VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wānanga o te Ūpoko o te Ika a Māui

School of Engineering and Computer Science
Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Identifying Redundant Test Cases

Marc Shaw : 300252702

Supervisors: David J Pearce and A/Prof. Lindsay Groves

Submitted in partial fulfilment of the requirements for Bachelor of Engineering with Honours.

Abstract

This project investigates tracing the method call information of Java JUnit Tests and using the information to identify redundant tests in a test suite. There are a variety of different methods implemented (and experimented on) to identify redundancy within a suite. The techniques involve retrieving different information from the tests and analysing it differently. The experiments show two main points. First, that a set of the method calls work well as a heuristic. This allowed for a pipeline approach to be implemented where the output of the heuristic is pipelined into more thorough analysis. Second, they showed that taking into account for the parameter information of the method calls gave insight into the context of the call. This increased the confidence that the identified tests were truly redundant.

Acknowledgments

I wish to thank Dr. David J. Pearce for the support throughout the project. This involved providing feedback, advice and general help. Without such, this project would have suffered greatly. I would like to thank Dr. Roman Klapaukh, Micheal Winton and Hai Tran for the encouragement throughout the year and A/Prof Lindsay Groves for helping with some proof reading.

Contents

Figures

List of Tables

Chapter 1

Introduction

Test suites are an important part in every major software project [13]. They check that a specified set of behaviours are met by a software program. Meeting the behaviours increases the confidence in the program, but as a consequence, a large number of tests need to be executed and may take up to several hours to run.

Test cases are added to the test suite throughout the project, often when a piece of code is altered, new code is developed or a bug gets fixed [25, 26]. With tests being added throughout, it is desirable to ensure any new test case is not replicating the behaviour of any of the previous. This is difficult to ensure, even with careful planning. The goals of the project are: create a tool to identify these test cases within a test suite and explore the tools capabilities to identify redundant tests through experiments.

The execution of a test case leaves a trail of data. This trail contains information ranging from low level to high level runtime data. A low level example would be machine code while a high level example could be the method execution data. A variety of previous research [28, 29, 23, 24, 14, 31, 16] discusses using some of this information to identify redundant test cases. The papers identify redundant tests using statement coverage while the majority examine benchmarks with under 200 test cases. For benchmarks with a large number of test cases storing every statement execution could be costly, therefore the principal aim of our project investigates using method execution data to identify redundant test cases. One of the issues that previous studies reported with statement information was a high level of false positives. Method execution data gives different ways to explore this issue which are explored further on in Chapter 3.

The tool developed can analyse method execution details, known as *test spectra*, to determine the level of redundancy between two tests. This analysis can identify potentially redundant test cases. Figure 1.1 shows a basic visualisation of the idea. The spectra are three different tests where colours represent method executions. It is clear that Test 1 is different from Tests 2 and 3 as the method executions are different between them however, there are similarities between Tests 2 and 3 which could identify some level of redundancy between the test cases.

Once the tests are identified as being redundant, they may be removed, however it is important to understand the dangers of removing test cases. Unless two test cases are exactly the same, it is difficult to guarantee whether one is redundant, even if one subsumes another. This creates the need to be able to use different techniques, depending on the use case. To expand on the goal of the project, the developed tool should give developers different approaches for identifying redundant test cases. The tool should allow a developer to configure different analysis metrics and view the results. Overall, the tool will be useful for gaining an overview and understanding of the condition of the test suite, allowing for manual inspection to determine if the identified redundant tests should be removed.

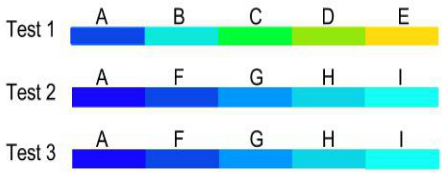

Figure 1.1: A spectra where each colour represents different method execution details. We see similarities between Tests 2 and 3. In contrast, Test 1 is different from 2 and 3.

Another potential use case of the tool is to redistribute the test cases. This can be achieved by splitting the non redundant tests into another test suite and running this suite in place of the original. This saves time to run the suite. To ensure there is no lose of fault detection ability, the original test suite may be run over night when no development is occurring. This separation of tests based on redundancy allows for a testing process, for example regression testing, to occur in a timely fashion while ensuring the original bug finding ability is retained. This is applicable to David Pearce, who is currently writing a language called Whiley. The language contains an extended static checking tool to eliminate runtime exceptions through formal verification techniques. In the main compiler module alone, there are roughly 20,000 tests. Relocating some of these tests into another suite would allow him to increase development speed. This is due to the reduction in the time taken to run his test suite. This reduction also gives David incentive to execute his suite more often. This increase in frequency allows him to trace bugs to code changes easier and reduces his time spent debugging.

The project's contributions are split into two groups:

- Create a tool for identifying redundant JUnit Test cases
- Analyse different strategies for identifying redundant test cases through experimenting on realistic benchmarks

1.1 Outline

The report is structured as follows. Chapter 2 discusses previous research in this area of interest. It also examines background information and explores the concepts needed to understand the following chapters. The design and implementation of the tool is then examined in Chapter 3. Chapter 4 investigates the different techniques then conducts and discusses some experiments. Finally, Chapter 5 concludes the report and identifies future work.

Chapter 2

Background

This chapter explores the current research conducted in this area. It then covers the key ideas needed to understand the remainder of the report. These ideas are – approaches to storing method coverage, edit distance metrics, evaluating performance of Java programs and the use of grid computing.

2.1 Related Work

2.1.1 Identifying Redundant Tests

Testing is a critical part to any software engineering process, primarily to stop incidents stemming from the product. The increasing popularity of agile methodologies [5] results in testing becoming more prominent throughout the development process [6] [3]. This requires the testing to be run in a timely fashion as agile processes often use regression testing during continuous integration [22]. The critical nature of testing and the increased importance of the time taken to execute them have influenced researchers to examine the different approaches taken to identify redundant test cases and reduce the size of test suites [28, 29, 23, 24, 14, 31, 16].

It is unclear whether programmatically reducing a test suite's size is worth the trade off in the ability to locate bugs. Wong et al. [28, 29] explored the impact that reducing the size of the test suite based off of code coverage had on the fault detection capability. By introducing faults into several programs and comparing the performance between the original and reduced test suites, they found that programmatically test suite reduction did not severely impact fault detection capability. In contrast to this finding, Rothermel et al. [23, 24] explored Wong's work by using different benchmark's and performance metrics. They found that test-suite reduction can severely impact the fault detection capability. The conflicting studies create uncertainty. This motivates our research to decouple the removal of tests from the tool and move the responsibility onto developers.

A popular technique used in detecting redundancy involves analysing the statement executions, known as statement coverage. Maurer, Garousi and Koochakzadeh [14] attempt to answer the question, *is coverage information enough to determine redundant test cases?* They state a redundant test case as being one that does not improve a specific criteria. For example, Figure 2.1 represents the statement criteria data from test cases T1 to T5. The figure shows that T4 and T5 are fully redundant as T3 covers the statements executed by these tests. The authors looked at two other criteria, branch coverage and granularities. Granularity criteria involved splitting the tests into setup, exercise (execution), verify (assert) and lastly teardown then performing analysis over each section. They implemented two different metrics in which both used the criteria described above. The first metric examined each

individual test with every other test. It was calculated by measuring the percentage of a test case that is also a subset of another test case. The second metric examined each test case with the test suite as a whole. It was calculated by measuring the percentage of the test case that the test suite covered without the test in it. By comparing with manual inspection, they were able to determine the level of false positive and actual redundant tests. Of the redundant tests manually identified, the algorithm matched 95% of those. Of the tests that were manually identified as being non-redundant, 52% of these were identified as being redundant by the algorithm. They concluded that coverage-based information is vulnerable in giving false-positives when identifying redundant test cases, suggesting common code paths as being a root cause. Statement coverage criteria gave a high rate of detection, but the implementation allowed for false positives to impact the results.

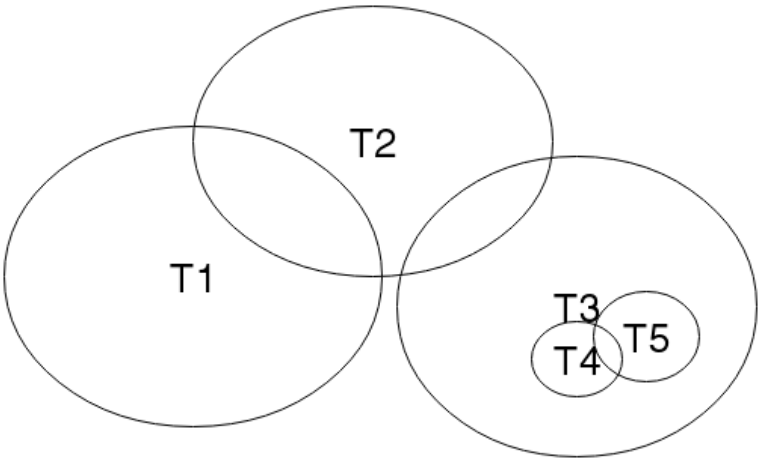

Figure 2.1: The coverage of each test is shown by a circle. It shows that T4 and T5 are redundant as T3 already covers those statements.

Zhang, Marinov, Zhang and Khurshid [31] examined the use of a greedy technique in comparison to heuristics. This required a criteria to be set by the tester, for example, statement coverage. The greedy technique would greedily select a test case that satisfies the maximum number of unsatisfied test requirements. It would continue until all the test requirements had been satisfied. The new test suite will contain exactly the same coverage as the old test suite while removing redundant test cases. This means that if a test subsumed another, then it would always be removed. The heuristic implementation was first conceived by Harrold, Gupta and Soffa [12] where it selects essential test cases as early as possible. Essential being that only one test case satisfies a test requirement exclusively. The heuristic approach resulted in the most cost-effective reduction, this being the amount of reduction and the fault-detection capability of the reduced test suites. This showed that although greedy approach worked, there were better techniques available.

In situations where it is not possible to generate a spectrum to analyse, static analysis can be used to determine the level of redundancy. Robinson, Li and Francis [16] explore this. The tests for the benchmark they use are written in a high level automation framework and consist of a list of commands. The commands perform actions such as file copying and loading configurations. To identify redundant test cases they examine the test case commands as well as the instructions within the procedures that the test case loads. To calculate the similarities between two test cases, they consider three different metrics. These are the Manhattan distance, unigram cosine similarity and bigram cosine similarity. They each measure how closely related two tests are based on the sequence of commands and

procedures loaded. Their findings were similar to Maurer et al. [14] in that there are a large number of false positives. Static checking has several limitations in comparison to dynamic. The main disadvantage is the availability of data. During run-time is when a large portion of the data trail is available. This is important when needing to know the exact method calls and parameters passed to these methods. Static checking would provide useful information in a framework where dynamic data can not be collected. The framework would also need to be applicable to being examined in a static fashion such as the one used by Robinson, Li and Francis [16].

Taking into account the related work discussed, each paper used the coverage of a test suite at several different criterion levels but none looked at the method execution explicitly. This leaves a potentially useful approach to the problem that may help determine the level of redundancy within a test suite.

2.1.2 Calling Tree

When methods execute there is a trail of data that is left behind. Piecing together this data can show the relationships between methods, which is known as a *call graph*. The call graph can be retrieved statically or dynamically [11]. A static graph represents every possible run of the program. A dynamic graph represents one particular run. Comparing them, a static graph requires more information to be held. In the particular case of profiling test cases, dynamic would be more suitable due to the nature of a test case being the same for every run. Another advantage of a dynamic graph is the ability to collect functional parameters. Xiaotong Zhuang et al. [32] describe a call tree as the overall tree of the dynamic method execution information. To store the full call tree would involve an excess of memory, therefore they discuss the use of a calling context tree. This tree represents the same method executions however is compressed where the edge between the nodes contains a weighting, the weighting being the number of occurrences of that method execution (calling frequency). Examining Figure 2.2 shows the different variations that can be observed with the same data. The top left hand picture shows the method execution trail, where A calls B, then B calls D twice and A calls C, then C calls D once. The call graph image (top right) condenses the information into one node per method call. This method is the most condensed variation, but gives the least amount of information. Directly below this, the calling context tree condenses it slightly less. It separates each path into its own branch when it is unique to the tree. The connection between nodes contains a weighting which represents the frequency. Finally, the bottom left (call tree) creates a new path for every new method execution regardless of the uniqueness. This allows for the tree to retain the most information about the context.

