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

Module Title: CSMPRJ
Submission Date: 07/09/07

Pareto Efficient Multi-Objective Test Case Selection

By
Syed Ali Shahid

Supervised by
Prof. Mark Harman

Report:
Final Report

ABSTRACT

This project will build on the previous work by Mark Harman and Shin Yoo on Pareto Efficient Test Case Selection pertaining to two objective formulation. It will introduce the concept of Pareto Efficient Test Case selection and the benefits of this approach. This project will illustrate the benefits of Pareto efficient test case selection with empirical studies of two objective formulations.

Shin Yoo and Mark Harman used multiple objectives such as code coverage, past fault-detection history and execution cost, and constructed a group of non-dominating, equivalently optimal test case subsets. This project will develop on the code coverage and execution cost objectives.

This project will consider a wider range of software artifacts with different meta-heuristic multi-objective optimization techniques. Also, a new evolutionary algorithm will be developed.

The artifacts have been obtained from SIR – a software archive. The programs in the project will be developed in C language.

AKNOWLEDGEMENTS

I would like to thank Shin Yoo and Mark Harman for their ongoing help in this project. Also, I am grateful to Zheng Li, who provided the coverage analysis data, which was obtained using a commercial tool called Canatata++. Shin Yoo provided the execution cost data for the Siemens suite and Space program which was obtained via Valgrind.

Table of Contents

Table of Figures

PROBLEM DEFINITION

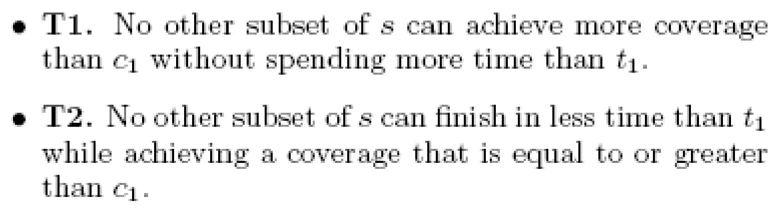
Previous work has treated test case selection as a single objective optimization problem. It was treated as two/three objective formulation only recently by Mark Harman and Shin Yoo [2]. This project builds on their work on two objective formulation of coverage and cost by providing more artifacts and multi-objective meta heuristic optimization techniques.

Multi Objective optimization can be found in various fields: finance, aircraft design, or whenever optimization designs have to be taken in the presence of tradeoffs between two conflicting objectives. Maximizing profit and minimizing cost of a product or maximizing performance and minimization fuel consumption of a car are both examples of multi-objective optimization problem. This helps a decision maker make a well-informed decision that balances the tradeoffs between the objectives.

The project describes the potential benefits of Pareto efficient multi-objective test case selection, illustrating with empirical studies of two objective formulations.

TECHNICAL SPECIFICATION

This project will be based on multi objective formulation of regression test case selection problem and will work with two objective formulation, which uses statement coverage and computational cost of test cases as objectives.
Thus, code coverage that is one of the objectives will be maximized for a given cost. The other objective time should be minimized for a given cost.



- **T1.** No other subset of s can achieve more coverage than c_1 without spending more time than t_1.
- **T2.** No other subset of s can finish in less time than t_1 while achieving a coverage that is equal to or greater than c_1.

Fig 1 – Obtained from Pareto Efficient Test Case Selection by Mark Harman & Shin Yoo Given test suite s with coverage c_1 and time t_1

The project will present seven algorithms for solving the two objective instances of the test case selection problem: a reformation of the single-objective greedy algorithm, Hill climbing algorithm, the Non Dominating Sorting Genetic Algorithm (NSGA-2) of Deb et al. [6], Island genetic algorithm variant of NSGA-2 (vNSGA-2), which was introduced in [2], the Strength Pareto Evolutionary Algorithm 2 called SPEA-2 by Zitler et al. [8], the Pareto Envelope-based Selection Algorithm PESA2 was introduced by Corne et al. and a new algorithm (evolutionary algorithm) will be developed. Also, an exhaustive search will be carried out on the smaller artifacts to ascertain the true Pareto frontier.
The project will present results from an empirical study of the application of several greedy, meta-heuristic and evolutionary search algorithms to 8 programs ranging from 374 to 11,148 lines of code. The results of these algorithms, when applied to two objective version of the problem using, as subjects four programs from the Siemens suite [4], together with **space** developed for the European Space Agency. The Siemens programs used are **print tokens**, **printtokens2**, **schedule** and **schedule2**. In addition to these three larger programs will be tested. Two of these programs (**Grep** and **Flex**) were used in [5] and another program **SED**. They were obtained from SIR - Software-artefacts Infrastructure Repository [6].

Table 1
Experiment Objects

Program	Lines Of Code	Test Pool Size	Avg. test suite size
print_tokens	726	4130	16
print_tokens2	570	4115	17
schedule	412	2650	8
schedule2	374	2710	8
space	6199	13585	153

Table 2
Additional Experiment Objects

Program	Lines Of Code	Test Pool Size
flex	10459	567
grep	10068	809
sed	11148	1293

AIMS AND OBJECTIVES

The aim of this project is to build on the work of two objective test case selection by [2]. To this end the project will use the artifacts used in [2] which are Siemens suite and Space program and than provide additional larger programs.

The objectives of the project are as follows:
1) To obtain coverage and cost data from larger artifacts i.e. SED, Grep and Flex.
2) To introduce a new algorithm and by means of empirical results ascertain its impact on the Pareto optimal front.
3) To introduce a wider range of multi-objective meta-heuristic algorithms.

BACKGROUND

Regression Testing ensures that new or modified features do not regress (make worse) existing features. Aim is to test the parts that have changed and ensure there has not been a detrimental effect on the rest of the system. It is an important and sometimes very frequent activity.
Regression testing is essential to ensure quality of software but it is an expensive maintenance activity [1].

> **IEEE software glossary defines regression testing as "selective retesting of a system or component to verify that modifications have not caused unintended effect and the system or component still complies with its specified requirements."**

Due to time limitations on regression testing using retest-all method, other techniques such as Test case selection and prioritization are considered [2].

