Test Suite Analysis: Minimization and Ordering

ENSE 623: Systems Engineering
Dr. Mark Austin

Fred Faber
Julie McNeil

December 8, 2004

Table of Contents

Index of Figures

Index of Tables

Abstract

Our research has two important aspects. The first focuses on the issue of test suite minimization. We developed a tool based on a LP to aid in the selection of test cases. The model seeks to maximize the error detection rating and the coverage factor of the selected test suite, while keeping within a given time and cost budget. As an extension, we next considered how to order the given set of test cases. We developed an algorithm which orders test cases based on the goal of minimizing the time to identify failures, thus minimizing the overall test cost.

1 Introduction

This document begins with an overview of the verification testing process and the motivation for our project. Next, relevant research is discussed that contributed to our focus area. This is followed by a detailed discussion of the Test Suite Minimization tool, including theory, approach, implementation, and its integration with the Dynamic Requirements Selector tool. We then discuss a new Test Suite Prioritization technique, including theory, implementation and results. Finally, we present our conclusions and areas for future research.

1.1 Overview of Verification Testing Process

Once a system has been designed and developed, it must undergo verification. This is the process of ensuring that the system and all its components meet the requirements and specifications of the design. The goal of the verification testing phase is to find all errors (incorrect internal system state) and failures (incorrect external behavior of system) and fix the underlying faults, or causes (Waters 1991).

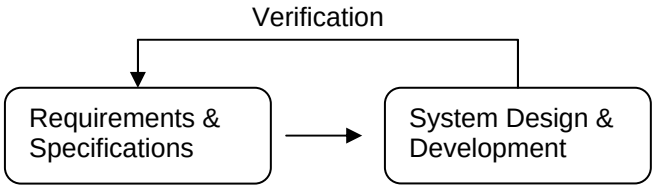


Figure 1: Verification Process

This testing process must be structured and comprehensive to facilitate detection of all system issues. The standard practice is to test first each component of the system at the unit-level. The next level of testing is system integration testing (SIT), in which tests are run that verify several

requirements at once. These tests are called test cases, and they are comprised of a set of execution conditions and inputs for a particular objective. The test results are checked against the expected results to verify compliance for the applicable requirements. The set of test cases used for verification of a system is called the test suite. The test suite must cover all requirements. The verification stage is not complete until every test case in the suite has passed successfully.

2 Test Suite Minimization

2.1 Motivation

When testing a system, the end goal is to test all requirements thoroughly. Ideally, the test suite will detect errors quickly and in a manner in which they could be assigned to root causes and resolved efficiently. One method of testing is to cover extensively every single requirement atomically and every potential interaction. This setup, however, is not feasible for large-scale systems as the number of required test cases in the suite would grow unwieldy and become cost prohibitive. Of course, if the test suite does not completely cover all requirements, then errors may remain undetected. The test manager must trade-off these issues when selecting the appropriate test cases for a test suite. The goal of our project is to develop a tool to aid in this selection process.

2.2 Literature Review

In recent years, there has been an increase in research focus on test suite analysis. Much of this research concentrates on how to reduce the size of test suites. Chen et al focused on reducing test suite size for software systems [1998]. They reviewed a greedy heuristic that reduces the suite size by selecting test cases based on the degree of 'essentialness.' This ranking is derived from a factor that measures the redundancy of test cases. They also created a new heuristic to improve upon the base heuristic. Cohen et al focused their research on reducing test suite size for test cases that involved interaction testing [2003]. They reviewed applicable heuristics and developed an integrated approach for finding coverage arrays. Of particular interest was the recent research by Black et al in which an LP model was developed to determine the optimal set of test cases [2004]. They developed a bi-criteria model which maximized error detection while minimizing the test suite size. Their research did not focus on any factors regarding time or costs.

2.3 Test Suite Minimization Theory

After reviewing this research, we decided to build upon the bi-criteria model by including additional factors beyond simple error detection. The scenario of interest is that in which an existing system is upgraded with new features. Therefore, during the test phase, testing must be done on both the new requirements and the existing functionality of the system. Because the system is being extended, there are existing regression test cases that could be used again. These cases, however, may not be the most efficient. Furthermore, they may cover requirements that are no longer valid. Finally, new test cases must be developed for the new functionality. The test manager must sort through all these test cases and select the few that will make up the test suite.

Unfortunately, the bi-criteria model is difficult to use in the situation of selecting new, unproven test cases. Estimates can be provided on the predicted error detection factor of each test case, but this can produce mixed results. By adding new aspects to the bi-criteria minimization model, the selection is based on a well-balanced set of factors and is not skewed by the use of just one factor.

2.3.1 Terminology

A discussion of the some of the key terminology that is used in this paper is provided below.

Error detection rating: The measure given to each test case as to the estimation of errors that will be detected. This is based on historic performance of the test case or an educated prediction. The higher the rating, the more likely the test case is to reveal errors. This rating is given at the test case level, not per requirement in the test case.

Coverage rating: The measure given to each requirement in a test case as to how well it is tested by that particular test case. This is a subjective rating based on the thoroughness of the test case to test each particular requirement. As some requirements are complex, it is clear that not every test case will cover the requirement to the same degree. Some may cover the basics of the requirement, but not all the intricacies involved. This measure enables differentiation of test cases based on requirement coverage levels.

Requirement importance rating: The measure given to each requirement as to its importance for testing. Essentially, this is a rating that is assigned to each requirement. The higher the rating, the more important that requirement is. Features that are crucial to system performance can, and should be tested more rigorously to ensure their success. Including this factor in test suite selection enables extensive testing for the most important requirements.

2.3.2 Test Suite Minimization Approach

Our goal was to develop a tool to select an optimized set of test cases from a larger set. We wanted this tool to perform a comprehensive analysis of test cases. Further, we felt it should consider not only the error detection rating, but also the coverage rating of requirements in the test cases. Additionally, we wanted the tool to select those test cases which most rigorously test the more important requirements of a system.

We based the logic for the tool on a linear program to optimize the test suite selection. The goal of this was to enable it to compute an optimal test suite when it is given various data parameters as inputs. The following section details this linear program formulation.

2.4 Model Formulation

Due to the binary constraints, the model was formulated as an integer program. The model determines which test cases should be included in the test suite (binary decision variables). The objective of the model is to maximize the overall error detection rating and coverage factor of the selected test suite. The weight of each term in the objective function can be altered based on the test manager's preferences. The coverage factor is a compound term that is based not only on the individual coverage rating of each test case for a requirement, but also how important each requirement is in the overall testing process.

The model is constrained by several issues. First and foremost, every requirement must be covered at least once by the test suite. Additionally, the test suite must be within the time and monetary budgets. The formulation is below:

Indices:

$m \in M$ = Set of requirements

$n \in N$ = Set of test cases

Decision Variables:

T_n = Indicator for test case selection

$TR_{m,n}$ = Indicator for test case selection (requirement-level)

Parameters:

te_m = Error detection rating for each test case

$tv_{m,n}$ = Coverage rating for each requirement in test case

$tn_{m,n}$ = Indicator that test case covers the requirepment

rv_n = Coverage importance rating for each requirement

tc_m = Total cost of test case

tt_m = Total time to run a test case

t = Total time budget for test phase

c = Total monetary budget for test phase

w' = Weight of error detection term in objective function

w'' = Weight of coverage factor term in objective function

Objective Function:

Maximize test suite error detection and coverage rating

$$w' \cdot \sum_{m \in M} te_m \cdot T_m + w'' \cdot \sum_{m \in M, n \in N} tv_{m,n} \cdot rv_{m,n} \cdot TR_{m,n}$$

Subject to:

If a test case is selected, all requirements in the test case are also selected:

$$T_{m,n} \cdot tn_{m,n} \leq TR_{m,n} \text{ for all } m \in M, n \in N$$

The test suite must cover all requirements at least once:

$$\sum_{m \in M} TR_{m,n} \geq 1 \text{ for all } n \in N$$

Within monetary budget for test phase:

$$\sum_{m \in M} tc_m \cdot T_m \leq c$$

Within time budget for test phase:

$$\sum_{m \in M} tt_m \cdot T_m \leq t$$

Define variable types :

$TR_{m,n}$ *is binary* $(0,1)$ for all $m \in M, n \in N$

T_m *is binary* $(0,1)$ for all $m \in M$

2.5 Tool Implementation

The model was developed using Excel. Details for each test case and requirement are entered into simple data tables (see below tables), which are then used to populate more extensive tables to construct the constraints. This information is passed to the Excel Solver which returns the optimal test suite to the user.

TC #	Test Case	Cost Estimate	Time Estimate (hours)	Error Detection Rating	Requirement Coverage: Indicator of coverage and respective rating					
					1		2		3	
					Incl.?	Cvg.	Incl.?	Cvg.	Incl.?	Cvg.
1	Generic (1, 4, 6, 10)	$45,000	90	25	1	50				
2	Generic (2, 3, 5, 8, 9)	$98,000	180	50			1	50	1	50
3	Generic (1, 3, 7, 11, 12)	$85,000	105	50	1	75			1	75
4	Generic (3, 6, 8, 12)	$65,000	110	50					1	75
5	Generic (2, 5, 6, 12)	$25,000	140	25			1	50		
6	Generic (1, 4, 9, 10)	$80,000	180	75	1	25				
7	Generic (2, 4, 5, 9, 11)	$40,000	80	50			1	50		
8	Generic (3, 5, 8, 12)	$30,000	95	25				25	1	25
9	Generic (2, 6, 8, 11, 12)	$85,000	180	50			1	25		
10	Generic (4, 7, 8, 9)	$45,000	200	50						
11	Generic (5, 7, 10, 11)	$28,000	120	75						
12	Generic (1, 5, 8, 10)	$15,000	80	25	1	50				

Table 1: Excerpt of Test Case Data Input Table

R#	Requirement/Features	Test Coverage Importance
1	Extended battery life (200 min talk, 200 hours standby)	8
2	Digital camera with zoom and flash	7
3	Color display (65,000 TFT)	3
4	MS Pocket software, Media Player	7
5	Mobile Web	3
6	Ultra Lightweight (under 3 oz)	4
7	Speaker phone capability	5
8	32 MB RAM	6
9	Flip phone-style	7
10	FM Radio and MP3 player	4
11	Polyphonic MIDI/iMelody Sound	3
12	Alarm clock, calculator, currency converter	3

Table 2: Excerpt of Requirement Data Input Table

The model efficiently handles small scale problems. Due to the linear nature and simplicity of constraint construction, larger scale problems should not cause any issues. Commercial optimization packages can easily solve this model for requirement and test case sizes in the thousands.

2.6 Integration with Dynamic Requirements Selector

2.6.1 Overview of DRS

In our prior research, we developed the Dynamic Requirements Selector tool. Given that a major issue in requirements engineering is the business trade-off analysis among system requirements, we developed a tool to facilitate this process. The particular scenario for use is that of a system upgrade in which new features are to be added to an existing system. With constraints concerning time, budget and other technical issues, not every feature can be selected for inclusion in the system. As such, the system stakeholders must perform trade-off analysis among the proposed features to maximize the utility of the new system design.

The tool considers conflicting interests of various stakeholders, budget limitations, and interaction of requirements. Of particular interest are the relationships among requirements. For instance, if one requirement is selected, it might reduce the cost of implementing another particular requirement. Or, perhaps the inclusion of one requirement increases the importance of another. The below figure helps illustrate the principles of the model.

