement coverage, is shown in table 1. If the aim is to achieve full statement coverage in as fewest test cases as possible, a Greedy Algorithm will select A, B, C or A, C, B. Test case A is selected first since it covers six statements; the maximum covered by a single test case. Test cases B and C both cover the same amount of statements, so the Greedy algorithm could return either A, B, C or A, C, B as the solution. However, it is clear that the optimal test case combinations for this example are B, C and C, B.

Test Case	Statement							
	1	2	3	4	5	6	7	8
A	X	X	X			X	X	X
B	X	X	X	X				
C					X	X	X	X

Table 2.1: A case in which the Greedy Algorithm will not produce an optimal solution

Consider the example of statement coverage for a program containing m statements and a test suite containing n test cases. For the Greedy Algorithm, the statements covered by each test case should be counted first, which can be accomplished in $O(m\ n)$ time; then sort the test cases according to the coverage. In the second step, quicksort can be used thereby increasing the time complexity by $O(n \log n)$. Typically, m is greater than n, in which case the cost of this process is $O(m\ n)$.

2.7 Additional Greedy Algorithm

The 'Additional' Greedy Algorithm is variation of the Greedy Algorithm, with a different strategy. It combines feedback from previous selections. It iteratively selects the maximum fitness element of the problem from that part that has not already been covered by previously selected elements [8].

In the example in table 2.1, test case A is selected first because it covers the most statements. This leaves statements 4 and 5 still uncovered. Test cases B and C both cover one of those two remaining uncovered statements. Thus, the Additional Greedy Algorithm will return either A, B, C or A, C, B – neither of which are the optimum solution.

Consider statement coverage: the Additional Greedy Algorithm requires coverage information to be updated for each unselected test case following the choice of a test case. Given a program containing m statements and a test suite containing n test cases, selecting a test case and readjusting coverage information has cost $O(m\ n)$ and this selection and readjustment must be performed $O(n)$ times. Therefore, the cost of the Additional Greedy Algorithm is $O(m\ n2)$.

2.8 2-Optimal Greedy Algorithm

The 2-Optimal Greedy Algorithm is an instantiation of the K-Optimal Greedy Algorithm when K=2. The K-Optimal approach selects the next K elements that, taken together, consume the largest part of the problem. In the case of K-Optimal *Additional* Greedy, it is the largest *remaining* part of the problem that is selected [8].

The K-Optimal approach has been studied in the area of heuristic search to solve the Travelling Salesman Problem (TSP) that is defined as *"find the cycle of minimum cost that visits each of the vertices of a weighted graph G at least once"* [11]. Experiments suggest that 3-Optimal tours are usually within a few percent of the cost of optimum tours for the TSP. When K is greater than 3, the computation time increases considerably faster than the quality of the solution. The 2-Optimal approach has been found to be fast and effective [14].

For the example in table 2.1, test cases B and C are selected first since the combination of test cases B and C cover more statements than any other combination. There are no over statements to cover, so the 2-Optimal Greedy Algorithm returns B, C (which is the global optimum).

Consider statement coverage: the 2-Optimal Greedy Algorithm updates coverage information for each unselected test case following the choice of each pair of test cases. Given a program containing m statements and a test suite containing n test cases, selecting a pair of test cases and readjusting coverage information has cost $O(m\ n2)$ and this selection and readjustment must be performed $O(n)$ times. Therefore, the time complexity of the 2-Optimal Greedy Algorithm is $O(m\ n3)$.

2.9 Hill-Climbing Algorithm

Hill-Climbing is a search algorithm is a local search algorithm. The current path is extended with a successor element which is closer to the locally optimal solution than the end of the current path.

There are two primary variations of Hill-Climbing: steepest ascent and next best ascent. In next best ascent ascent, the first element that is closer to the solution is chosen. Steepest ascent is more complicated and comprises of the following steps, taken from Li and Harman [8]:

1. Pick a random solution state and make this the current state.
2. Evaluate all the neighbours of the current state and choose the neighbour with maximum fitness value.
3. Move to the state with the largest increase in fitness from the current state. If no neighbour has a larger fitness than the current state then no move is made.
4. Repeat the previous two steps until there is no change in the current state.
5. Return the current state as the solution state.

Hill-Climbing is simple and computationally cheap to implement and execute. However, it is common for the search to yield sub-optimal results that are locally optimal, but not globally optimal.

2.10 Genetic Algorithm

A genetic algorithm is a programming technique that mimics the process of natural genetic selection according to Darwinian theory of biological evolution as a problem solving strategy [5, 8]. Genetic algorithms represent a class of adaptive search techniques, based on biological evolution, which are used to approximate solutions.

Given a specific problem to solve, the input to the genetic algorithm is a set of potential solutions to that problem and a metric called a fitness function that allows each candidate solution to be quantitatively evaluated. One application of genetic algorithms is regression test suite minimization. The candidate solutions are test suites, with the aim of the genetic algorithm being to minimise it.

Figure 1 shows a typical genetic algorithm procedure, taken from Li and Harman (2006). The procedure begins by initialising a population P randomly, evaluating the fitness of candidate solutions (individuals), selecting pairs of individuals that are combined and mutated to generate new individuals, and forming the next generation. The algorithm continues through a number of generations until the termination condition has been met.

Genetic Algorithm:

```
Begin
t ← 0
initialise P(t)
evaluate P(t)
while (not termination condition) do
    begin
        t ← t + 1
        select P(t) from P(t – 1) according to evaluation
        crossover P(t) according to crossover rate
        mutate P(t) according to mutation rate
        evaluate P(t)
    end
end
```

Figure 2.1: Genetic Algorithm structure

The initial population is a set of randomly generated candidate solutions (individuals), also known as chromosomes. Each individual is represented by a sequence of variables/parameters, known as the genome. The genome encodes a possible solution to a given problem. The standard encoding of the solution is an array of binary bits, but other structures and forms of encoding can be used (for example, real-valued or character-based). The main property that makes this form of encoding convenient is the ease in which binary bits are aligned due to their fixed size. This facilitates simple crossover operation.

Selection

A proportion of the existing population is selected and later used to breed the next generation. The selection process is biased, and individuals are selected through a fitness based process, where the fitter individuals are more likely to be selected.

There are several genetic selection algorithms. Some rate the fitness of each candidate solution and preferentially select the best solutions. Other methods of selection rate only a random sample of the population, as this process may be time consuming. Most selection functions are stochastic and designed so that a small proportion of less fit solutions are selected. This helps keep the diversity of the population large prevents premature convergence onto local minima in the search landscape. Two of the most popular and well studied selection methods are roulette wheel selection and tournament selection:

In roulette wheel selection, individuals are assigned a fitness value by the fitness function. This fitness level is used to assign a probability of selection to each individual. While individuals with a higher fitness will be less likely to be eliminated, there is still a chance they may be. There is also a chance some weaker individuals may survive the selection process. This is an advantage because, although an individual may be weak, it may include some component which could prove useful following the crossover process.

In tournament selection, a “tournament” is run among a few individuals chosen at random from the population and the winner (individual with the best fitness) is selected for crossover. Selection pressure can be adjusted by changing the tournament size. If the tournament size is larger, weak individuals have a smaller chance to be selected. Again, this is an advantage because those weak individuals may contain some useful component.

A less sophisticated selection algorithm is truncation selection, which will eliminate a fixed percentage of the weakest individuals. The individuals are ordered by fitness, and some proportion of the fittest individuals are selected and reproduce.

Crossover

Crossover is a genetic operator that generates a new population from those individuals selected through the selection process. For each new individual to be produced, two individuals (the “parents”) are selected for breeding from the pool selected previously. A probability of crossover determines whether crossover should be performed or not. There are several methods of crossover that may be used:

"One point crossover" selects a crossover point on the parent individual. All data beyond that point in the individual is swapped between the two parent individuals. The resulting individuals are the children, as shown in figure 2.2.

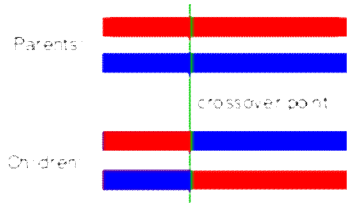


Figure 2.2: One point crossover

In "two point crossover", there are two points selected on the parent individuals. All data between the two points is swapped between the parent organisms, giving two child individuals, shown in figure 2.3.

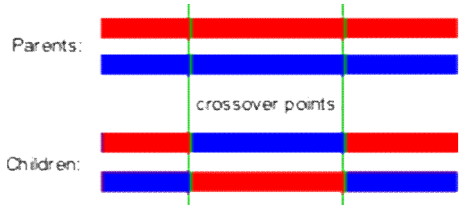


Figure 2.3: Two point crossover

Another crossover method is "cut and splice", in which each parent individual has a separate choice of crossover point. This causes a change in the length of the children individuals, as shown in figure 2.4.

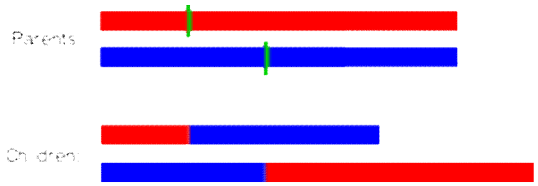


Figure 2.4: Cut and splice crossover

In uniform crossover, the two parent individuals are combined to produce two new offspring. Individual bits in the individual are compared between the two parents. The bits are swapped with a fixed probability, typically 0.5.

Half uniform crossover is similar to uniform crossover. The two parent individuals are combined to produce two new offspring. Exactly half of the non-matching bits in the parent individuals are swapped. This gives the number of differing bits (the Hamming distance), which is divided by two. The resulting number is how many of the bits that do not match between the two parents are swapped.

Under some circumstances, a direct swap may not be possible. One such case is when the chromosome is an ordered list, such as an ordered list of the cities to be travelled for the travelling salesman problem. A crossover point is selected on the parent individuals. Since the

chromosome is an ordered list, a direct swap would introduce duplicates and remove necessary individuals from the list. Therefore, the chromosome up to the crossover point is retained for each parent. The information after the crossover point is ordered as it is ordered in the other parent. For example, if the two parents are ABCDEFGHI and IGAHFDBEC and the crossover point is after the third character, then the resulting children would be ABCIGHFDE and IGABCDEFH.

By producing a "child" solution using one of the above methods of crossover, a new individual is created which typically shares many of the characteristics of its parents. New parents are selected for each child, and the process continues until a new population of individuals of appropriate size is generated.

Mutation

The mutation action alters one or more gene values in the individual, depending on the probability of mutation. It is a genetic operator used to maintain genetic diversity from one generation of the population to the next.

The most common mutation operator involves a probability that an arbitrary bit in an individual will be changed. A random variable is generated for each bit in the sequence, and this random variable determines whether or not a particular bit will be modified.

The purpose of mutation in genetic algorithms is to allow the algorithm to avoid local minima in the search landscape by preventing the population of individuals from becoming too similar to each other, which will slow or even stop evolution.

The selection, crossover and mutation processes ultimately result in the next generation population being different from the initial generation. Generally the average fitness will have increased by this procedure for the population, since only the best individuals from the first generation are selected for breeding (along with a very small proportion of less fit individuals).

Termination

The generational process is repeated until a termination condition has been met. There is no guarantee that the genetic algorithm will converge upon a single solution. Some common terminating conditions are:

- A solution is found that satisfies minimum criteria
- Fixed number of generations reached
- Allocated budget (computation time/money) reached
- Highest ranking solution fitness has reached a plateau such that successive iterations no longer produce better results.

2.11 Quicksort

The greedy algorithms need a method to sort test cases according to program coverage.
Quicksort is a well-known sorting algorithm that, on average, makes O(n log n) comparisons to sort n items. However, in the worst case, it makes O(n2) comparisons. Typically, quicksort is significantly faster in practice than other O(n log n) algorithms.

Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists. The steps are:

1. Pick an element, called a pivot, from the list.

2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The base case of the recursion are lists of size zero or one, which are always sorted. The algorithm always terminates because it puts at least one element in its final place on each iteration.

```
function quicksort(q)
    var list less, pivotList, greater
    if length(q) ≤ 1
        return q
    select a pivot value pivot from q
    for each x in q except the pivot element
        if x < pivot then add x to less
        if x ≥ pivot then add x to greater
    add pivot to pivotList
    return concatenate(quicksort(less), pivotList, quicksort(greater))
```

Figure 2.5:Pseudocode of the quicksort algorithm

3. Design and Implementation

3.1 Research Questions

The objective of this paper is to determine which algorithm should be used to minimise a test suite for regression testing. In order to achieve this, there are two specific research questions that must be posed:

- Which algorithm is most effective at minimising a test suite whilst retaining the same program coverage as the original test suite?
- What factors affect the efficiency of the algorithms for minimising a test suite whilst retaining the same program coverage as the original test suite?

These two questions concern the quality and the computation cost of regression testing and, as such, it is essential that they be answered before deciding on a choice of algorithm.

3.2 Effectiveness measure

To address the two research questions, there needs to be a measure with which to determine the effectiveness of each test suite minimisation technique. Obviously, the smaller the test suite after the minimisation technique has been applied, the more effective that technique is. However, simply comparing the number of test cases in the original test suite with the number of test cases in the minimised test suite is not an not a suitable measure of the effectiveness of the minimisation algorithms because this does not check that the same proportion of the program is being covered by this minimised test suite. This fitness function takes all of this into consideration.

Fitness = Program Coverage / Number of Test Cases in the Minimised Test Suite

It could be suggested, on examining this fitness function, that high fitness could be achieved by selecting only one test case that, on its own, achieved a high coverage but did not cover the same proportion of the program as the original test suite. Therefore, the fitness will simply be:

Fitness = Number of Test Cases removed from Original Test Suite

[Providing the same program coverage is achieved in the minimised test suite as in the original test suite].

3.3 Coverage

It is only true that a test suite minimisation technique is effective provided the resulting test suite still achieves its main purpose. That is, it is still able to test the entire program and expose any faults that may be present.

It is not generally possible to know the faults exposed by a test suite before each test case in that test suite has been run. Therefore, program coverage can be used as an alternative. Coverage is also an important concern in its own right. Software companies often require new software to be tested to full coverage as standard. For example, the avionics testing standard [17] has a coverage mandate. The presence of these mandates means that software must be tested to this level, irrespective of whether doing so will expose all the faults present.

Coverage type was used as a variable in the experiments. Each test suite minimisation technique must ensure that the minimised test suite achieves the same coverage as the original test suite.

There are three forms of coverage criteria: block, decision and statement:

- Block Coverage
  The amount of blocks in the program covered by the test suite. In a program, a block is a sequence of consecutive statements, containing no branches except at the end, so that if one statement of the block is executed all are.
- Decision Coverage
  The amount of decisions (branches) in the program covered by the test suite. In a program, a branch is either an entry point to a module, or an outcome to a decision. In a decision tree a branch is any line emerging from any node.
- Statement Coverage
  The amount of statements in the program covered by the test suite. A statement is any line in a program.