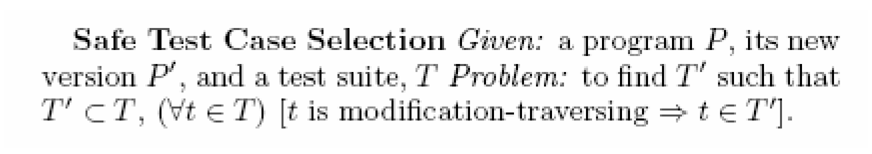
Existing literature categorizes three major areas:

1) Test Case Selection (Screen)
2) Test Suite Minimization (Remove)
3) Test Suite Prioritization (Order)

This project is concerned with multi-objective test case selection. Regression test selection techniques select some subset of an existing test suite to try to reduce the time required to retest the modified program [1]. Test suite reduction techniques permanently eliminate test cases from test suites. Prioritization techniques try to order the test cases in such a way that faults are detected early.

REGRESSION TEST CASE SELECTION

Test case selection techniques try to improve the retest all approach by selecting a subset of the entire test suite based on some test criteria.
One of the criteria to select test cases is the safe selection of test cases. Lets assume we have program P, its new version P', and a test suite, T. A test case is modification-traversing if it if it executes code that was changed or inserted into P, or deleted from P [2]. Subset T' of T is safe if it includes all the modification traversing test cases of T:



Safe Test Case Selection *Given:* a program P, its new version P', and a test suite, T *Problem:* to find T' such that $T' \subset T$, $(\forall t \in T)$ [t is modification-traversing $\Rightarrow t \in T'$].

Fig1.1 Test Case Selection from Mark Harman & Shin Yoo (2007)

META HEURISTIC

Metaheuristic is a heuristic method for solving a general class of computational problems by combining user given procedures – usually heuristic themselves – in an efficient way.
Metaheuristic are generally applied to problems where there are no satisfactory problem specific algorithms or heuristic; or when it is not practical to implement such a method [14].
The most commonly used Meta-Heuristics are targeted to optimization problems, but they can tackle any problem that can be recast in that form.

Many exciting and challenging problems in Software Engineering lie within the reach of metaheuristic search [13].

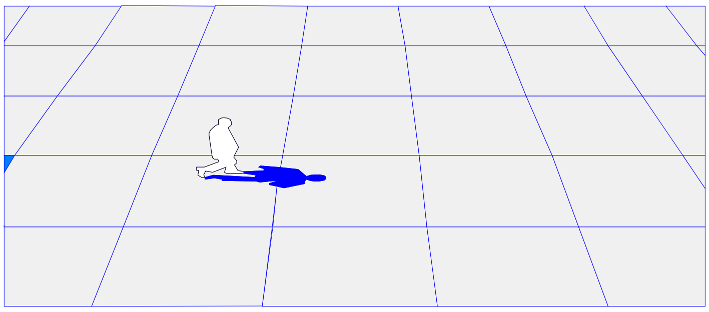
EVOLUTIONARY ALGORITHM

Evolutionary Algorithm (EA) 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 [15].

GENETIC ALGORITHM

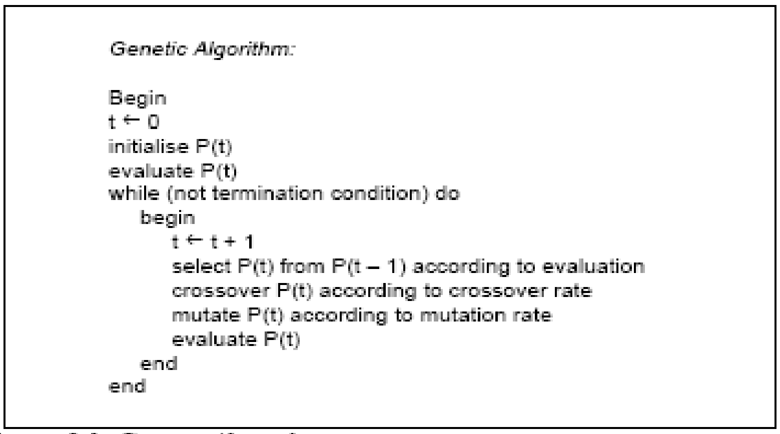
“Genetic Algorithms are good at taking large, potentially huge search spaces and navigating them, looking for optimal combinations of things, solutions you might not otherwise find in a lifetime.”



- Fig 1.2 from Salvatore Mangano *Computer Design*, May 1995

Genetic Algorithms (GA) are the most popular type of Evolutionary algorithm. Genetic Algorithms, is a search technique, based on biological evolution, which are used to approximate solutions to optimization.
To solve a specific problem, the genetic algorithm takes set of potential solutions to that problem as input 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 selection. The candidate solutions are test suites, with the aim of the genetic algorithm being to select it based on some criterion.