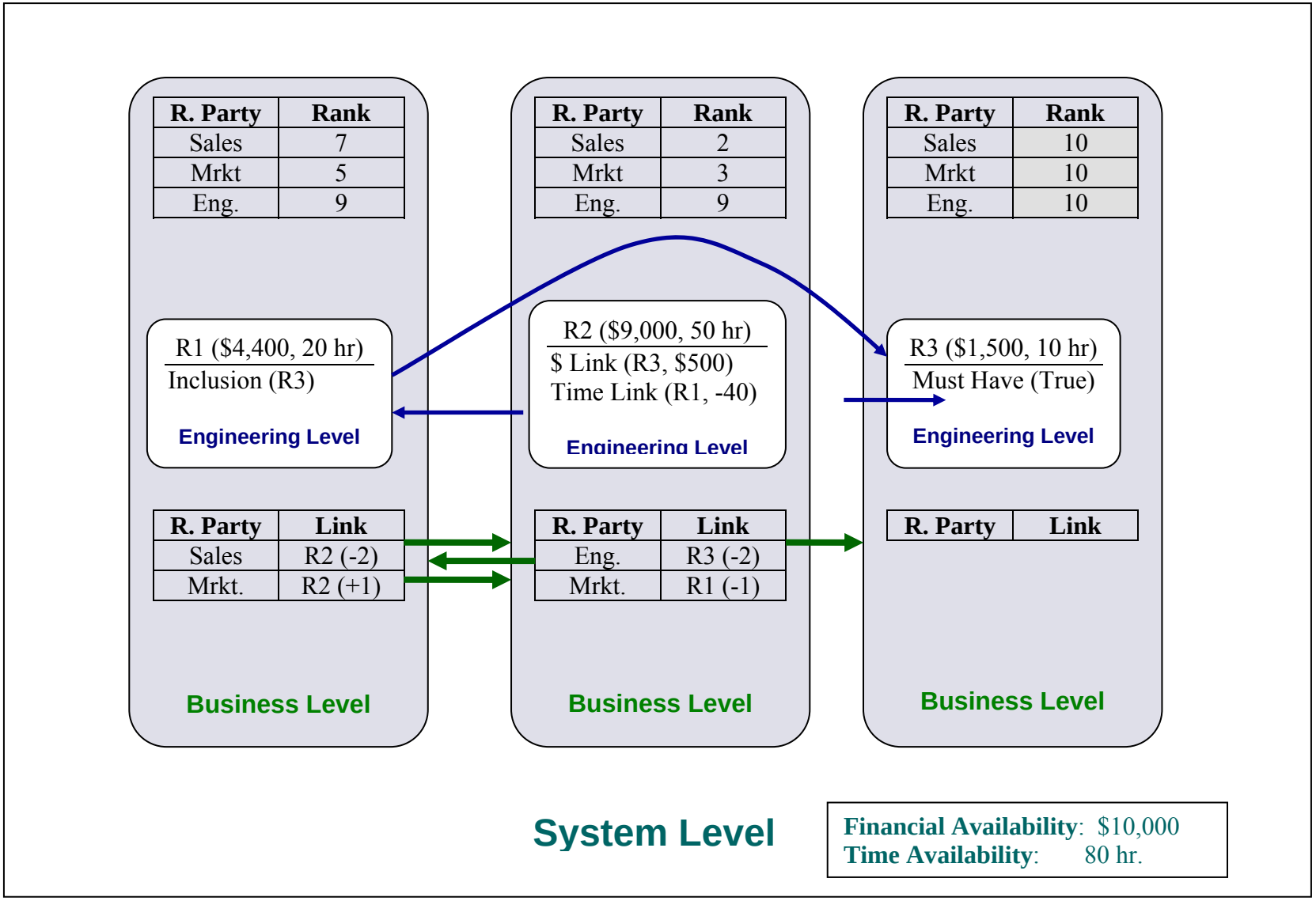


Figure 2: Details of DRS Concept

Given these constraints, a non-linear model was developed to select the optimal requirements for implementation. The model seeks to maximize the total importance rating of the requirements selected, while maintaining the constraints of the project. Clearly, the dependencies of requirements impose the non-linearity aspect on the model.

2.6.2 Integration of Dynamic Requirements Selector and Test Suite Minimizer

By integrating the two models, a powerful model was created that selects not only the requirements, but also the optimal test suite for those requirements. This model is particularly useful for certain systems where test costs are excessively high. Thus, it would make sense to consider the testing cost when selecting the requirements.

The resulting model essentially keeps all the constraints from the individual models. The objective function now considers three terms – the requirements ranking, the error detection rating and the coverage factor of the test suite. The decision variables and constraints are linked through the addition of the requirement selection indicator in the calculations for test suite selection. Thus, requirements that are not selected do not need to be tested. As a side note, a selected test case may include extraneous requirements that were not selected. For the purposes of this model, we assumed those aspects of the test case could be ignored and would not impact the cost/time or error detection rating of the test case.

2.6.3 Mathematical Formulation of DRS-TSM

The model was formulated as a non-linear program, due to the correlation of rank, time and costs of requirements and the correlation of test cases and requirements. The objective of this model is to maximize the three terms discussed earlier. The decisions are the selection of requirements and test cases for the system (binary variables). The constraints for requirements include time and cost budgets, requirement correlations, inclusion/exclusion relationships and 'Must Have' determinations. The constraints on test cases are similar to the earlier formulation.

Indices:

$m \in M$ = Set of requirements

$n \in N$ = Set of test cases

Decision Variables:

T_n = Indicator for test case selection (0,1)

$TR_{m,n}$ = Indicator for test case selection (requirement-level) (0,1)

R_n = Indicator for requirement selection (0,1)

TK_n = Total rank of requirement n

AC_j = Adjusted cost of requirement j ($j = 1..n$)

AT_j = Adjusted time of requirement j ($j = 1..n$)

AR_{js} = Adjusted rank of requirement j by shareholder s ($j = 1..n$)

Parameters:

Test Cases:

te_m = Error detection rating for each test case

$tv_{m,n}$ = Coverage rating for each requirement in test case

rv_n = Coverage importance rating for each requirement

$tn_{m,n}$ = Indicator that test case covers the requirement

tc_m = Total cost of test case

tt_m = Total time to run a test case

t = Total time budget for test phase

c = Total monetary budget for test phase

w' = Weight of error detection term in objective function

w'' = Weight of coverage factor term in objective function

Requirements:

m_n = Indicator that requirement n is a 'Must Have' (0,1)

k_{ns} = Rank of requirement n by shareholder s

w_{ns} = Weight given by shareholder s for requirement n

c_n = Cost to implement requirement n

t_n = Time to implement requirement n

tc = Total budget for project

tt = Total time for project

in_{nj} = Indicator of inclusion: if req. n is selected, req. j must be included (0,1), ($j = 1..n$)

ex_{nj} = Indicator of exclusion: if req. n is selected, req. j must be excluded (1,2), ($j = 1..n$)

cd_{nj} = Cost differential of req. j if req. n is selected ($j = 1..n$)

td_{nj} = Time differential of req. j if req. n is selected ($j = 1..n$)

rd_{njs} = Rank differential by shareholder s of req. j if req. n is selected ($j = 1..n$)

w''' = Weight of requirement ranking in objective function

Objective Function:

Maximize test suite error detection and coverage and requirement ranking

$$w' \cdot \sum_{m \in M} te_m \cdot T_m + w'' \cdot \sum_{m \in M, n \in N} tv_{m,n} \cdot TR_{m,n} + w''' \cdot \sum_{n \in N} R_n * TK_n$$

Subject to:

Within monetary budget for test phase:

$$\sum_{m \in M} tc_m \cdot T_m \leq c$$

Within time budget for test phase:

$$\sum_{m \in M} tt_m \cdot T_m \leq t$$

A requirement of a test case is factored only when both the requirement and test case are selected:

$$T_{m,n} \cdot tn_{m,n} \cdot R_n \leq TR_{m,n} \text{ for all } m \in M, n \in N$$

The test suite must cover all requirements at least once:

$$\sum_{m \in M} TR_{m,n} \geq 1 \text{ for all } n \in N$$

All 'Must Have' requirements included:

$$R_n \geq m_n \text{ for all } n \in N$$

Adjustment to Rank :

$$AR_{js} = \sum_{n=1}^{n-1} R_n * rd_{nj} + \sum_{n=n+1}^{n} R_n * rd_{nj} \quad \text{for all } j \in N, s \in S$$

Total Rank of Requirement :

$$TK_n = \sum_{s \in S} \left(AR_{js} + k_{ns} \right) * w_{ns} \quad \text{for all } n \in N,\ j \in N$$

Adjustment to Time :

$$AT_j = \sum_{n=1}^{n-1} R_n * td_{nj} + \sum_{n=n+1}^{n} R_n * td_{nj} \quad \text{for all } j \in N$$

Adjustment to Cost :

$$AC_j = \sum_{n=1}^{n-1} R_n * cd_{nj} + \sum_{n=n+1}^{n} R_n * cd_{nj} \quad \text{for all } j \in N$$

All Inclusion Relationships are Satisfied :

$$R_n + R_j \geq 2 * in_{nj} * R_n \quad \text{for all } n \in N,\ j \in N$$

All Exclusion Relationships are Satisfied :

$$R_n + R_j \leq ex_{nj} \quad \text{for all } n \in N,\ j \in N$$

The requirements selected are within monetary budget :

$$\sum_{n \in N(j=n)} \left(AC_j + c_n \right) * R_n \leq tc$$

The requirements selected are within time budget :

$$\sum_{n \in N(j=n)} \left(AT_j + t_n \right) * R_n \leq tt$$

Define variable types :

$TR_{m,n}$ *is binary* $(0,1)$ for all $m \in M, n \in N$

T_m *is binary* $(0,1)$ for all $m \in M$

R_n *is binary* $(0,1)$ for all $n \in N$

Smaller scale models can be solved in Excel using the Solver in an efficient time frame. Due to the complexity of the non-linear program, larger scale problems require a more advanced optimization solver, such as Dash Optimization's Xpress SLP.

2.7 Conclusion

Our primary objective was to create a tool to analyze an existing test suite in order to create a reduced, optimized test suite. We modeled the logic for this tool as a linear program and successfully and implemented it in MS Excel. We then integrated it with an existing tool, the Dynamic Requirement Selector. As such, we successfully completed our research objective.

3 Test Case Ordering Algorithm

Test case prioritization involves sequencing the test cases within a test suite in a manner that will increase some measure of effectiveness. A commonly used metric is the *rate of fault detection*. This is a measure of how quickly faults are detected after the commencement of a test suite run. This is a sensible metric to use as it is desirable to identify faulting components as soon as possible in order to repair them quickly.

An extension of this idea is the goal of identifying one particular faulting requirement with as much certainty as possible. This is advantageous in the case that one faulting requirement causes several test cases to fail, hence masking other potentially faulting requirements. Additionally, this early identification may facilitate a quick repair to the system. This is especially of interest in the case of nightly software builds; being able to identify a faulting requirement quickly may enable a quality-assurance team to contact the responsible engineer before he leaves the office for the day (instead of having to page him in the middle of the night).

An additional facet of a test case prioritization technique is the inclusion of cost-efficiency methods. These methods help address the cost-effectiveness trade-off that is inherently present when selecting and running a test suite.

Given this motivation, the goal of the model presented below is:

Given:

1. A set of test cases
2. A set of requirements that are tested by these test cases
3. That one of these requirements has caused a previously run test case to fail

Do:

1. Determine a cost-minimizing sequence of test cases to run to identify a faulting requirement
2. Update the sequence with new information a test case is run

Terminate when:

1. A goal state fails; when this happens, the requirements associated with this goal state can be examined further to determine exactly which one is faulting (unless the goal state is atomic)

3.1 Literature Review

Most recent studies on test case prioritization focus on ordering an entire test suite at its onset in order to increase the effectiveness of some performance measure. An example of a metric used is the Weighted Average of the Percentage of Faults Detected (APFD) as introduced by Elbaum et al. This metric is an indicator of the rate of fault detections uncovered by the testing process [2000]. There have been several extensions to this work, including that done by Elbaum et al [2002], and that done by Aggrawal et al [2004]. These work all assumes that once a test suite is ordered it will be run until completion.

3.2 Definitions

In addition to the terms described above, the following terms are used throughout the discussion of test suite reduction:

Requirement of Interest: A requirement that is potentially faulting. In other words, it is a requirement that is of interest to test. This is compared to a *requirement not of interest*, which is a requirement that is known not to be faulty.

Test Case Joined Cost: This term refers to the value that is calculated by combining a weighted measure of a test case's financial cost and temporal cost. This is done to create one cost value per test case. For example:

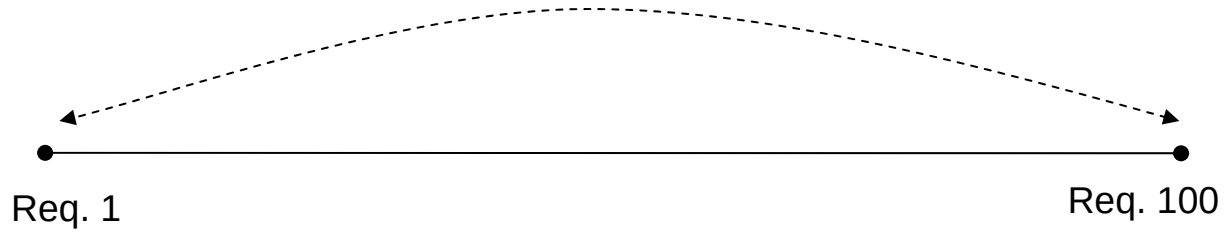
- Test Case X Financial Cost = $30
- Test Case X Temporal Cost = 10 hours
- alpha value = .50
- Test Case X Joined Cost = (alpha) * (*FinancialCost*) + (1 – alpha) * (*TemporalCost*)

- Test Case X Joined Cost = 25

There are no units to this cost as its use is only to compare the cost of one test case to the cost of another test case.

Closeness of Requirements (closeness): This term refers to the relationships among the requirements contained in test case; it is, in fact, a simplified measure of the correlation among requirements in a test case *with respect to* the frequency with which they appear together in a test case. This is defined as a percentage [1-100]. This measure is typically used as input when constructing test cases, as illustrated in the following example:

- Assume the range of requirements in a test suite is 1 - 100. Envision this is a looped continuum such as:



- Assume there are to be 10 requirements included in the test case
- Assume the measure of *closeness* is 90%
- Assume that the test case will contain Requirement 40 (an arbitrary decision, but necessary as a starting point)

Given these parameters, the following calculations will be performed:

1. Pivot Requirement for test case = Requirement 40
2. Number of Requirements eligible to be added to this test case:
    - # of Requirements = *size_test_case* * [100% + (100% - *closeness*)]
    - # of Requirements = 10 * [2 - .90]
    - # of Requirements = 10 * 1.10
    - # of Requirements = 11

3. Range in which requirements may be selected[1]:
    - Range = Pivot +/- [1 + (# of Requirements / 2)]
    - Range = 40 +/- [1 + 11/2]
    - Range = 40 +/ 6

As a result of this *closeness* parameter, among the 9 additional requirements that will be added to this test case will fall within +/- 6 of Requirement 40 (i.e. between Requirement 34 and Requirement 46). If, for instance, Requirement 3 were used instead of Requirement 40 as the pivot, the range would be [Requirement 1 – Requirement 9 UNION Requirement 97 – Requirement 100] (which is why the notion of the looping continuum was introduced above).