3.4 Subjects

This paper used five C programs as subjects for regression test selection algorithms, as summarized in table 2.0 (taken from Li and Harman [8]). The faulty programs Print_tokens, Print_tokens2, Schedule and Schedule2, along with their associated test cases, were assembled by researchers at Siemens Corporate Research for a study of the fault detection capabilities of control-flow and data-flow criteria. Space is a program developed for the European Space Agency.

Program	Lines of Code	Blocks	Decisions	Total test pool size	Average small test suite size	Average large test suite size
Print_tokens	726	126	123	4,130	16	318
Print_tokens2	570	103	154	4,115	12	388
Schedule	412	46	56	2,560	19	228
Schedule2	374	53	74	2,710	8	230
Space	8,564	869	1,068	13,585		1,293

Table 2.2 Experiment Subjects

For each of the Siemens programs, the researchers at Siemens created a test pool containing possible test cases for the program. Li and Harman [8] created sample test suites for each program by first taking the test pools and testing branch coverage information about each test case in those pools. In order to produce small sized test suites, a test case is selected at random

and is added to the test suite only if it adds to the cumulative branch coverage. This process is repeated until total branch coverage is achieved. In order to produce large sized test suites, test cases were selected at random and added to the test suite irrespective of whether they added to the cumulative branch coverage or not. Again, this process was repeated until full branch coverage had been achieved.

Rothermel et al. [12] constructed a test pool of 13,585 for the program space. They then used space's test pool to obtain a range of large sized test suites that each achieved full branch coverage.

The programs, test suites and test cases described above were used in this paper as subjects for determining the most suitable algorithm to use in test suite minimisation.

3.5 Analysis tools

Block, decision (branch) and statement coverage information was obtained from Zheng Li [8]. He used the testing tool Cantata++ to identify, for each program, which blocks, branches and statements were covered by each test case in the test pool.
Zheng Li [8] used Cantata++ to find, for each program, the blocks, branches and statements covered by each test case in the test pool.

3.6 Experimental Design

To make the results reported in this paper more universally applicable, each test suite minimisation technique was instantiated with several values of the three primary variables that govern the nature and outcomes of the search:

- The program to which the regression testing is applied.
  Five programs were studied, ranging from 374 to 11,148 lines of code.
- The coverage criterion to be achieved.
  The three choices of coverage were block, decision and statement for each program.
- The size of the test suite.
  The test suites were classed as either small (8 – 155 test cases) or large (228 – 4,350 test cases).

The experiments involved five C programs. There were 1,000 small test suites and 1,000 large test suites available for each of those programs. To reduce the computation time for the experiments, without significant loss of generality, half of these test suites were used in the experiments. Thus the results obtained are averages obtained over 500 executions of the associated search algorithm, with each execution using a different test suite.

In summary, for each algorithm, for each program, for each coverage criterion and for each test suite, an instantiation of the experiment was carried out. This meant a more general result could be obtained by averaging the results generated by each different test suite experiment. For example, using one algorithm on one program, using one coverage criteria and one test suite size, the result was obtained by running this experiment on 500 different test suites and taking an average.

3.7 Algorithm Design and Implementation

6 search algorithms were used as test suite minimisation techniques:

- Random
- Greedy
- Additional Greedy
- n-Optimal Greedy
- Hill-Climbing
- Genetic Algorithm

Random

The random algorithm was used as the benchmark for the effectiveness of each test suite minimisation technique. If an algorithm performed better than random, it was at least somewhat successful.
For each execution, the random algorithm worked by randomly selecting test cases from the original test suite until the same level of coverage as the original test suite had been obtained. Test cases could only be selected once.

Greedy

For each instantiation, the greedy algorithm used the quick sort algorithm to order the test cases from the original test suite according to their fitness (coverage). The test case with the highest coverage was selected, followed by the test case with the second greatest coverage, and so on, until the same level of coverage as the original test suite had been obtained.

Additional Greedy

For each execution, the additional greedy algorithm would first use the quick sort algorithm to sort the test cases from the original test suite according to fitness (coverage). The test case with the greatest coverage was selected. At this point, the coverage information for the test suite was recalculated so that only those aspects of the program, as yet still uncovered, would be considered. The quick sort algorithm was used to resort the test cases according to the updated coverage information and again, the test case with the greatest coverage was selected. This process continued until the same level of coverage as the original test suite had been obtained.

2-Optimal Greedy

For each instantiation, two test case were selected that, when combined, covered the greatest amount of the program. The coverage information for the test suite was then recalculated so that only those aspects of the program, as yet still uncovered, would be considered. The process repeated until the same level of coverage as the original test suite had been obtained.

Hill-Climbing

The steepest ascent form of Hill-Climbing was used. For each execution, the hill-climbing algorithm used the random algorithm to generate an initial solution containing n test cases. An additional n solution individuals were generated to be the neighbours, each one of these containing n-1 test cases taken from the initial solution.

In the next step, the neighbour individual which covered the most of the program was selected. If it covered as much of the program than the initial solution, the process repeats but with this neighbour individual as the initial solution. If it covered less of the program that the initial solution, the algorithm was terminated and the previous solution became the solution.

Genetic Algorithm

It is common with the application of Genetic Algorithms to find that the parameters of the algorithm need to be changed in order to determine the values that yield the best results. Therefore, some initial experiments were performed in order to optimise the algorithm for the different programs and test suite sizes being studied. The size of the population determines the diversity of the initial population. Insufficient diversity can lead to premature convergence to a suboptimal solution in the search landscape.. Larger programs have a larger search spaces and therefore require a larger population in order to maintain diversity. Population size was set at 50 individuals for the small programs (those with fewer than 1,000 lines of code), and 100 for the large programs (those with 1,000 lines of code or more) [8].

For each instantiation, a number of random solutions are generated using the random algorithm (50 for small programs and 100 for large programs). Instead of selecting only the fittest individuals, a stochastic function was used for the selection process. These algorithms are designed so that a small proportion of less fit solutions are selected in order to help keep the diversity of the population large and prevent premature convergence on poor solutions. The stochastic selection method “roulette wheel” was used.

A fitness value was assigned to each solution. This fitness value was used as the probability of selection for each test case. The fitness metric would change with each new generation of individual solutions. Initially, the fitness metric was defined as the number of test cases removed from the original test suite. After a new generation had been created, the fitness value was changed to be the coverage achieved by each individual solution. After the next generation, the fitness metric was changed back to be the number of test cases removed, and so on. This was to ensure that as test cases were being exchanged in crossover, those individual test suite solutions that no longer achieved adequate program coverage were less likely to be selected.

The uniform crossover method was used. This method ensured more crossover and, therefore, more genetic diversity amongst the test suite solutions. The uniform crossover method combined the two parents to produce two new offspring. Each test case in the parent individuals was compared and swapped with a fixed probability of 0.5 per test case.

Mutation was applied after crossover. It randomly selected one test case in every individual and, either deleted it, or exchanged it with another test case randomly selected from the entire test suite, with a fixed probability of 0.1.

The genetic algorithm was terminated after 50 generations for small programs and after 100 generations for large programs. The expectation is that the average fitness of the population will increase each round. So, by repeating this process for many times, high fitness test suites can be discovered.

The effectiveness of the algorithm at test suite minimisation was measured based on the fitness of the best individual in the population at the end of the 50 or 100 generations.

4. Evaluation of results

4.1 Experiments using small test suites

Figure 4.1: Test suite minimisation of schedule1, using block coverage and small sized test suite

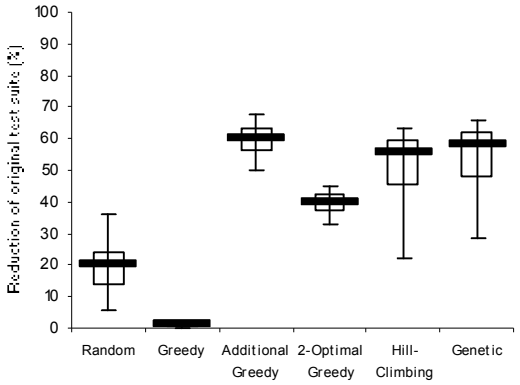


Figure 4.2: Test suite minimisation of schedule1, using branch coverage and small sized test suite

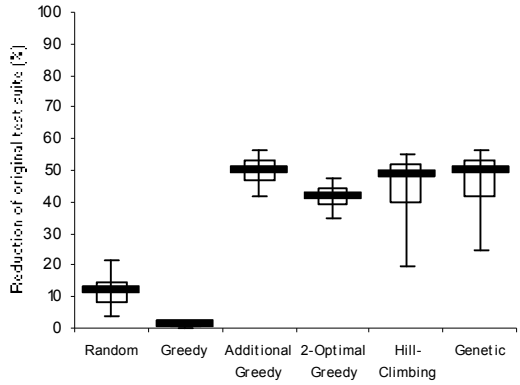


Figure 4.3: Test suite minimisation of schedule1, using statement coverage and small sized test suite

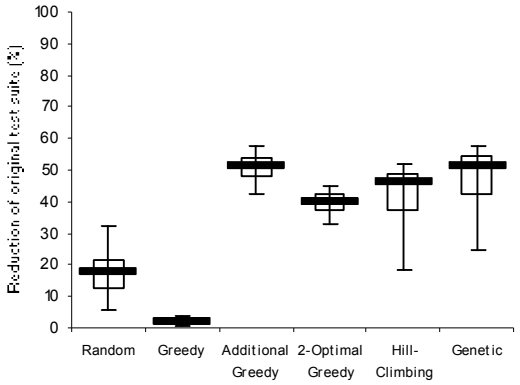


Figure 4.4: Test suite minimisation of schedule2, using block coverage and small sized test suite

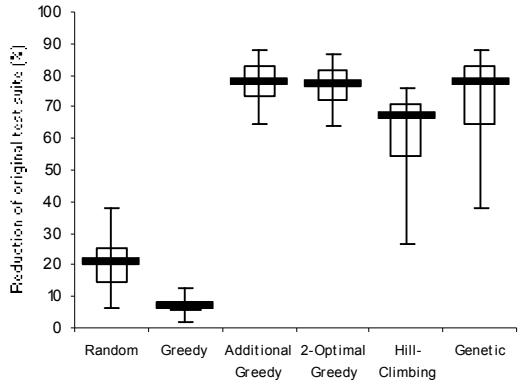


Figure 4.5: Test suite minimisation of schedule2, using branch coverage and small sized test suite

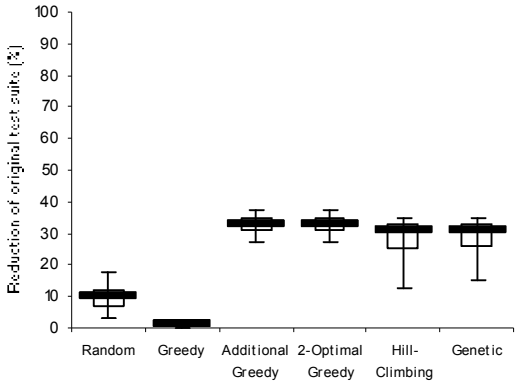


Figure 4.6: Test suite minimisation of schedule2, using statement coverage and small sized test suite

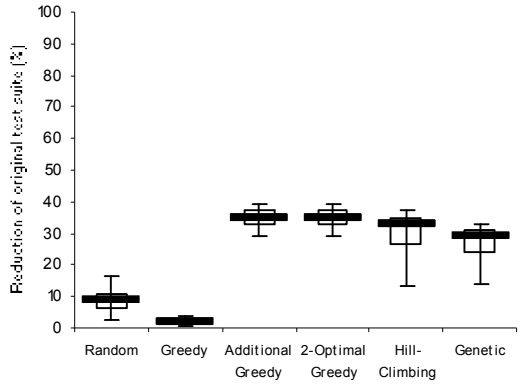


Figure 4.7: Test suite minimisation of print_tokens, using block coverage and small sized test suite

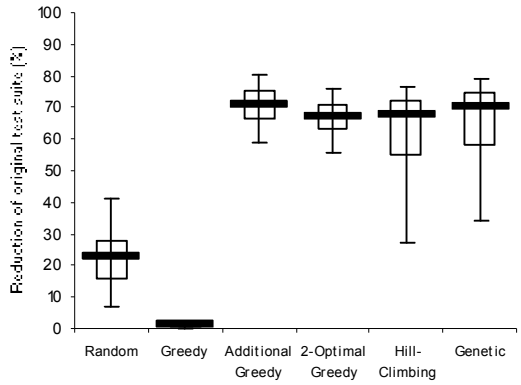


Figure 4.8: Test suite minimisation of print_tokens, using branch coverage and small sized test suite

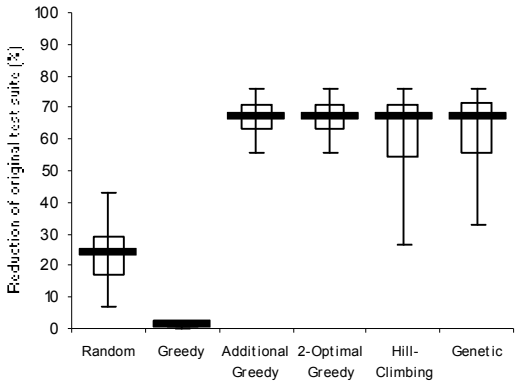


Figure 4.9: Test suite minimisation of print_tokens, using statement coverage and small sized test suite

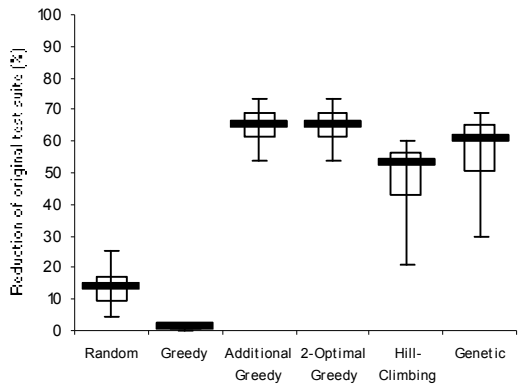


Figure 4.10: Test suite minimisation of print_tokens2, using block coverage and small sized test suite

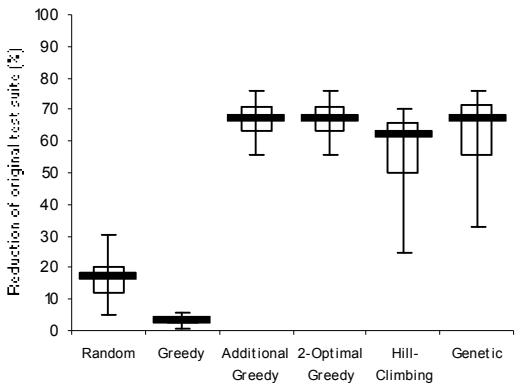


Figure 4.11: Test suite minimisation of print_tokens2, using branch coverage and small sized test suite