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 [16].



```
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

- Fig 1.3 Genetic Algorithm from Mark Harman and Zheng Li (2006)

Population could be:

☐ Bit strings	(0101 ... 1100)
☐ Real numbers	(43.2 -33.1 ... 0.0 89.2)
☐ Lists of rules	(R1 R2 R3 ... R22 R23)
☐ Program elements	(genetic programming)
☐ ... any data structure …	

Selection: Parents are chosen at random according to some criteria (fitness function)

Cross over Mutation:

- Alters one or more values in a genome

Before: (1 0 1 1 0 1 1 0)
After: (0 1 1 0 0 1 1 0)

Cross over Recombination:

- It leads to effective recombination of subsolutions

P1	(0 1 1 0 1 0 0 0)	(0 1 0 0 1 0 0 0)	C1
P2	(1 1 0 1 1 0 1 0)	(1 1 1 1 1 0 1 0)	C2

PARETO OPTIMALITY

According to Carlos [3], multiobjective optimization problem (MOP) can be defined as the problem of finding (Osyczka, 1985):
"a vector of decision variables which satisfies constraints and optimizes a vector function whose elements represent the objective functions. These functions form a mathematical description of performance criteria which are usually in conflict with each other. Hence, the term "Optimize" means finding such a solution which would give the values of all the objective functions acceptable to the decision maker."

Pareto optimality is a notion found in economics with a wide range of applications. The basic definition of Pareto optimality explained in [2] is that given a set of allocations and a set of population, allocation A is an improvement over allocation B, if A can make a person better off B without making any person worse off.

Based on this Mark Harman and Shin Yoo defined multiobjective optimization problem as to find a vector of decision variable x, which optimizes a vector of M objective functions $f_i(x)$ where i = 1,2,3,4,……,M. Objective functions are mathematical descriptions of optimization criteria (which are often in conflict with one another). So, if we want to maximize f_i, where i =1,2,…,M. A decision vector x is said to dominate decision vector y if and if their objective vectors satisfies:

$$f_i(x) \geq f_i(y) \forall i \in \{1, 2, \ldots, M\}; and$$

$$\exists i \in \{1, 2, \ldots, M\} | f_i(x) > f_i(y)$$

- Fig 1.4 Pareto Optimality from Mark Harman and Shin Yoo (2007)

Pareto optimality concept always gives a set of non-dominated solutions called the Pareto optimal set, while the corresponding vectors form the Pareto frontier. The vectors corresponding to solutions in a Pareto optimal set are called non-denominated.

So, the multiobjective problem can be described as:

Given: a vector of decision variables, x, and a set of objective functions, $f_i(x)$ where $i = 1, 2, \ldots, M$
Definition: maximize $\{f_1(x), f_2(x), \ldots, f_M(x)\}$ by finding the Pareto optimal set over the feasible set of solutions.

- Fig 1.5 Pareto Optimality from Mark Harman and Shin Yoo (2007)

Identifying the Pareto frontier is useful because it can be used to make well-informed decisions that balance trade-offs between the objectives.
The multi-objective test case selection problem with respect to Pareto efficient subset can be defined as:

Multi Objective Test Case Selection *Given:* a test suite, T, a vector of M objective functions, f_i, $i = 1, 2, \ldots, M$.
Problem: to find a subset of T, T', such that T' is a Pareto optimal set with respect to the objective functions, f_i, $i = 1, 2, \ldots, M$.

- Fig 1.6 Pareto Optimality from Mark Harman and Shin Yoo (2007)

MULTIOBJECTIVE ALGORITHMS

We usually consider two generation of MOEAs [3]:

1) **First Generation**
This is characterized by the use of Pareto ranking and niching (or fitness sharing). These algorithms are quite simple. Some were population based i.e. VEGA (Vector Evaluated Genetic Algorithm) while other used Pareto approach.
2) **Second Generation**
The introduction of elitism as a concept with two main forms: selection and using an external population.

Evolutionary algorithms offer many advantages pertaining to multi objective optimization. These algorithms can deal with a set of possible solutions simultaneously. This means that we can find several members of the Pareto optimal set in a single run. Also, evolutionary algorithms are less susceptible to the shape or continuity of the Pareto frontier.

Pareto-based techniques were introduced by Goldberg (1989). This technique uses non-dominated ranking and selection to move towards the Pareto Frontier. Also, to maintain diversity in the population it requires a technique and ranking procedure [3]. The advantage of Pareto approach is that it is relatively easy to implement and the disadvantage is the problem to scale the approach (i.e. checking non dominance is O (kM^2), where k=amount of objectives and M = population size).

There are normally three issues that are considered for design in a good metric [3]:

1. Minimize the distance of the Pareto front produced by our algorithm with respect to the true Pareto front
2. Maximize the speed of solutions found.
3. Maximize the elements of Pareto optimal set found.