Test Case Size Skew: This term refers to the distribution of the sizes of test cases within a suite (i.e. the number of requirements contained in a test case). It, like the *closeness* parameter, is used primarily when constructing a test suite. Measures of *test case size skew* (skew) are used in combination with a normal distribution that is used as a random number generator (RNG). The output of this RNG is the size of the next test case that is to be created (an additional input necessary for this calculation is the total number of test cases that are to be created). The *skew* measure is simply a descriptor of the mean and the standard deviation that are used to create the normal distribution. In the scope of this paper, there is only a discrete set of values used to describe the *test case size skew*:

- VERY_LEFT_SKEW
    - distribution mean = 20% * *total_number_test_cases*
    - distribution standard deviation = 20% * *total_number_test_cases*
- LEFT _SKEW
    - distribution mean = 33% * *total_number_test_cases*
    - distribution standard deviation = 20% * *total_number_test_cases*
- CENTER_SKEW
    - distribution mean = 50% * *total_number_test_cases*
    - distribution standard deviation = 20% * *total_number_test_cases*
- RIGHT_SKEW

[1] The calculation is roughly this, with some adjustments for integer division in the implementation

    - distribution mean = 66% * *total_number_test_cases*
    - distribution standard deviation = 20% * *total_number_test_cases*
- VERY_RIGHT_SKEW
    - distribution mean = 80% * *total_number_test_cases*
    - distribution standard deviation = 20% * *total_number_test_cases*

Note that if the RNG generates a number of out range (e.g. -3), the RNG will be re-queried until a valid number is produced.

These measures may be used to shift the skew of the size of test cases to a particular range of sizes relative to the total number of test cases used. As an alternative, a uniform distribution may also be used as a RNG; this distribution takes the form:

- CENTER_NO_SKEW
    - distribution min = 0
    - distribution max = *total_number_test_cases*

Test Case Goal State (Goal State): A goal state refers to a test case that is used to determine if a particular requirement is faulting. There is one goal state for each requirement; if this goal state fails during testing, it is assumed that the associated requirement is at fault.

Ideally, a goal state will be atomic. That is, it will contain one and only one requirement. In a given test suite, such test cases may not be present; in such a scenario, non-atomic test cases will be used as goal states (i.e. test cases that test more than one requirement). These cases will be selected a measure of the certainty with which it can be determined that a test case failure is related to a particular requirement. Obviously, in the atomic case, this correlation is 100%; if the test case fails, it is due to the one requirement it contains. In the non-atomic case, this correlation may be measured as a rough probability: essentially, the higher the probability that a test case has failed due to a particular requirement, the better that test case is as a goal state for that requirement. A simple example is given below, and more quantitative calculations are given later in the paper:

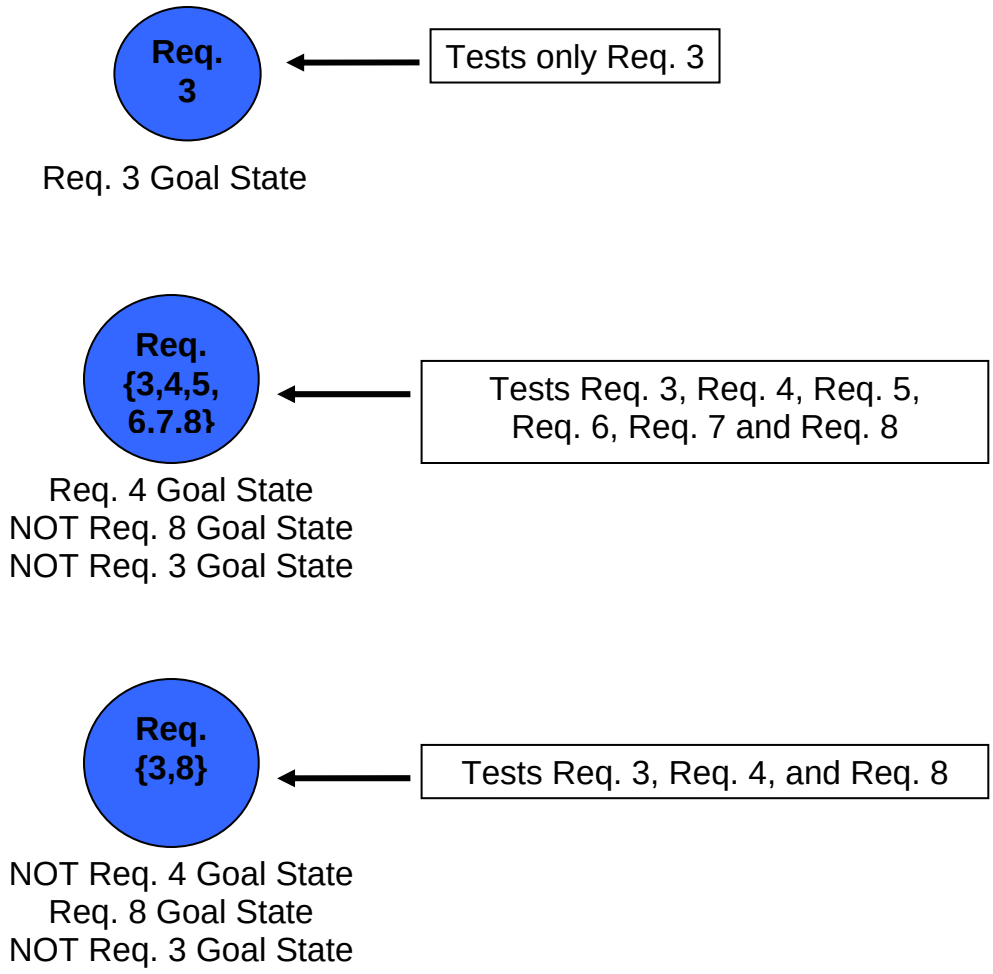


Figure 3: Goal State Example

3.3 Assumptions

There were several assumptions made during the work described in the paper. The major assumptions were:

- **Smaller test cases are more expensive than larger test cases**

  In many systems, it is easier and less expensive to test an assembled system than to test components individually. An example of this is a financial firm's enterprise information system. Assume this system receives data feeds from several different disparate sources. Assume also that the enterprise information system needs one module per feed. In this case, it may be significantly less expensive to test the system once all modules are constructed. This may be the case because no additional work

must be done to simulate the feed reception. On the other hand, it may be considerably difficult to test the system if all reception modules are not complete. Simulated data feeds would have to be constructed, which would increase the cost of testing.

- **Duplicate goal states can be used**

  In some instances, a goal state may be associated with more than one requirement. This is allowable in this model.

- **A test case failure reveals only that one of its included requirements fails and no more**

  This assumption is based on the fact that many test cases provide only binary results: fail or pass. If a test case provided more results (e.g. the requirement that caused it to fail), then the objective of this model would be met immediately upon *any* test case failure.

- **A test case only needs to be run once to determine if any of its included requirements are faulting**

  To illustrate why this is an important assumption, consider the following:

  - Test Case X has an EDR of 50%
  - Test Case X includes Requirement Z
  - Test Case X is run, and it succeeds

  At this point the model presented in this paper will assume that Requirement Z is *not* faulting. Unfortunately, there is room for Type II error in this assumption. That is, because the EDR of Test Case X is only 50%, there is a chance that Requirement Z is faulty but Test Case X failed to notice the fault. To limit this error, the test case may be run consecutively several times until the probability that Requirement Z is faulting AND Test Case X *misses* the failure becomes less than some nominal amount. An example of how to do this is:

  - Assume the same parameters as above
  - Assume a certainty measure of 99%; that is, do not assume a test case provides accurate results until there is less than a 1% chance of a Type II error. This is achieved by running the test case several times consecutively. This calculation is:

P(Type II Error) =

P(*test_case_misses_failure* AND *req_faulting*)$^{(\text{Times TC is run})}$

- Repeat the test case until it fails, or it has been run the appropriate number of times without failing

- **Absolute calculations can be substituted for relative calculations**

  Along with minimizing the Type II error, there are a series of additional Bayesian calculations that may be included to update the probabilities of failure of each test case as the suite is run. For the large part, however, simplifications have eliminated such calculations in the prioritization model herein. This is done because it is only the relative cost of test cases (compared with each other) that is important in the model. That is, it is not important to know that a test case costs exactly *x* dollars and *y* hours to run, but it is only important to know that a test case costs *z* times more than another test case.

- **All requirements have equal probabilities of faulting**

  Although the model can account for heterogeneously distributed probabilities, it is assumed that all requirements have the same probability of failing. This is done to simplify the experimental results as well as ease the computational stress of testing running the algorithms.

3.4 Theory

The test case prioritization techniques presented below are used to identify, with certainty, only one faulting requirement. These are not meant to be techniques that are used to order test cases at the onset of a test suite run. On the contrary, they are meant to be used as truncation techniques once a test case fails during the normal run of a test suite. This is done in order to save the cost of running the entire test suite once it's known that at least one requirement is faulting. The alternative to truncation is to continue running the entire suite *even after it's known that at least one requirement is faulting*. This may result in wasted resources as the one faulting requirement may cause other test cases to fail, which will have to be re-run once the requirement is repaired. Recognizing this immediately after the first failure of the test suite occurs will prevent such unnecessary resource use. Notice that a truncation technique will *never* be more expensive than the non-truncation technique. This is because a truncation technique will *never* run a test case that already has been run by the test suite, or that is not included in the test suite.

The worst case scenario for a truncation technique is that it runs all the remaining test cases in a suite, which is exactly what will happen every time in the non-truncation method.

The problem as stated above is similar to a typical search algorithm: given a set of items (potentially faulting requirements), and a set of test cases (the remaining test cases within a suite), search the test case set to identify one particular item (the goal state of the faulting requirement). An additional goal is to minimize the cost of this search. This cost is calculated as the sum total of the costs of each test case that is run.

Given these objectives, one may immediately consider the simple and intuitive solution of implementing a simple linear search algorithm. This algorithm simply would run each goal state sequentially until one goal state fails. This method would be simple to implement, but it would suffer from an O(n) running time. For these reasons, this paper will refer to such an algorithm as the "Naïve" algorithm.

An alternative to the Naïve solution would be to perform a binary search on the requirements. Considering the goal states for the requirements as leaves, and the other test cases as nodes may help the reader visualize this method. Such an algorithm would be non-trivial to construct, but it would yield an efficient O(log(n)) running time. This is the algorithm on which the prioritizing models presented in this paper are based.

3.5 Test Case Comparisons

To prioritize the test suite sequence, it was necessary to develop quantitative methods to compare test cases. This was especially important in two instances: selecting goal states, and ordering the sequence of test cases to run.

3.5.1 Selecting Goal States

Each requirement included in the test suite was linked to exactly one goal state. This goal state, by definition, is the state whose failure is most probabilistically associated with a given requirement. In other words, for a given requirement, the requirement's goal state is the test case that "tests a bigger percentage of that requirement than of any other requirement." This is part of

the criteria used to calculate the value that is used to rank potential goal states. The test case with the minimum ranking value is selected as the goal stated. The first step of its calculation is:

- Percentage tested = coverage(Req. in Question) / ∑ coverage(All Non-Faulting Reqs. Tested)
- Rank value = 1 / Percentage tested

Note that the metric, "coverage," was used. This is done to scale the percentage in accordance with how well a test case covers its requirements. Additionally, the EDR of a test case is used in order to determine how well it tests its requirements. This is incorporated as:

- Rank value = 1 / (Percentage tested * EDR)

Goal states are also selected based on cost-efficiency. To incorporate this into the measurement, the following calculation is done:

- Rank value = (Cost Test Case) / (Percentage tested * EDR)

This is the final value that is used to determine the attractiveness of a test case as a goal state for a given requirement.

3.5.2 Test Case Comparisons

Determining the least expensive test case, given a set of candidates, is a similar process to selecting goal states: a rank value is calculated for each test case, and the test case with the **minimum** value is selected to be run. The calculation for this is:

- Total Req. Coverage = ∑ coverage(All Non-Faulting Reqs. Tested)
- Rank value = (Cost Test Case) / (Total Req. Coverage * EDR)

Qualitatively, a test case that tests more requirements is more attractive than a test case that tests fewer requirements. This is because a test case that covers more requirements effectively contains more information; when it passes, all its requirements can be considered as non-faulting; when it fails, only its requirements become of interest.