2.1.3 Edit Distance Metrics

Edit distance algorithms play an important part in many domains, ranging from signal processing to mutations in genome sequences [19]. They calculate the number of edit operations needed to go from one string to another, in our research they are used to calculate the difference between test spectrums. There are a variety of different implementations of edit distance metrics. Cohen, Ravikumar and Fienberg [8] explore different edit distance metrics for name matching. They concluded that the Monge & Elkan edit distance metric [18] performed the best for string edit-distance metrics. The metric works by splitting the two strings into tokens, the best matching tokens are then calculated to find the similarity using other common metrics such as Levenshtein distance [15]. One of the other techniques they explored was the Levenshtein distance metric by itself. This metric is the minimal number

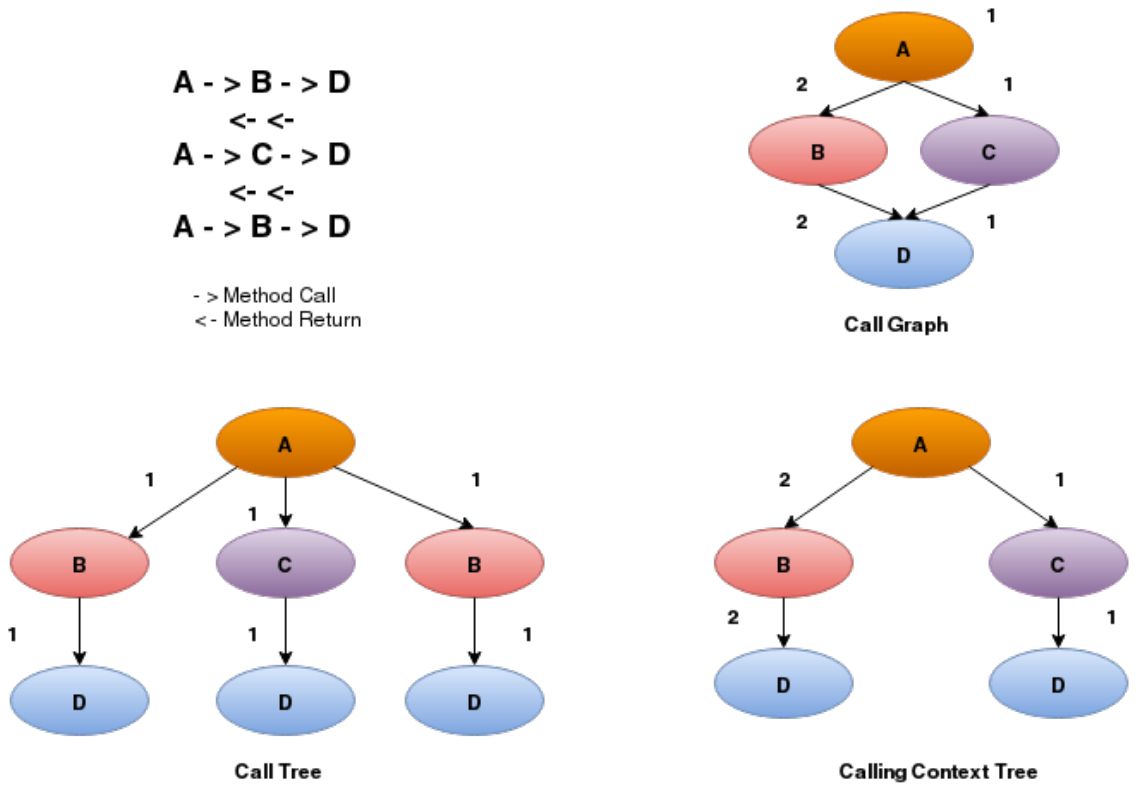

Figure 2.2: A diagram showing the three different variations of Call Graph information. Each have a varying level of information that is stored.

of operations to change one test's spectrum into another. These operations are inserting, deleting or substituting and a cost is associated with completing an operation. The maximum difference is the size of the larger spectrum. The amount of redundancy is calculated by dividing the cost of operations with the max difference in order to normalize the value. This value is the percentage of the test that is not redundant, the value 1 is then subtracted by the value to give the redundant percentage.

Table 2.1 shows an example of Levenshtein. The example changes 'kitchen' into 'kitten'. It shows the number of operations needed is 2, and the max potential needed if the two strings were completely different is 7, as kitchen contains 7 characters. The redundancy is calculated by subtracting 1 from the cost over the max potential cost $(1 - (2/7))$. The outcome being that the words contain 71% redundant information.

Previous State	Current State	Operation
-	kitten	-
kitten	kitcen	Substitution of 't' with 'c'
kitcen	kitchen	Insertion of 'h' between 'c' and 'e'

Table 2.1: Using Levenshtein edit distance metric to transform kitten to kitchen.

Table 2.2 shows an example of Monge & Elkan edit distance metric. It compares the strings 'paul johnson' and 'johson paule'. Each of the strings get broken down into two tokens and the best matches are identified and used for the final score. The table shows that the final score of the Monge & Elkan edit distance metric is taking into account 'paul' &

'paule' and 'johnson' & 'johson'. Using these two matches, the final result is 1/2 * (.85 + .80) = 0.825.

Input string 1:	paul johnson
Input string 2:	johson paule
Levenshtein("paul","johson")	0.00
Levenshtein("paul","paule")	0.80
Levenshtein("johnson","paule")	0.00
Levenshtein("johnson","johson")	0.85

Table 2.2: The Monge & Elkan edit distance metric to calculate the similarity between "paul johnson" and "johson paule"

2.1.4 Wilcoxon Signed Rank Test

The Wilcoxon Signed Rank test was first proposed by Frank Wilcoxon in 1945 [27]. It allows us to infer whether there are any significant differences between the experiment results. It is a nonparametric test for comparing two pair groups. A nonparametric test is a collective term that is given to inferences that are valid under less confining assumptions than classical statistical inferences [10]. The test does not assume that the data follows a normal distribution which is ideal for the data presented in our research. It is used when two nominal variables and one measure variable is being measured [17].

2.2 Performance Evaluation

Throughout the report, the design and implementation of different techniques are discussed. To evaluate the techniques, we conduct experiments. The purpose for evaluating the performance of a given technique is to create an understanding of the typical performance. For this typical performance to be valid, it has to be rigorous in design and implementation. This involves taking into consideration a variety of factors that are examined in reference to research papers that explore the issues.

Andy Georges et al. [9] and Steve Blackburn et al. [7] present issues and alternatives to the current Java performance methodologies used in research papers. They note that in premier conferences, 16 of the 50 papers examined did not discuss the methodology they used. This limits the validity of the results and creates a situation where others can not reproduce the experiment. One of the main issues identified is specifying singular numbers, without expressing what the number is referring to (average, median, best, worst). This leads to a misleading comprehension of the results. The paper explores a set of parameters that are of interest. The parameters include: heap size, start up performance, number of virtual machine (VM) invocations and handling of garbage collection. An experiment should also take into consideration the environment it is executed on.

- **Heap Size** – A change in heap size can impact the garbage collection process. The smaller the heap size, the more often the garbage collector is run [7].
- **Start Up Costs** – The costs associated with starting the application up. This will always involve class loading and just in time (JIT) (re)compilation. It can also involve reading from a database and setting the application up [7].
- **Number of VM invocations** – The number of application runs that a single VM instance will execute.

- **Garbage Collection** – The process in identifying and removing objects that are no longer referenced.
- **Environment** – The hardware that the application is being run on.

2.3 Grid Computing

The time intensive nature of conducting experiments resulted in a single computer not being adequate. A grid computing system was used instead. It is a distributed system of multiple machines that have no interactive tasks between each other. No interactions results in task independence being a necessity. The particular grid system used for this report is located at Victoria University of Wellington. The machines in the grid are idle machines in the School of Engineering and Computer Science department. There are a limited number of machines, therefore a total of 150 jobs can be run at a given time. Since the grid is located around an active community, the system pauses the application if a user logs on while an application is running. After the user logouts, the grid system unpauses the application.

Chapter 3

Design and Implementation

3.1 Overview

The goal of the project as mentioned in Section 1 is to create a tool to allow developers to identify redundant test cases. The tool is split into several different sections and the chapter will discuss each. The first objective of the tool was to trace the test data. We explore two different frameworks and compare the usability of each. After the tool was able to trace the data, we determine what spectra we are interested in. These are then discussed. The next section discusses using edit distance metrics to compare test case trace information. Throughout the testing of the tool, one of the main impediments was the time taken to analyse the large amount of data. Finally, how breaking the analysis into a series of pipeline stages decreases the time taken is explored.

3.2 Tracing

A fundamental requirement for this project was to trace the methods executed by a test. David Pearce's language Whiley is written in Java, therefore it was decided to use Java frameworks to trace the tests. There were two tracing frameworks considered. The first framework considered was AspectJ. The technique that AspectJ uses is Aspect Oriented Programming (AOP). AOP can be used to add code to an existing program without modifying the source code, known as weaving. To distinguish what code to weave, and where in the existing code to weave it, AspectJ uses a *pointcut* [1]. An *aspect* contains multiple point cuts. Weaving the aspect can be achieved through the following methods:

- **Compile time** – The classes are compiled with the aspect woven into them. This means when the program is executed, the pointcuts are already added to the program. This requires the source to be compiled with AspectJ's compiler.
- **Load time** – The weaving process is deferred until the point at which a class loader attempts to load in a class file. Load time weaving is achieved by using a command line argument notifying Java to use the AspectJ class loader.

The other framework we considered was the Java Debugging Interface (JDI). Using JDI is similar to using an observer pattern. It allows for a list of classes to be selected to observe. When a method within a selected class is called, the listener class is notified. This class is then passed the information about the method call.

AspectJ provided several advantages over JDI. The ability to choose which methods to record was easier to define and it returns the actual object when retrieving the parameters

of the method call. This detail becomes important when discussing the ability to trace parameter values in Section 3.6. In comparison, JDI was faster to execute, however there was a limited amount of documentation available and the parameters returned were not the actual objects. The decision to use AspectJ was based off this trade off between information and performance. Decoupling the data gathering and data analysis within the tool removed the emphasis on retrieval performance. This meant that the tracing performance was less of an issue.

Our pointcuts record every method execution. Listing 3.1 shows a simplified version. There are two pointcuts within the aspect. The first pointcut gets called for every method call which has a JUnit Test annotation attached to it. This will then call a static service passing it the method information of the new test. The second pointcut traces everything apart from the methods with a JUnit Test annotation attached. It then passes the information to the static service. The next stage was to weave the aspect into a benchmark. Using compile time weaving would have meant that every benchmark had to be recompiled using the aspect compiler. In contrast, load time only requires the AspectJ class loader to be passed through a command line argument. Load time was chosen as it was easy to use when working with external benchmarks.

```
1  pointcut testMethod(): execution(@org.junit.Test * *(..));
2
3  before(): testMethod(){
4      StaticService.addMethodCall("New Test:" + thisJoinPointStaticPart.
       getSignature().getDeclaringTypeName());
5  }
6
7  pointcut traceMethods() : (!within(@org.junit.Test *) && execution(* *(..));
8
9  before(): traceMethods(){
10     StaticService.addMethodCall("New Method:" + thisJoinPointStaticPart.
       getSignature().getDeclaringTypeName());
11 }
```

Listing 3.1: A simplified pointcut within AspectJ. The first pointcut is for when a new test is executed and the second when a new method is executed.

3.2.1 Saving Trace Information To Disk

To execute the analysis on a grid computing system, the test data had to be accessible on the School of Engineering's local network. To achieve this, the trace data was saved to the local network. It allowed for the analysis to be executed on the grid system which is key to the experiments. It also allowed for the data to be reanalysed without having to re trace the test suite.

3.3 Filtering By Spectra

The idea of a test spectrum was previously identified in Chapter 1. Reexamining the idea, a spectrum is some abstraction of the method information. There are three different types of spectrum. These are *unique method calls*, *all method calls* and *call tree*, each with its own characteristic. It is important to understand how we use a spectra. When the tool is tracing the tests, it traces the amount of information needed based on a settings file. A spectra can then be applied to this trace information to filter out data to match the spectra's characteristic. The tool ensures that the traced information is able to be filtered by the spectrum.

Figure 3.1 helps with discussing the spectra characteristics. The different spectra require different levels of resources, these resources are: time taken and memory.

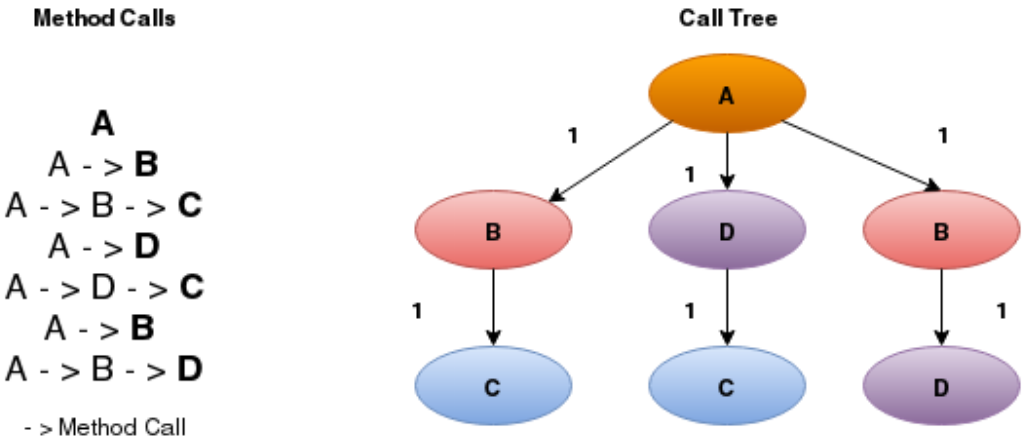

Figure 3.1: A simple representation of a call tree of three that is traced from a test. Each letter represents a method call and the bold letter is the active method in each line of the call tree.