So, the question arises what kind of metrics can we use?

Well, there are many metrics that can be used and some examples of metrics follow:

1) **Spread:** It is used as a statistical metric such as the chi-square distribution to measure "spread" along the Pareto front.
2) **General Distance:** Estimates how far is our current Pareto front from the true Pareto front of a problem using Euclidean distance (measured in objective space)
3) **Coverage:** Measures the size of objective value space area covered by a set of non dominated solutions.

GREEDY ALGORITHM

A greedy algorithm is implemented on the ‘next best’ search approach. The principle is that the element with the maximum weight is taken first, followed by the element with the second highest weight and so on, until a complete solution is constructed [9].

ADDITIONAL GREEDY ALGORITHM

The additional greedy algorithm is a kind of greedy algorithm but adopts a different strategy. The feedback from previous selections is combined. According to [9] it iteratively selects the maximum weight element of the problem from the part not already consumed by previously selected elements. The cost cognizant version of the two objective formulation was implemented for this algorithm.

HILL CLIMBING ALGORITHM

Hill climbing is local search algorithm. It has two variations in strategy: steepest ascent and next best ascent. In this project steepest ascent approach for Hill Climbing is adopted which comprises of the following steps, taken from Li and Harman [9].

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.

NSGA-2 ALGORITHM

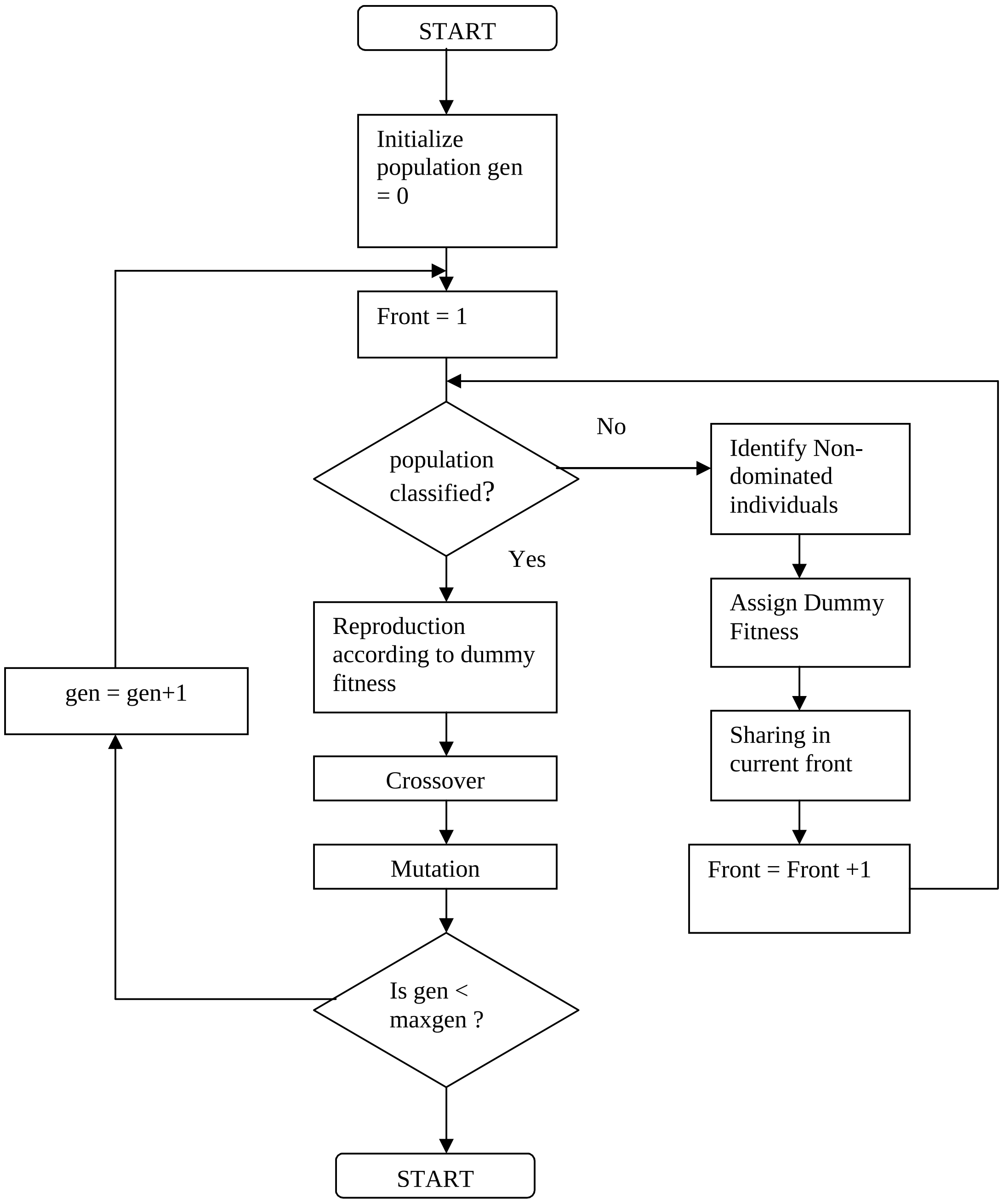


Fig 1.7 Nondominated Sorting Genetic Algorithm (NSGA) flowchart

NSGA was proposed by Srinivas and Deb in 1994. NSGA is based on several layers of classifications of individuals. Nondominated individuals get a dummy value and then are removed from the population. The process is repeated until the entire population has been classified [3]. Classified individuals are shared with their fitness value to maintain diversity.

NSGA-2 was introduced by Srinivas and Deb [11] as a new version to NSGA [12]. The main changes were as follows:

1) NSGA-2 is more efficient (computationally).
2) It uses elitism.
3) A crowded comparison operator is used which keeps diversity.

vNSGA-2 ALGORITHM

vNSGA-2 algorithm is an island generic algorithm variant of NSGA-2 and was developed by [2]. vNSGA-2 makes two major modifications to NSGA-2. First, the algorithm uses a group of sub-populations which are separate from one another, to achieve a wider Pareto Frontier. It performs a pairwise tournament selection on individuals that from a non dominated pair and each of the subpopulations prefers different objectives inorder to advance the Pareto frontier in all the directions. Also, it improves the elitism of NSGA-2.

SPEA-2 ALGORITHM

SPEA was introduced in 1999 by Zitzler & Thiele. It main characteristic are that it stores non dominated solutions externally in a second; continuously updated population, it evaluates an individual's fitness dependent on the number of external nondominated points that dominate it, it preserves population density using the Pareto dominance relationship and it incorporates a clustering procedure in order reduce the nondominated set without destroying its characteristics [10]. The clustering technique is called "average linkage method".
A revised version of SPEA was proposed by Zitzler et al. [8] called SPEA2. It has three main differences with regards to SPEA [3]:
1) SPEA2 incorporates a fine grained fitness assignment strategy which takes into account for each individual the number of individuals that dominate it and the number of individuals which it dominates.
2) It uses a nearest neighbour density estimation which results in a more efficient search.
3) Its enhanced archive truncation method, guarantees the preservation of boundary solutions.

NEW ALGORITHM

A new evolutionary algorithm will be introduced to multiobjective optimization. This algorithm will be a combination of NSGA-2 and Hill climbing approaches.

SUBJECTS

In the project a total of eight C programs were studied. Four small programs Print_tokens, Print_tokens2, Schedule, Schedule2 assembled by researchers at Siemens Corporate Research were studied together with four large programs, Space Sed, Grep and Flex. Space was developed by European Space Agency and Sed is the UNIX stream editor string processing utility. Grep (global regular expression print) and Flex (Fast Lexical Analyzer) are both UNIX utility programs [7]. Grep utility searches a file to see if it contains a specified string of characters and Flex is a lexical analyzer generator. These programs together with coverage information and test suites are available from SIR online repository.

DESIGN AND IMPLEMENTATION

RESEARCH QUESTIONS

The research questions are as follows:
1. Do there exist situations where the Pareto efficient approaches produce more points on the Pareto front than the greedy or hill climbing algorithm?
2. How well do the algorithms perform compared to one another and to the global optimum for the 2-objective formulation?