Of critical importance is that this value gets updated as the test suite runs. To understand this, consider the following case:

- Test Case X contains requirements { 2, 3, 4 } (all with coverage of 100%), has an EDR of 100%, and a cost of 50
- Test Case Y contains requirements { 1, 3, 4 } (all with coverage of 100%), has an EDR of 100%, and a cost of 55
- Test Case Z contains requirements { 6, 7 } (all with coverage of 100%), has an EDR of 100%, and a cost of 60

The rank values of these test cases will be:

- Test Case X: = 50 / (3 * 1.00) = 50/3 = 16.67
- Test Case Y: = 55 / (3 * 1.00) = 55/3 = 18.33
- Test Case Z: = 60 / (2 * 1.00) = 60/2 = 30.00

As such, Test Case X is selected first to be run. Assume that it passes. This means that Requirement 3 and Requirement 4 are non-faulting. Therefore, the only information that is offered by Test Case Y is whether Requirement 1 is faulting. This needs to be reflected in the rank values, and is appropriately done so by:

- Test Case Y: = 55 / (1 * 1.00) = 55/1 = 55.00
- Test Case Z: = 60 / (2 * 1.00) = 60/2 = 30.00

At this point Test Case Z will be selected to be run next as it provides more information than Test Case Y.

3.6 Algorithm Evolution

3.6.1 Naïve Algorithm

The Naïve algorithm of identifying a faulting requirement is as follows:

- Identify the goal state for each requirement of interest
- Select randomly a goal state and run it
    - If the goal state fails, the search is complete
    - If the goal state passes, randomly select another goal state and repeat

This algorithm is the simplest to implement. It has a running time of O(*# requirements*) and provides no optimization.

3.6.2 Naïve Algorithm, Ordered by Efficiency

This algorithm is identical to the Naïve algorithm above, except that it more deterministically chooses the sequence in which it will run goal states. To do this, the goal states are sorted in order of an efficiency measure, as defined below:

- Assume goal state X is associated with Requirement 1
- Requirement 1 has probability to fail = *pf*(1)
- Efficiency[2] = (Test Case Joined Cost) / *pf*(1)

This algorithm is used as the baseline Naïve algorithm in the experimental runs that are presented below.

An example of this algorithm is provided below. The ovals represent test cases. The requirements included in each test case are included within each oval and are abbreviated, for example as "R1" for Requirement 1. Requirement 3 is the faulting requirement in this example.

[2] Notice that a lower Efficiency value is more desirable. This is an arbitrary measure that was adopted for primarily implementation purposes, the details of which are given in 3.5.1 - Selecting Goal States

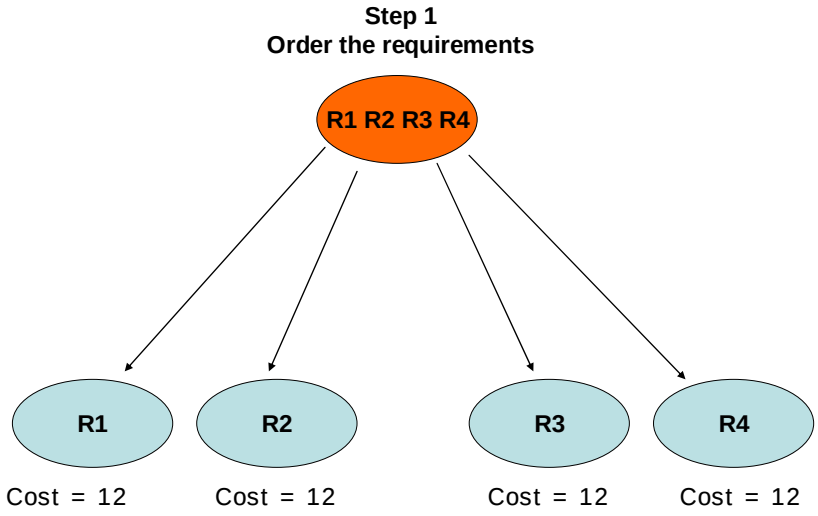


Figure 4: Identify the requirements of interest

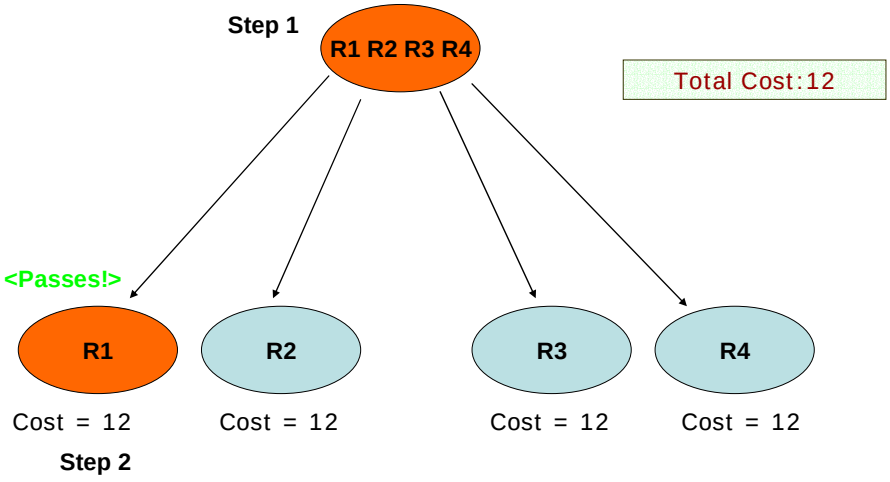


Figure 5: Test Requirement 1

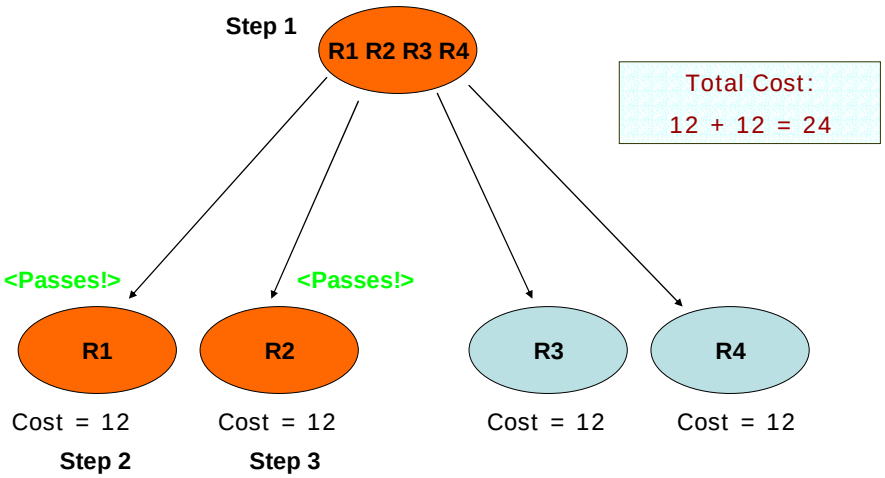


Figure 6: Test Requirement 2

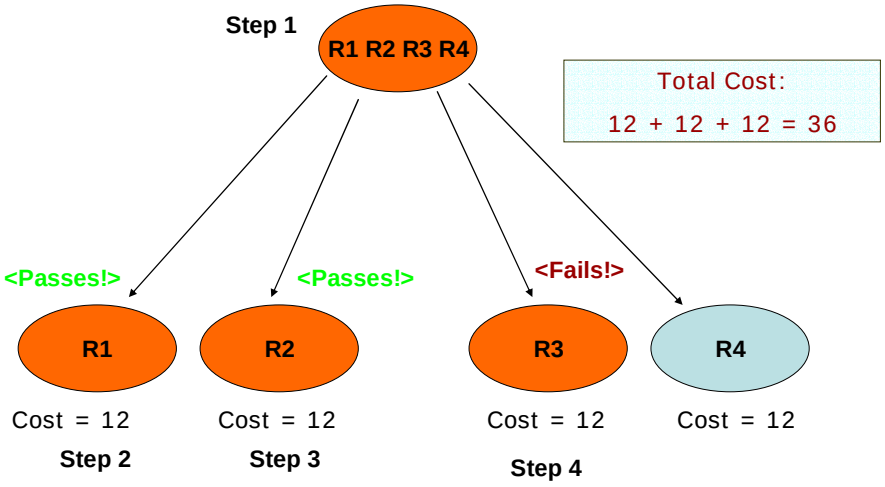


Figure 7: Test Requirement 3

3.6.3 Tree Algorithm, Ordered by Efficiency

The Tree Algorithm adopts a binary search-like procedure to identify the faulting requirement. Essentially, this is done by the following:

- Identify the goal state for each requirement of interest
- A tree-like data structure is built.
    - The root of the tree is a dummy node. Its children are the set of test cases that are not a subset of any other test case
    - Every test case is added to the such that each parent node is a superset of all its children
    - By the constraint above, the goal states are added as the leaf nodes
- A depth-first search on the tree is performed
    - Each node that is visited represents a test case. As soon as the node is visited, it is marked as "visited." The test case that is associated with the node is then run.
        - If the test case fails, the search continues to all of its unvisited children. If the node is a goal state, the search is terminated.
        - If the test case passes, all children of the node are deemed "visited". The search then returns to the parent of the node and continues.

An example of this algorithm is provided below. The ovals represent test cases. The requirements included in each test case are included within each oval and are abbreviated, for example as "R1" for Requirement 1. Requirement 3 is the faulting requirement in this example.

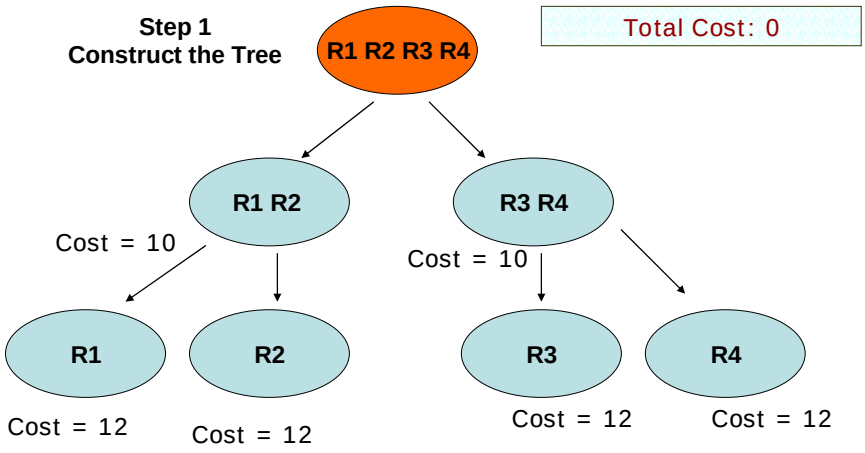


Figure 8: Four Potentially Faulting Reqs. Identified

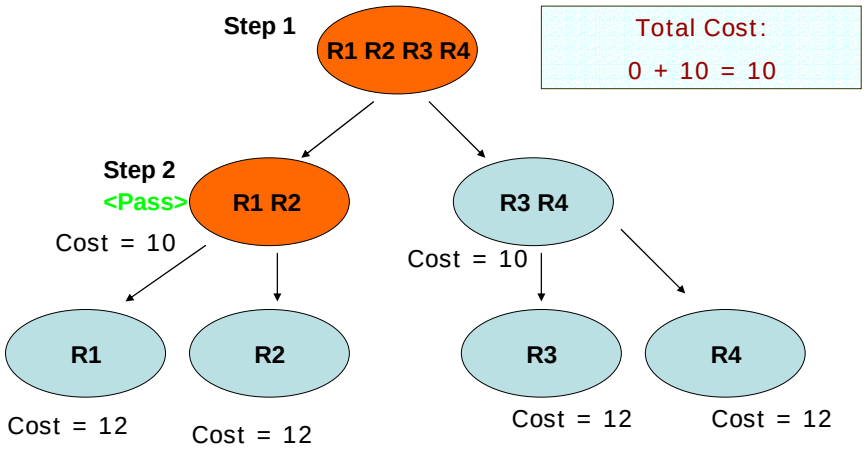


Figure 9: Testing Req. 1 and Req. 2

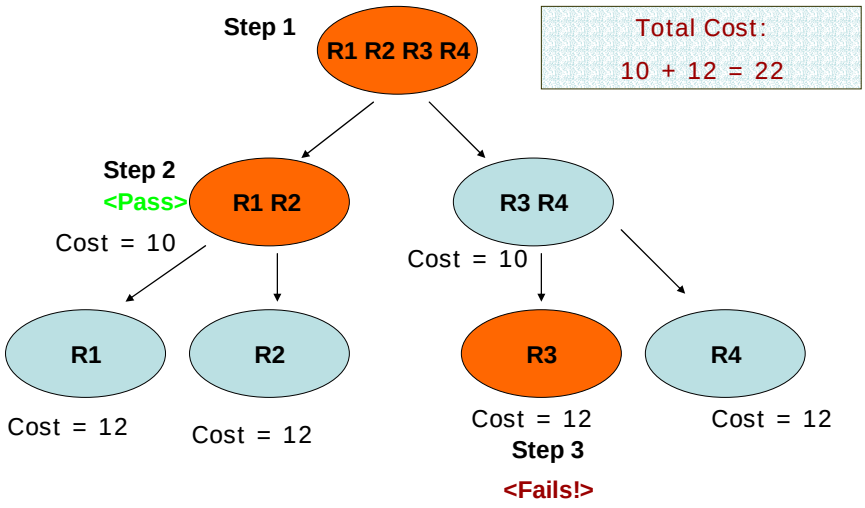


Figure 10: No new information given by testing Req. 3 and Req. 4 together. Proceed directly to goal states.