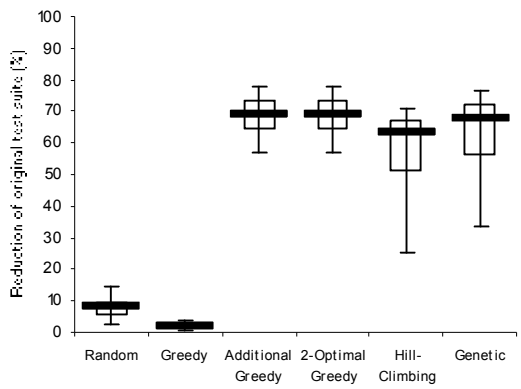


Figure 4.12: Test suite minimisation of print_tokens2, using statement coverage and small sized test suite

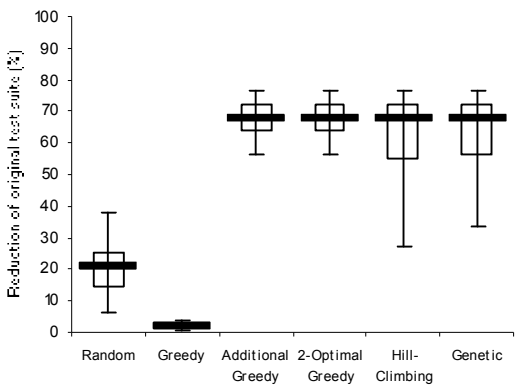


Figure 4.13: Test suite minimisation of space, using block coverage and small sized test suite

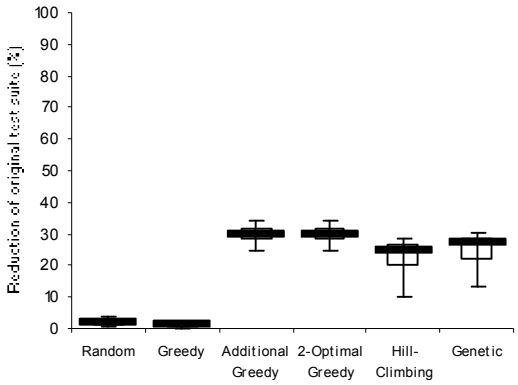


Figure 4.14: Test suite minimisation of space, using branch coverage and small sized test suite

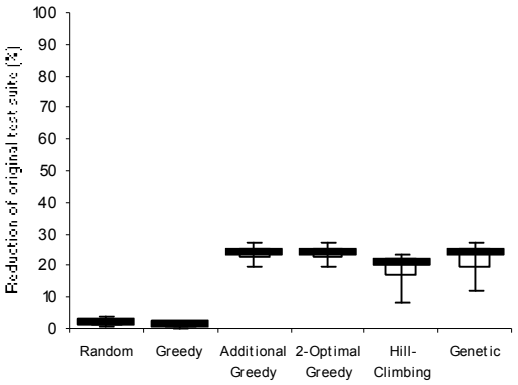
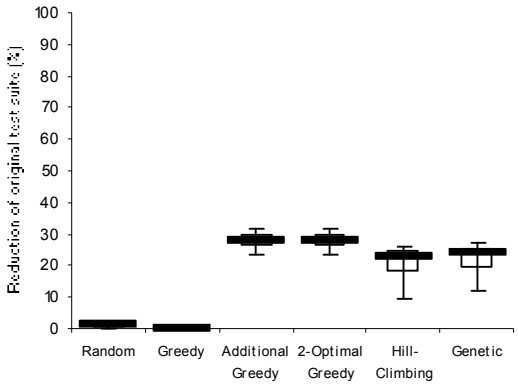


Figure 4.15: Test suite minimisation of space, using statement coverage and small sized test suite



4.2 Evaluation of results obtained using small test suites

Additional Greedy consistently performs better than the other algorithms

Across different programs and different coverage criterion, the Additional Greedy Algorithm consistently performs better than all other algorithms. This is unsurprising because the Additional Greedy Algorithm only selects test cases that cover new aspects of the program as yet uncovered. The Random Algorithm and Greedy Algorithm both select new test cases that may merely cover what has already been covered, a technique which is bound to be worse than the Additional Greedy method. Both Meta-Heuristic Algorithms begin by selecting a number of random test cases that may merely cover what has already been covered by previous test cases. Although these algorithms both attempt to remove redundant test cases, their solution is only ever going to be as good as the solution obtained by selecting only non-redundant test cases from the start – as is the Additional Greedy Algorithm method.

The reasons why the 2-Optimal Greedy Algorithm, which is very similar to the Additional Greedy Algorithm, does not always perform best are described in the next section.

Where applicable, the cheap-to-implement-and-execute Additional Greedy Algorithm should always be used to minimise small sized test suites.

2-Optimal Greedy is sometimes worse than Additional Greedy

Across different programs and different coverage criterion, the 2-Optimal Greedy Algorithm often performs worse than the Additional Greedy Algorithm. This is surprising because it appears the 2-Optimal Greedy Algorithm should overcome the weakness of the Additional Greedy Algorithm defined in the Literature Review section, so the 2-Optimal Greedy Algorithm should be better, or no worse, than the Additional Greedy Algorithm.
This phenomenon is due to the 2-Optimal Greedy Algorithm selecting test cases in multiples of two. It will select the two test cases that, when combined, cover the largest part of the program. Another two test cases are selected, and so on, until the entire program is covered. The problem comes when complete coverage has almost been achieved and only one more test case is needed to achieve it but, by the definition of the algorithm, two test cases must be selected. The second test case is not needed and is unnecessarily added to the test suite, reducing the fitness.
The Additional Greedy Algorithm selects test cases individually and, therefore, does not suffer from this phenomenon. The phenomenon is only significant when using small test suite sizes.

Greedy consistently performs worse than any other algorithm (including random)

Across different programs and different coverage criterion, the Greedy Algorithm consistently performs worse than all other algorithms. This is because the test cases that achieve high coverage cover almost identical parts of the program. The Greedy Algorithm selects these test cases because they achieve high coverage although, because they tend to all cover the same parts, total program coverage cannot be achieved until test cases are selected that cover the difficult to reach areas of the program.
By definition, there are much fewer test cases covering difficult to reach areas of the program. These test cases tend to avoid the easy to reach areas of the program in order to cover the difficult

to reach program areas and, as a result, cover far less of the program as a whole than other test cases. Therefore, the test cases covering difficult to reach areas in the program are, although important, not selected by the Greedy Algorithm until very close to the end of its cycle.
The Greedy Algorithm consistently performs worse than the Random Algorithm and, as such, does not achieve the minimum performance threshold and should not be used in practice.

Meta-Heuristic Algorithms consistently perform nearly as well as Additional Greedy and 2-Optimal Greedy

Across different programs and different coverage criterion, Hill-Climbing and Genetic Algorithm consistently perform nearly as well as, or as well as, the Additional Greedy and 2-Optimal Greedy algorithms. This suggests that in the search landscape there are many sub-optimal solutions that are very near to the optimal solution. It also suggests that there is more than one optimal solution, although one of these optimal solutions is always found by the Additional Greedy and 2-Optimal Greedy Algorithms.

Genetic Algorithm always performs at least well as, and often better than, Hill-Climbing

Across different programs and different coverage criterion, Genetic Algorithm always performs at least as well as, and often better than, the Hill-Climbing Algorithm. This is because it is easy for the Hill-Climbing Algorithm to yield sub-optimal results that are local optima in the search landscape, but not globally optimal. The Genetic Algorithm performance suggests that the solutions obtained using Hill-Climbing are indeed merely local optima and better solutions are available. It aims to avoid these local optima by keeping a diverse test suite population and introducing mutation. The Genetic Algorithm may be able to avoid the same local optima found by Hill-Climbing but, as the results show, it is not always possible.

All algorithms perform poorly for the only 'large' program

Across different coverage criterion, all algorithms perform much worse when using the only 'large' program, Space, than the rest of the 'small' programs'. This is to be expected, as there is a large amount of program space to be covered using a small test suite containing a small number of test cases. This initial test suite is already near optimal, so can only be further reduced by a small amount.

4.3 Experiments using large test suites

Figure 4.16: Test suite minimisation of schedule1, using block coverage and large sized test suite

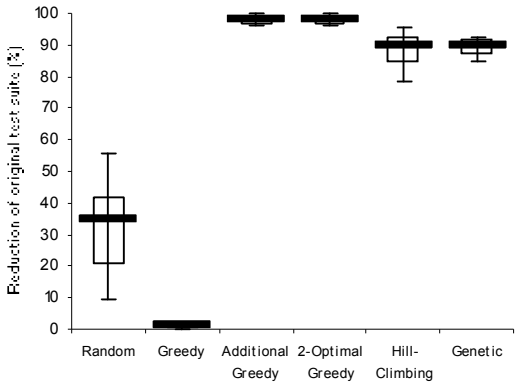


Figure 4.17: Test suite minimisation of schedule1, using branch coverage and large sized test suite

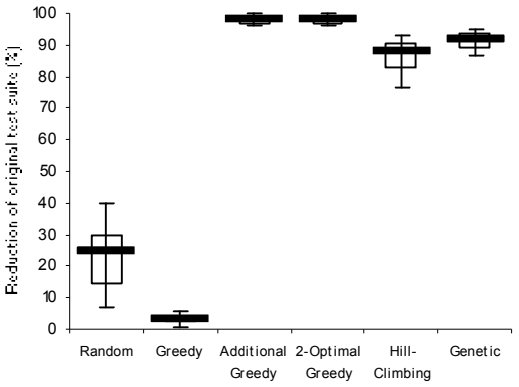


Figure 4.18: Test suite minimisation of schedule1, using statement coverage and large sized test suite

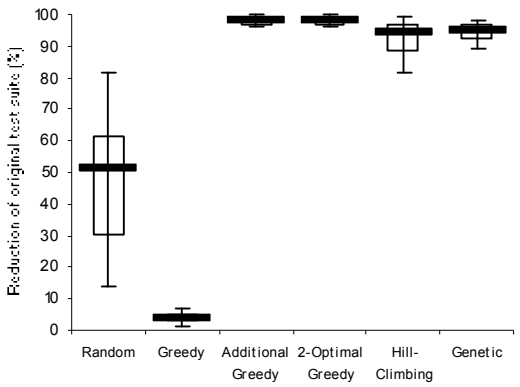


Figure 4.19: Test suite minimisation of schedule2, using block coverage and large sized test suite

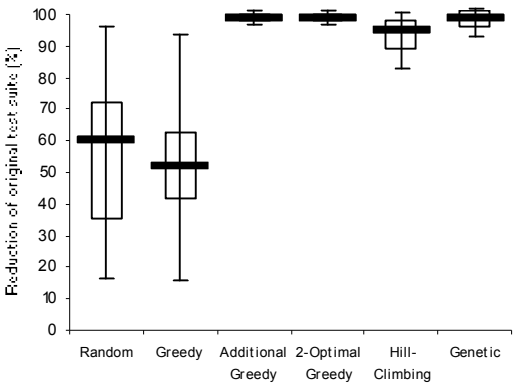


Figure 4.20: Test suite minimisation of schedule2, using branch coverage and large sized test suite

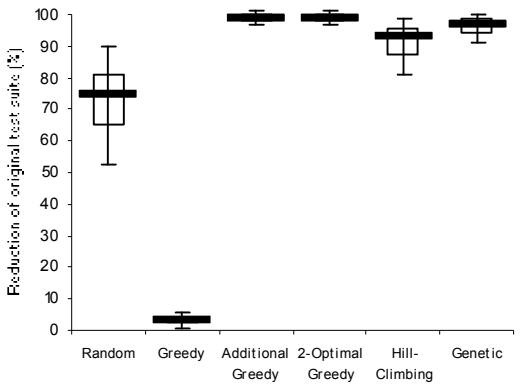


Figure 4.21: Test suite minimisation of schedule2, using statement coverage and large sized test suite

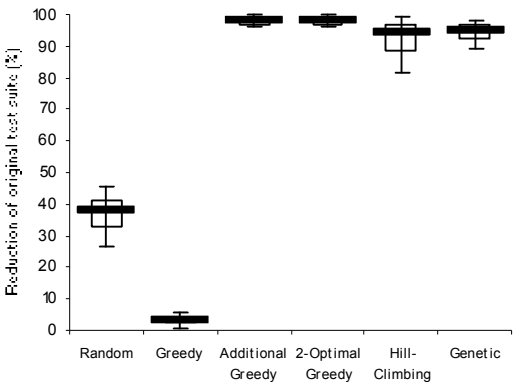


Figure 4.22: Test suite minimisation of print_tokens, using block coverage and large sized test suite

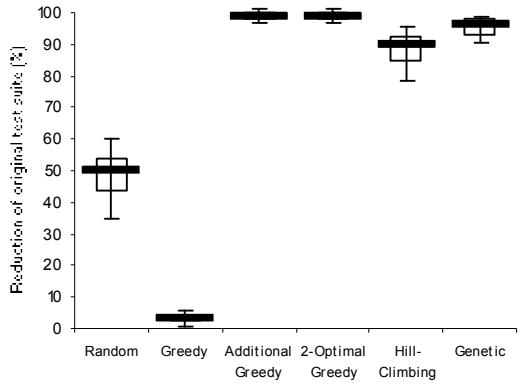


Figure 4.23: Test suite minimisation of print_tokens, using branch coverage and large sized test suite

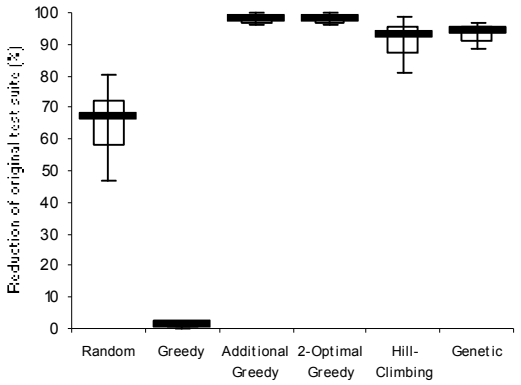


Figure 4.24: Test suite minimisation of print_tokens, using statement coverage and large sized test suite