3. What can be said about the shape of the Pareto Optimal frontiers, both approximated and optimal? What insights do they reveal concerning the tester's dilemma as to how to balance the trade-offs between objectives?

ARTIFACTS

The project will present results from an empirical study of 8 programs ranging from 374 to 11,148 lines of code. The results of these algorithms, when applied to two objective version of the problem using, as subjects four programs from the Siemens suite [4], together with **space** developed for the European Space Agency. The Siemens programs used are **print tokens**, **printtokens2**, **schedule** and **schedule2**. In addition to these three larger programs will be tested Flex, Grep and **SED**. They were obtained from SIR - Software-artefacts Infrastructure Repository [6].

Table 1
Experiment Objects

Program	Lines Of Code	Test Pool Size	Avg. test suite size
print_tokens	726	4130	16
print_tokens2	570	4115	17
schedule	412	2650	8
schedule2	374	2710	8
space	6199	13585	153

Table 2
Additional Experiment Objects

Program	Lines Of Code	Test Pool Size
flex	10459	567
grep	10068	809
sed	11148	1293

OBJECTIVES

The project instantiates the two objective formulation, with code coverage as a measure of test adequacy and execution time as a measure of cost. Code coverage or the statement coverage is the amount of statements in the program covered by the test suite. A statement is any line in the program.

EFFECTIVENSS MEASURE

To address the research questions there has to be a measure which determines the effectiveness of each Pareto frontier. In order to optimize both cost and coverage the additional greedy algorithm will not only be able to measure cost but coverage per unit time.

ANALYSIS TOOLS

Zheng Li, provided the coverage analysis data, which was obtained using a commercial tool called Canatata++. Shin Yoo provided the execution cost data for Siemens suite and Space program which was obtained via Valgrind.
The coverage data and execution data for SED, Grep and Flex will be obtained using Cantata++ and Valgrind.

EXPERIMENTAL DESIGN

Each program has a large number of available test suites. Four test suites were randomly selected for each program. So, a total of 32 test suites were used as input to the multi-objective Pareto optimization. Statement coverage and execution time were used as objectives.

ALGORITHM DESIGN AND IMPLEMENTATION

A total of eight algorithms will be used for multi-objective test case selection techniques:

- Random
- Greedy
- Additional Greedy
- Hill Climbing
- NSGA-2
- vNSGA-2
- SPEA-2
- New Algorithm – hNSGA-2

Exhaustive search was used to locate the true Pareto frontier for the Siemens suite. For the Space Program the reference Pareto frontier was formed was formed differently.
Random algorithm was used as a benchmark to measure the effectiveness of other algorithms. If an algorithm had a pareto frontier close to the reference frontier, it was atleast somewhat successful.
The **Greedy** algorithm will be implemented using the next best approach. The cost cognizant version of the **Additional greedy** algorithm will be implemented. Greedy and Additional Greedy algorithms would first use the quick sort algorithm to sort the test cases.
Hill Climbing algorithm will be used with steepest ascent. For each execution the hill climbing algorithm used the random algorithm to generate an initial solution containing **n** test cases.
Population size is set at 100 for all programs.
NSGA-2 [11] and **vNSGA-2** algorithms will be executed 20 times for each test suite to account for their inherent randomness. Two major modifications were made to NSGA-2 for vNSGA-2. First, the algorithm uses a group of sub-populations to achieve a wider Pareto frontier and it extends the elitism of NSGA-2. NSGA-2 will be

configured with the recommended setting of {population=100, and maximum fitness evaluation = 25,000}. Also, it will use a single point crossover and bit-flip mutation.
SPEA-2 algorithms will be implemented as mentioned in [8].
hNSGA-2 is an algorithm that combines the best features of NSGA-2 and hill climbing. The solutions obtained from the last Pareto front produced by NSGA-2 are inserted into a hill climber which tries to find a better front.

EVALUATIONS OF RESULTS

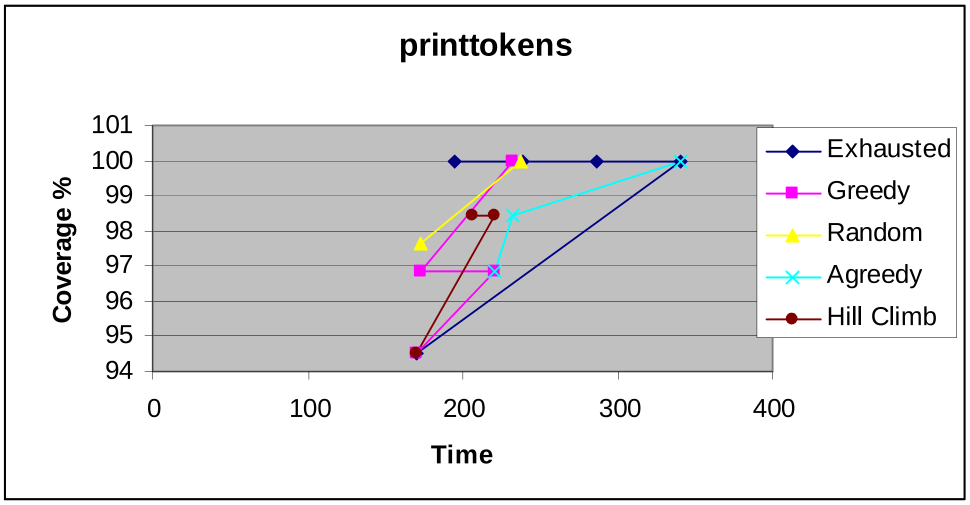


Fig 1.2 printtokens

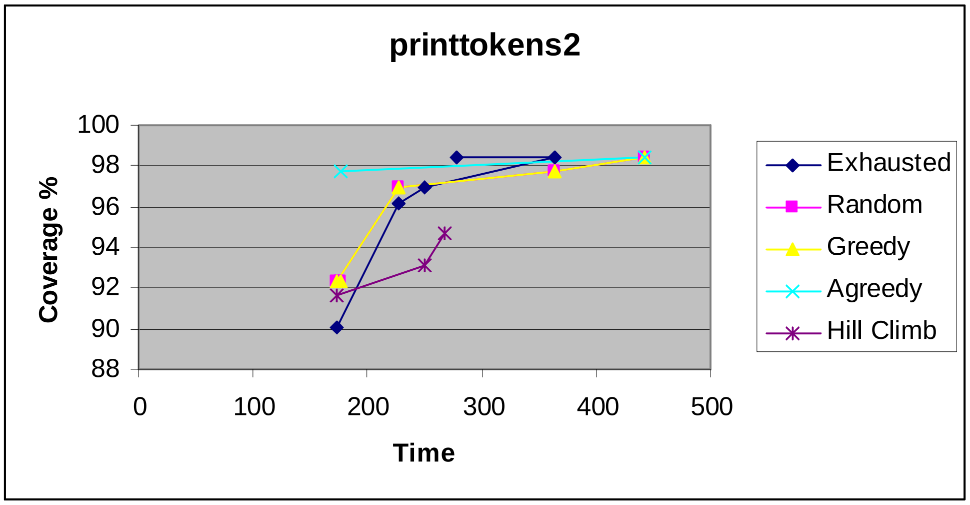


Fig 1.3 printtokens2

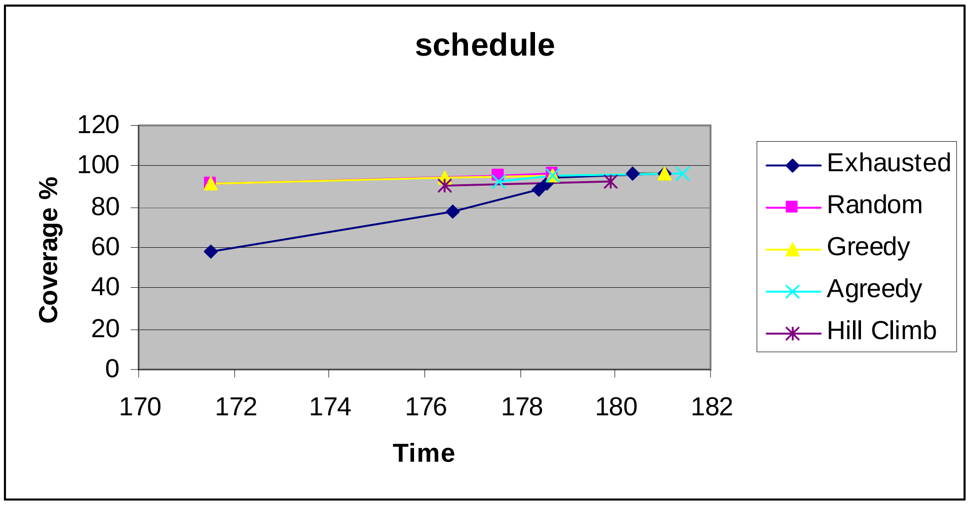


Fig 1.4 schedule

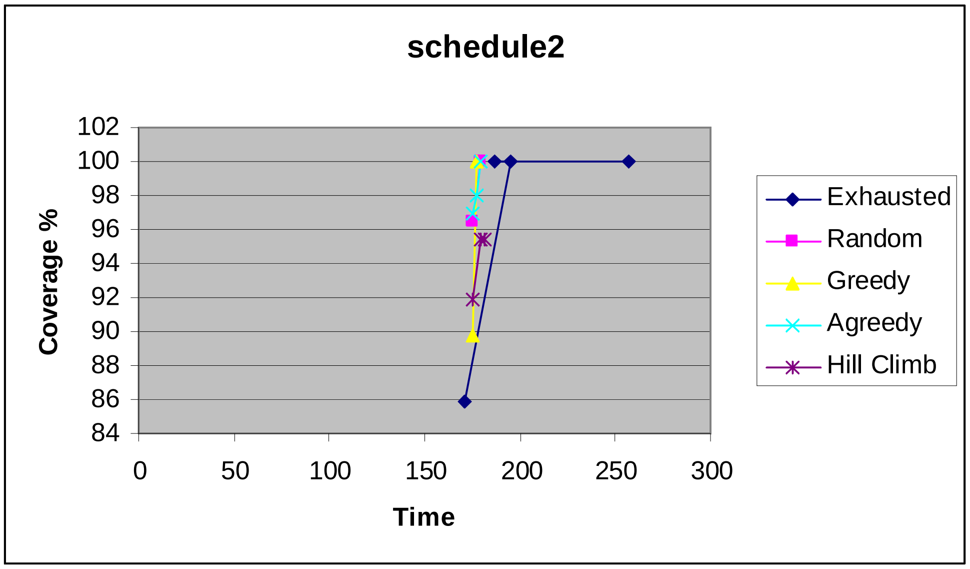


Fig 1.5 schedule2

EVALUATIONS OF RESULTS OBTAINED

Across the different results obtained Additional Greedy Algorithm performs better than all the other algorithms. This is because Additional Greedy algorithm only selects test cases that cover new aspects of the program.
Greedy algorithm performs worst than Additional Greedy Algorithm. Random algorithm performs the worst in nearly all the artifacts. The reason for this is that it selects new test cases that merely cover what has already been covered.