3.3.1 Unique Method Calls

The "unique method call" spectrum filters out the repetition of methods. The data after applying the filter only has each method represented a single time. The motivation of this spectrum is to substantially reduce the time taken and memory used. Consequently, decreasing the amount of data leads to a decrease in the confidence that the tests identified are truly redundant.

Examining Figure 3.1, on the left hand side are the method calls where each line represents one call. The active method is in bold and the parent methods are before it. On the right hand side is the call tree. With no filtering applied the method calls are "A,B,C,D,C,B,D". When we apply a "unique method calls" spectrum filter, the data will become "A,B,C,D". This shows that we are filtering out a large portion of data while retaining an accurate representation of the original trace information. For the Whiley Compiler benchmark there are 494 unique methods called for a particular test and 80,000 method executions. Utilising the unique method call spectrum, the tool will only use 494 method executions when comparing the test with another, rather than the 80,000. This is $^{6}/_{1000}$ of the original amount.

3.3.2 All Method Calls

The "all method calls" spectrum filters out information regarding the call tree. The motivation behind this approach is to analyse a larger subset of trace data than the "unique method calls" spectrum. Comparing the two, "all method calls" increases the accuracy of the results however, this increases the time taken and memory used to analyse the information.

Examining Figure 3.1, the output of filtering the method calls with this spectrum would be "A,B,C,D,C,B,D". This shows a loss of the context data. When using an "all method calls" spectrum on the Whiley Compiler benchmark example, the tool will use all 80,000 method executions when comparing the test with another.

3.3.3 Call Tree

A "call tree" may filter out some of the parent methods. The number of calls retained is referred to as the K depth [32]. The motivation behind the approach is to increase the accuracy of the results by taking into account the *context* of a call. The consequence of increasing the amount of data is a further increase in the time taken and memory used to analyse the trace data in comparison to the other two spectra.

We can filter the Figure 3.1 using a K depth of two. This would lead to the filtered data returning 'A, A $\rightarrow$ B', 'B $\rightarrow$ C', 'A $\rightarrow$ D', 'D $\rightarrow$ C', 'A $\rightarrow$ B' and 'B $\rightarrow$ D'. We can see that by using a K depth of two, we lose some of the information from the trace data. To stop this loss of data, we could extend the depth to three. Using the Whiley Compiler benchmark as an example, a call tree filter would examine all 80,000 method executions. The difference to "all method calls" filter is that each execution contains a total of K calls.

To retrieve the calling context in Java, the current stack trace of a method call is examined. It is then parsed to retrieve the relevant information. This parsing involves removing the method calls that are used to retrieve the stack trace and any memory location details.

3.4 Analysis Metrics

Having traced the information and being able to filter out specific data. This filtered data needs to be then compared with another test cases data to produce a redundant level between them. Edit distance metrics were choosen to perform this comparison. The two different edit distance metrics considered were discussed in Section 2.1.3. They were: Monge & Elkan edit distance metric [18] which splits the strings into sections and compares the sections and Levenshtein [15] which compares tokens that are generally single characters.

For both of the algorithms, there were publicly available frameworks that implemented them. Both implementations allowed for a call tree to be represented by a token. The trace information of the test cases were represented by a list of tokens. The issue with Monge & Elkan was it attempted to find the best match with no regard to the ordering. Attempting to find the best match would increase the time taken, perform unnecessary computation for our requirements and increase the false positive rate. The trace information in Figure 3.2 can be used to show the difference between the metrics. The implementation of Monge & Elkan would match 'Method Call' in test case 1 to the 'Method Call' in test case 2, even though they were called at different times. Levenshtein did not search for a match, rather the metric only looked at the opposing method call. We prefer the direct matching as we were interested in the order of the method calls as well as what methods were called.

3.5 Pipeline

For the tool to help developers reduce the number of redundant tests, it has to analyse the information within an acceptable time frame. Comparing every test with every other while using all the available information (e.g. call tree) in some cases resulted in the analysis taking several days to complete. This is not considered an acceptable time frame relative to the test suite sizes. This issue arises when each of the spectrum of the test cases contain tens of thousands of method calls. We implemented a pipeline approach to get around this issue of lengthy analysis. Figure 3.3 visualises the idea. The figure shows the trace information is input into the start of the pipeline. The purpose of a stage is to determine the redundancy of test pairs. If a pair is more redundant than amount specified, it will be stored and analysed during the next stage. The goal is that each subsequent stage should soundly eliminate pairs

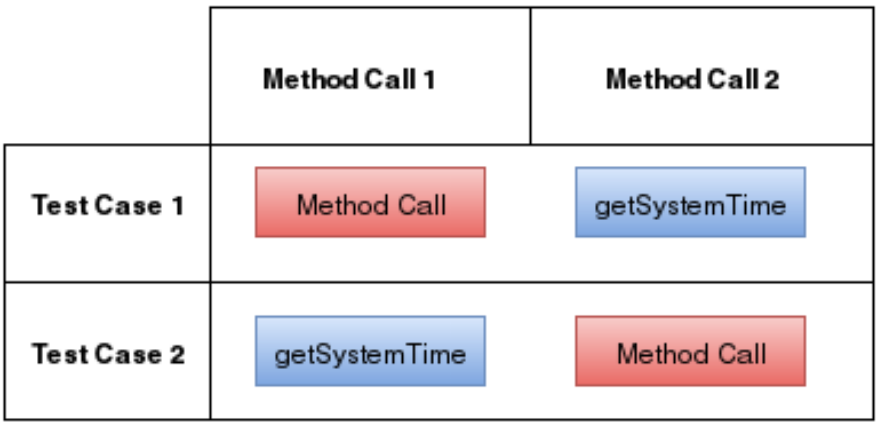

Figure 3.2: Trace information used to show the difference between Monge & Elkan and Levenshtein metric implementations. The Monge & Elkan metric will disregard ordering and measure these test cases as being the same. The Levenshtein metric examines one to one and will measure the test cases as having no similarities.

from consideration that the next stage would have removed. Heuristics can help achieve this goal. A heuristic would examine a subset of data while still eliminating test pairs that would have been removed in the subsequent stages.

Figure 3.3: Trace information goes in at the start of the pipeline. After each stage there should be a reduction of comparisons that the next stage has to complete. The last stage should be the most computationally heavy.

To further expand on the heuristic idea, the tool can use the spectra filters to examine the trace information in different perspectives. The particular filter of interest is the "unique method calls" spectrum. Inspecting a subset of the trace data increased the speed of the comparison but as a consequence, it decreased the confidence of the output being truly redundant. We can increase the confidence by then analysing the output test pairs with a "context tree" filter. An important part to emphasize is that no information is lost when information is filtered, rather some of it is ignored for that pipeline stage only. We see a pipeline set up in Figure 3.4. The first stage uses a "unique method calls" spectrum and pipelines the output into a "call tree" of K depth three. The three test cases at the top are the trace information input into the pipeline (Single method A and Z removed). The remainder of this section explores this example.

Pipeline Stage 1 The tool first filters the trace information with a "unique method call" spectrum. The three test cases are filtered into:

1. "B,C,D"
2. "E,H,J,M,Q"

3. “B,C,D”

Calculating the Levenshtein distance between each, it would show that test cases 1 and 3 are similar but test case 2 has no similarities to either. The test cases that are passed onto the next pipeline stage are test cases 1 and 3.

Pipeline Stage 2 The information used in this stage is shown at the top of the figure. The only test cases compared are test cases 1 and 3. Looking at the test cases, when filtered with a call tree of K depth three there are no similarities between the two method calls. This is because the top level calls are different. Hence the output of the pipeline as a whole is that there are no redundant test cases in the trace information.

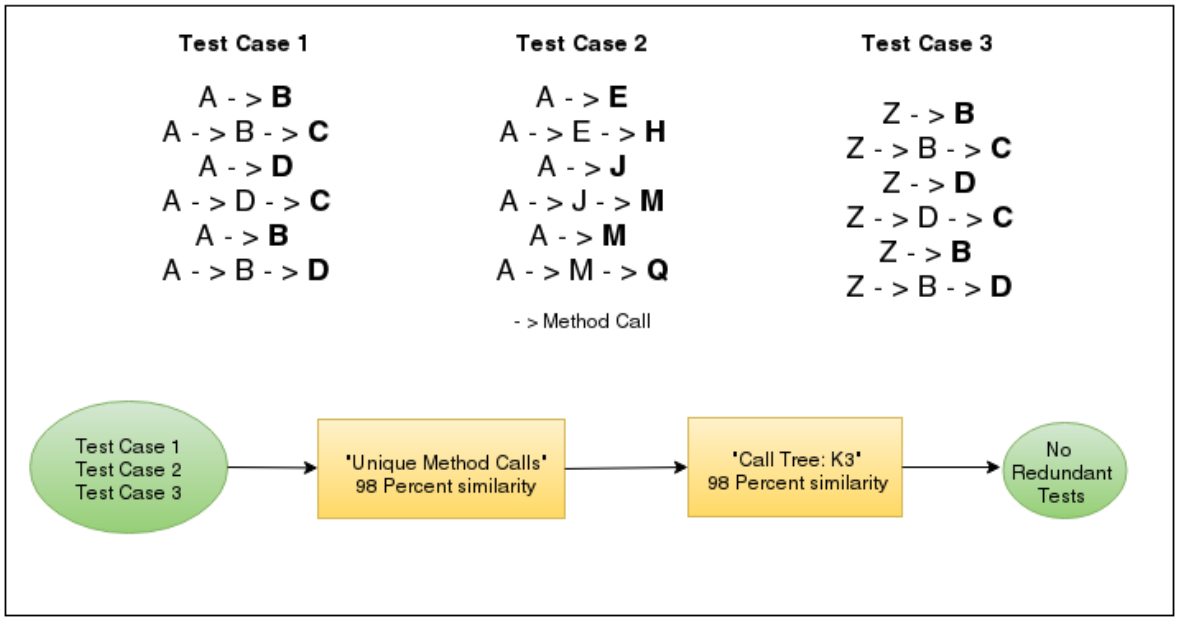

Figure 3.4: A two stage pipeline that analyses the trace information shown. The first stage uses a “unique method calls” filter with a 98% similarity and the second stage filters with a call tree of K Depth three with a 98% similarity. After analysing the data, no test cases are returned as being redundant.

An alternative heuristic to use is the “all method calls” spectrum. In comparison to the “unique method calls” filter, the “all method calls” filter further decreases the amount of comparisons that the final pipeline does. But it increases the time taken and memory consumed to analyse the stage. This approach may be used as the last stage in a pipeline of the length two. The other option, would be to have it as the middle stage in a pipeline of length three.

3.6 Tracing Parameter Values

The related work in this research area has explored using statement coverage while ignoring the parameter values of each method. It could be argued that parameters are redundant when the statement path is known. This is because the execution path of the method is known, and parameters do not add any more information. When identifying redundant tests with method information, parameters become crucial. They give insight into the exe-

cution paths that methods take and act as a proxy to the statement path, without storing the same amount of the data.

There were two variations of parameters that were considered to trace. Firstly, primitive types only. Primitives only represented a small number of parameters when running pre-experiments on the benchmarks. The parameters collected had a limited effect on the number of false positives identified and time taken. The second approach was to use reflection. Reflection is the ability to examine and modify objects at run time [2]. Since AspectJ gives direct access to the object's within the method parameters, reflection can be used. By using reflection, the tool can retrieve the fields of the object and turn them in a string to represent the state of the object. If there are objects within the object, a reference is returned to ensure no cycles occur. This string is then examined to remove any Java reference locations and then stored. The reflection is done through the Apache commons language library. We decided to use reflection as it was more representative of a method call in comparison to primitive types only.

3.7 Weighting

Maurer et al. [14] and Robinson et al. [16] found that test suites often had a set of methods that were in every test case, such as setup and tear down. These common methods could create false positives. To understand why, a redundant test is one where it is nearly or exactly a replication of another test. Since each method call within a spectra has the same weighting, the more setup and teardown calls made means that the exercise stage has decreased weighting overall. We can see an example of this in Figure 3.5. The figure shows test cases with different proportions of setup and tear down in relation to the exercise stage. It also shows how the size of the test case may have different proportions. This is when a test suite contains both large and small test cases that use the same setup and teardown methods. The figure shows that this would cause the small test case to have an increased portion of setup and teardown methods relative to the size of the test case.

Two different variations of weighting were considered. The first variation involved giving each method execution a weighting based on its call frequency, the higher the frequency, the lower the weighting. This would cause the more common methods to have less impact on the final result, but not be completely removed. The other variation that was considered, and used, was completely removing the most used method calls for each test suite. This involved determining method calls that were called in more than 80% of the test cases. This approach allowed us to remove only high frequent calls. The method executions to these calls were then removed at the start of the pipeline stage. During initial experiments, the first variation had less impact. The reason was that the frequent method calls were still having a large impact, even with a lower weighting. This meant that each benchmark needed to use different weightings. We used the removal variation as it was easier to tune to work well on every benchmark. The removal variation was applied at the test suite level and is referred to as weighting technique throughout the remainder of the report.