3.6.4 Graph Algorithm, Greedy Heuristic

The Tree Algorithm presented above presents an attractive alternative to the Naïve Algorithm. It is not, however, a completely accurate representation of the problem. To understand why this is, recall that the algorithm looked only at the test cases that were children of a node as candidates for the next test case to run; this was an adaptation that was made to reduce the algorithm into a variant of depth-first search. Unfortunately, this logic limits the set of test cases that are available to run at a given time. This, however, is not representative of the true problem scenario in which *any* test case in the test suite may be run following termination of *any other* test case.

To account for this discrepancy, an algorithm more representative of the true problem must be used. The algorithm developed to fill this, the Graph Algorithm, is similar to the Tree Algorithm, but with greater flexibility. This algorithm translates the tests cases not into a tree, but into a fully connected graph. The links of this graph represent transitions from one test case to another. Because it is fully connected, there are no restrictions on the sequence of test cases that may be run.

Predictably, the Graph Algorithm, Greedy Heuristic (GGH) uses a greedy heuristic to determine the sequence of test cases to run. This is conceptually a simple technique, as the following steps illustrate:

- Identify the goal state for each requirement of interest
- Rank all the test cases using the algorithm given in 3.5.2 - Test Case Comparisons
- Select the least-cost test case and run it
    - If the test case fails
        - If the test case is a goal state, the search terminates
        - Else, this information is fed back into the algorithm, the test cases are re-ranked, and the algorithm repeats by selecting the next least-cost test case
    - If the test case passes
        - This information is fed back into the algorithm, the test cases are re-ranked, and the algorithm repeats by selecting the next least-cost test case

This is a greedy algorithm because it looks only at the cost of the *next* test case to run; it does not perform any further look-ahead. In fact, this can be considered an adoption of Dijkstra's minimum cost algorithm used on a fully connected graph.

Due to the nature of the problem, the complete GGH algorithm is more complex than the algorithm outlined above. This complexity arises in the way in which the GGH may prune candidate test cases from the set of available test cases. That is, the GGH algorithm may find it unnecessary to run a test case, even if it is the least-cost test case in the available set. This typically is done when it is determined that a test case offers no new information, or redundant information. The reasons why this may be done include the following:

- The test case examines only one requirement, and is not a goal case
- The test case does not examine a subset of the previously run test case

This pruning is done after the selection of the least-cost goal state.

An example of the GGH Algorithm compared to the Tree algorithm is given below. First the Tree Algorithm is shown, and then the GGH is shown. Note that not all connections are given so as to illustrate the difference between the Tree Algorithm and the GGH Algorithm.