Hill Climbing algorithm has performed well in some of the artifacts. The main problem with its performance is that the solutions obtained are local optima and other better solutions are available. It does perform better than Greedy and Random algorithms.
In the Fig 1.2 printtokens, Additional Greedy algorithm outperforms all the other algorithms studied. While in Fig 1.3 printtokens2, both Greedy and Additional Greedy algorithms perform well. In Fig 1.4 schedule and Fig 1.5 schedule2, all algorithms have performed well. Space artifact could not be checked because of a file opening error.

FUTURE WORK

Pareto Optimality has been treated as single objective optimization problem in previous works. This project showed the results achieved by some of the algorithms Random, Greedy, Additional Greedy and Hill Climbing. Some of the objectives of this project could not be met which could have been helpful in further our understanding of multi-objective test case selection. Future work will consider the artifacts proposed in this project tighter with other meta-heruistic techniques. This project focused on Multi Objective test case selection, other studies could use test case prioritization to further the work done in this project.

This project has described some of the benefits of Pareto Optimality and presented empirical study that investigates the effectiveness of some algorithms. This paper described four algorithms for multiobjective test case selection. It presented results of an empirical study to investigate there effectiveness. The objectives of the projects were to introduce 8 algorithms and 3 artifacts. This project has introduced four algorithms.

Across the different programs considered, the Additional Greedy algorithm performed the better than the rest.

REFERENCES

[1] Gregg Rothermel and Mary Jean Harrold. Analyzing Regression Test Selection Techniques. IEEE transactions on Software Engineering, V.22, no. 8, August 1996, pages 529-551
[2] Shin Yoo and Mark Harman. Pareto Efficient Multi-Objective Test Case Selection. ISSTA '07, July 9-12, London, U.K.
[3] Carlos A. Coello Coello. Metaheuristics for Multiobjective Optimization.
[4] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow and control-flow-based test adequacy criteria. In Proceedings of the 16th International Conference on Software Engineering, pages 191-200. IEEE Computer Society Press, May 1994.
[5] Sebastian Elbaum, Alexey G. Malishevsky and Gregg Rothermel. Test Case Prioritization: A Family of Empirical Studies. IEEE transactions on software engineering, Vol. 28, No. 2, February 2002
[6] H. Do, S.G.Elbaum and Gregg Rothermal. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering: An International Journal, 10(4):405-435,2005.
[7] Mark S. Sobell. A Practical Guide To The Unix System, 3rd Edition.
[8] Eckart Zitzler, Marco Laumanns and Lothar Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. CIMNE Barcelona, Spain, 2002.
[9] Zheng Li, Mark Harman and Robert M. Hierons. Search Algorithms for Regression Test Case Prioritization.
[10] Eckart Zitzler and Lothar Thiele. Multiobjective Evolutionary Algorithms: A comparative Case study and the strength Pareto Approach. IEEE Transactions on evolutionary computation, Vol. 3, No. 4, November 1999.
[11] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A Fast and Elitist Multi-Objective Genetic Algorithm: NSGA-2. KanGAL Report No. 200001
[12] N.Srinivas and K. Deb. Multi-Objective function optimization using non-dominated sorting genetic algorithms. Evolutionary Computation, Vol. 2, pp. 221-248, 1995
[13] Mark Harman and Bryan F. Jones .Search-Based Software Engineering
[14] http://en.wikipedia.org/wiki/Metaheuristic
[15] http://en.wikipedia.org/wiki/Evolutionary_algorithm
[16] http://en.wikipedia.org/wiki/Genetic_algorithm
[17] http://en.wikipedia.org/wiki/Fitness_function
[18] David W. Corne, Joshua D. Knowles, Martin J. Oates The Pareto Envelope-based Selection Algorithm for Multiobjective Optimization

APPENDIX

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


#define SMALL 50
#define LARGE 100

#define EXT "executiontime"
#define SCH1 "schedule1"
#define SCH2 "schedule2"
#define PRI1 "printtokens1"
#define PRI2 "printtokens2"
#define SPAC "space"

#define FLEX "flex"
#define GREP "Grep"
#define SED "Sed"

#define STA "statement_cov"

#define NAMESIZE 30



int exhaustive(char **testSuiteCoverage, char **testSuite, int *solution, int noLines, int
testSuitePop, int totalNoCovered,double time[]);
int random(char **testSuiteCoverage, char **testSuite, int *solution, int noLines, int
testSuitePop, int totalNoCovered,double time[]);
void greedy(char **testSuiteCoverage, char **testSuite, int *solution, int noLines, int
testSuitePop, int totalNoCovered,double *time);
void additionalGreedy(char **testSuiteCoverage, char **testSuite, int *solution, int
noLines, int testSuitePop, int totalNoCovered, double time[]);
void hillClimbing(char **testSuiteCoverage, char **testSuite, int *solution, int noLines,
int testSuitePop, int totalNoCovered, double time[]);
void printResults(char name[], char **testSuiteCoverage, char **testSuite, int solution[], int solutionPop,