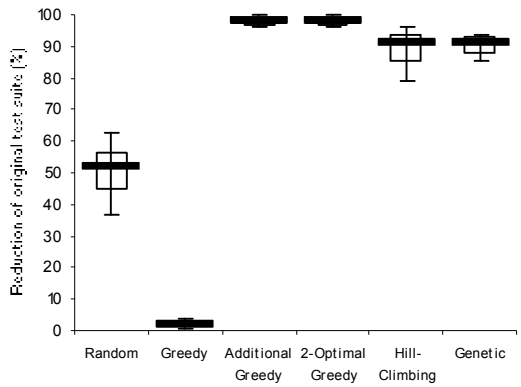


Figure 4.25: Test suite minimisation of print_tokens2, using block coverage and large sized test suite

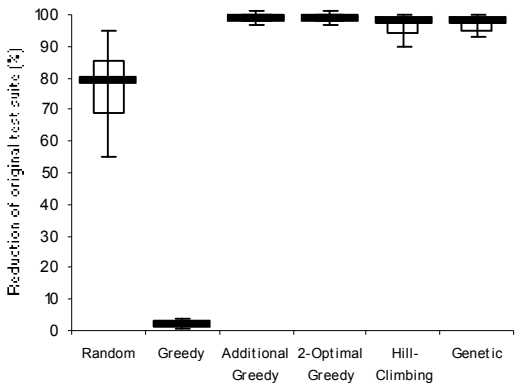


Figure 4.26: Test suite minimisation of print_tokens2, using branch coverage and large sized test suite

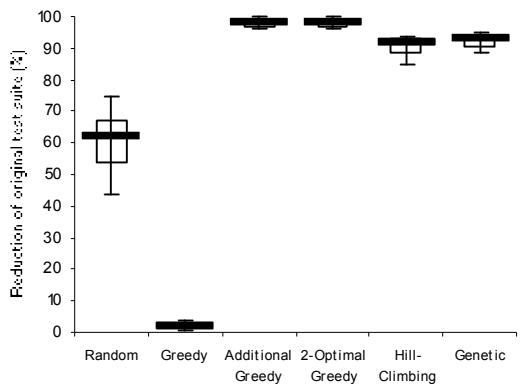


Figure 4.27: Test suite minimisation of print_tokens2, using statement coverage and large sized test suite

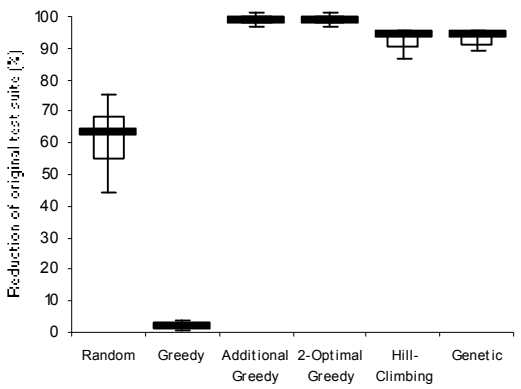


Figure 4.28: Test suite minimisation of space, using block coverage and large sized test suite

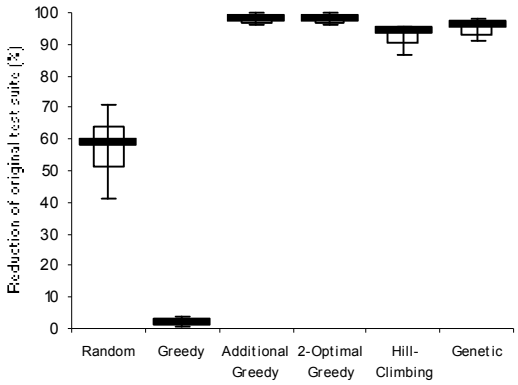


Figure 4.29: Test suite minimisation of space, using branch coverage and large sized test suite

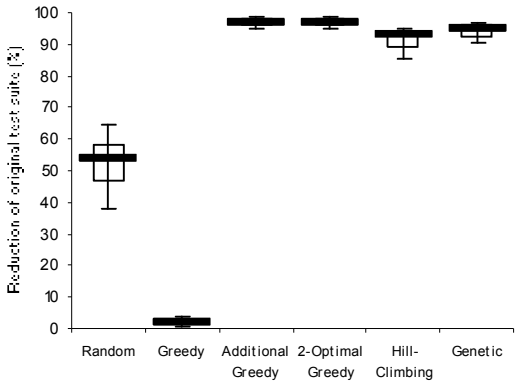
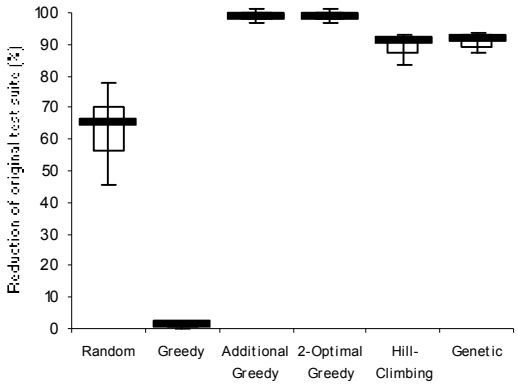


Figure 4.30: Test suite minimisation of space, using statement coverage and large sized test suite



4.4 Evaluation of results obtained using large test suites

Additional Greedy and 2-Optimal Greedy consistently perform better than the other algorithms

Across different programs and different coverage criterion, the Additional Greedy and 2-Optimal Greedy Algorithms consistently perform better than all other algorithms. This is unsurprising because the Additional Greedy and 2-Optimal Greedy Algorithms only select test cases that cover new aspects of the program as yet uncovered.

The Random Algorithm and Greedy Algorithm both select new test cases that may merely cover what has already been covered, a technique which is bound to be worse than the Additional Greedy and 2-Optimal Greedy methods. Both Meta-Heuristic Algorithms begin by selecting a number of random test cases that may merely cover what has already been covered by previous test cases. Although these algorithms both attempt to remove redundant test cases, their solution is only ever going to be as good as the solution obtained by selecting only non-redundant test cases from the start – as are the Additional Greedy and 2-Optimal Greedy methods.

The Additional Greedy Algorithm is far quicker than the 2-Optimal Greedy Algorithm at deciding which test cases to select. This gives the 2-Optimal Greedy Algorithm a higher computational cost than the Additional Greedy Algorithm. This paper focuses on the effectiveness of each algorithm, without measuring their cost, but, for applications where cost is of concern, the Additional Greedy Algorithm should be used to minimise large sized test suites.

Greedy consistently performs worse than any other algorithm (including random)

Across different programs and different coverage criterion, the Greedy Algorithm consistently performs worse than all other algorithms. This is because the test cases that achieve high coverage cover almost identical parts of the program. The Greedy Algorithm selects these test cases because they achieve high coverage although, because they tend to all cover the same parts, total program coverage cannot be achieved until test cases are selected that cover the difficult to reach areas of the program.
By definition, there are much fewer test cases covering difficult to reach areas of the program. These test cases tend to avoid the easy to reach areas of the program in order to cover the difficult to reach program areas and, as a result, cover far less of the program as a whole than other test cases. Therefore, the test cases covering difficult to reach areas in the program are, although important, not selected by the Greedy Algorithm until very close to the end of its cycle.
The Greedy Algorithm consistently performs worse than the Random Algorithm and, as such, does not achieve the minimum performance threshold and should not be used in practice.

2-Optimal Greedy and Additional Greedy consistently perform as well as each other

Across different programs and different coverage criterion, the Additional Greedy and 2-Optimal Greedy Algorithms consistently perform as well as each other. This is because they both utilise very similar methods for test suite minimisation. Additional Greedy selects the test case that covers the largest part of the program, then selects the test case that covers that largest part of the program as yet still uncovered, and so on until full coverage is achieved. 2-Optimal Greedy selects the two test cases that, when combined, cover the largest part of the program, then selects the two test cases that, when combined, cover that largest part of the program as yet still

uncovered, and so on until full coverage is achieved. Although there may be very small differences in their performance, these are not significant when minimising a large sized original test suite.

The Meta-Heuristic Algorithms consistently perform nearly as well as Additional Greedy and 2-Optimal Greedy

Across different programs and different coverage criterion, Hill-Climbing and Genetic Algorithm consistently perform nearly as well as, or as well as, the Additional Greedy and 2-Optimal Greedy algorithms. This suggests that in the search landscape there are many sub-optimal solutions that are very near to the optimal solution. It also suggests that there is more than one optimal solution, although one of these optimal solutions is always found by the Additional Greedy and 2-Optimal Greedy Algorithms.

Genetic Algorithm always performs at least well as, and often better than, Hill-Climbing

Across different programs and different coverage criterion, Genetic Algorithm always performs at least as well as, and often better than, the Hill-Climbing Algorithm. This is because it is easy for the Hill-Climbing Algorithm to yield sub-optimal results that are local optima in the search landscape, but not globally optimal. The Genetic Algorithm performance suggests that the solutions obtained using Hill-Climbing are indeed merely local optima and better solutions are available. It aims to avoid these local optima by keeping a diverse test suite population and introducing mutation. The Genetic Algorithm may be able to avoid the same local optima found by Hill-Climbing but, as the results show, it is not always possible.

The results obtained from experiments with large test suites are consistent with those obtained for the small test suites. This provides additional evidence to support the claim that the results capture properties of the test suite minimisation problem, rather than some artifact of the choice of test suite or the overall size of the test suites being tested.

4.5 Evaluation of results obtained over all experiments

Additional Greedy performs the best

Across different programs and different coverage criterion, the Additional Greedy Algorithm consistently performed better than all other algorithms when applied to the *small* sized test suites. Under the same situation, both the Additional Greedy and 2-Optimal Greedy Algorithms consistently performed better than all other algorithms when applied to the *large* sized test suites.

These findings are unsurprising because the Additional Greedy and 2-Optimal Greedy Algorithms only select test cases that cover new aspects of the program as yet uncovered. The Random Algorithm and Greedy Algorithm both select new test cases that may merely cover what has already been covered, a technique which is bound to be worse than the Additional Greedy and 2-Optimal Greedy methods. Both Meta-Heuristic Algorithms begin by selecting a number of random test cases that may merely cover what has already been covered by previous test cases. Although these algorithms both attempt to remove redundant test cases, their solution is only ever going to be as good as the solution obtained by selecting only non-redundant test cases from the start – as are the Additional Greedy and 2-Optimal Greedy methods.

Situations occur during small sized test suite minimisation in which the 2-Optimal Greedy Algorithm performs worse than the Additional Greedy Algorithm. This is because the 2-Optimal Greedy Algorithm selects test cases in multiples of two. It will select the two test cases that, when combined, cover the largest part of the program. Another two test cases are selected, and so on, until the entire program is covered. The problem comes when complete coverage has almost been achieved and only one more test case is needed to achieve it but, by the definition of the algorithm, two test cases must be selected. The second test case is not needed and is unnecessarily added to the test suite, reducing the fitness. The Additional Greedy Algorithm selects test cases individually and, therefore, does not suffer from the same phenomenon as the 2-Optimal Greedy Algorithm when applied to small size test suites. Therefore, the Additional Greedy Algorithm should be used to minimise small sized test suites.

During large sized test suite minimisation, the difference in time complexities between the Additional Greedy and 2-Optimal Greedy algorithms become very apparent. Additional Greedy Algorithm is quicker than the 2-Optimal Greedy Algorithm at deciding which test cases to select. This gives the 2-Optimal Greedy Algorithm a higher computational cost than the Additional Greedy Algorithm. This paper focuses on the effectiveness of each algorithm, without measuring their cost, but, for applications where cost is of concern, the Additional Greedy Algorithm should be used to minimise large sized test suites.

Greedy performs the worst

Across different programs and different coverage criterion, the Greedy Algorithm consistently performs worse than all other algorithms when applied to both sized test suites. This is because the test cases that achieve high coverage cover almost identical parts of the program. The Greedy Algorithm selects these test cases because they achieve high coverage although, because they tend to all cover the same parts, total program coverage cannot be achieved until test cases are selected that cover the difficult to reach areas of the program.

By definition, there are much fewer test cases covering difficult to reach areas of the program. These test cases tend to avoid the easy to reach areas of the program in order to cover the difficult to reach program areas and, as a result, cover far less of the program as a whole than other test cases. Therefore, the test cases covering difficult to reach areas in the program are, although important, not selected by the Greedy Algorithm until very close to the end of its cycle.
The Greedy Algorithm consistently performs worse than the Random Algorithm and, as such, does not achieve the minimum performance threshold and should not be used in practice.

2-Optimal Greedy performs as well as, but never better than, Additional Greedy

Across different programs and different coverage criterion, the 2-Optimal Greedy Algorithm can perform as well as, but never better than, the Additional Greedy Algorithms when applied to both sized test suites.
The 2-Optimal Greedy Algorithm should overcome the weakness of the Additional Greedy Algorithm described in the Literature Review section. However, it appears that this weakness is not exposed enough during the experiments to significantly alter the results and make the 2-Optimal Greedy Algorithm perform better than the Additional Greedy Algorithm.

All algorithms perform better on large test suites than small test suites

Across different programs and different coverage criterion, all algorithms performed better when applied to large test suites than small test suites. This is unsurprising because large test suites contain a large amount of redundant test cases. Redundant test cases do not cover any parts of the program that are not covered by a test case that covers more of the program in general. There also may be a large amount of duplicate test cases – those test cases that have different inputs but cover exactly the same parts of the program. There are far more of these redundant and duplicate test cases found in large test suites than small test suites. The algorithms will perform better because there is more of these test cases to be removed.

Both Meta-Heuristic Algorithms perform significantly better on large test suites than small
Across different programs and different coverage criterion, the Hill-Climbing Algorithm and Genetic Algorithm both perform significantly better on large test suites than small. In addition to the reasoning described in the previous section, there are other reasons why this observation is true. Both Meta-Heuristic Algorithms use a search landscape in which to converse in order to achieve the best possible solution. By having more test cases in the original test suite, the population diversity is increased in the search landscape. This means that, using the same number of test cases, there are many more different combinations of test cases that will achieve complete program coverage. With more globally optimal combinations of test cases, it is less likely that the Meta-Heuristic Algorithms will yield sub-optimal results that are merely local optimal, but not globally optimal.

There is no significant difference between results obtained using different coverage metrics (block/branch/statement)

Across different programs, test suite sizes, and using different algorithms, there was no significant difference between the results obtained using block, branch and statement coverage as the program coverage metric. This provides evidence for the robustness of these results. That is,

the results suggest that the nature of the search problem denoted by regression test suite minimisation has been captured, rather than merely some aspect of a particular choice of program coverage criterion.