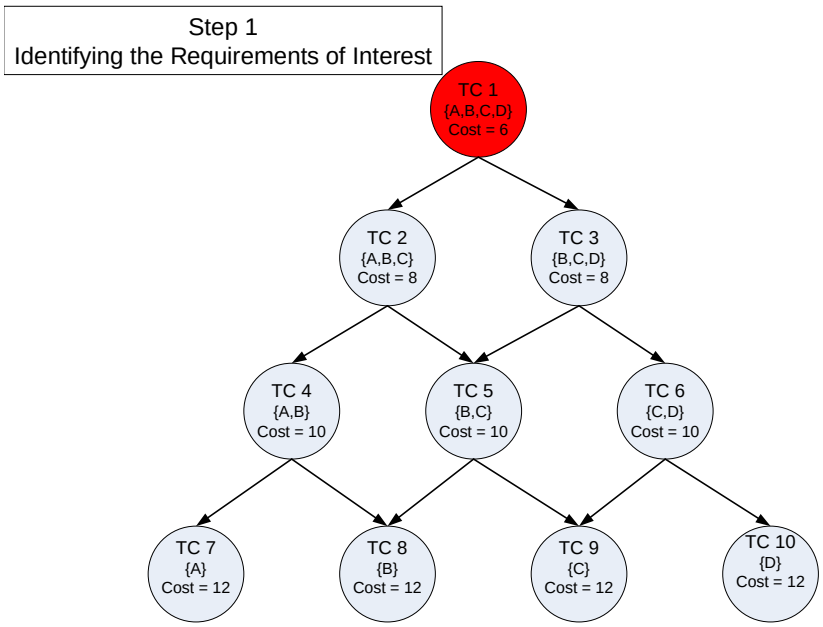


Figure 1 (Tree) Identifying the Requirements of Interest

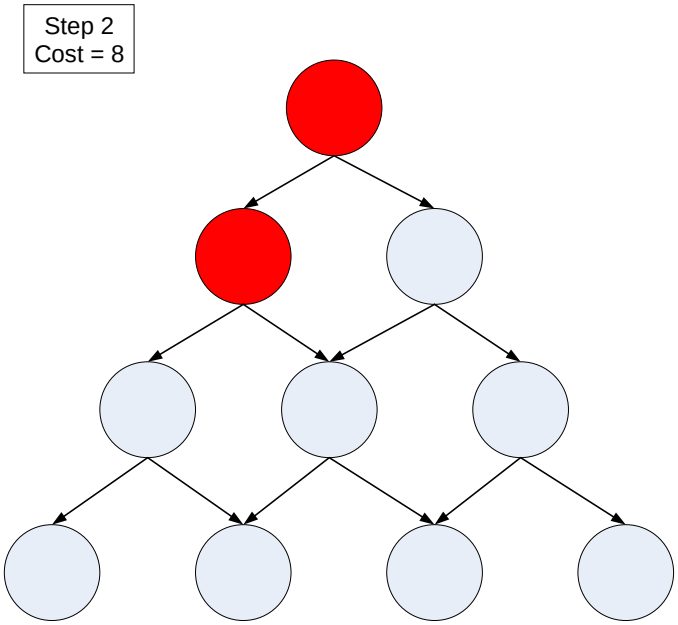


Figure 22: (Tree) Running a least-cost test case

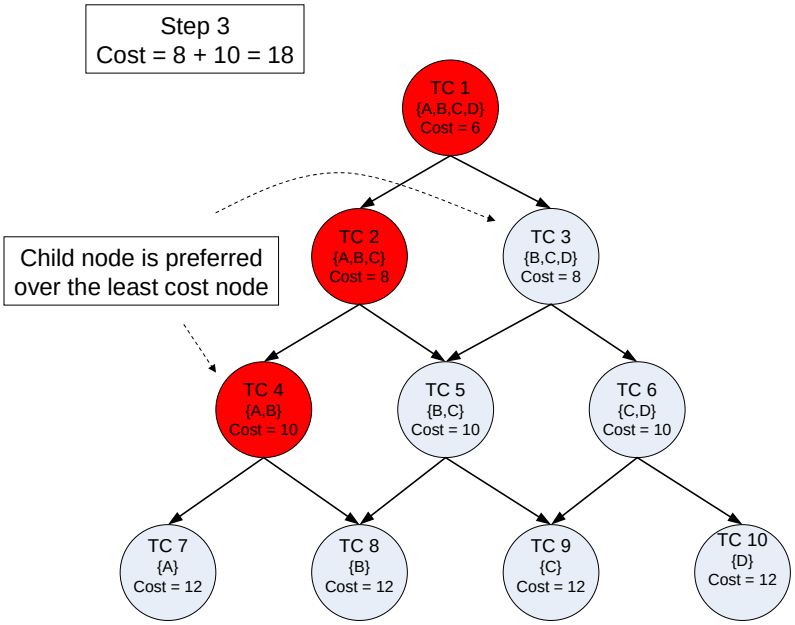


Figure 33: (Tree) Selecting a child test case

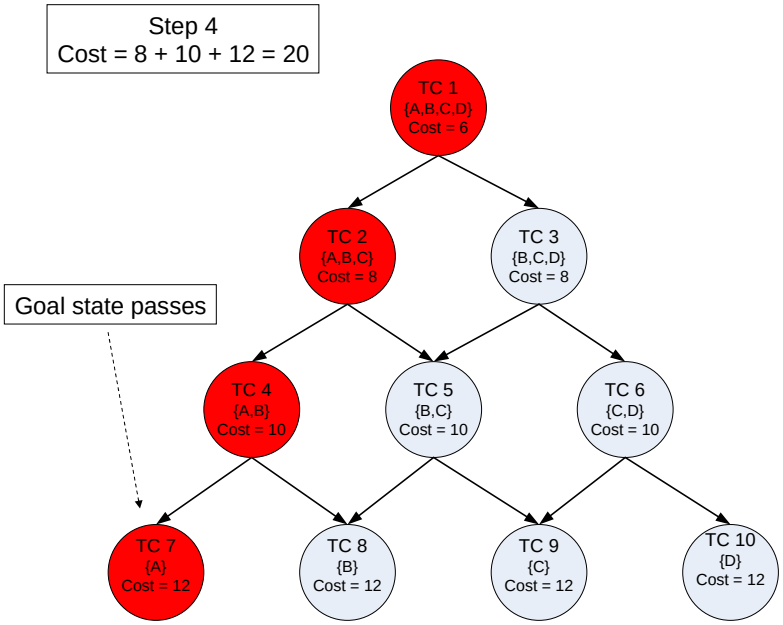


Figure 14: (Tree) Running a goal state

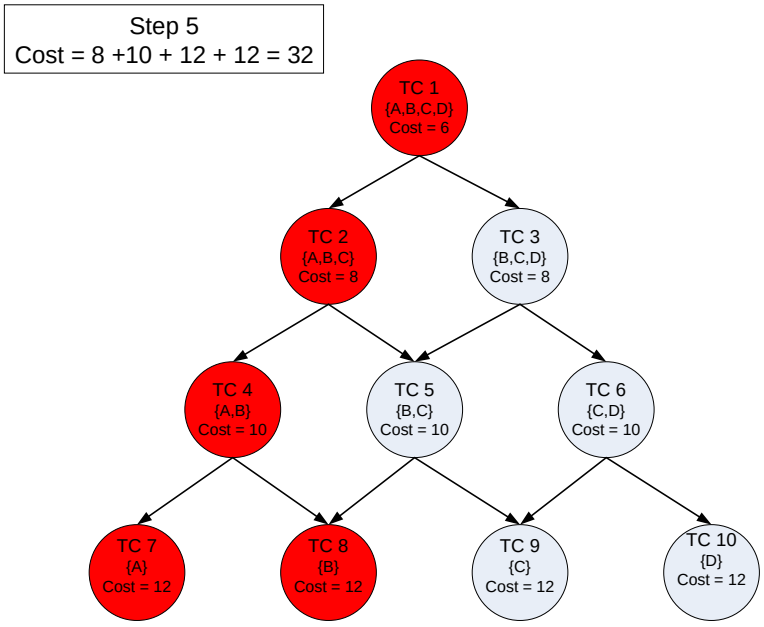


Figure 15: (Tree) Identifying the faulting requirement

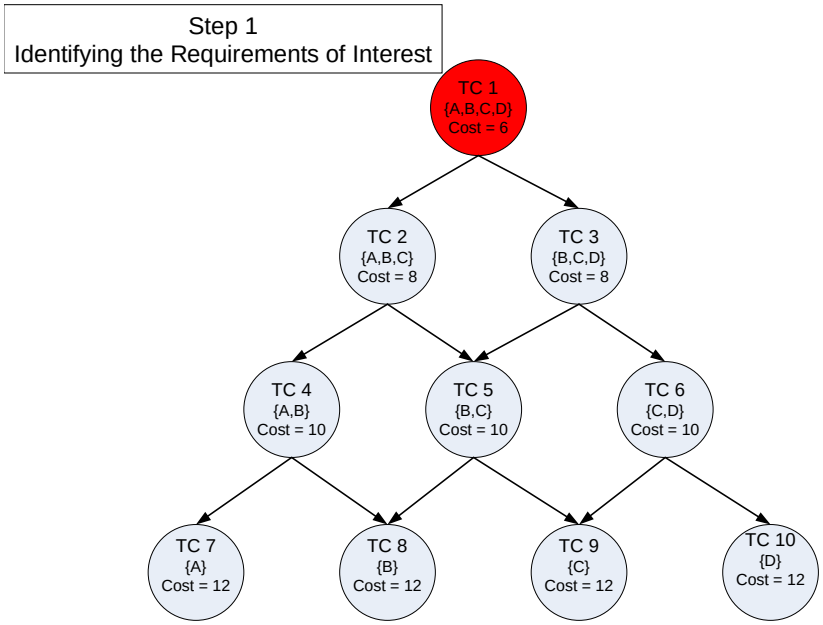


Figure 46: (GGH) Identifying the Requirements of Interest

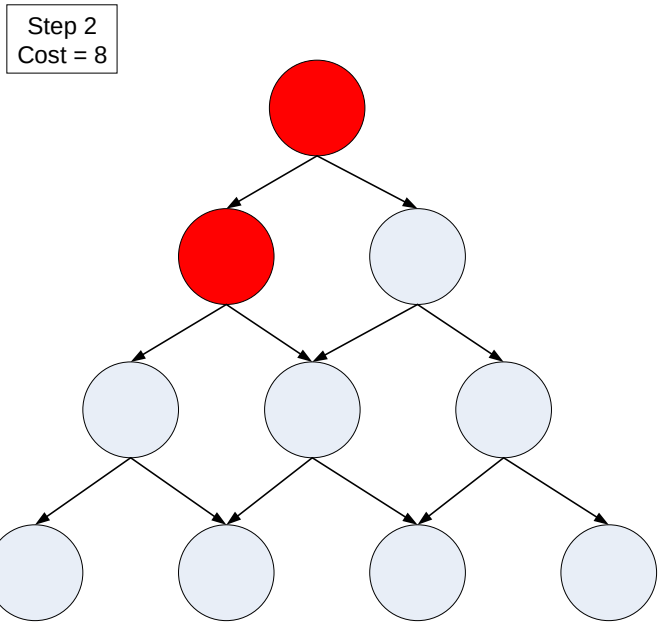


Figure 5 (GGH) Running a least cost test case

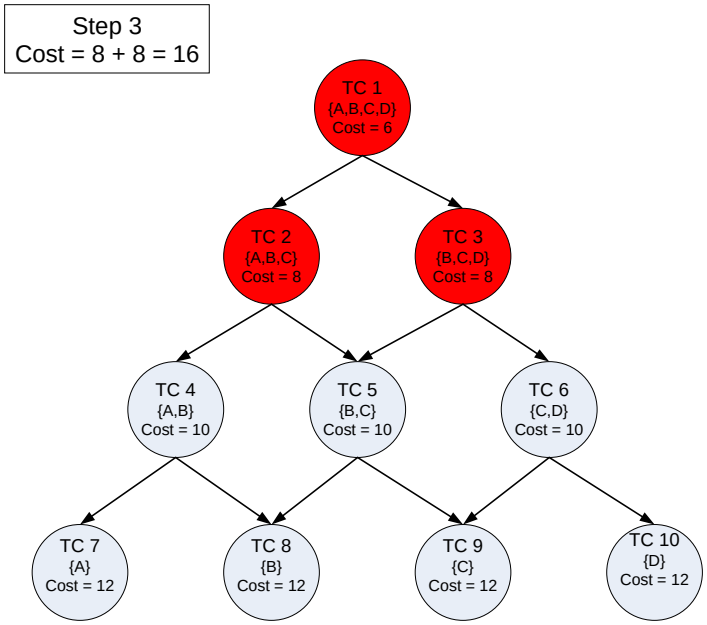


Figure 18: (GGH) Running the least cost test case

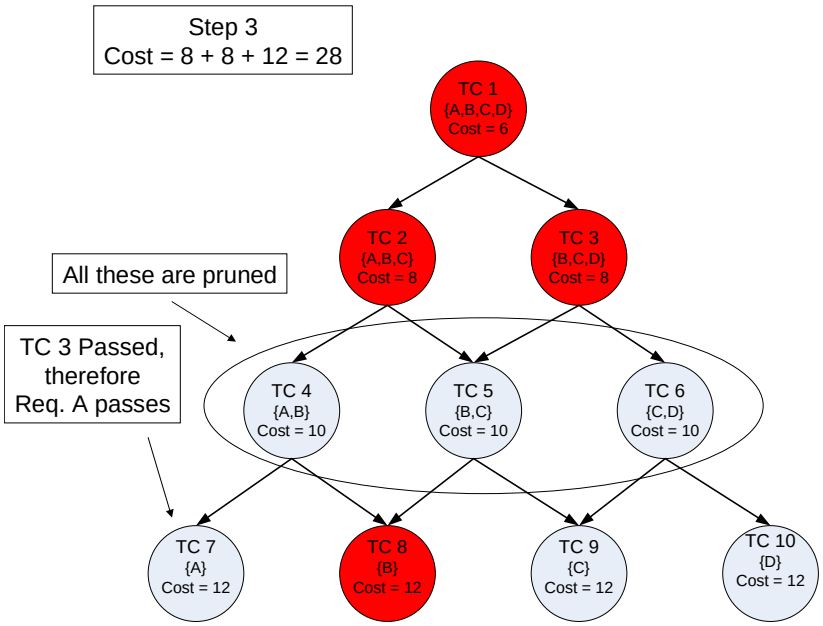


Figure 19: (GGH) Goal state is found

For this same test suite and the same faulting requirement, the Tree Algorithm will have a cost that is less than the GGH Algorithm. This illustrates the advantage of selecting the next test case to be run from the entire set of available cases. It also illustrates the advantage of pruning this set.

3.6.5 Graph Algorithm, Exhaustive Search

An alternative to the Greedy Heuristic is a completely exhaustive search. This technique would examine every possible path that may be taken, and would return the probabilistically least cost path as the sequence to run. At every test case run, this path would be updated.

Because of the nature of the search included, this Exhaustive Search algorithm would be able to provide lesser-cost sequences than the prioritization better algorithm than the Greedy Heuristic. Despite this, the algorithm is not a better practical choice. This is because the algorithm would have running time $O(n^n)$ (n = # of test cases). Therefore, the Greedy Heuristic was determined to be the best algorithm to use to test the graph prioritization method.

3.7 Implementation

A framework was created to test the Greedy Graph Algorithm against the Naïve Algorithm. This was done so that large test suites with large sets of requirements could be analyzed. Moreover, given key parameters, the framework was used to create test suites with different key characteristics. This was done to determine the key variables that affected the success of the algorithms tested.

The framework was written as a Java™ application that read and produced XML and text-delimited files. XML files were used as a simple way to define a test suite; these were useful for small examples. The text-delimited files were used to analyze the data in MS Excel.

3.7.1 Technical Overview

The testing framework was a relatively simple application. To some extent it was interfaced as a decoupled OO application, but to the same extent it was a design-once, run-once framework. The basic structure is shown below:

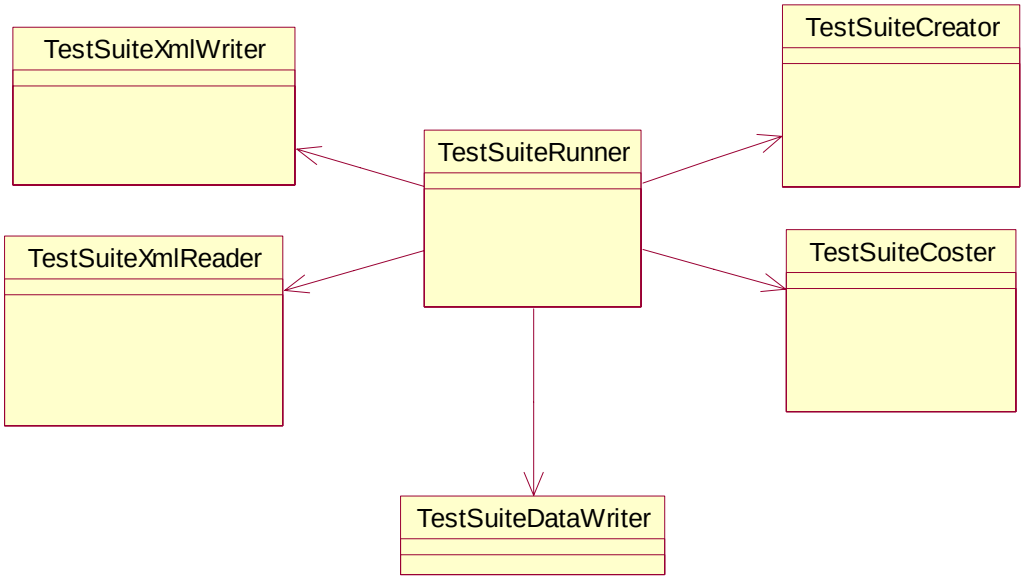


Figure 20: Application Class Diagram

3.7.2 TestSuiteCreator

The TestSuiteCreator is a class that was used to create test suites. The key parameters given to the TestSuiteCreator were as follows:

- Number of test cases: This indicates how many test cases to include in the suite
- Number of requirements: This indicates how many requirements are to be tested by the suite
- Closeness of requirements: This indicates the frequency with which the same set of requirements are included in test cases together
- Test Case Size Skew: This describes the shape curve that represents the distribution of the sizes of the test cases created
- Test Case EDR Mean: This is the mean value of the normally distributed Error Detection Rate value that is associated with each created test case
- Test Case EDR Sigma: This is the standard deviation of the normally distributed Error Detection Rate value that is associated with each created test case
- TC - Requirement Coverage Mean: This is the mean value of the normally distributed Coverage value that is associated with each requirement tested by a particular test case

- TC – Requirement Coverage Sigma: This is the standard deviation of the normally distributed Coverage value that is associated with each requirement tested by a particular test case

3.7.3 TestSuiteCoster

The TestSuiteCoster is a class that determines the cost of each test case for a given test suite. This is a key component of the framework, as minimizing cost is a primary objective of the research problem. The TestSuiteCoster is driven by the assumption that smaller test cases will cost more than larger test cases, but is also parameterized to be flexible in this notion. Such parameterization is done using the following variables (note that costing for the Financial and Temporal costs are done identically, and so only the financial costs are described below):

- Financial Base Cost: This is the base financial cost, down from which all pricing adjustment will be made. In other words, it is the maximum financial cost that a test case can be given
- Financial Overhead Amount: This is the percentage of the financial cost of a test case than can not be affected by scaling down for a lower EDR. This is described below in more detail.
- Financial Random Amount: This is the maximum percentage of the financial cost that can be either added or subtracted to the amount as a result of a random fluctuation

The TestSuiteCoster follows the proceeding method when costing a test case:

1. Sets the financial cost (*fc*) to be the Financial Base Cost
2. Scales the *fc* linearly according to:
    a. how many requirements the test case covers
    b. how well the test case covers each of these requirements
3. Determines the Financial Overhead Amount
    a. Determines the scaleable amount = *fc* – (Financial Overhead Amount)
    b. Scales the scaleable amount according to the EDR of the test case
    c. Adds this reduced amount back to the Financial Overhead Amount
    d. Sets this sum to be the new value of *fc*
4. Introduces a random fluctuation using the Financial Random Amount parameter
    a. Scales *fc* by a random percent of the Financial Random Amount parameter

This costing method is truly an approximation, but the consistency it introduces allows for reasonable creation of test suites and experimentation with test suites.

3.7.4 TestSuiteRunner

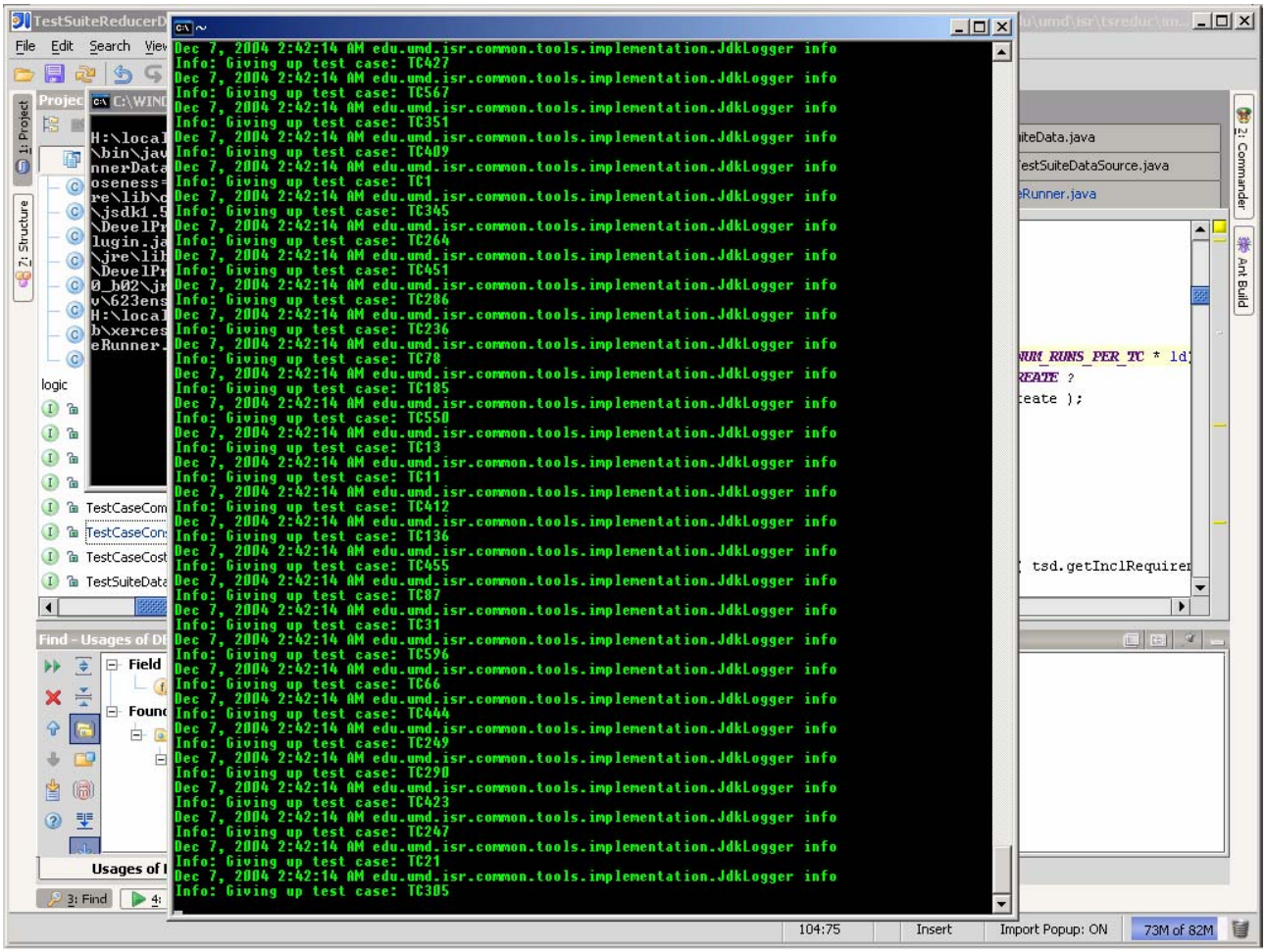
The TestSuiteRunner provides the main functionality of the framework. It is this class that implements the algorithms that are tested.