int totalNoCovered, int noLinesCovered, int testSuitePop, double time[]);

void quickSort(int numbers[], int positions[], int array_size);
void qsort(int numbers[], int positions[], int left, int right);

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 *extTimeName; //input Execution Time name
char *suiteName; //suite type name
char *testName; // for test name
char universeFile[35] = "./";
char matrixFile[35] = "./";
char suiteFile[35] = "./";
char timeFile[35] = "./";
int populationNo = 0; //starting population number for the NSGA-2,hNSGA-2,SPEA-2 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("6. Flex\n");
printf("7. Grep\n");
printf("8. Sed\n");
printf("Select input program (1-8): ");
scanf("%d", &progNo);

switch(progNo)
{
case 1:
progName = SCH1;
extTimeName = SCH1;
populationNo = SMALL;
break;
case 2:
progName = SCH2;
extTimeName = SCH2;
populationNo = SMALL;
break;
case 3:
progName = PRI1;
extTimeName = PRI1;
populationNo = SMALL;
break;
case 4:
progName = PRI2;
extTimeName = PRI2;
populationNo = SMALL;
break;
case 5:
progName = SPAC;
extTimeName = SPAC;
populationNo = LARGE;
break;
case 6:
```

```
progName = SPAC;
extTimeName = SPAC;
populationNo = LARGE;
break;
case 7:
progName = SPAC;
extTimeName = SPAC;
populationNo = LARGE;
break;
case 8:
progName = SPAC;
extTimeName = SPAC;
populationNo = LARGE;
break;
default:
progName = SCH1;
extTimeName = SCH1;
populationNo = SMALL;
printf("Using default program - %s\n", SCH1);
}

// --------------------------------Copy Universal Data-------------------------
strcat(universeFile, progName);
strcat(universeFile, "/universe");
//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");

// --------------------------------Statement Coverage Data-----------------------
printf("\n ------------------------Statement Coverage     -------------------------------------\n");
covName = STA;

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("Number Of Lines= %d\n", noLines);
printf("Number Of Tests= %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");
// --------------------------------Coverage Data Copied--------------------------- //


// --------------------------------Time Execution Data--------------------------- //
printf("\n ------------------------Time Execution Data------------------------------------\n");



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

//int tnoLines=0;
int tnoTests=0;

tnoTests = noTests;

printf("Number of Execution Data = %d\n", tnoTests);
inFile = fopen(timeFile, "r");


char line[8];
double* time=NULL;
time = new double[tnoTests];

int counter1=0;

double timeData;
```

```
while(fgets(line, 8, inFile) != NULL ) //get the whole line of text
{

            //convert the string to a float
            timeData=atof(line);

            if(timeData != 0.0)
            {
    time[counter1]=timeData;
            counter1++;
            }
  }

// -------------------------------Time Execution Data Copied-----------------------   //
printf("\n -----------------------Select Test Suite Size------------------------------\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 line 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++;
}
}
// --------------------------------Test Suite Data Copied------------------------   //
//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 selected test suite: %d\n", testSuitePop);
printf("Number of lines in the program: %d\n", noLines);
printf("Number of lines covered by the selected test suite: %d\n",totalNoCovered);

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. Hill Climbing algorithm\n");
printf("5. NSGA-2 algorithm\n");
printf("6. vNSGA-2 algorithm\n");
printf("7. hNSGA-2 algorithm\n");
printf("8. Exhaustive 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,time);
break;
case 2:
printf("\nGreedy algorithm selected\n");
greedy(testSuiteCoverage, testSuite, solution, noLines,testSuitePop, totalNoCovered,time);
break;
case 3:
printf("\nAdditional Greedy algorithm selected\n");
additionalGreedy(testSuiteCoverage, testSuite, solution, noLines,testSuitePop,totalNoCovered, time);
break;
case 4:
printf("\nHill Climbing algorithm selected\n");
hillClimbing(testSuiteCoverage, testSuite, solution, noLines,testSuitePop, totalNoCovered,time);
break;
case 5:
printf("\nNSGA-2 algorithm selected\n");

break;
case 6:
printf("\nvNSGA-2  algorithm selected\n");

break;
case 7:
printf("\nhNSGA-2 algorithm selected\n");

break;
case 8:
printf("\nExhausted Search algorithm selected\n");
exhaustive(testSuiteCoverage, testSuite, solution, noLines,testSuitePop, totalNoCovered,time);
break;
default:
printf("\nDefault algorithm (Random) selected\n");
random(testSuiteCoverage, testSuite, solution, noLines,testSuitePop, totalNoCovered,time);
}

return 0;
}


/******************************** Exhaustive Algo ********************************/
int exhaustive(char **testSuiteCoverage, char **testSuite, int *solution, int noLines, int
testSuitePop, int totalNoCovered,double time[])
{
printf("\nEXHAUSTIVE ALGORITHM\n");

for(int i=0;i <testSuitePop;i++)
{
        solution[i] = i+1;
}

int index = testSuitePop;

int noLinesCovered = 0;
```