The effect of program size on Algorithm Performance

Considering the differences in results for Hill-Climbing instantiations, both the size of the program and the size of the test suite appear to influence the difference in results. However, larger programs typically have larger test suites, since more test cases are required in order to fully test them. For example the program Space is about ten times larger than the other programs, and the corresponding test suite is about ten times larger than the average test suite for the other programs. The size of the program does not directly affect the complexity of test suite minimisation, whereas the size of the test suite does, since it determines the size of the search space. However, it takes longer to determine the fitness of a test case for a larger program, since more coverage information has to be examined. This suggests that the deterministic search techniques (the Greedy Algorithms) are scalable to larger programs, subject to the associated increase in test suite sizes, since the fitness metrics are computed only once. For meta-heuristic algorithms, the cost-benefits should be considered first before they are applied to large programs.

Fitness Landscapes

The results obtained using Hill-Climbing, both for small and large sized test suites, reveal the layout of the search landscape and the factors that govern the attributes of the fitness landscape.

For the problem of test suite minimisation, the search space is the combination of test cases included the minimised test suite. The global optimum is a minimised test suite for which no other test suite contains fewer test cases without suffering a subsequent reduction in coverage. A local optimum is a minimised test suite containing a population of test cases that is no greater than those of its neighbours, but is not necessarily the global optimum. For Hill-Climbing, and other Meta-Heuristic Algorithms, if the fitness of the final minimised test suite is less than the global optimum then this order is deemed ‘suboptimal’.

The fact that the Hill-Climbing Algorithm produced so many results that were as good as, or almost as good as, the results obtained using the best performing algorithm (Additional Greedy) suggests that the search landscape contains many globally optimum solutions and many locally optimum solutions that are nearly as good as the global optimum.

With an increase in program size and test suite size there is a significant increase in the vertical lengths of the boxplots of the Hill-Climbing Algorithm compared to other algorithms. This suggests that the search landscape becomes much more complicated with the increase in program size and test suite size.

5. Future Work

The purpose of minimising a test set is to reduce the associated cost. This cost, computed as the sum of the costs of its test cases, can be measured in several ways. One measurement considers the computation time needed to execute each test case. Another measurement considers the tester time spent on constructing and analysing these test cases. In this paper, the cost of a test suite is the simply the number of test cases in the suite. Work could be done into associating different costs with each test case in the test pool, adding another dimension to the test suite minimisation problem.

The criteria studied were based on code coverage, which is different from criteria based on fault detection. The application of meta-heuristic search algorithms to fault detection based test suite minimisation problems could yield different results.

The Genetic Algorithm has many different parameters that often need to be tuned in order to optimise the algorithm. Selection methods include Roulette Wheel, Tournament and Truncation. Some Crossover methods are One Point Crossover, Two Point Crossover, Cut and Splice, Uniform Crossover and Half Uniform Crossover. There are also variations in the way mutation is carried out to an individual.

There are many different combinations of selection method and probability, crossover method and probability, and mutation method and probability. This, in itself, could form the basis of some future study in order to determine an optimal Genetic Algorithm for test suite minimisation for regression testing.

6. Conclusions

This paper described six algorithms for the test suite minimisation problem in regression testing. It presented the results of an empirical study that investigated their relative effectiveness.

Several observations have been made by analysing the data:

There are definite differences between the five algorithms and, with the increase in the size of the original test suite, these differences may become more apparent.

Across different programs and different coverage criterion, the Additional Greedy Algorithm consistently performed better than, or at least as well as, all other algorithms when applied to both small and large sized test suites. This is because the Additional Greedy Algorithm is one of the only algorithms that will only select a test case if it covers parts of the program not currently being covered.

Under the same conditions, the Greedy Algorithm consistently performed worse than all other algorithms. This is because the test cases selected all tend to cover the same, or very similar, parts of the program.

The 2-Optimal Greedy Algorithm performed as well as, but never better than, the Additional Greedy Algorithm during experiments. In terms of effectiveness, there is never a significant difference between their performances. This suggests that, where applicable, the cheaper-to-implement-and-execute Additional Greedy Algorithm should be used.

For different programs and different coverage criterion, all algorithms performed better when applied to large test suites than small test suites. This is unsurprising because there are far more redundant and duplicate test cases found in large test suites than small test suites. These are more likely to be deleted, hence increasing the algorithm's effectiveness.

The choice of coverage criterion has no effect on the efficiency of minimisation techniques. The size of the test suite determines the size of the search space, therefore affecting the complexity of the test suite minimisation problem. The size of the program does not have a direct effect, but increases the computational effort of computing fitness values.

Empirical studies regarding the performance of meta-heuristic algorithms led to several conclusions. The results produced by the Hill-Climbing Algorithm indicate that the nature of the fitness of the landscape is multi-modal. The results obtained using the Genetic Algorithm show that it never performs better than the Additional Greedy and 2-Optimal Greedy Algorithms, but in most cases the difference between their performances is not significant.

However, in order to find the *absolute* minimum test suite solution, it is necessary to know every possible combination of test cases for a minimised test suite. For such cases, the Greedy Algorithms are unlikely to be appropriate because the Greedy technique must determine which *part* of the original solution to add next. Instead, algorithms will be required that can be guided by a fitness function that takes into account the entire ordering. Genetic Algorithms may be suitable for this. They are far more transferable whilst consistently achieving similar performance to the Greedy Algorithms.

7. References

[1] B. Beizer. *SoftwareTesting Techniques*. Van Nostrand Reinhold, New York, NY, 1990.

[2] Larry J. Eshelman, *The CHC Adaptive Search Algorithm: How to Have Safe Search When Engaging in Nontraditional Genetic Recombination*, in Gregory J. E. Rawlins editor, Proceedings of the First Workshop on Foundations of Genetic Algorithms. pages 265-283. Morgan Kaufmann, 1991.

[3] Fogel, David B. (1998) *Evolutionary Computation: The Fossil Record*, IEEE Press, New York

[4] Goldberg, David E (2002), *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*, Addison-Wesley, Reading, MA.

[5] J. H. Holland. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. University of Michigan Press, Ann Arbor, Michigan. 1975.

[6] Koza, John (1992), *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press.

[7] H. K. N. Leung and L. White. Insights Into Regression Testing. In *Proceedings of the Conference on Software Maintenance*, pages 60-69, October 1989 (20).

[8] Z. Li and M. Harman. Search Algorithms for Regression Test Case Prioritisation. April 2006.

[9] Michalewicz, Zbigniew (1999), *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag.

[10] K. Onoma, W-T Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 31(6):676-686, June 1988. (24).

[11] G. Reinelt. *TSPLIB – A traveling salesman problem library*. ORSA Journal on Computing, 3(4):376-384, 1991.

[12] G. Rothermel, R. Untch, C. Chu, M. Harrold (2003), *Prioritizing Test Cases For Regression Testing*, IEEE Transactions on Software Engineering, 27(10):929-948, Oct. 2001

[13] Schmitt, Lothar M (2001), *Theory of Genetic Algorithms*, Theoretical Computer Science (259), pp. 1-61

[14] S. S. Skiena. *The algorithm design manual*. Springer-Verlag, New York, NY, USA, 1998.

[15] A. Srivastava and J. Thiagarajan. *Effectively prioritising tests in development environment*, ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, pages 97-106, New York, NY, USA, 2002. ACM Press.

[16] Vose, Michael D (1999), *The Simple Genetic Algorithm: Foundations and Theory*, MIT Press, Cambridge, MA.

[17] Wong, W. E. and Horgan, J. R. and London, S. and Agrawal, H. *A study of effective regression testing in practice*, Proceedings of the Eighth International Symposium on Software Reliability Engineering, pages 230-238. IEEE Computer Society, Nov. 1997

[18] Wong, W. E. and Horgan, J. R. and London, S. and Mathur, A. P. (1995), *Effect of Test Set Minimisation on Fault Detection Effectiveness*

Appendix

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctime>

#define SMALL         50
#define LARGE         100
#define SCH1          "schedule1"
#define SCH2          "schedule2"
#define PRI1          "printtokens1"
#define PRI2          "printtokens2"
#define SPAC          "space"
#define BLO                  "block_cov"
#define STA                  "statement_cov"
#define BRA                  "decision_cov"
#define NAMESIZE      30

void quickSort(int numbers[], int positions[], int array_size);
void q_sort(int numbers[], int positions[], int left, int right);
int random(char **testSuiteCoverage, char **testSuite, int *solution, int noLines, int
testSuitePop, int totalNoCovered);
void greedy(char **testSuiteCoverage, char **testSuite, int *solution, int noLines, int
testSuitePop, int totalNoCovered);
void additionalGreedy(char **testSuiteCoverage, char **testSuite, int *solution, int
noLines, int testSuitePop, int totalNoCovered);
void twoOptimalGreedy(char **testSuiteCoverage, char **testSuite, int *solution, int
noLines, int testSuitePop, int totalNoCovered);
void hillClimbing(char **testSuiteCoverage, char **testSuite, int *solution, int noLines,
int testSuitePop, int totalNoCovered);
void genetic(char **testSuiteCoverage, char **testSuite, int *solution, int noLines, int
testSuitePop, int totalNoCovered, int populationNo);
void modifiedGreedy(char **testSuiteCoverage, char **testSuite, int *solution, int
noLines, int testSuitePop, int totalNoCovered);
void printResults(char name[], char **testSuiteCoverage, char **testSuite, int
solution[], int solutionPop, int totalNoCovered, int noLinesCovered, int testSuitePop);


/******************************** MAIN *******************************/

int main()
{
       srand((unsigned)time(0));  //seeds the random number generator

       bool found=false;  //flag as to when a test case has been found in the universe
array
       int pos = 0;  //position in the test case name/input
       int progNo;  //input program number
       int covNo;  //type of coverage number
       int suiteType;  //type of test suite (large or small)
       int testSuiteNo;  //test suite number
       char *progName;  //input program name
       char *covName;  //coverage type name
       char *suiteName;  //suite type name
       char universeFile[35] = "./";
       char matrixFile[35] = "./";
       char suiteFile[35] = "./";
       int populationNo = 0;  //starting population number for the genetic algorithm
       int inChar;
       FILE *inFile;

       printf("1. Schedule\n");
       printf("2. Schedule 2\n");
       printf("3. Print Tokens\n");
       printf("4. Print Tokens 2\n");
       printf("5. Space\n");
       printf("Select input program (1-5): ");
       scanf("%d", &progNo);
       switch(progNo)
       {
              case 1:
                     progName = SCH1;
```

```
                populationNo = SMALL;
                break;
        case 2:
                progName = SCH2;
                populationNo = SMALL;
                break;
        case 3:
                progName = PRI1;
                populationNo = SMALL;
                break;
        case 4:
                progName = PRI2;
                populationNo = SMALL;
                break;
        case 5:
                progName = SPAC;
                populationNo = LARGE;
                break;
        default:
                progName = SCH1;
                populationNo = SMALL;
                printf("Using default program - %s\n", SCH1);
}
strcat(universeFile, progName);
strcat(universeFile, "/universe");
printf("\nUsing the universe file: %s\n", universeFile);

//Open the universe file and store the test case names in an array
printf("Opening universe file: %s\n", universeFile);
inFile = fopen(universeFile, "r");
if(!inFile)
{
printf("Error: Could not open universe file\n");
exit(EXIT_FAILURE);
}
printf("%s selected\n", universeFile);

int universePop = 1;
while(inChar != EOF)  //calculate number of test cases in the universe file
{
        inChar = fgetc(inFile);
        if(inChar == '\n')
                universePop++;
}
printf("universePop: %d\n", universePop);

//char is a 2D array to hold the name of each test case
char** universe = NULL;  //declare the 2D array as a pointer to a pointer
universe = new char*[universePop];  //allocate the main array
for (int i=0; i<universePop; i++)
        universe[i] = new char[NAMESIZE];  //allocate each member of the main
array

for(int a=0; a<universePop; a++)
{
        for(int b=0; b<NAMESIZE; b++)
                universe[a][b] = ' ';
}

int y=0;
int x=0;

inFile = fopen(universeFile, "r");
inChar = 0;
while(inChar != EOF)
{
        inChar = fgetc(inFile);
        if(inChar != '\n')
        {
                universe[y][x] = inChar;
                x++;
```

```
        }
        else
        {
                y++;
                x=0;
        }
}
printf("Universe data copied\n");

printf("\n1. Block coverage\n");
printf("2. Statement coverage\n");
printf("3. Branch coverage\n");
printf("Select type of coverage to test (1-3): ");
scanf("%d", &covNo);
switch(covNo)
{
        case 1:
                covName = BLO;
                break;
        case 2:
                covName = STA;
                break;
        case 3:
                covName = BRA;
                break;
        default:
                covName = BLO;
                printf("Using default coverage method - %s\n", BLO);
}

strcat(matrixFile, progName);
strcat(matrixFile, "/");
strcat(matrixFile, covName);
strcat(matrixFile, "/matrix.out");
printf("\nUsing coverage (matrix) file: %s\n", matrixFile);

//Open the matrix file, retrieve coverage data and store in a 2 dimensional array
inFile = fopen(matrixFile, "r");
printf("Opening %s file...\n", matrixFile);
if(!inFile)
{
printf("Error: Could not open matrix file\n");
exit(EXIT_FAILURE);
}
printf("%s selected\n", matrixFile);

int noLines=0;
int noTests=0;

inChar = 0;
while(inChar != EOF)
{
        noLines=0;
        while(inChar != '\n')
        {
                inChar = fgetc(inFile);
                noLines++;  //increment line counter
        }
        noTests++;  //increment test case counter
        inChar = fgetc(inFile);
}

printf("noLines= %d\n", noLines);
printf("noTests= %d\n", noTests);

//char is a 2D array to hold coverage information for each test case
char** coverage = NULL;  //declare the 2D array as a pointer to a pointer
coverage = new char*[noTests];  //allocate the main array
for (i=0; i<noTests; i++)
        coverage[i] = new char[noLines];  //allocate each member of the main array
```

```
        inFile = fopen(matrixFile, "r");
        int counter = 0;
    for(y=0; y<noTests*2; y++)  //fill the 2D array with data from the input text file
        {
                for(x=0; x<noLines; x++)
                {
                        inChar = fgetc(inFile);
                        if(inChar == '\n')  //during a line break, to avoid leaving a blank
mini array, the
                        {                   //counter is incremented.  this number is then
subtracted from
                                counter++;      //the main array number when entering data.
                                break;
                        }
                        else
                                coverage[y-counter][x] = inChar;
                }
        }

        printf("\nMatrix data retrieved.\n");

        printf("1. Large test suites\n");
        printf("2. Small test suites\n");
        printf("Select test suite size (1 or 2): ");
        scanf("%d", &suiteType);
        switch(suiteType)
        {
                case 1:
                        suiteName = "testplans-bigcov";
                        break;
                case 2:
                        suiteName = "testplans-cov";
                        break;
                default:
                        suiteName = "testplans-bigcov";
                        printf("Using default test suite size - %s\n", LARGE);
        }

        strcat(suiteFile, progName);
        strcat(suiteFile, "/");
        strcat(suiteFile, suiteName);
        strcat(suiteFile, "/suite");
        printf("\nUsing suite: %s\n", suiteFile);

        printf("Enter test suite number to use (1-999): ");
        scanf("%d", &testSuiteNo);

        char ext[33];
        itoa (testSuiteNo,ext,10);
        strcat(suiteFile, ext);
        printf("suiteFile: %s\n", suiteFile);

        inFile = fopen(suiteFile, "r");
        if(!inFile)
        {
        printf("Error: Could not open test suite file %d\n", i);
                exit(EXIT_FAILURE);
        }

        int testSuitePop = 1;
        while(inChar != EOF)  //calculate number of test cases in the test suite
        {
                inChar = fgetc(inFile);
                if(inChar == '\n')
                        testSuitePop++;
        }
        printf("testSuitePop is: %d\n", testSuitePop);

        char** testSuite = NULL;  //2d array to hold a test suite
        testSuite = new char*[testSuitePop];  //allocate the main array
        for (i=0; i<testSuitePop; i++)
```