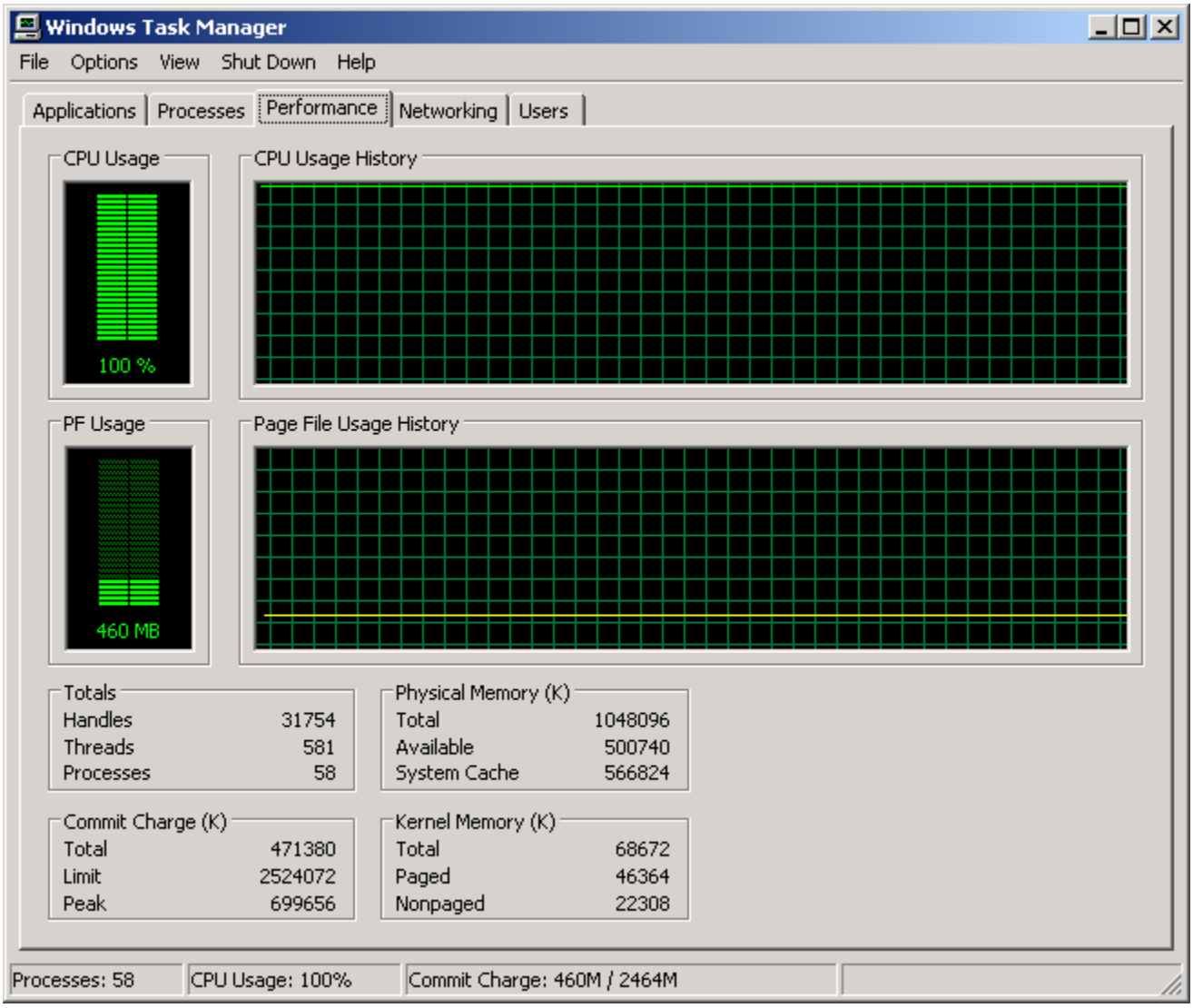
3.7.5 Comparators

The majority of the sorting was performed using custom implementations of the *Comparator* class. This was done to take advantage of the JDK1.5 built-in merge sort implementation. Additionally, the use of *Comparator*s facilitated algorithm enhancement and adjustments through the use of sub-classing existing *Comparator*s.

3.7.6 Screen Shots

For the authors' amusement, a screen shot of the application is provided below:


Dec 7, 2004 2:42:14 AM edu.umd.isr.common.tools.implementation.JdkLogger info
Info: Giving up test case: TC427
Info: Giving up test case: TC567
Info: Giving up test case: TC351
Info: Giving up test case: TC409
Info: Giving up test case: TC1
Info: Giving up test case: TC345
Info: Giving up test case: TC264
Info: Giving up test case: TC451
Info: Giving up test case: TC286
Info: Giving up test case: TC236
Info: Giving up test case: TC78
Info: Giving up test case: TC185
Info: Giving up test case: TC550
Info: Giving up test case: TC13
Info: Giving up test case: TC11
Info: Giving up test case: TC412
Info: Giving up test case: TC136
Info: Giving up test case: TC455
Info: Giving up test case: TC87
Info: Giving up test case: TC31
Info: Giving up test case: TC596
Info: Giving up test case: TC66
Info: Giving up test case: TC444
Info: Giving up test case: TC249
Info: Giving up test case: TC290
Info: Giving up test case: TC423
Info: Giving up test case: TC247
Info: Giving up test case: TC21
Info: Giving up test case: TC305
104:75
Insert
Import Popup: ON
73M of 82M



3.8 Experiments and Results

To test the effectiveness of the Graph Greedy Heuristic Algorithm (GGH), four experiments were run. These experiments sought to measure the effect of changing one of four major test suite parameters. In each experiment, three different scenarios were tested:

1. Set the parameter in question at a low value
2. Set the parameter at a medium value
3. Set the parameter at a high value.

For each of these scenarios, 30 simulation runs were completed. In each run, the GGH Algorithm and Naïve Algorithm were applied to the same test suite, containing the same faulting

requirement, in order to find the cost of identifying the appropriate goal state. These costs were then compiled and t-tested.

3.8.1 Four Factor Experiment Overview

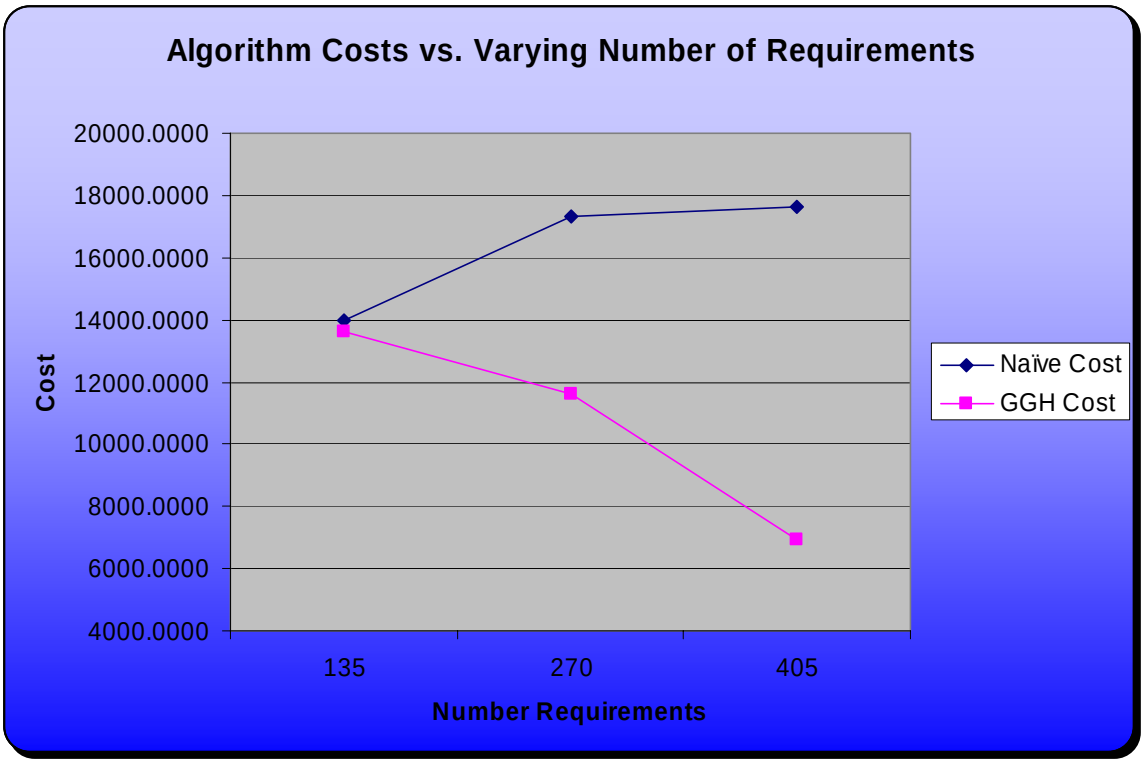
Four factors were isolated and tested in order to determine their influence on the effectiveness of the GGH Algorithm as compared to the Naïve Algorithm. For each of the experiments run, the baseline test was the same. Its parameters were:

- Number Requirements of Interest: 270
- Number of Test Cases: 600
- Closeness: 90%
- Test Case Size Skew: LEFT_SKEW

Each parameter was then varied twice in order to test its effect on the efficiency of both algorithms. The results are given below.

3.8.1.1 Number of Requirements

The Naïve Algorithm has running time O(n), where n is the number of requirements of interest. As such, for a small number of requirements, it may be more efficient to use this algorithm than the GGH. Investigating the accuracy of this theory was the driver behind first experiment.



Number of Reqs.	Naïve Cost	GGH Cost	T-Value	95% T-Crit.
135	13976.1901	13630.7524	0.1318	2.0017
270	17346.3226	11598.9739	1.7247	2.0017
405	17626.1015	6911.2533	4.4221	2.0017

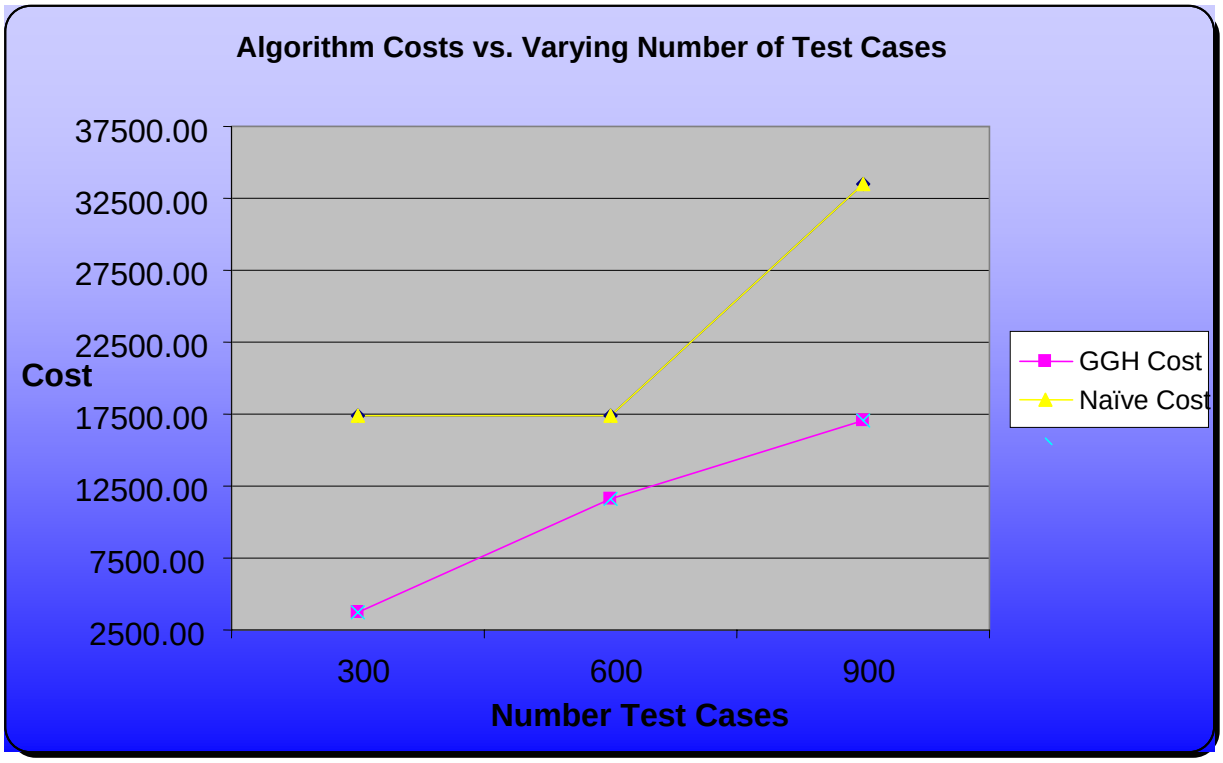
As the graph illustrates above, as the number of requirements increases, it becomes more attractive to use the GGH Algorithm. This is confirmed by the T-Value of the last run as it is above the critical value for a two-tailed 95% interval. Therefore, for test suites with large numbers of requirements, it is significantly less expensive to use the GGH Algorithm than the Naïve Algorithm.[3]

3.8.1.2 Number of Test Cases

Both the Naïve and GGH Algorithms benefit by having a large number of test cases from which to choose goal states. This is a simple concept – the more cases from which to chose, the more effective and efficient the selected goal cases should be. In addition to this, the GGH Algorithm

[3] The exact determination of what a "large" number has not been made yet

may benefit by larger sets of test cases because of the additional options it has from which to choose. That the cost differential between the Naïve and GGH algorithms would lessen as the number of test cases increases was the hypothesis tested by this experiment.