3.8 Summary

The tool implemented has a range of abilities to achieve the goal of identifying redundant tests in a test suite. To identify redundant tests, the tool can trace the information of a test suite. The analysis can then be executed on this trace information with some settings selected by the user. The user can determine the pipeline, call tree depth, whether to analyse

Figure 3.5: A diagram showing how the different size of the test case can be affected by setup and teardown methods. The "Exercise" refers to the core test method calls. The figure shows the case where a test suite contains both large and small test cases that use the same setup and teardown methods. It shows that this would cause the small test case would be made up of a larger portion of setup and tear down in comparison to the large test.

parameters and whether weighting is applied. The redundant tests are then output for the user to inspect and determine the appropriate action.

Chapter 4

Results and Discussion

For our tool to be useful it has to achieve the goal of identifying redundant test cases. To evaluate this goal, we need to execute and analyse experiments. The key idea of this chapter is to discuss our approach to the experiments and discuss the results. This chapter first discusses the method we used to perform the experiments. It then discusses the benchmarks used to execute these experiments on. After this, the remainder of the chapter reports on the outcome of the experiments. These experiments explore the use of our tool on a realistic benchmark suite and the important factors of interest are: the time taken, number of comparisons, the types of redundant tests identified and the overall cost of performance (time) vs precision (number and types of tests identified).

There are two points that the reader needs to be aware of. First, in the tables below, a '+' represents a significant increase, '-' a significant decrease and '=' represents no significant difference. Second, the graphs use a logarithm scale.

4.1 Experimental Method

The experimental method is key to producing believable and reproducible results. The next three sections discuss the process of our experiments. Before we could conduct the experiments, the benchmarks had to have their test suites traced. This involved setting the benchmarks up locally and then tracing the tests with the AspectJ class loader on a local machine. The trace information was then used in the experiments. The experiments consisted of using the tool with the different techniques discussed in Chapter 3. The experiments explored the following: Pipeline length, K depth, Parameters and Weighting. We solved the issue of running a large number of tests by using a grid computing system. After the results had been gathered, we used a Wilcoxon signed rank test [27] to determine whether the results were significantly different or not. The significance level used was 95% with a null hypothesis of the two sample's median being equal. Overall, seven different settings were run per benchmark with each setting being run 30 times. For every experiment, they consisted of a first pipeline stage using a "unique method calls" spectrum filter.

4.2 Environmental Considerations

As discussed in Section 2.2 there are a variety of challenges that present themselves when using Java to evaluate performance. The following section discusses these challenges.

4.2.1 Grid Computing

The experiments were executed on a grid computing system, with a total of 150 jobs queued on the grid at a single time. When using a grid system it is important to ensure every experiment is executed with the same hardware. The grid system allowed for particular types of hardware to be specified – 8GB DDR3 RAM, Linux 4.0.5 64 bit system and an Intel Core i7-3770 CPU running @ 3.40GHz.

4.2.2 Measuring Time

The time taken is an important variable in evaluating the performance of any application. Our tool uses the notation of CPU time to measure the time taken. The CPU time is the measure of time in nanoseconds that the tool spent on the CPU. A concern is if a user logs in during analysis, the grid system pauses the tool and the machine may move it into the on disk virtual memory. This is dependent on the amount of resources that the user needs. When the user logs out, the grid system will unpause the tool. The occurrence of pausing and unpausing may cause non-deterministic behaviour. We limited this issue by running the tests overnight when users were unlikely to log in.

We had to decide what the tool should measure in regard to the time taken. Ideally, it would measure the start up cost of the tool as well as the time taken to analyse the trace information. The start up of the tool involved loading the trace information into memory. The issue was when a large number of instances of the tool were running concurrently on the grid. These instances were attempting to access the same trace information file stored on the network. The stress on the network introduces a large amount of non-deterministic behaviour. Therefore, the tool disregards the start up time when measuring the total time taken.

Heap Size

The heap is the location in memory that the JVM uses to store objects that our tool creates. The amount of heap that was allocated to the JVM was 6GB for every benchmark on every run.

4.2.3 Software Environment

We used a concurrent-mark-sweep garbage collection strategy (default). This approach uses multiple threads to scan the heap, mark unused objects and recycle them [21]. It allows for a high throughput however, tends to use more CPU time than other strategies. This was deemed worth the trade off as memory was the bottleneck rather than the CPU.

4.3 Benchmarks

The benchmarks had to be realistic real world projects otherwise the experiments would be less credible. They were located by looking at popular Java frameworks, Github repositories and David Pearce's personal projects. For us to consider a benchmark, it had to be Java based, have a reasonable number of tests (40+) and be open source. The benchmarks that used either Ant or Gradle build tools were favoured due to their easier build process.

A variety of test types and sizes of benchmarks were used to fully evaluate the tool. Tables 4.1 , 4.2, 4.3 and 4.4 show information on the benchmarks. The information shown is a description of each benchmark, the authors, types of tests, number of tests traced, lines of

code and version used. We chose a range of test types. David J Pearce's projects contain end to end tests which run through a whole module at once. The other benchmarks test units of code at a time. Having a range of test types and sizes gives a more representative view and insight into the potential situations where different settings may not be helpful in achieving the goal of identifying redundant test cases.

An important thing to note is that the number of tests is only the amount that were traced. Some of the benchmarks produced up to 100,000 method calls per test, each with parameter information. This required the tool to use a large amount of memory when tracing the tests. Whiley and Ant were the benchmarks that did not have all their tests traced.

Benchmark	**Description**	**Authors**
Whiley - Wyc	The language contains an extended static checking tool in order to eliminate runtime exceptions through formal verification techniques.	David J Pearce
Spring - Core	"The Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications" [4] .	Community
Metric-x - Core	Records metrics about the JVM and application. Used for observing the behaviour of code in production.	Community
Jasm	A library used to disassemble or assemble Java Bytecode.	David J Pearce

Table 4.1: A table of the large benchmarks. The table describes each benchmark and displays the author(s).

Benchmark	**Description**	**Authors**
Ant	A tool used to automate the software build process.	Community
Imcache	A caching library. The library provides applications a means to manage cached data.	Cetsoft

Table 4.2: A table of the small benchmarks. The table describes each benchmark and displays the author(s).

Benchmark	**Type of tests**	**Number of Tests Traced**	**Lines of code**	**Version number**
Whiley - Wyc	End to End	304/498	26012	0.3.34
Spring - Core	Unit Tests	96442	58957	4.1
Metric-x - Core	Unit Tests	181	6211	3.3
Jasm	End to End	208	14651	1.0 (Master Branch)

Table 4.3: A table of the large benchmarks. The table shows the types of tests, numbers of tests, lines of code and version number.

Benchmark	**Type of tests**	**Number of Tests Traced**	**Lines of code**	**Version number**
Ant	Unit Tests	40/878	222011	1.9.6
Imcache	Unit Tests	102	8434	0.1.2

Table 4.4: A table of the small benchmarks. The table shows the types of tests, numbers of tests, lines of code and version number.

4.4 Experiment I - Pipeline Length Comparison

4.4.1 Motivation

The pipeline length would be expected to impact the time taken and comparisons performed in the final pipeline stage. Selecting a pipeline too small or too large will have varying negative effects on these. This experiment explores this trade off.

The goal of pipelining is to decrease the number of comparisons that the final stage has to perform. By increasing the length of the pipeline, it would be expected to lead to a decrease in the time taken and improve the trade off between precision (number and types of tests identified) and performance. This is because each stage will perform a decreased number of comparisons than the previous. Hypothesis 1 reflects this.

Hypothesis 1 *Increasing the length of the pipeline decreases the time taken and comparisons performed in the final pipeline stage*

4.4.2 Settings

There were two different pipeline lengths tested, two and three respectively. For both length's, the first and final pipeline stage was the same. The first stage was a "unique method call" filter. For the pipeline of length three, the middle stage used a "all method calls" filter. The final stage was a "call tree" of K depth 3. The pipeline did not use any other technique.

4.4.3 Results

Table 4.5 shows the results from comparing the pipeline of length two and three. It shows a mixture of results for the total time taken to analyse the data. Metrics-x, Ant, Spring and Jasm performed significantly better with a pipeline of length two in comparison to a length of three. Imcache had no significant difference and Whiley performed significantly better with a pipeline of length three in comparison to length two. Every benchmark had no significant difference between the number of redundant tests identified. They all produced the exact same number of redundant tests in both pipeline lengths.

Figure 4.1 shows how each benchmark reacted to the change in the pipeline length. It shows the number of test case comparisons that the final stage of the pipeline had to conduct. We are able to infer the number of comparisons that a single pipeline would have to perform by looking at the comparisons in the first stage. We are not able to measure the time taken for a pipeline of length one as it required a too much memory and took several days to complete.

4.4.4 Discussion

Inspecting Figure 4.1, it becomes clear that the pipeline length decreases the comparisons in the final stage. In appears that the length does not scale in every benchmark. This is shown by every pipeline of length two or three being a significant improvement on a length of one. Although they both perform better than a length of one, the majority of the benchmark's perform better with a two stage over a three. These results imply that the majority were spending more time on the second stage than they were saving from the reduced number of comparisons in the third stage. An explanation for this is that there may be little difference between the number of comparisons performed in the final stage in the pipelines of length two and three. Figure 4.1 shows that this is what is happening. The figure shows that between the '2' and '3' on the x-axis, there is limited difference. This means the third stage is

	Total Time	Redundant Tests Identified
Whiley	+	=
Jasm	-	=
Ant	-	=
Spring	-	=
Imcache	=	=
Metrics-x	-	=

Table 4.5: A table showing the significant relationship between the use of a pipeline with two stages and a pipeline with three for each benchmark. The table shows that pipeline of length two performs better than a pipeline of length three for all the benchmarks apart from Whiley. It is important to note that a '+' represents a significant increase and a '-' represents a significant decrease. This is for the total time taken and redundant tests identified. See appendix A.1 for the time taken graph.

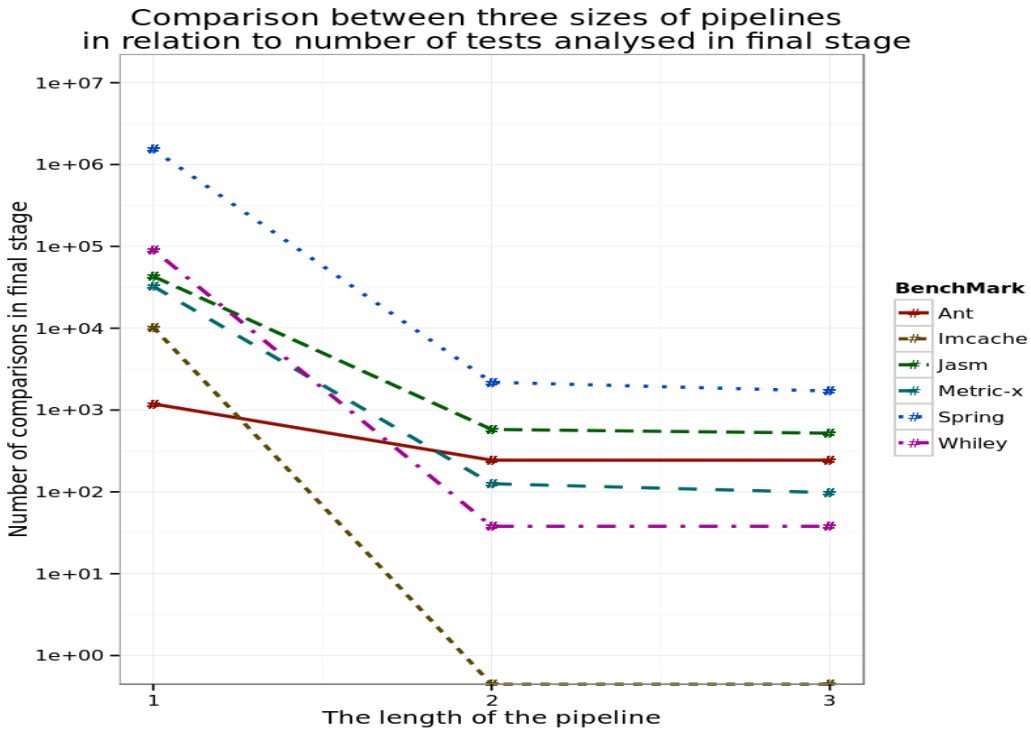

Figure 4.1: A figure showing the effect that the different pipeline lengths have on the number of comparisons during the final stage (most computationally heavy). It is noted that the number of comparisons for a pipeline of length can be infered from the number of comparisons performed in the first pipeline stage in a pipeline of length two.

only performing slightly less comparisons. To understand how this affects the outcome, we can inspect Figure 4.2. The first stage in the pipelines has a large reduction in the number of comparisons between the input and output. Inspecting the pipeline of length three, the second stage reduces the number of test pairs by a limited amount. This creates a situation where the number of identified redundant test cases has little change from stage two to three while still performing the extra comparisons. Taking this discussion into account, as well as the factor that a pipeline of length one could take days to complete, if at all. This leads us to

accept the first hypothesis, that increasing the pipeline length decreases the time taken and number of comparisons in the final stage. The optimal length of the pipeline may not be the same in every benchmark.

The change in the number of redundant tests identified is the other result to examine. The pipeline length of two had no significant difference to a length of three. If there was a change between the pipeline outputs this may be an issue in the pipeline setup. Since both pipelines of length two and three share the same settings in final stage, for the pipeline of length three, the second stage may be analysing with more information than the final. This would cause a different output between the two lengths.