```
                testSuite[i] = new char[NAMESIZE];  //allocate each member of the main
array
        for(a=0; a<testSuitePop; a++)
        {
                for(int b=0; b<NAMESIZE; b++)
                        testSuite[a][b] = ' ';
        }

        char** testSuiteCoverage = NULL;  //2d array to hold coverage information for a
test suite
        testSuiteCoverage = new char*[testSuitePop];  //allocate the main array
        for (i=0; i<testSuitePop; i++)
                testSuiteCoverage[i] = new char[noLines];  //allocate each member of the
main array
        for(a=0; a<testSuitePop; a++)
        {
                for(int b=0; b<noLines; b++)
                        testSuiteCoverage[a][b] = '0';
        }

        x=0;
        y=0;

        inFile = fopen(suiteFile, "r");
        inChar = 0;
        //copy the info in the file to a 2d array
        while(inChar != EOF)
        {
                inChar = fgetc(inFile);
                if(inChar != '\n')
                {
                        testSuite[y][x] = inChar;
                        x++;
                }
                else
                {
                        y++;
                        x=0;
                }
        }

        printf("\nTest case values/names:\n");

        testSuitePop--;  //there is an extra liine break at the end of the test suite
files - so reduce by 1

        for(a=0; a<testSuitePop; a++)
        {
                for(int b=0; b<NAMESIZE; b++)
                        printf("%c", testSuite[a][b]);
                printf("\n");
        }

        printf("\nTest case coverage:\n");

        for(a=0; a<testSuitePop; a++)  //match the entries in this array to the entries in
the universe array
        {
                found = false;
                pos=0;
                while(found==false)
                {
                        if(strcmp(testSuite[a],universe[pos]) == 0)  //if the two test
cases are identical
                        {
                                for(int c=0; c<noLines; c++)
                                        testSuiteCoverage[a][c] = coverage[pos][c];
                                for(c=0; c<noLines; c++)
                                        printf("%c", testSuiteCoverage[a][c]);
                                printf("\n");
                                found=true;
```

```
                        }
                        pos++;
                }
        }

        //Count how many lines are covered by the test suite
        int totalNoCovered = 0;  //total number of lines covered by the full test suite
        bool *covered = NULL;  //the lines covered by the test suite
        covered = new bool[noLines];
        for(i=0; i<noLines; i++)
                covered[i] = false;

        for(y=0; y<testSuitePop; y++)
        {
                for(x=0; x<noLines; x++)
                {
                        if(testSuiteCoverage[y][x] == '1')
                                covered[x] = true;
                }
        }

        for(i=0; i<noLines; i++)
        {
                if(covered[i] == true)
                        totalNoCovered++;
        }

        printf("\nNumber of test cases in the entire pool: %d\n", noTests);
        printf("Number of test cases in the starting test suite: %d\n", testSuitePop);
        printf("Number of blocks/branches/lines in the program: %d\n", noLines);
        printf("Number of blocks/branches/lines covered by the starting test suite: %d\n",
totalNoCovered);

        //Calculate the percentage coverage
        float percentCovered = 0;
        float floatNoCovered = (float)totalNoCovered;
        float floatNoLines = (float)noLines;
        percentCovered = (floatNoCovered/floatNoLines)*100;
        printf("Starting percentage coverage: %.2f%%\n", percentCovered);

        int* solution = NULL;
        solution = new int[testSuitePop];
        for(i=0; i<testSuitePop; i++)
                solution[i] = -1;

        int algNo;
        printf("\n1. Random algorithm\n");
        printf("2. Greedy algorithm\n");
        printf("3. Additional Greedy algorithm\n");
        printf("4. 2-Optimal Greedy algorithm\n");
        printf("5. Modified Greedy algorithm\n");
        printf("6. Hill-Climbing algorithm\n");;
        printf("7. Genetic algorithm\n");
        printf("Select algorithm number: ");
        scanf("%d", &algNo);

        switch(algNo)
        {
                case 1:
                        printf("\nRandom algorithm selected\n");
                        random(testSuiteCoverage, testSuite, solution, noLines,
testSuitePop, totalNoCovered);
                        break;
                case 2:
                        printf("\nGreedy algorithm selected\n");
                        greedy(testSuiteCoverage, testSuite, solution, noLines,
testSuitePop, totalNoCovered);
                        break;
                case 3:
                        printf("\nAdditional Greedy algorithm selected\n");
```

```
                        additionalGreedy(testSuiteCoverage, testSuite, solution, noLines,
testSuitePop, totalNoCovered);
                        break;
                case 4:
                        printf("\n2-Optimal Greedy algorithm selected\n");
                        twoOptimalGreedy(testSuiteCoverage, testSuite, solution, noLines,
testSuitePop, totalNoCovered);
                        break;
                case 5:
                        printf("\nModified Greedy algorithm selected\n");
                        modifiedGreedy(testSuiteCoverage, testSuite, solution, noLines,
testSuitePop, totalNoCovered);
                        break;
                case 6:
                        printf("\nHill-Climbing algorithm selected\n");
                        hillClimbing(testSuiteCoverage, testSuite, solution, noLines,
testSuitePop, totalNoCovered);
                        break;
                case 7:
                        printf("\nGenetic algorithm selected\n");
                        genetic(testSuiteCoverage, testSuite, solution, noLines,
testSuitePop, totalNoCovered, populationNo);
                        break;
                default:
                        printf("\nDefault algorithm (Random) selected\n");
                        random(testSuiteCoverage, testSuite, solution, noLines,
testSuitePop, totalNoCovered);
        }
        return 0;
}


/****************************** QUICKSORT ALGORITHM ******************************/

void quickSort(int numbers[], int positions[], int array_size)
{
  q_sort(numbers, positions, 0, array_size - 1);
}

void q_sort(int numbers[], int positions[], int left, int right)
{
  int pivot, pivotPos, l_hold, r_hold;

  l_hold = left;
  r_hold = right;
  pivot = numbers[left];
  pivotPos = positions[left];
  while (left < right)
  {
    while ((numbers[right] >= pivot) && (left < right))
      right--;
    if (left != right)
    {
      numbers[left] = numbers[right];
          positions[left] = positions[right];
      left++;
    }
    while ((numbers[left] <= pivot) && (left < right))
      left++;
    if (left != right)
    {
      numbers[right] = numbers[left];
          positions[right] = positions[left];
      right--;
    }
  }
  numbers[left] = pivot;
  positions[left] = pivotPos;
  pivot = left;
  left = l_hold;
  right = r_hold;
```

```
  if (left < pivot)
    q_sort(numbers, positions, left, pivot-1);
  if (right > pivot)
    q_sort(numbers, positions, pivot+1, right);
}


/****************************** RANDOM ALGORITHM ******************************/

int random(char **testSuiteCoverage, char **testSuite, int *solution, int noLines, int
testSuitePop, int totalNoCovered)
{
	printf("\nRANDOM ALGORITHM\n");
	int randomInt = 0;  //random integer
	int noLinesCovered = 0;  //number of lines not covered
	int index = 0;  //index of the 'solution' array

	bool* lineCovered = NULL;  //array to keep track of which lines have been covered
by test cases
	lineCovered = new bool[noLines];

	for(int i=0; i<noLines; i++)
		lineCovered[i] = false;

	for(i=0; i<testSuitePop; i++)
		solution[i] = -1;

	while(noLinesCovered<totalNoCovered)
	{
		noLinesCovered = 0;
		randomInt = rand() % testSuitePop;
		printf("Test case: %d  ", randomInt);

		for(int x=0; x<noLines; x++)  //sets elements of the linesCovered array to
be true if covered by the current test case
		{
			if(testSuiteCoverage[randomInt][x] == '1')
				lineCovered[x] = true;
		}

		for(x=0; x<noLines; x++)  //increments noLinesCovered for each element in
the linesCovered array that is true
		{
			if(lineCovered[x] == true)
				noLinesCovered++;
		}

		printf("Number of lines covered: %d", noLinesCovered);

		solution[index] = randomInt;  //remember each test case selected

		for(i=index-1; i>=0; i--)
		{
			if(solution[i]==randomInt)  //if the test case has already been
selected then remove it
			{
				printf(" (This test case has already been selected so will
not be included)");
				solution[index] = -1;
				index--;
			}
		}
		printf("\n");
		index++;
	}
	printf("Solution is: ");
	i=0;
	while(solution[i] != -1 && i<testSuitePop)
	{
		printf("%d ", solution[i]);
		i++;
```

```
	}
	printResults("Random", testSuiteCoverage, testSuite, solution, index,
totalNoCovered, noLinesCovered, testSuitePop);
	return index;
}



/****************************** GREEDY ALGORITHM ******************************/

void greedy(char **testSuiteCoverage, char **testSuite, int *solution, int noLines, int
testSuitePop, int totalNoCovered)
{
	printf("\nGREEDY ALGORITHM\n");
	int solutionPop = 0;  //population of solution array
	int noLinesCovered = 0;  //number of lines not covered
	int pos = testSuitePop-1;  //keeps track of the current test case in the testCase
array
	int* testCase = NULL;  //array to hold the position of each test case number
during sorting
	testCase = new int[testSuitePop];
	for(int y=0; y<testSuitePop; y++)
		testCase[y]=y;
	int* score = NULL;  //array to hold a score (number of lines covered) for each
test case
	score = new int[testSuitePop];
	for(y=0; y<testSuitePop; y++)  //initialise the score array
		score[y] = 0;

	for(y=0; y<testSuitePop; y++) //assigns a score to each test case
	{
		for(int x=0; x<noLines; x++)
		{
			if(testSuiteCoverage[y][x]=='1')
				score[y]++;
		}
	}

	quickSort(score, testCase, testSuitePop);

	bool* lineCovered = NULL;  //array to keep track of which lines have been covered
by test cases
	lineCovered = new bool[noLines];
	for(int i=0; i<testSuitePop; i++)
		lineCovered[i] = false;

	while(noLinesCovered<totalNoCovered)
	{
		noLinesCovered = 0;

		for(int x=0; x<noLines; x++)  //sets elements of the linesCovered array to
be true if covered by the current test case
		{
			if(testSuiteCoverage[testCase[pos]][x] == '1')
				lineCovered[x] = true;
		}

		for(x=0; x<noLines; x++)  //increments noLinesCovered for each element in
the linesCovered array that is true
		{
			if(lineCovered[x] == true)
				noLinesCovered++;
		}

		solution[solutionPop] = testCase[pos];//add the current test case to the
solution array
		printf("Test case %d selected ", testCase[pos]);
		printf("(%d lines covered)", score[pos]);
		printf(" ...Running total number of lines covered: %d\n", noLinesCovered);
		pos--;
		solutionPop++;
```

```
	}
	printResults("Greedy", testSuiteCoverage, testSuite, solution, solutionPop,
totalNoCovered, noLinesCovered, testSuitePop);
}


/****************************** ADDITIONAL GREEDY ALGORITHM
******************************/

void additionalGreedy(char **testSuiteCoverage, char **testSuite, int *solution, int
noLines, int testSuitePop, int totalNoCovered)
{
	printf("\nADDITIONAL GREEDY ALGORITHM\n");
	int solutionPop = 0;  //number of test cases in the solution
	int noLinesCovered = 0;  //number of lines covered in the solution
	int* testCase = NULL;  //array to hold the position of each test case number
during sorting
	testCase = new int[testSuitePop];
	int* score = NULL;  //array to hold a score (number of lines covered) for each
test case
	score = new int[testSuitePop];
	char** originalCoverage = NULL;  //original coverage information
	originalCoverage = new char*[testSuitePop];  //allocate the main array
	for(int i=0; i<testSuitePop; i++)
		originalCoverage[i] = new char[noLines];  //allocate each member of the
main array
	for(i=0; i<testSuitePop; i++)
	{
		for(int x=0; x<noLines; x++)
			originalCoverage[i][x] = testSuiteCoverage[i][x];
	}

	while(noLinesCovered<totalNoCovered)
	{
		for(int y=0; y<testSuitePop; y++)  //number the test case number array
			testCase[y]=y;
		for(y=0; y<testSuitePop; y++)  //initialise the score array
			score[y] = 0;
		for(y=0; y<testSuitePop; y++) //assigns a score to each test case
		{
			for(int x=0; x<noLines; x++)
			{
				if(testSuiteCoverage[y][x]=='1')
					score[y]++;
			}
		}

		quickSort(score, testCase, testSuitePop);

		printf("Test case %d selected ", testCase[testSuitePop-1]);
		printf("(%d lines covered)", score[testSuitePop-1]);
		solution[solutionPop] = testCase[testSuitePop-1];  //adds the solution
number at the top of the sorted testCse array

		for(int x=0; x<noLines; x++)  //recalculate testSuiteCoverage info
		{
			if(testSuiteCoverage[testCase[testSuitePop-1]][x]=='1')
			{
				noLinesCovered++;
				for(int y=0; y<testSuitePop; y++)
					testSuiteCoverage[y][x] = '0';
			}
		}
		solutionPop++;
		printf(" ...Running total number of lines covered: %d\n", noLinesCovered);
	}

	printResults("Additional Greedy", originalCoverage, testSuite, solution,
solutionPop, totalNoCovered, noLinesCovered, testSuitePop);
}
```