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




/******************************** Quicksort Algo ********************************/
void quickSort(int numbers[], int positions[], int array_size)
{
qsort(numbers, positions, 0, array_size - 1);
}
void qsort(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)
qsort(numbers, positions, left, pivot-1);
if (right > pivot)
qsort(numbers, positions, pivot+1, right);
}


/******************************** Random Algo ********************************/

int random(char **testSuiteCoverage, char **testSuite, int *solution, int noLines, int
testSuitePop, int totalNoCovered, double *time)
{
```

```
printf("\nRANDOM ALGORITHM\n");
int randomInteger = 0; //random integer
int noLinesCovered = 0; //number of lines 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;
randomInteger = rand() % testSuitePop;
printf("Test case: %d ", randomInteger);
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[randomInteger][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] = randomInteger; //remember each test case selected
for(i=index-1; i>=0; i--)
{
if(solution[i]==randomInteger) //if the test case has already been selected then remove it
{
printf(" (Test Case Duplication - Exclude TC)");
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, time);
return index;
}

/******************************** End of Random Algo ********************************/
```

```
/******************************** Greedy Algo *********************************/
void greedy(char **testSuiteCoverage, char **testSuite, int *solution, int noLines, int
testSuitePop, int totalNoCovered, double *time)
{
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(" Total nos lines covered: %d\n", noLinesCovered);
pos--;
solutionPop++;
}
printResults("Greedy", testSuiteCoverage, testSuite, solution, solutionPop,totalNoCovered,
noLinesCovered, testSuitePop,time);
}
```

```
/******************************** End of Greedy Algo ********************************/



/******************************** Additional Greedy Algo******************************/
void additionalGreedy(char **testSuiteCoverage, char **testSuite, int *solution, int
noLines, int testSuitePop, int totalNoCovered, double *time)
{
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 numberduring 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(" Total nos lines covered: %d\n", noLinesCovered);
}
printResults("Additional Greedy", originalCoverage, testSuite, solution,
solutionPop, totalNoCovered, noLinesCovered, testSuitePop, time);
}
/********************************* End of Additional Greedy Algo *********************/

/******************************** Hill Climbing Algo********************************/
void hillClimbing(char **testSuiteCoverage, char **testSuite, int *solution, int noLines,
int testSuitePop, int totalNoCovered,double *time)
{
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,time); //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, time);
}
/********************************* End of Hill Climbing Algo*************************/

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

void printResults(char name[], char **testSuiteCoverage, char **testSuite, int
solution[], int solutionPop, int totalNoCovered, int noLinesCovered, int testSuitePop,double time[])
{


bool* lineCovered = NULL;

lineCovered = new bool[totalNoCovered];

for (int l=0;l>totalNoCovered;l++)
{
        lineCovered[l]=false;
}

int finalPop=testSuitePop;
float* perCovered=NULL;
perCovered = new float[solutionPop];
float* timeCovered=NULL;
timeCovered = new float[solutionPop];


for (int y =0;y<solutionPop;y++)
        {
                noLinesCovered = 0;
                for (int j=0; j<(totalNoCovered+2); j++)
                {
                        if(testSuiteCoverage[solution[y]][j] =='1')
                        {
                                lineCovered[j] = true;
                        }

                }
                for(int x=0; x<(totalNoCovered+2);x++)
                {
                        if(lineCovered[x] == true)
                        noLinesCovered++;
                }



                float floatNoCovered = (float)noLinesCovered;
                float floatNoLines = (float)totalNoCovered;
                perCovered[y]= (floatNoCovered/floatNoLines)*100;
                timeCovered[y]=(float)time[solution[y]];
        }




printf("\n\n********* %s Algorithm Results *********\n", name);
printf("\n\n--------- %s Non-Dominated Solutions---------\n", name);


/******************************** Non Dominating Sorting********************/

float* ndArrayCov=NULL;
ndArrayCov = new float[solutionPop];
float* ndArrayTime=NULL;
ndArrayTime = new float[solutionPop];



int k =0;
int counter=0;

for (int i=0;i<solutionPop;i++)
{

for (int j=0;j<solutionPop;j++)
{


        if(i!=j)
        {
//printf("\t %d %d \t",j,i);
if ((perCovered[j]>perCovered[i]) && (timeCovered[j]<timeCovered[i]))
{
//This is only to test if the coverage is there or not
}else{counter++;}

        } // end of if statement for i is not equal to j

}

if(counter==(solutionPop-1))
{
ndArrayCov[k] = (float)perCovered[i];
ndArrayTime[k] = (float)timeCovered[i];
k++;
} //end if for counter
        counter=0;
}

//Display the result on the screen
printf("Coverage \t");
printf("Time \n");
```

```
for( int nt=0; nt< solutionPop;nt++)
{
            if(ndArrayTime[nt]>0 && ndArrayCov[nt]>0)
            {

printf("%.3f%8s",ndArrayCov[nt],"");
printf("\t %.3f\n",ndArrayTime[nt]);

            }
}

/********************************End of Non Dominating Sorting*********************/
FILE *inFile;
inFile = fopen("result.dat", "w"); // Writing Data to a File


fprintf(inFile,"Coverage \t");
fprintf(inFile,"Time \n");
for( int t=0; t< solutionPop;t++)
{
            if(ndArrayTime[t]>0 && ndArrayCov[t]>0)
            {

fprintf(inFile,"%.3f",ndArrayCov[t]);
fprintf(inFile,"\t %.3f\n",ndArrayTime[t]);

            }
}

 fclose(inFile);

}

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