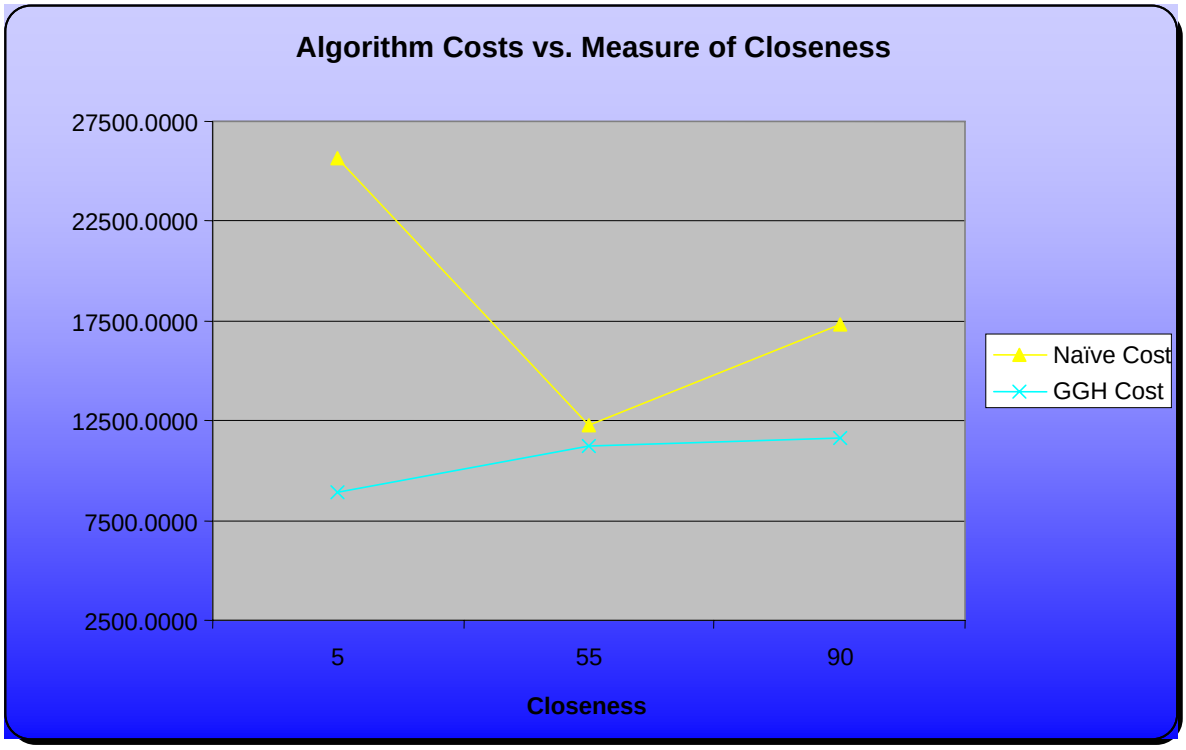
Number of TC's	Naïve Cost	GGH	T-Value	T-Crit
300	17424.1237	3745.2008	4.7883	2.0017
600	17346.3226	11598.9739	1.7247	2.0017
900	33455.5291	17040.9244	2.7882	2.0017

As seen above, it is not entirely conclusive that the cost differential between the Naïve and GGH Algorithms lessens as the number of test cases increases. A possible reason for this is that the percentage increase in test cases from 300 to 900 was too small to reflect a general trend. Note also that the costs for both algorithms increases as the number of test cases increases. This might be a result of a greater number of near-atomic test cases becoming available to use as goal states. The costs of such near-atomic test cases are generally higher than the cost of a large test case, and therefore this may contribute to the increasing trend. If this is the reason for this general increase, its effect would be expected to diminish as the number of test cases grew large enough

to provide one atomic test case as a goal state for each requirement. Beyond this number of test cases there would be no reason for the general cost to increase because of this hypothesized effect.

3.8.1.3 Closeness of Requirements in Test Cases

An increase in the closeness of requirements in test cases was expected to benefit the GGH Algorithm more than the Naïve algorithm. This is because a high correlation of requirements in test cases was predicted to provide a greater amount of information than a low correlation of requirements. For example, in a highly correlated suite, if a test case containing Requirements {1,2,3,…..10} passed, then it would be expected that many smaller test cases would contain an exact subset of these requirements and therefore could all be eliminated. In a lesser correlated suite, not as many exact subsets would be expected to be found. This was the reason why an increase in closeness was expected to result in a lesser cost for the GGH algorithm.



Closeness	Naïve Cost	GGH	T-Value	T-Crit
5	25677.4144	8883.5462	3.9174	2.0017

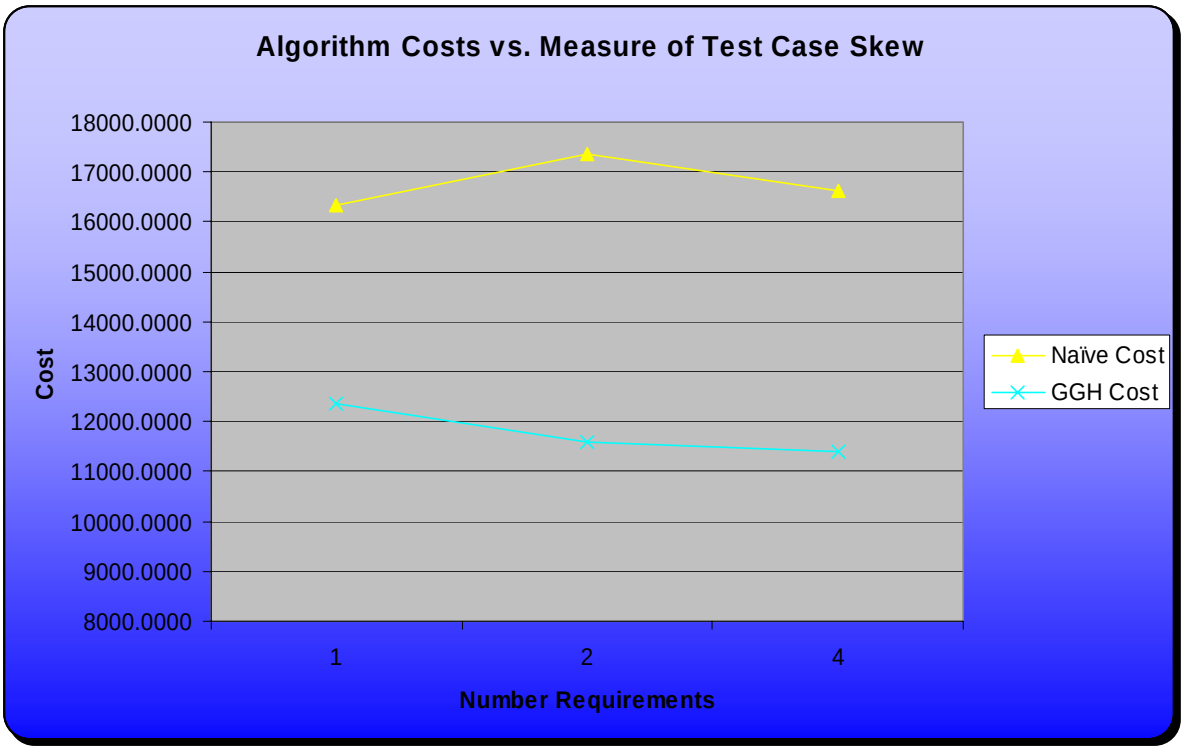
55	12296.5426	11194.0877	0.4311	2.0017
90	17346.3226	11598.9739	1.7247	2.0017

As seen above, it seems as if the opposite of the predicted theory is true. That is, the less correlated the requirements are in a test case, the more effective the GGH algorithm is. Upon reflection of the results, this seems to be a sensible theory. This is related to the pruning that is done in the GGH algorithm: Having a large set of test cases that contains randomly placed requirements offers a large opportunity to discover new information; the pruning encourages new discovery and discourages selecting test cases that offer redundant information. As such, each new test case that is selected and avoids being pruned will offer a relatively significant amount of new information. For this reason, a test suite with a lower measure of *closeness* may be a better candidate on which to run the GGH than a test suite with a higher measure of *closeness*.

3.8.1.4 Test Case Cardinality Skew

The measure of test case size skew was predicted to affect the costs of the algorithms run in the following ways:

- A left skew will provide larger amounts of small test cases. These will provide a larger, but more expensive, set of goal cases from which to chose. As a result, as the skew was moved to the left, the overall cost of both algorithms was expected to increase
- A right skew will provide larger amounts of large test cases. These will offer a large amount of information to the GGH Algorithm. As a result, as the skew was moved to the right, the cost of the GGH Algorithm was expected to decrease.



Skew	Naïve Cost	GGH	T-Value	T-Crit
1	16333.9106	12358.3748	0.9975	2.0017
2	17346.3226	11598.9739	1.7247	2.0017
4	16610.6174	11410.8743	1.3496	2.0017

As predicted above, the cost of the GGH Algorithm decreased as the skew moved to the right. The first point of the hypotheses, however, was not observed by the experiment. A cause for this may be the number of requirements used in the experiment. A larger number of requirements may have driven both algorithms to select a much larger set of small test cases as goal states. As a result this predicted effect may have been amplified to a point where it would have been observable. This not being the case, however, it cannot be concluded that moving the skew to the left has a general effect on the cost of both algorithms.

Note additionally that there is no significant difference between the costs of the algorithms over the range of *test case skew* that was tested in this experiment. This suggests that neither algorithm offers a clear advantage for use in an environment that has a particular value of *test case skew*.

3.8.2 Experiment Conclusions

In the four experiments that were run, there was not one instance in which the Naïve algorithm proved to be significantly more efficient than the GGH Algorithm. This is to say that in every scenario tested, the GGH Algorithm offered *at least* the same performance as the Naïve Algorithm. Additionally, in several cases, the GGH Algorithm proved that it offered significantly greater performance. As such, the GGH Algorithm has been shown to be a consistently better technique for the test case prioritization problem than the Naïve method.

4 Conclusion

This paper covered test suite minimization and ordering. We developed a non-linear program to select both requirements and the optimal test suite to test those requirements. We then created an algorithm to identify a faulting requirement in a more efficient manner than a simple linear implementation.

5 References

1. Aggrawal, K.K., Y. Singh, A. Kaur. 2004. Code coverage based technique for prioritizing test cases for regression testing. *ACM SIGSOFT Software Engineering Notes*. **29**(5).

2. Austin, M. 2004. System validation and verification. *University of Maryland - ENSE 623 Systems Engineering Course Notes*. 1-94.

3. Black, J., E. Melachrinoudis, D. Kaeli. 2004. Bi-criteria models for all-uses test suite reduction. *Proceedings of the 26th International Conference on Software Engineering*.

4. Chen, T.Y., M.F. Lau. 1998. A new heuristic for test suite reduction. *Information and Software Technology*. **(40)** 347-354.

5. Cohen, M.B., P.B. Gibbons, W.B. Mugridge, C.J. Colbourn. 2003. Constructing Test Suites for Interaction Testing. *IEEE*. 38-47.

6. Elbaum, S., A.G. Malishevsky, G. Rothermel. 2002. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*. **28**(2) 159-182.

7. Elbaum, S., A.G. Malishevsky, G. Rothermel. 2000. Prioritizing test cases for regression testing. *ISSTA '00*. 102-112.

8. A.G. Malishevsky, G. Rothermel, S. Elbaum. 2002. Modeling the cost-benefits trade-offs for regression testing techniques. *Proceedings of the International Conference on Software Maintenance*.

9. Offutt, A.J., J. Pan, J. Voas. Procedures for reducing the size of coverage-based test sets. *NSF-CCR-93-11967*.

10. Vaysburg, B., L. Tahat, B. Korel. 2002. Dependence analysis in reduction of requirement based test suites. *ACM*. 107 – 111.

11. Waters, R. 1991. System validation via constraint modeling. *MIT AI Laboratory ACM SIGPLAN Notices*. **26**(8).