```
/******************************** TWO-OPTIMAL GREEDY ALGORITHM
*******************************/

void twoOptimalGreedy(char **testSuiteCoverage, char **testSuite, int *solution, int
noLines, int testSuitePop, int totalNoCovered)
{
	printf("\n2-OPTIMAL GREEDY ALGORITHM\n");
	int solutionPop = 0;  //number of test cases in the solution
	int noLinesCovered = 0;  //number of lines covered in the solution
	int topScore = 0;
	int topTestCase = -1;
	int secondTopTestCase = -1;
	int currentScore = 0;
	int* score = NULL;  //array to hold a score (number of lines covered) for each
test case
	score = new int[testSuitePop];
	bool* covered = NULL;  //array to hold whether or not each line is covered by a
test case
	covered = new bool[noLines];
	char** originalCoverage = NULL;  //original coverage information
	originalCoverage = new char*[testSuitePop];  //allocate the main array
	for(int i=0; i<testSuitePop; i++)
		originalCoverage[i] = new char[noLines];  //allocate each member of the
main array
	for(i=0; i<testSuitePop; i++)
	{
		for(int x=0; x<noLines; x++)
			originalCoverage[i][x] = testSuiteCoverage[i][x];
	}

	while(noLinesCovered<totalNoCovered)
	{
		topScore = 0;  //reset the top score
		topTestCase = -1;  //reset the number of the best two test cases
		secondTopTestCase = -1;
		for(int z=0; z<testSuitePop; z++)  //look at each test case in the test
suite and find the combination
		{                                                                    //of the
two test cases that achieve best coverage
			for(int y=0; y<testSuitePop; y++)
			{
				for(int i=0; i<noLines; i++)
					covered[i] = false;  //reset covered array
				for(int x=0; x<noLines; x++)
				{    //if one of the two test cases covers the line
					if(testSuiteCoverage[z][x]=='1' ||
testSuiteCoverage[y][x]=='1')
						covered[x] = true;
				}
				currentScore = 0;  //reset the current score
				for(i=0; i<noLines; i++)  //count the number of lines
covered by this combination
				{
					if(covered[i] == true)
						currentScore++;
				}
				//printf("Test cases %d and %d cover %d lines\n", z, y,
currentScore);
				if(currentScore > topScore)  //if the current score is
better than the best score
				{
					topScore = currentScore;  //make current score the
top score
					topTestCase = z;  //remember the best two test cases
					secondTopTestCase = y;
				}
			}
		}
		printf("\nBest combination of test cases is: %d and %d\n", topTestCase,
secondTopTestCase);
```

```
                solution[solutionPop] = topTestCase;  //add the top two test cases to the
solution array
                solution[solutionPop+1] = secondTopTestCase;
                for(int line=0; line<noLines; line++)  //recalculate coverage information
                {
                        if(testSuiteCoverage[topTestCase][line]=='1')
                        {
                                for(int test=0; test<testSuitePop; test++)
                                        testSuiteCoverage[test][line]='0';
                        }
                        if(testSuiteCoverage[secondTopTestCase][line]=='1')
                        {
                                for(int test=0; test<testSuitePop; test++)
                                        testSuiteCoverage[test][line]='0';
                        }
                }
                noLinesCovered+=topScore;
                solutionPop+=2;
                printf("Running total number of lines covered: %d\n\n", noLinesCovered);
        }
        printResults("2-Optimal Greedy", originalCoverage, testSuite, solution,
solutionPop, totalNoCovered, noLinesCovered, testSuitePop);
}


/****************************** HILL-CLIMBING ALGORITHM
******************************/

void hillClimbing(char **testSuiteCoverage, char **testSuite, int *solution, int noLines,
int testSuitePop, int totalNoCovered)
{
        printf("\nHILL-CLIMBING ALGORITHM\n");
        //initial random solution, neighbours are the same solution with one test case
removed, fittest one is the solution with
        //best coverage (if none of them have complete coverage then keep current
solution)

        int noSolutions = random(testSuiteCoverage, testSuite, solution, noLines,
testSuitePop, totalNoCovered); //find the number of test cases in the solution
        printf("\nNumber of test cases in the randomly generated solution: %d\n",
noSolutions);
        printf("Random solution:\n");
        for(int i=0; i<noSolutions; i++)
                printf("%d ", solution[i]);

        int* savedSolution = NULL;
        savedSolution = new int[noSolutions];
        int** neighbours = NULL;  //2d array to hold the test case numbers in each
neighbour solution
        neighbours = new int*[noSolutions];  //allocate the main array
        for (i=0; i<noSolutions; i++)
                neighbours[i] = new int[noSolutions-1];  //allocate each member of the
main array
        for(int y=0; y<noSolutions; y++)  //initialise the 2d array
        {
                for(int x=0; x<noSolutions-1; x++)
                        neighbours[y][x] = -1;
        }

        bool* covered = NULL;  //array to hold whether or not each line is covered by a
test case
        covered = new bool[noLines];

        int currentScore = 0;  //score of the current neighbour
        int bestScore = 0;  //coverage of the best neighbour
        int bestNeighbour = -1;  //the best neighbour (highest coverage)
        int miss = 0;  //position in the solution array to not copy over (miss out)
        int pos = 0;  //position in the solution array
        bool terminate = false;  //flag to determine when to terminate the algorithm
        int solutionPop = 0;  //number of test cases in the solution array
        int noLinesCovered = 0;  //number of lines covered by the solution individual
```

```
	for(i=0; i<noSolutions; i++)
		savedSolution[i] = solution[i];  //save the initial solution

	while(terminate==false)
	{
		for(i=0; i<noSolutions; i++)
			savedSolution[y] = solution[y];  //save the current solution
		noLinesCovered = bestScore;

		printf("\nCurrent Best Solution:\n");
		for(i=0; i<noSolutions; i++)
			printf("%d ", solution[i]);
		printf("\n");

		miss = 0;

		for(y=0; y<noSolutions; y++)  //generate neighbours
		{
			pos = 0;
			for(int x=0; x<noSolutions-1; x++)
			{
				if(pos==miss)  //if the position in the solution array is
the one supposed to be missed
				{
					pos++;  //skip to the next position in the solution
array
					neighbours[y][x] = solution[pos];
				}
				else
					neighbours[y][x] = solution[pos];
				pos++;
			}
			miss++;
		}

		printf("Its neighbours are:\n");
		for(y=0; y<noSolutions; y++)
		{
			printf("%d. ", y);
			for(int x=0; x<noSolutions-1; x++)
				printf("%d ", neighbours[y][x]);
			printf("\n");
		}

		bestNeighbour = -1;
		bestScore = 0;

		for(int sol=0; sol<noSolutions; sol++)  //find the neighbour with the best
coverage
		{
			for(int i=0; i<noLines; i++)
				covered[i] = false;  //reset covered array
			for(int y=0; y<noSolutions-1; y++)  //check to see which lines are
covered by the test cases in the neighbour
			{
				for(int x=0; x<noLines; x++)
				{
					if(testSuiteCoverage[neighbours[sol][y]][x]=='1')
						covered[x] = true;
				}
			}
			currentScore = 0;  //reset the current score
			for(i=0; i<noLines; i++)  //count the number of lines covered by
this neighbour
			{
				if(covered[i] == true)
					currentScore++;
			}
			printf("Neighbour %d covers %d lines\n", sol, currentScore);
```

```
                        if(currentScore > bestScore)  //if the current score is better than
the best score
                        {
                                bestScore = currentScore;  //make current score the top
score
                                bestNeighbour = sol;  //remember the best two test cases
                        }
                }

                printf("Best score= %d\n", bestScore);
                printf("Best neighbour= %d\n", bestNeighbour);

                for(i=0; i<noSolutions; i++)
                        solution[i] = neighbours[bestNeighbour][i];  //make the current
best neighbour the solution

                if(bestScore < totalNoCovered)  //if the best neighbour covers fewer lines
than the original test suite
                {
                        printf("Terminating algorithm - none of the above neighbours are
better than the current solution\n");
                        terminate = true;  //terminate the algorithm

                        for(i=0; i<noSolutions; i++)
                                solution[i] = savedSolution[i];  //make the solution the
previously saved test suite
                        solutionPop = noSolutions;

                        printf("The minimised test suite is:\n");
                        for(i=0; i<solutionPop; i++)
                                printf("%d ", solution[i]);
                }
                noSolutions--;
        }

        printResults("Hill-Climbing", testSuiteCoverage, testSuite, solution, solutionPop,
totalNoCovered, noLinesCovered, testSuitePop);
}


/****************************** GENETIC ALGORITHM ******************************/

void genetic(char **testSuiteCoverage, char **testSuite, int *solution, int noLines, int
testSuitePop, int totalNoCovered, int populationNo)
{
        int lifespan = populationNo/2;  //lifespan of the generation
        int currentPop = populationNo;  //current population
        int solutionPop = 0;  //number of test cases in the solution test suite
        int noLinesCovered = 0;  //number of lines covered by the solution test suite
        int noSolutions = 0;  //number of test cases in the current individual
        int currentScore = 0;  //number of lines covered by the current individual
        int x=0;
        int a=0;
        int b=0;
        int randomInt = 0;  //random integer
        bool terminate = false;  //flag to determine when to terminate the algorithm
        int start = 0;
        int range = 0;
        bool found = false;  //whether an individual was found or not
        bool outsideFound = false;  //whether an individual was found or not
        int noFound = 0;  //number of selected individuals
        int noParents = populationNo/2;  //number of parent individuals
        int fCount = 0;  //number of test cases in the father
        int mCount = 0;  //number of test cases in the father
        int max = 0;  //maximum range a random number can be generated within
        int pos = noParents;  //position in the population array
        int loop = 0;  //loop counter
        int plot = 0;  //place in the population array
        int testCaseCount = 0;  //number of test cases in a solution
        int bestScore = 0;  //fitness score of the best individual
        int bestIndividual = -1;  //best individual
```

```
	//2D array to hold population information (population id and which test cases it
uses)
	int** population = NULL;  //declare the 2D array as a pointer to a pointer
	population = new int*[populationNo];  //allocate the main array
	for(int i=0; i<populationNo; i++)
		population[i] = new int[testSuitePop];  //allocate each member of the main
array

	bool* covered = NULL;  //array to hold whether or not each line is covered by a
test case
	covered = new bool[noLines];
	int* selection = NULL;  //array to hold whether an individual hasn't been selected
to reproduce (0), has been (1), or is an offspring (2)
	selection = new int[populationNo];
	bool* parentSelected = NULL;  //array to hold whether or not a parent individual
has already been selected in crossover
	parentSelected = new bool[populationNo/2];
	int* score = NULL;  //array to hold a score (number of test cases removed) for
each individual in the population
	score = new int[populationNo];
	int* individual = NULL;  //array to hold the position of each individual during
quicksort
	individual = new int[populationNo];

	int* father = NULL;  //array to hold the 'father' individuals (the index in the
parent array)
	father = new int[noParents/2];
	for(int y=0; y<noParents/2; y++)
		father[y]=-1;
	int* mother = NULL;  //array to hold the 'mother' individuals (the index in the
parent array)
	mother = new int[noParents/2];
	for(y=0; y<noParents/2; y++)
		mother[y]=-1;
	int* parents = NULL;  //array to hold the parent individuals (the index in the
population array)
	parents = new int[noParents];
	for(y=0; y<noParents; y++)
		parents[y]=-1;
	int* offspring1 = NULL;  //array to hold the test case numbers used by offspring1
	offspring1 = new int[testSuitePop];
	for(y=0; y<testSuitePop; y++)
		offspring1[y]=-1;
	int* offspring2 = NULL;  //array to hold the test case numbers used by offspring2
	offspring2 = new int[testSuitePop];
	for(y=0; y<testSuitePop; y++)
		offspring2[y]=-1;

	for(y=0; y<populationNo; y++)  //initialise the population array
	{
		for(int x=0; x<testSuitePop; x++)
			population[y][x] = -1;
	}

	for(y=0; y<populationNo; y++)   //populate the population array with randomly
generated solutions
	{
		noSolutions = random(testSuiteCoverage, testSuite, solution, noLines,
testSuitePop, totalNoCovered);
		for(int x=0; x<noSolutions; x++)
			population[y][x] = solution[x];
	}

	for(y=0; y<populationNo; y++)  //initialise the individual array
		individual[y]=y;

	printf("Genetic Algorithm\n");
//GENERATION LOOP
	for(int generation=0; generation<lifespan; generation++)
```

```
	{

		printf("***** NEW GENERATION *****\n");
		for(y=0; y<populationNo; y++)
		{
			printf("Individual: %d\t", y);
			for(int x=0; x<testSuitePop; x++)
			{
				if(population[y][x] != -1)
					printf("%d ", population[y][x]);
			}
			printf("\n");
		}
		printf("\n\n");

		/************ selection ************/
		//kill any individuals that do not achieve same coverage as original test
suite
		for(int z=0; z<populationNo; z++)
		{
			for(int i=0; i<noLines; i++)  //reset covered array
				covered[i] = false;
			currentScore = 0;  //reset currentScore

			for(y=0; y<testSuitePop; y++)  //calculate coverage for the test
suite
			{
				for(x=0; x<noLines; x++)
				{
					if(population[z][y]!=-1)
					{
						if(testSuiteCoverage[population[z][y]][x] ==
'1')
							covered[x] = true;
					}
				}
			}
			for(i=0; i<noLines; i++)
			{
				if(covered[i] == true)
					currentScore++;
			}
			//printf("\nIndividual %d: %d", z, currentScore);
			if(currentScore<totalNoCovered)  //if the score of this test suite
is less than the original test suite
			{
				score[z] = -1;
				for(y=0; y<testSuitePop; y++)
					population[z][y] = -1;  //delete from the population
			}
		}

		//order the remaining individuals according to fitness - those with fewer
test cases are fitter
		for(y=0; y<populationNo; y++)  //reset the individual score array
			score[y] = 0;

		for(y=0; y<populationNo; y++)  //assigns a score to each individual in the
population
		{
			testCaseCount = 0;
			for(int x=0; x<testSuitePop; x++)
			{
				if(population[individual[y]][x] != -1)
					testCaseCount++;
			}
			if(testCaseCount==0)
				testCaseCount=testSuitePop;
			score[y]=testSuitePop-testCaseCount;
		}
```

```
            quickSort(score, individual, populationNo); //quicksort will put the
individuals covering

        //more lines at the end of the array
            printf("\nQuicksorted\n");

            start = 0;
            range = 0;
            found = false;  //whether an individual was found or not
            noFound = 0;  //number of selected individuals
            for(y=0; y<populationNo; y++)  //reset the individual selection array
                selection[y] = 0;

            for(y=0; y<populationNo; y++)  //determine the range of the roulette wheel
            {
                if(score[y]!=-1)
                    range+=score[y];
            }

            loop = 0;