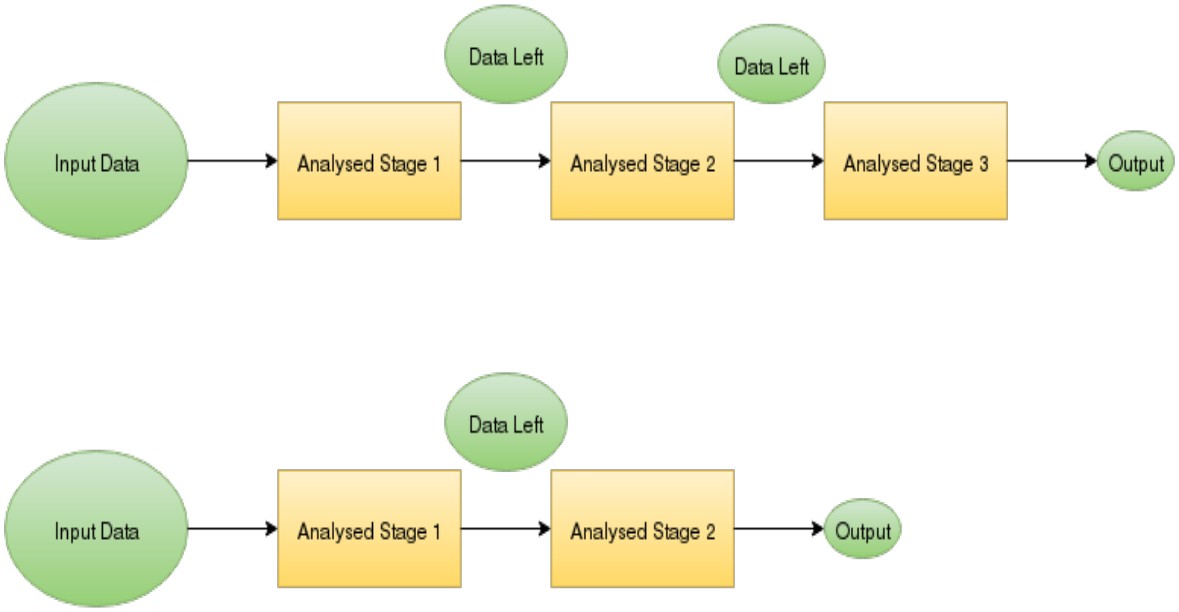

Figure 4.2: The circle size represents the data that is left in the pipeline. The figure shows that the difference that is between a two stage pipeline and a three stage only has limited room for a reduction in test cases identified. This shows the time taken to execute the extra stage costs more than it saves.

4.5 Experiment II - K Depth Comparison

4.5.1 Motivation

One of the abilities of the tool is to trace a call tree to the depth of K. This experiment is to determine the impact that altering the depth of K has on the number of tests identified.

Increasing the depth of the call tree leads to an increase in the amount of data that is analysed, intuitively making it more difficult for test cases to be the same. We expect a reduction in the number of tests identified to be reflect this. The extra analysis performed implies an increase in the time taken. This time taken should not be substantial and we expect that the decreased number of tests identified is worth the performance decrease. Hypothesis 2 reflects this.

Hypothesis 2 *The decreased number of tests identified from increasing K Depth outweighs the performance decrease.*

4.5.2 Settings

There are three K depth's explored, one, two and three respectively. It is important to note that a Wilcoxon signed-rank test was not performed as it is a two-sample test. We considered other significant tests to perform the test for three samples, however we felt the graphs were enough to convey the results. The first stage was a "unique method call" spectra and the final (second) stage was a "call tree". The depth of the final stage varied between one to three.

4.5.3 Results

Ant, Whiley and Spring appear to be the most responsive to the depth of K as shown in Figure 4.3 although by a limited amount. The use of a log scale should be taken into account when looking at the graph as it may make the changes appear less notable. Figure 4.4 shows the differences in time taken. Overall, for every benchmark there was not a large amount of change however it shows that as the K depth increases, the time taken increases.

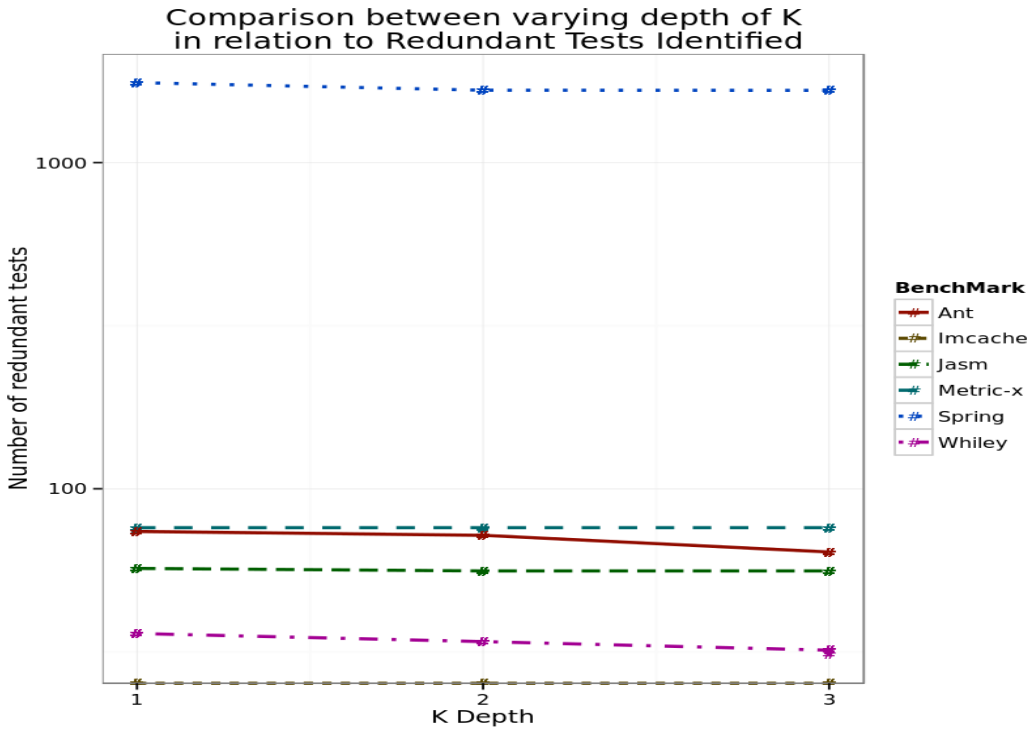

Figure 4.3: A figure showing the effect that a change in the depth of the call tree has on the number of redundant tests are identified.

4.5.4 Discussion

There are two main take away points from the results. First, it implies that when no other settings such as weighting or parameters are used, the depth has limited effect on the number of redundant test cases identified. Second, the number of comparisons that were output from the first pipeline stage may cause there to be a limited differentiation. When the first pipeline stage outputs a limited number of pairs, the final number of redundant tests identified would be similar regardless of the K depth specified. This situation is similar to the

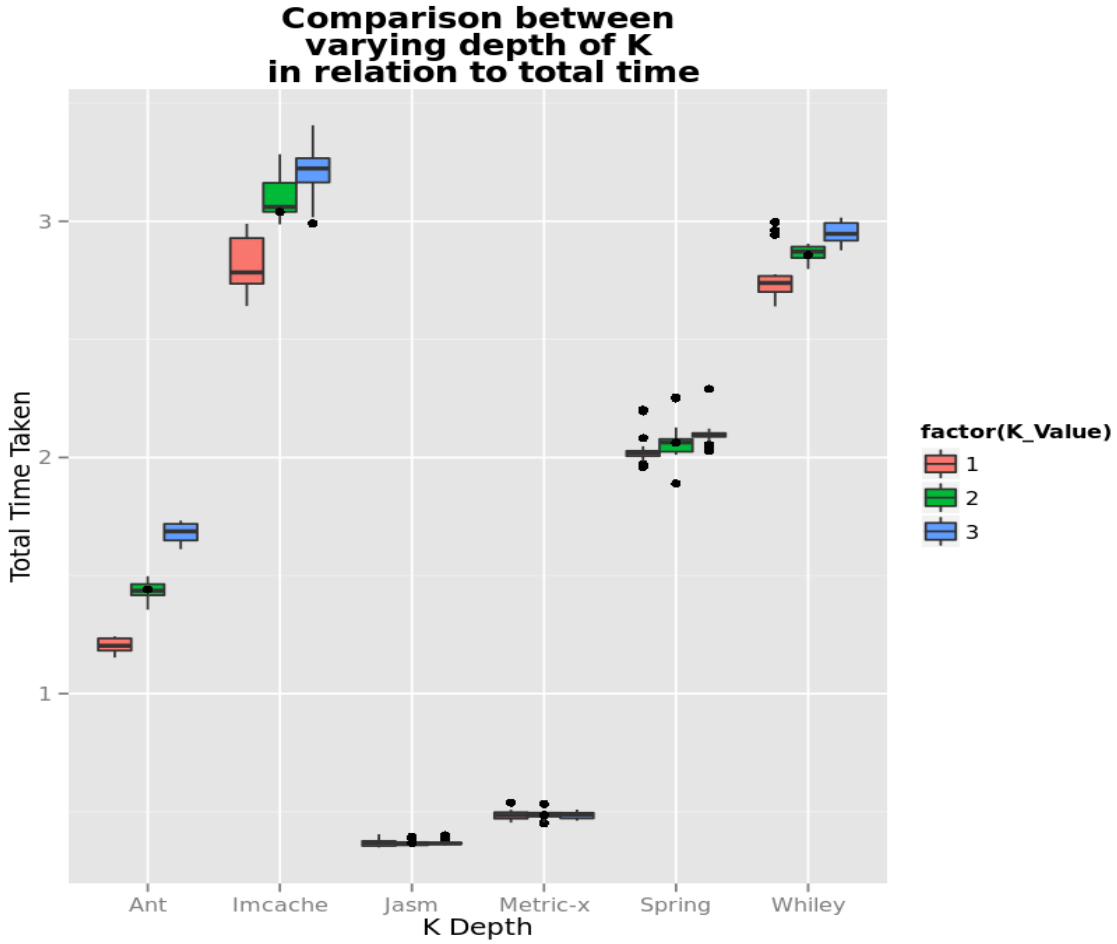

Figure 4.4: A figure showing the relationship that using a different K Depth has on the total time taken to analyse the data.

one discussed in Section 4.4 and occurs in Whiley, Metric-x, Imcache and Jasm. The results from this experiment and Experiment I indicate that by using a "unique method call" filter and analysing a subset of the data, the tool is able to remove a large majority of the non-redundant tests. We can examine Figure 4.5 to confirm this. Inspecting the figure, it shows that the first stage ("unique method calls") in the pipeline removes a large portion of the comparisons that the following stages has to perform. However, we need to inspect the number of output. The difference between pipeline stage two and the output is a limited amount. This shows that the heavy computational analysis stage only removes a small set of extra tests. This implies that the "unique method calls" has a good accuracy of identifying redundant tests.

To further discuss the hypothesis, we need to examine the change in time taken. Increasing the depth of K has varying effect's on the time taken to analyse a benchmarks trace data. Figure 4.4 shows the differences. It shows that Ant, Imcache and Whiley have larger differences than the rest. In perspective, these equate to around half a minute difference. Although increasing the depth has an smaller effect than expected on the number of tests identified, the impact on time taken is not substantial. One of the reasons is that methods generally have a uniform call structures. When applications have low cohesion, the internal functions of a class (or module) are tightly bound [30]. This means that for tests, the call

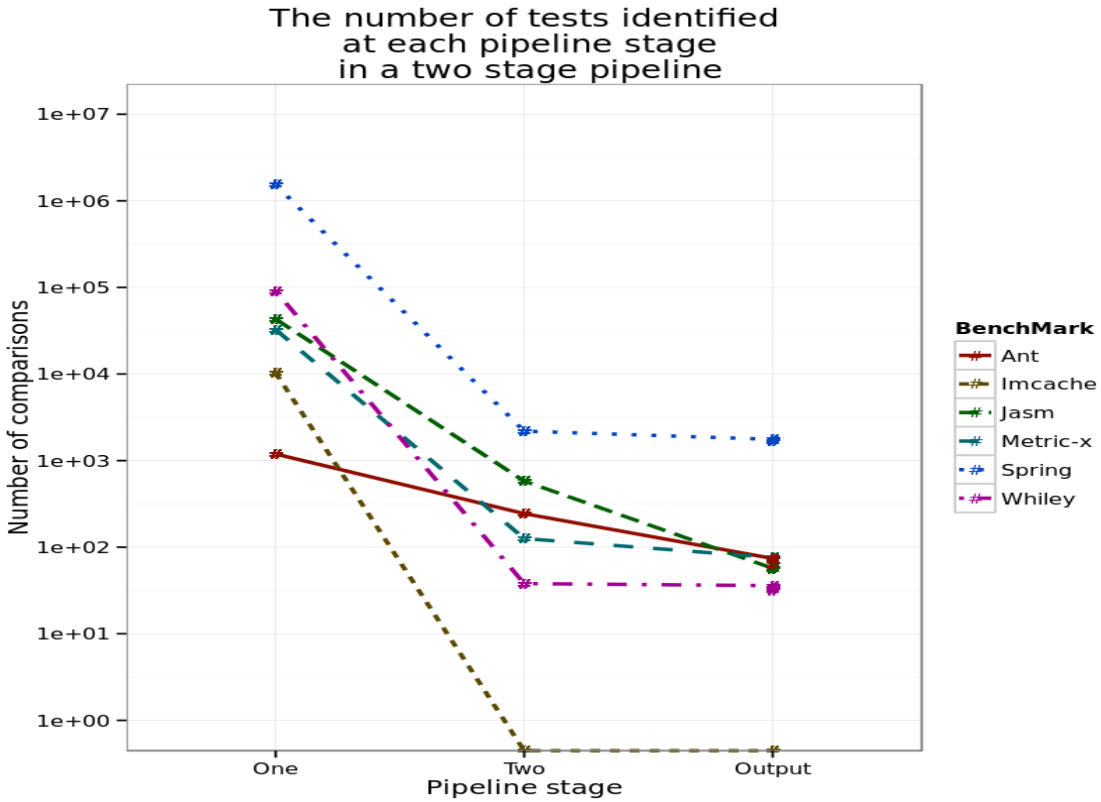

Figure 4.5: A figure showing the the number of comparisons that each stage performs. The pipeline contains two stages and is from Experiment I. The graph shows the first stage is able to reduce a large number of the comparisons that the subsequent stage has to perform. **Note** this differs from Figure 4.1 as this looks at the number of comparisons in each stage. Figure 4.1 examines the comparisons performed in the final stage with varying pipeline lengths.

path will often be the same if the method set is the same, with the difference often being parameters passed. Taking these details into account, we can accept the hypothesis 2, with the results implying that for some projects the calling context does not have to be particularly deep.