            //calculate number of active individuals
            currentPop=0;
            for(i=0; i<populationNo; i++)
            {
                if(individual[i]!=-1)
                    currentPop++;
            }
            noParents=currentPop/2;
            if((noParents%2)!=0)  //noParents/2 is odd
                noParents--;

            while(noFound<noParents)  //select those individuals that will breed
            {
                start=0;
                randomInt = rand() % range;  //generate a random number in the
range
                for(int x=0; x<populationNo; x++)
                {
                    if(randomInt>start && randomInt<=(start+score[x]))  //find
out which solution that number applies to
                    {
                        if(score[x]!=-1 && selection[x]!=1)  //check the
solution hasn't been deleted or already selected
                        {
                            selection[x]=1;  //mark as being selected
                            noFound++;
                        }
                    }
                    start+=score[x];
                    loop++;
                }
                if(loop>populationNo*populationNo)  //if, for an especially poor
set of initial solutions, none get selected,
                    noFound=noParents;     //leave the loop
            }

            if(loop>populationNo*populationNo)
            {
                for(i=0; i<populationNo; i++)
                    selection[i]=0;
                for(i=populationNo/2; i<populationNo; i++)
                {
                    if(individual[i]!=-1)
                        selection[i]=1;  //mark as being selected
                }
            }

            /************ crossover ************/
            //one point crossover
```

```
		//count number of parents
		noParents=0;
		for(i=0; i<populationNo; i++)
		{
			if(selection[i]==1)
				noParents++;
		}
		if((noParents%2)!=0)  //noParents/2 is odd
			noParents--;

		y=0;
		for(i=0; i<populationNo; i++)
		{
			if(selection[i]==1)
			{
				parents[y]=individual[i];
				y++;
			}
		}

		printf("\nFathers:\n");
		for(y=0; y<noParents/2; y++)
			printf("%d\t", parents[y]);
		printf("\n");
		printf("Mothers:\n");
		for(y=noParents/2; y<noParents; y++)
			printf("%d\t", parents[y]);
		printf("\n\n");

		for(i=0; i<noParents/2; i++)
		{
			for(y=0; y<testSuitePop; y++)  //reset offspring arrays
				offspring1[y]=-1;
			for(y=0; y<testSuitePop; y++)
				offspring2[y]=-1;
			fCount = 0;  //reset father/mother counters (number of test cases
in these solutions)
			mCount = 0;
			max = 0;

			printf("Father %d: ", parents[i]);
			for(y=0; y<testSuitePop; y++)
			{
				if(population[parents[i]][y] != -1)
				{
					printf("%d ", population[parents[i]][y]);
					fCount++;
				}
			}
			printf("\n");

			printf("Mother %d: ", parents[i+noParents/2]);
			for(y=0; y<testSuitePop; y++)
			{
				if(population[parents[i+noParents/2]][y] != -1)
				{
					printf("%d ",
population[parents[i+noParents/2]][y]);
					mCount++;
				}
			}
			printf("\n");

			if(fCount > mCount)
				max = fCount;
			else
				max = mCount;

			randomInt = 1 + rand() % (max-1);  //generates a random integer
within the range
```

```
                        for(y=0; y<randomInt; y++)  //put the first randomInt elements into
the offspring
                        {
                                offspring1[y] = population[parents[i]][y];
                                offspring2[y] = population[parents[i+noParents/2]][y];
                        }

                        for(y=randomInt; y<max; y++)  //put the last max-randomInt elements
into the offspring
                        {
                                offspring1[y] = population[parents[i+noParents/2]][y];
                                offspring2[y] = population[parents[i]][y];
                        }

                        printf("Offspring 1: ");
                        for(y=0; y<testSuitePop; y++)
                        {
                                if(offspring1[y]!=-1)
                                        printf("%d ", offspring1[y]);
                        }
                        printf("\n");
                        printf("Offspring 2: ");
                        for(y=0; y<testSuitePop; y++)
                        {
                                if(offspring2[y]!=-1)
                                        printf("%d ", offspring2[y]);
                        }
                        printf("\n\n");

                        //save offspring1 to the population array
                        found=false;
                        plot=0;
                        while(found==false)
                        {
                                if(selection[plot]==0)
                                {
                                        for(x=0; x<testSuitePop; x++)
                                        {
                                                population[individual[plot]][x] =
offspring1[x];
                                                selection[plot]=2;  //mark as being an
offspring
                                        }
                                        found=true;
                                }
                                plot++;
                        }

                        //save offspring2 to the population array
                        plot=0;
                        found=false;
                        while(found==false)
                        {
                                if(selection[plot]==0)
                                {
                                        for(x=0; x<testSuitePop; x++)
                                        {
                                                population[individual[plot]][x] =
offspring2[x];
                                                selection[plot]=2;  //mark as being an
offspring
                                        }
                                        found=true;
                                }
                                plot++;
                        }
                }

                printf("***** POPULATION ARRAY *****\n");
                for(y=0; y<populationNo; y++)
```

```
		{
			printf("Individual: %d\t", y);
			for(int x=0; x<testSuitePop; x++)
			{
				if(population[y][x] != -1)
					printf("%d ", population[y][x]);
			}
			printf("\n");
		}
		printf("\n\n");

		/************ mutation ************/
		//one test case removed at random from randomly selected individuals
		for(i=0; i<populationNo; i++)
		{
			randomInt = rand() % 10;
			if(randomInt == 0)  //0.1 chance of success
			{
				printf("Mutation to individual %d - ", i);
				randomInt = rand() % testSuitePop;
				if(population[i][randomInt]==-1)  //if this test case does
not exist (has already been removed)
					printf("but no test case removed this time\n");
				else
					printf("test case %d deleted\n",
population[i][randomInt]);
				population[i][randomInt] = -1;
			}
		}
	}
//END OF GENERATION LOOP
	printf("\n---- End of generation loop ----\n\n");

	//remove individuals with coverage below original test suite
	for(int z=0; z<populationNo; z++)
	{
		for(int i=0; i<noLines; i++)  //reset covered array
			covered[i] = false;
		currentScore = 0;  //reset currentScore

		for(y=0; y<testSuitePop; y++)  //calculate coverage for the test suite
		{
			for(x=0; x<noLines; x++)
			{
				if(population[z][y]!=-1)
				{
					if(testSuiteCoverage[population[z][y]][x] == '1')
						covered[x] = true;
				}
			}
		}
		for(i=0; i<noLines; i++)
		{
			if(covered[i] == true)
				currentScore++;
		}
		if(currentScore<totalNoCovered)  //if the score of this test suite is less
than the original test suite
		{
			score[z] = -1;
			for(y=0; y<testSuitePop; y++)
				population[z][y] = -1;  //delete from the population
		}
	}

	//give the remaining individuals a fitness score those with fewer test cases are
fitter
	for(y=0; y<populationNo; y++)  //reset the individual score array
		score[y] = 0;

	//printf("\nNumber of test cases\n");
```

```
	for(y=0; y<populationNo; y++)  //assigns a score to each individual in the
population
	{
		testCaseCount = 0;
		for(int x=0; x<testSuitePop; x++)
		{
			if(population[individual[y]][x] != -1)
				testCaseCount++;
		}
		if(testCaseCount==0)
			testCaseCount=testSuitePop;
		score[y]=testSuitePop-testCaseCount;
		printf("Individual %d: %d\n", individual[y], score[y]);
	}
	//find the smallest solution and make it the solution
	for(y=0; y<populationNo; y++)
	{
		if(score[y]>bestScore)
		{
			bestScore = score[y];
			bestIndividual = individual[y];
		}
	}
	printf("best score: %d\n", bestScore);
	printf("best individual: %d\n", bestIndividual);
	for(y=0; y<testSuitePop; y++)
		solution[y] = population[bestIndividual][y];

	for(i=0; i<noLines; i++)  //reset covered array
		covered[i] = false;
	noLinesCovered = 0;  //reset noLinesCovered

	for(y=0; y<testSuitePop; y++)  //calculate coverage for the best individual
	{
		for(x=0; x<noLines; x++)
		{
			if(population[bestIndividual][y]!=-1)
			{
				if(testSuiteCoverage[population[bestIndividual][y]][x] ==
'1')
					covered[x] = true;
			}
		}
	}
	for(i=0; i<noLines; i++)
	{
		if(covered[i] == true)
			noLinesCovered++;
	}

	solutionPop=0;
	printf("\nSOLUTION: ");
	for(y=0; y<testSuitePop; y++)  //calculate the number of test cases in the
solution
	{
		if(solution[y]!=-1)
		{
			printf("%d ", solution[y]);
			solutionPop++;
		}
	}
	printf("\nSolutionPop: %d\n", solutionPop);

	printResults("Genetic", testSuiteCoverage, testSuite, solution, solutionPop,
totalNoCovered, noLinesCovered, testSuitePop);
}


/****************************** MODIFIED GREEDY ALGORITHM
******************************/
```

```
void modifiedGreedy(char **testSuiteCoverage, char **testSuite, int *solution, int
noLines, int testSuitePop, int totalNoCovered)
{
	int solutionPop = 0;  //population of solution array
	int noLinesCovered = 0;
	int* lineNo = NULL;  //array to hold the position of each line number number
during sorting
	lineNo = new int[noLines];
	int* lineScore = NULL;  //array to hold a score (number of test cases that execute
this line) for each line
	lineScore = new int[noLines];
	int* candidate = NULL;  //array to hold a candidate test cases for the least
executed line
	candidate = new int[testSuitePop];
	int* candidateScore = NULL;  //array to hold a candidate test case scores in the
event that there is more than one test case that executes the least executed line
	candidateScore = new int[testSuitePop];
	int leastExecuted = -1;  //line which is executed by the least number of test
cases
	int pos = 0;  //position in the candidate array
	int bestScore = -1;  //best candidate score
	int bestCandidate = -1;  //best candidate test case number

	printf("\nMODIFIED GREEDY ALGORITHM\n");

	while(noLinesCovered<totalNoCovered)
	{
		for(int x=0; x<noLines; x++)
			lineNo[x]=x;

		for(int i=0; i<noLines; i++)  //initialise the lineScore array
			lineScore[i] = 0;

		for(x=0; x<noLines; x++) //assigns a score to each line
		{
			for(int y=0; y<testSuitePop; y++)
			{
				if(testSuiteCoverage[y][x]=='1')
					lineScore[x]++;
			}
		}

		quickSort(lineScore, lineNo, noLines);

		for(i=0; i<noLines; i++)
		{
			if(lineScore[i]!=0)
			{
				leastExecuted = lineNo[i];
				break;
			}
		}

		printf("Line %d is executed by the fewest number of test cases\n",
lineNo[0]);
		//printf("least executed is line number %d\n", leastExecuted);


		for(i=0; i<testSuitePop; i++)  //initialise the candidate array
			candidate[i] = 0;
		for(i=0; i<testSuitePop; i++)  //initialise the candidateScore array
			candidateScore[i] = 0;

		pos = 0;  //position in the candidate array

		for(int y=0; y<testSuitePop; y++)  //find out which test cases execute
this line
		{
			if(testSuiteCoverage[y][leastExecuted]=='1')
			{
				candidate[pos]=y;
```

```
                        pos++;
                }
        }

        if(pos>1)  //if a draw, use the test case which, of the two, executes the
most lines in general
        {
                for(y=0; i<pos; i++)
                {
                        for(x=0; x<noLines; x++)
                        {
                                if(testSuiteCoverage[candidate[pos]][x]=='1')
                                        candidateScore[pos]++;
                        }
                }
        }

        bestScore = -1;
        bestCandidate = -1;

        for(i=0; i<pos; i++)
        {
                if(candidateScore[i]>bestScore)
                bestCandidate=candidate[i];
        }

        printf("Best test case is %d\n", bestCandidate);
        solution[solutionPop] = candidate[bestCandidate];  //adds the solution
number at the top of the sorted testCse array

        //recalculate coverage info as before

        for(x=0; x<noLines; x++)
        {
                if(testSuiteCoverage[bestCandidate][x]=='1')
                {
                        noLinesCovered++;
                        for(int y=0; y<testSuitePop; y++)
                        {
                                testSuiteCoverage[y][x] = '0';
                        }
                }
        }
        solutionPop++;
        printf(" ...Running total number of lines covered: %d\n", noLinesCovered);
    }
}


/****************************** PRINT RESULTS ******************************/

void printResults(char name[], char **testSuiteCoverage, char **testSuite, int
solution[], int solutionPop, int totalNoCovered, int noLinesCovered, int testSuitePop)
{
    //Calculate the percentage coverage
    float percentCovered = 0;
    float floatNoCovered = (float)noLinesCovered;  //float version of number of lines
covered by the minimised test suite
    float floatNoLines = (float)totalNoCovered;  //float version of number of lines
covered by the original test suite
    percentCovered = (floatNoCovered/floatNoLines)*100;

    //Calculate the percentage size of the minimised test suite and the decrease in
test suite size
    float percentSize = 0;
    float percentReduction = 0;
    float floatSolutionPop = (float)solutionPop;  //float version of number of test
cases in the minimised test suite
    float floatTestSuitePop = (float)testSuitePop;  //float version of number of test
cases in the original test suite
    percentSize = (floatSolutionPop/floatTestSuitePop)*100;
```

```
	percentReduction = 100-percentSize;

	printf("\n\n********* %s Algorithm Results *********\n", name);

	//The genetic algorithm may have deleted test cases (from mutation) in it's genome
so these must not be displayed
	if(name[0]=='G' && name[1]=='e')  //if the genetic algorithm calls this function
	{
		printf("\nThe minimised test suite is:\n");
		for(int y=0; y<testSuitePop; y++)
		{
			if(solution[y]!=-1)
			{
				for(int x=0; x<NAMESIZE; x++)
					printf("%c", testSuite[solution[y]][x]);
				printf("\n");
			}
		}
		printf("\nThe coverage information for these test cases is:\n");
		for(y=0; y<testSuitePop; y++)
		{
			if(solution[y]!=-1)
			{
				for(int x=0; x<totalNoCovered; x++)
					printf("%c", testSuiteCoverage[solution[y]][x]);
				printf("\n");
			}
		}
	}
	else  //if anything other than genetic algorithm called this function
	{
		printf("\nThe minimised test suite is:\n");
		for(int y=0; y<solutionPop; y++)
		{
			for(int x=0; x<NAMESIZE; x++)
				printf("%c", testSuite[solution[y]][x]);
			printf("\n");
		}
		printf("\nThe coverage information for these test cases is:\n");
		for(y=0; y<solutionPop; y++)
		{
			for(int x=0; x<totalNoCovered; x++)
				printf("%c", testSuiteCoverage[solution[y]][x]);
			printf("\n");
		}
	}

	printf("\nNumber of blocks/branches/lines covered by original test suite: %d\n",
totalNoCovered);
	printf("Number of blocks/branches/lines covered by minimised test suite: %d
(%.2f%%) \n", noLinesCovered, percentCovered);
	printf("Number of test cases in original test suite: %d\n", testSuitePop);
	printf("Number of test cases in minimised test suite: %d (%.2f%% size
reduction)\n", solutionPop, percentReduction);
}
```