4.6 Experiment III - Parameter Comparison

4.6.1 Motivation

Parameters show the different contexts that a method is called in. Retrieving this information gives us more confidence in determining whether two tests are redundant. This experiment explores the impact on performance and precision.

Two factors need to be taken into account when discussing the impact of parameters on the time taken. First, it is intuitive that analysing the extra data generated increases the time taken. The amount increased is interesting, parameters only add extra computation when the method execution of the given K depth is exactly the same. For example, if the test execution $A \rightarrow B \rightarrow C$ is compared to $A \rightarrow B \rightarrow F$ then the parameters are not taken into account. Therefore parameters may have limited effect on the time taken.

The amount of time that the VM spends garbage collecting would also increase the time

taken. Due to the increased amount of information in memory, the garbage collection process will have to execute more frequently for the analysis to continue to run. This would cause non-deterministic attributes to become more wide spread. Hypothesis 3 and 4 reflect that parameters increase the precision and the total time taken in comparison to the pipeline of length two in Experiment I.

Hypothesis 3 *Using parameters during the analysis increases the time taken when using a pipeline of length two*

Hypothesis 4 *Using parameters during the analysis increases the precision when using a pipeline of length two*

4.6.2 Settings

The use of parameters is compared directly to the pipeline of length two used in Experiment I. The first stage was a "unique method call" spectra and the final (second) stage was a "call tree" of K depth 3. The differences between executions were the analysis included parameters.

4.6.3 Results

Table 4.6 shows the comparison information between parameters and no parameters. It shows that there was a significant increase for every benchmark in regard to time taken. The table also shows that every benchmark had a significant decrease in the number of redundant test cases identified. This is not the case in Imcache due to it having zero redundant test cases identified in both. Examining Figure 4.6 – Ant, Imcache, Jasm and Spring appear to be affected by an increased variance between executions.

	Total Time	**Redundant Tests Identified**
Whiley	+	-
Jasm	+	-
Ant	+	-
Spring	+	-
Imcache	+	=
Metrics-x	+	-

Table 4.6: A table showing the significant relationship between the use of parameters and no parameters for each benchmark

4.6.4 Discussion

The Table 4.7 matches what is expected. For every benchmark, using parameters significantly increases the time taken. Examining Figure 4.6 shows how the benchmarks react with more detail. The most interesting would be Jasm. Without parameters, it analyses the data in less than one minute, with parameters, it takes just under five minutes. This may be indicative of the results discussed in Section 4.5. The section discussed how increasing the K depth had little impact on the final outcome. Taking this into account, the call trees were matching the majority of the time and led to parameter information being examined more often, sequentially causing an increase in time taken. A factor that may have an accumulative effect are the setup and tear down methods. If these methods are a large majority of

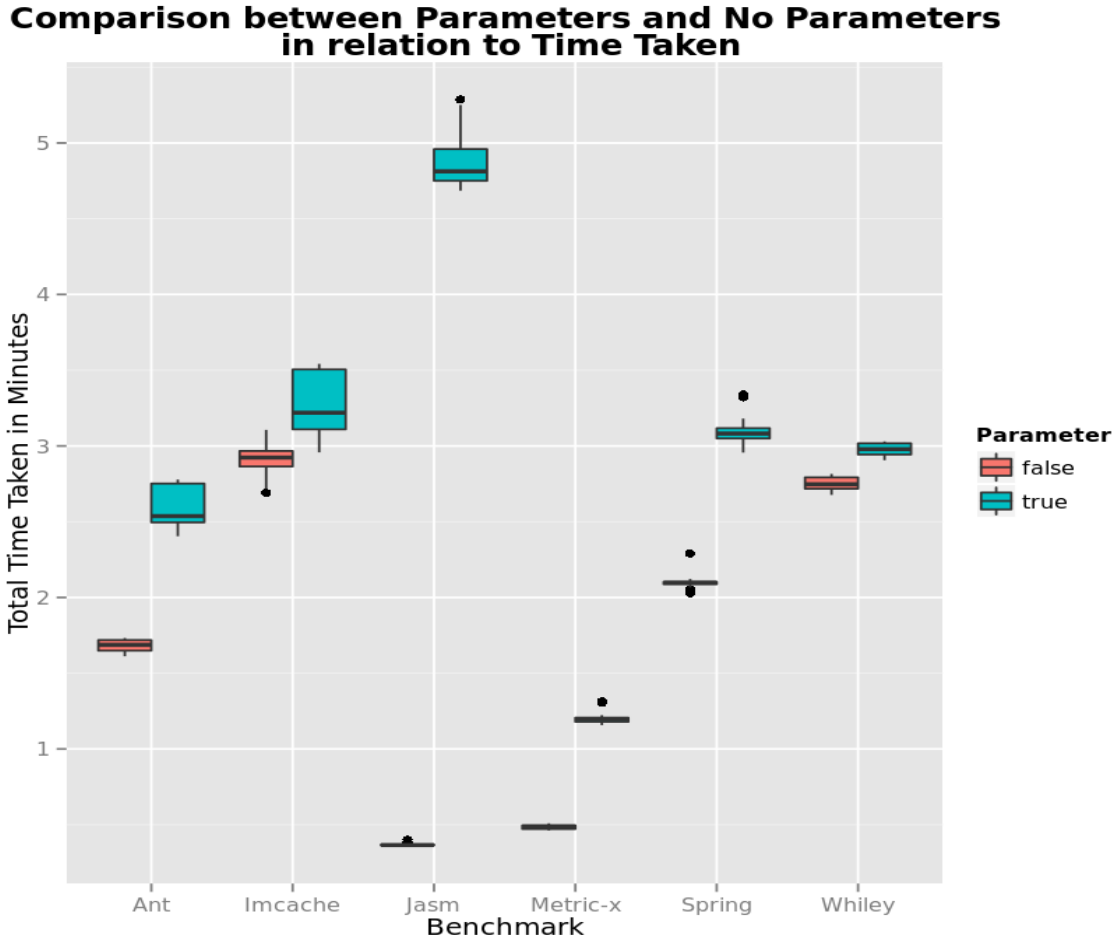

Figure 4.6: A figure showing the relationship that using parameters has on the total time taken to analyse the data.

the method calls, and match each other for the K depth, this would further increase the time taken. This allows us to accept hypothesis 3 that parameters increase the time taken.

Looking at Figure 4.7, it confirms that every benchmark reacted with a substantial decrease in redundant tests identified. Whiley was the least affected. This reiterates the discussion in Section 4.5 that K depth is not enough of a factor to identify redundant test cases. To determine the precision improvement, we need to analysis the number of false positives identified. At this stage, there is not enough information to discuss this, therefore Hypothesis 4 will be revisited in Section 4.8 where a stronger conclusion can be made.

4.7 Experiment IV - Weighting Comparison

4.7.1 Motivation

Utilising the weighting technique is an attempt to remove any false positive tests identified. The experiment attempts to identify if applying the weighting technique has potential to remove these false positives. It also provides information concerning the impact on performance.

As discussed in Chapter 2, much of the related work encountered difficulties with test

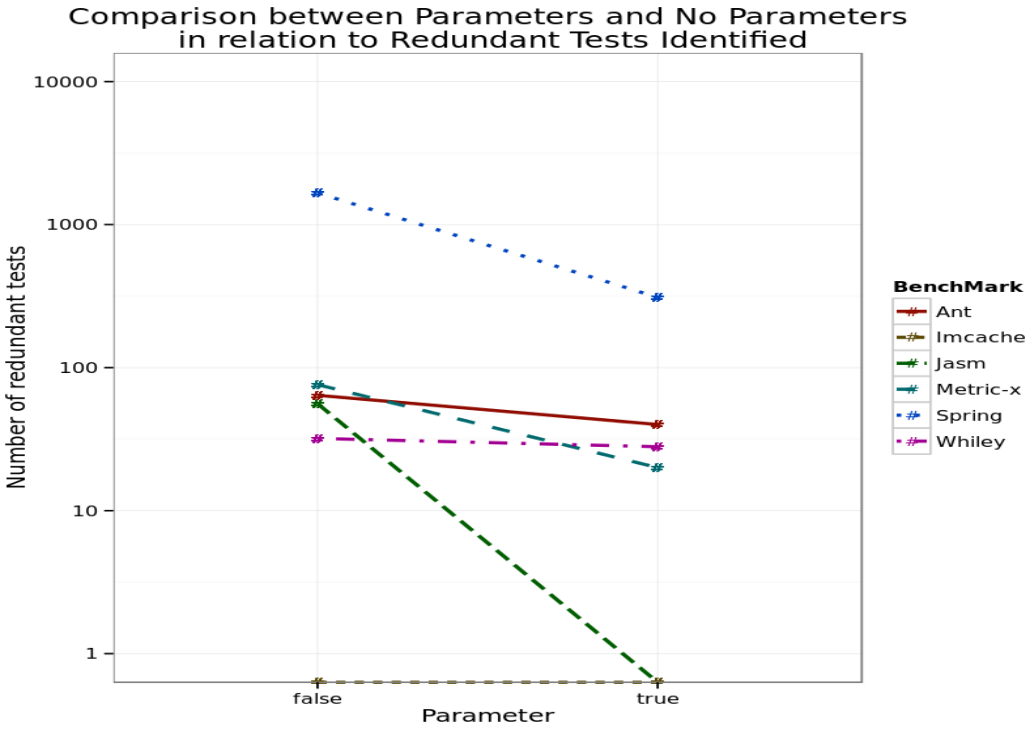

Figure 4.7: A figure showing the effect that using parameters has on the number of redundant tests are identified.

cases sharing setup and teardown method calls. This results in a high false positive rate. Intuitively, this makes sense when the test cases are small in comparison to the setup and teardown methods where these methods represent a large majority of the trace data. The approach of removing a part of the most executed method calls meant that the amount of data decreases. This should reflect onto the results by showing a decrease in the time taken, as per hypothesis 6. The effect the weighting technique has on the number of redundant test cases should be dependent on the benchmark. This is because removing the most common method calls should imply a decreased false-positive rate, but also may increase the number of actual redundant tests picked up. Hypothesis 5 reflects our expectation on how weighting will impact the false positives.

Hypothesis 5 *Weighting decreases the number of false positives compared to a standard two stage pipeline*

Hypothesis 6 *Weighting decreases the time taken compared to a standard two stage pipeline*

4.7.2 Settings

The use of weighting is compared directly to the pipeline of length two from Experiment I. The first stage was a “unique method call” spectra and the final (second) stage was a ”call tree” of K depth 3. The differences between executions was the use of the weighting technique.

4.7.3 Results

Weighting has no clear impact concerning the total time taken. We see this in Table 4.7 – Whiley, Ant and Imcache had a significant increase in the time taken to analyse. Jasm, Spring and Metric-x had a significant decrease in the time taken to analyse. The majority – Whiley, Ant, Jasm and Metrics-x had a significant decrease in the number of redundant tests identified. Spring was the only benchmark where weighting significantly increased the number of redundant tests identified.

	Total Time	Redundant Tests Identified
Whiley	+	-
Jasm	-	-
Ant	+	-
Spring	-	+
Imcache	+	=
Metrics-x	-	-

Table 4.7: A table showing the significant relationship between the use of weighting and no weighting for each benchmark

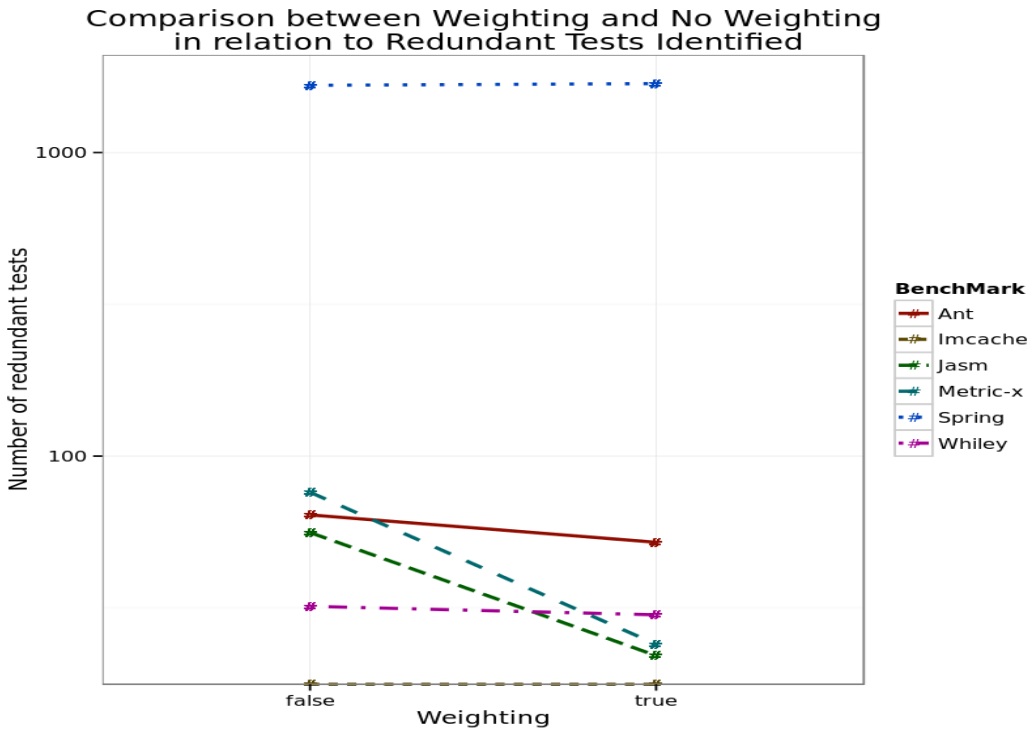

Figure 4.8: A figure showing the effect that using weighting has on the number of redundant tests are identified.

4.7.4 Discussion

A reduction in the false positive rate was the weighting technique's primary goal. Spring was the only benchmark that had a significant increase in the number of redundant test cases

identified. The other benchmarks had a significant decrease. At first, this makes it appear that by using weighting it may solve some of the issues Maurer et al [14] and Robinson et al. [16] identified. To confirm this, we need to inspect the types of tests that were identified. At this stage, there is not enough information to perform this discussion. Hypothesis 5 will be revisited in Section 4.8 where a stronger conclusion can be made.

Table 4.7 shows a mixture of results in regard to the total time taken. Two out of the three benchmarks that had a significant increase in the total time were small benchmarks – Whiley, Ant, Imcache. This may imply that the size of the benchmark has some relation to the effect of weighting on the time taken. One reason is that the tool calculates weighting once per analysis stage. The weighting calculation has a linear relation with the number of test cases. In comparison, the tool compares every test case to every other. This is a squared relation with the number of test cases. Figure 4.9 shows that the less test cases there are, the closer the linear relation is to the squared and the more impact the linear weight calculation has on the time taken. This may explain a relation between size and time taken. Appendix B.1 shows the overview of the impact on the time taken. The figure shows there was no general impact from using weighting on the benchmarks and for some benchmarks, weighting is taking more time than it saves. Taking the discussion into account, it allows us to invalidate hypothesis 7 and that weighting does not affect time taken.

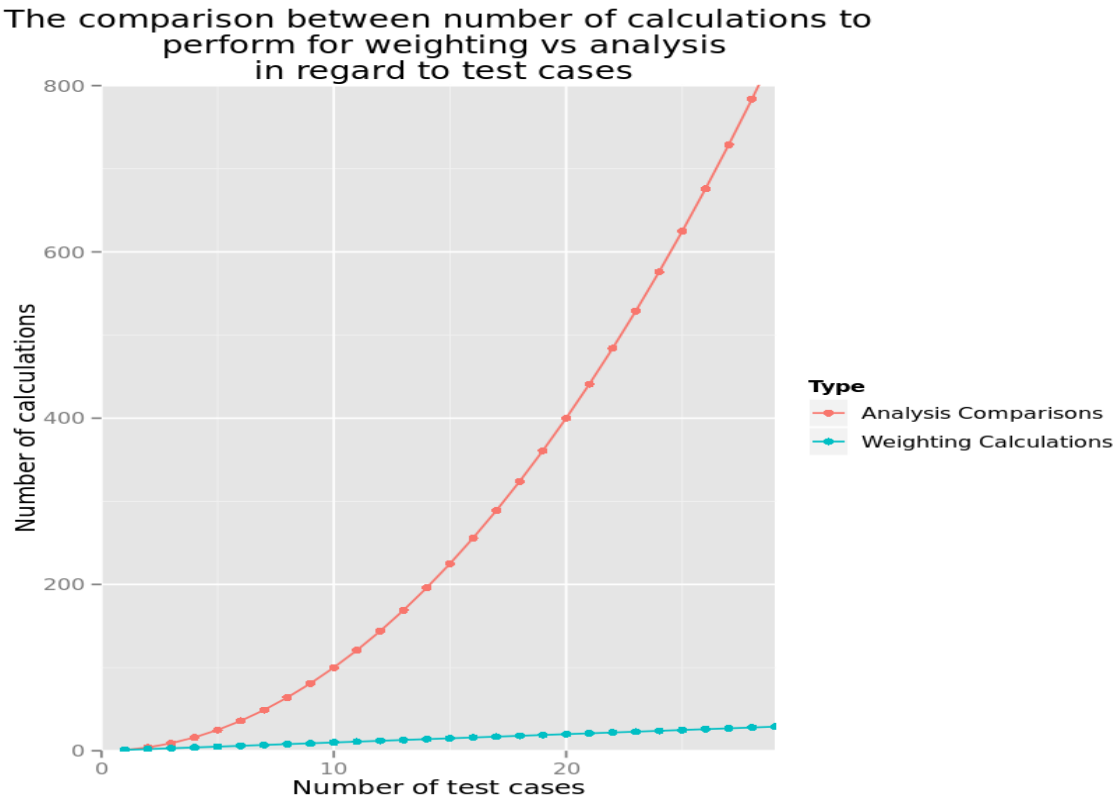

Figure 4.9: A graph showing the growth of a n squared relation in comparison to a linear relation.

4.8 Additional Analysis

It is not enough for the tool to identify tests. We need to examine these tests and verify that the tests identified are truly redundant and not false positives. To achieve this, we manually classified the identified tests from the experiments into types of redundancy. These are shown in Tables 4.8 and 4.9 and referred to as coding tables. The names given to the types are indicative of the main differences between the tests. We also classified data from a pipeline using both parameters and weighting. As the classification was a lengthy process, only Whiley and Metric-x had their output tests classified. This section will continue the discussion from Sections 4.6 and 4.7 and conclude the hypotheses 4 and 5.

4.8.1 Parameters

In Section 4.6, we discussed that parameters caused a decrease in the number of tests identified. We need to inspect the types of tests identified to conclude hypothesis 4. The hypothesis states that parameters increase the precision in comparison to a two stage pipeline. Inspecting Table 4.8 and 4.9 gives insight into the types of tests identified. The Whiley benchmark coding shows that parameters remove the limited redundant tests as well as some different array value tests. The Metric-x coding demonstrates a reduction in the number of the different parameter value tests identified substantially, from 52 to 8. However, we lose the 2 similar tests that were identified. It is important to note that since we classified the tests while not being the developers of the Metric-x benchmark, the similar tests may not be redundant and removing them may be the correct action. These codings show that we are able to reduce the number of redundancies identified but it may cost us potential similar tests. Taking these coding tables into account, it allows us to accept hypothesis 4 that parameters increase the precision.

4.8.2 Weighting

Revisiting hypothesis 5, it states that weighting decreases the number of false positives compared to a standard two stage pipeline. To confirm this, examining Table 4.8 and 4.9 gives insight into the effect of weighting. Comparing the two columns for the Whiley coding, Pipeline 2 and Weighting, the only difference is the removal of the two limited redundancy test cases that were picked up by Pipeline 2. This shows the weighting technique is able to reduce the impact that the similar set up and tear down methods have on the analysis. By removing the method executions that were common, this led to a decrease in the number of false positive tests identified. The Metric-x coding displays a similar result. The weighting technique reduces the number of parameter value redundancies identified as well as the limited redundancies, but does not completely remove them. This is interesting and implies that the number of setup and tear down methods that remained in the data analysed was still a large part of the data. We are able to accept hypothesis 5, although work remains to improve the weighting method to remove redundant tests completely.

4.8.3 Further Discussion

Neither weighting or parameters were able to fully remove the tests with limited redundancy in Table 4.9. The table suggests that the combination of parameters and weighting may have some appeal by removing all eight limited redundancy tests. At the same time, the similar tests are also removed which is a negative outcome. This suggests that the combination may have some applicability to other benchmarks and may warrant future experiments.

	Whiley			
Types of redundancy	**Pipeline 2**	**Weighting**	**Parameters**	**Parameters and Weighting**
Different Equation Value	6	6	6	6
Different Equation Sign	8	8	8	8
Different Array Values	2	2	0	0
Same	10	10	10	10
Limited Redundancy	2	0	0	0
Rearranged Equation	2	2	2	2
Extra if statement	2	2	2	2
Total	32	30	28	28

Table 4.8: A table displaying a list of coding's for the Whiley Benchmark for four of the different techniques used. The tests identified have been coded under a type of redundancy. These types of redundancies allow us to see how many false-positives were identified as being redundant.

	Metric-X			
Types of redundancy	**Pipeline 2**	**Weighting**	**Parameters**	**Parameters And Weighting**
Different Parameter Value	52	10	8	4
Different Object Type	6	2	2	0
Different Array Values	8	6	4	6
Similar	2	2	0	0
Limited Redundancy	8	4	6	0
Total	76	24	20	10

Table 4.9: A table displaying a list of coding's for the Whiley Benchmark for four of the different techniques used. The tests identified have been coded under a type of redundancy. These types of redundancies allow us to see how many false-positives were identified as being redundant.

4.9 Limitations

- A major limitation to the study was the amount of RAM used. For the Whiley and Ant benchmarks, the tool had high RAM usage when storing the parameter data and call tree information. This limited the amount of tests that the tool was able to trace. The experiments may not fully represent the total redundancy in these two benchmarks.
- The tool described throughout this report would not be able to pick up when a test case subsumes another. The tool only looks at the method call information, as such gains no new information by identifying method call subsumed tests. It is then difficult gauge this type of redundancy within the test suites.
- Some benchmarks performed better with a two stage pipeline in comparison to a three stage, and vice versa. A range of different settings were tested to identify close to optimal settings for the second stage. However, this does not mean that the settings chosen were optimal and may have skewed the data slightly.
- The tool applies the weighting technique once per stage where required. Performing the calculation once per stage adds unnecessary computations. This does not affect

the experiments as weighting is ever only executed once per pipeline.

- The setup time was not measured in the time taken. This means that to perform the analysis, it would normally take longer than the times shown on the graph.

Chapter 5

Future Work and Conclusions

This chapter will first discuss potential future work. The work will be aimed at increasing the tools ability to identify redundant tests. The tool and experiments are then reflected on and finally concluded.

5.1 Future Work

Throughout the project, three main areas have been identified for future work: The ability to handle larger test suites, tracing statement information and lastly, using both statement and method information together.

- **Handle larger test suites.** The main objective would be to increase the ability of the tool to handle larger test suites. Currently, the tool relies on RAM to hold the trace and analysis information however, the amount of data held can grow rapidly. A similar idea was implemented by Nelson [20] while profiling execution data. The approach would allow the tool to handle arbitrary large suites. The down side would that the speed of the tool may decrease as the read/write to the database can be slower than to RAM. This may be negligible as databases optimize their read and write by using a certain amount of memory. Giving developers this option would increase the capability of the tool.

- **Trace statement information.** The tool has the ability to trace method call information and has potential to be used on large test suites. This approach may not be the best when we are only looking at smaller test suites and wanting a better accuracy. Another approach that could be implemented is tracing the statement coverage information. This would allow us to compare the statement and method coverage information directly and give us more insight into the issue of identifying redundant tests. Tracing the statement information would allow us to further explore the use case mentioned in Chapter 1. The use case was the ability to split the test suites into two, one containing the non redundant test cases and the other was the original suite. This could be further expanded with statement coverage information. By identifying when a test subsumes another, the test subsumed could be moved into another suite. If the larger test fails, the subsumed tests could then be ran to help pin point the error in the code.

- **Combining Statement and method information.** The method call information acted as a good heuristic. It is difficult to judge how good method calls are at identifying redundant test cases without any comparison to statement coverage or large scale redundancy and bugs added. One application could be to combine the statement and

method information together. The tool could use the method information as a heuristic and statement coverage could then explore the test cases identified by the heuristic.

5.2 Conclusions

There were two main contributions of the paper as discussed in Chapter 1:

- Create a tool for identifying redundant JUnit Test cases (Chapter 3)
- Analyse the different strategies for identifying redundant test cases through experimenting on realistic benchmarks (Chapter 4)

Throughout the report, the tool has been discussed along with the different techniques implemented. The tool first had to trace the data. This was achieved through the AspectJ framework. After the test suites were able to be traced, the trace information then had to be filtered with different spectra. The ability to filter out information was one of the key features of the tool. This allows developers to trace information, then run a range of different filters over the data without being constrained to one spectra. The three spectrum filters were "unique method calls", "all method calls" and "call tree". The other key feature of the tool was the pipeline. Integrating it with the spectra filters allowed for the "unique method calls" to be used as a heuristic to reduce the number of comparisons that the subsequent stages had to perform. The tool then was altered to trace the parameter information from the test cases. This was achieved by using reflection on the parameter objects. Finally, the ability to apply a weighting was implemented. This was achieved by removing the top 20% of method calls.

Method Call Information The use of method call information was a good candidate to identify redundant tests. They allow the tool to look at test suites that produce a larger amount of total information, in particular end to end tests ie Whiley and Jasm. As mentioned in the Future Work section above and discussed in Experiment II, the method calls appeared to be particularly good at being used as a heuristic. Filtering with the "unique method calls" spectrum, the tool was able to remove a large portion of the test case comparisons in the final stage had to perform while retaining good precision.

Pipeline Pipelining was one of the critical aspects of the project. It not only lead to a decrease in the time taken but also the memory consumed. Important to note is that each benchmark had an optimal length of the pipeline, using a length smaller or larger than the optimal caused more time to be taken to analyse than was saved. The optimal of the benchmarks was generally a length of two. Pipelining particularly worked well with the use of a "unique method call" spectrum filter.

Call tree depth The experiments showed that as the depth of the call tree increased, there was a limited decrease in the number of redundant tests identified. The result was lower that what was expected. This implies that by increasing the K depth without any other technique, there is limited improvement. However the precision increase from a depth of three rather than one is still worth the increase in time.

Parameter Reflection allows for our tool to retrieve the object in a representative string. This allows more insight into the state of the parameter objects and the nature of them. It also meant more data had to be held and analysed resulting in an increase in the time taken. There exists a trade off between time taken and confidence of redundancy when using parameters that the developers can choose.

Weighting The weighting technique implemented attempts to improve the accuracy of the tool by removing false-positives while improving the time taken to analyse. Experiment IV discussed how weighting was able to reduce the level of false-positives but needs further improvement.

The project has explored the use of higher level information than previously explored for identifying redundant test cases while examining how different techniques can impact the results. Overall, the tool was useful in achieving the goal of identifying redundant test cases as shown through the experiments.

Appendices

Appendix A

The effect of pipeline size in regard to the time taken

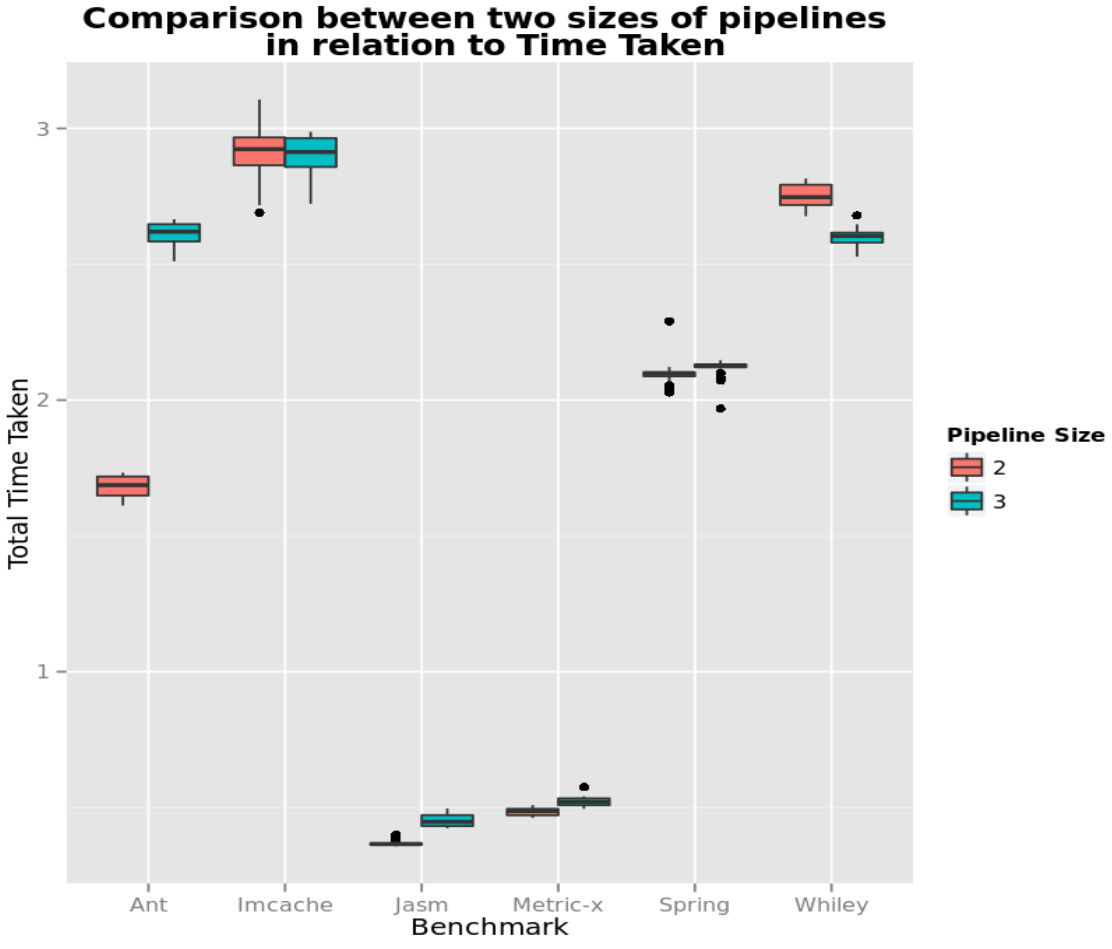

Figure A.1: A figure showing the relationship that using different pipeline sizes has on the total time taken to analyse the data.

Appendix B

The effect of weighting in regard to the time taken

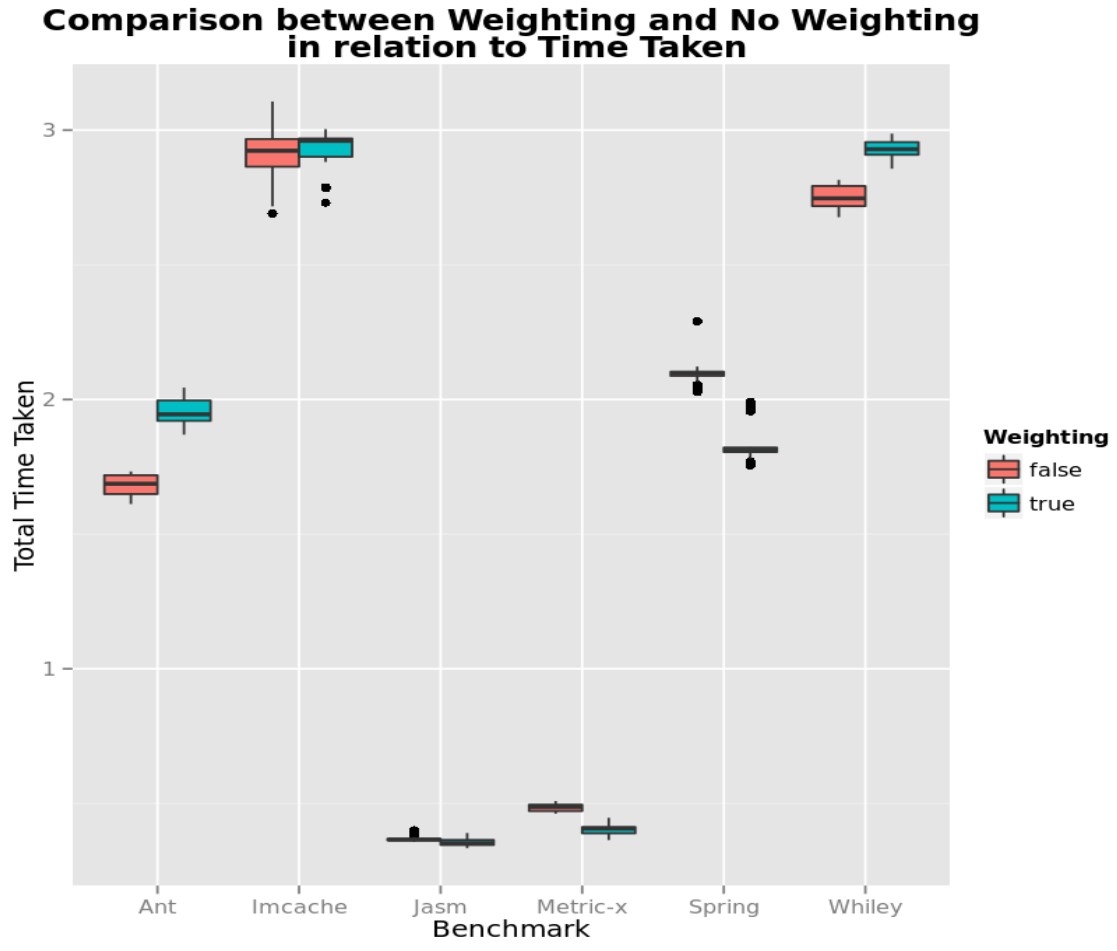

Figure B.1: A figure showing the relationship that using weighting has on the total time taken to analyse the data.

Appendix C

The effect of weighting and parameters combined in regard to the time taken

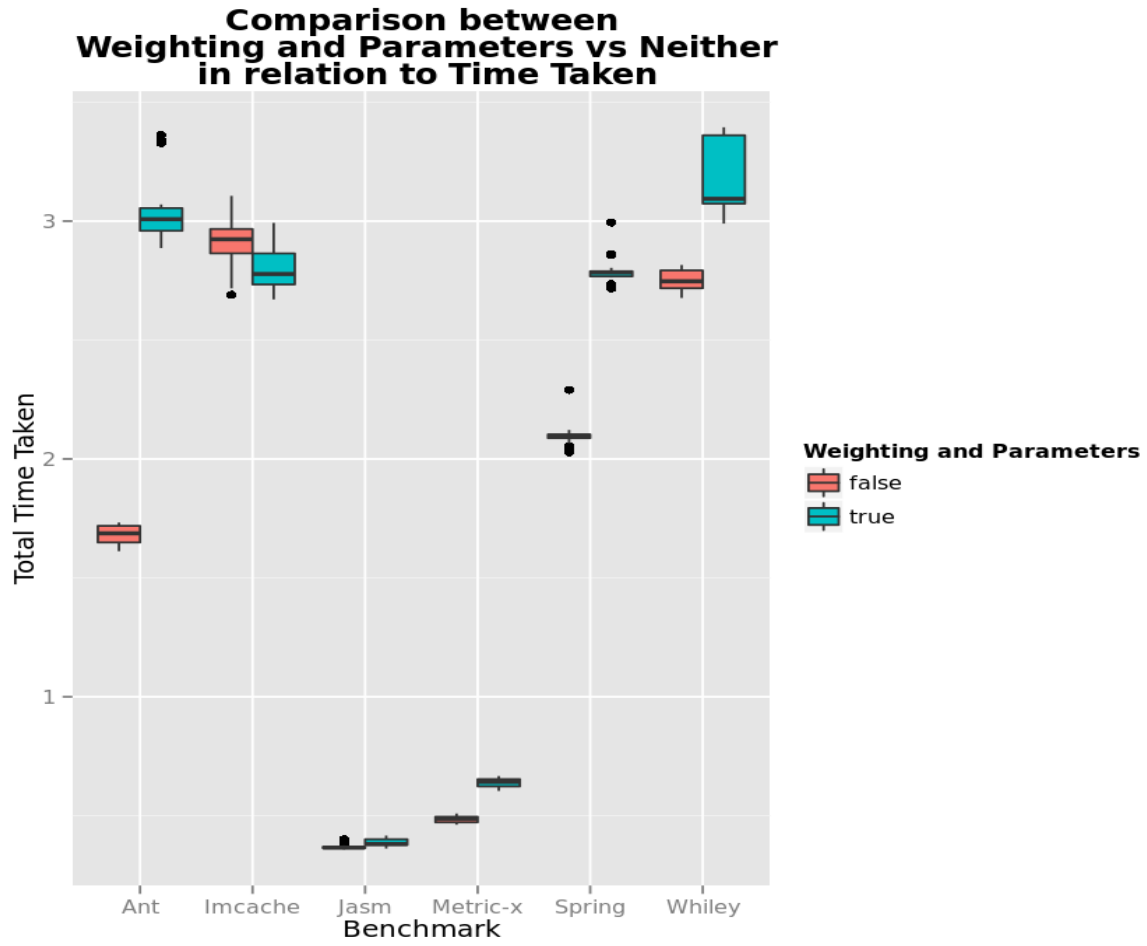

Figure C.1: A figure showing the relationship that using weighting and parameters combined has on the total time taken to analyse the data.

Appendix D

The effect of weighting and parameters vs parameters in regard to the time taken

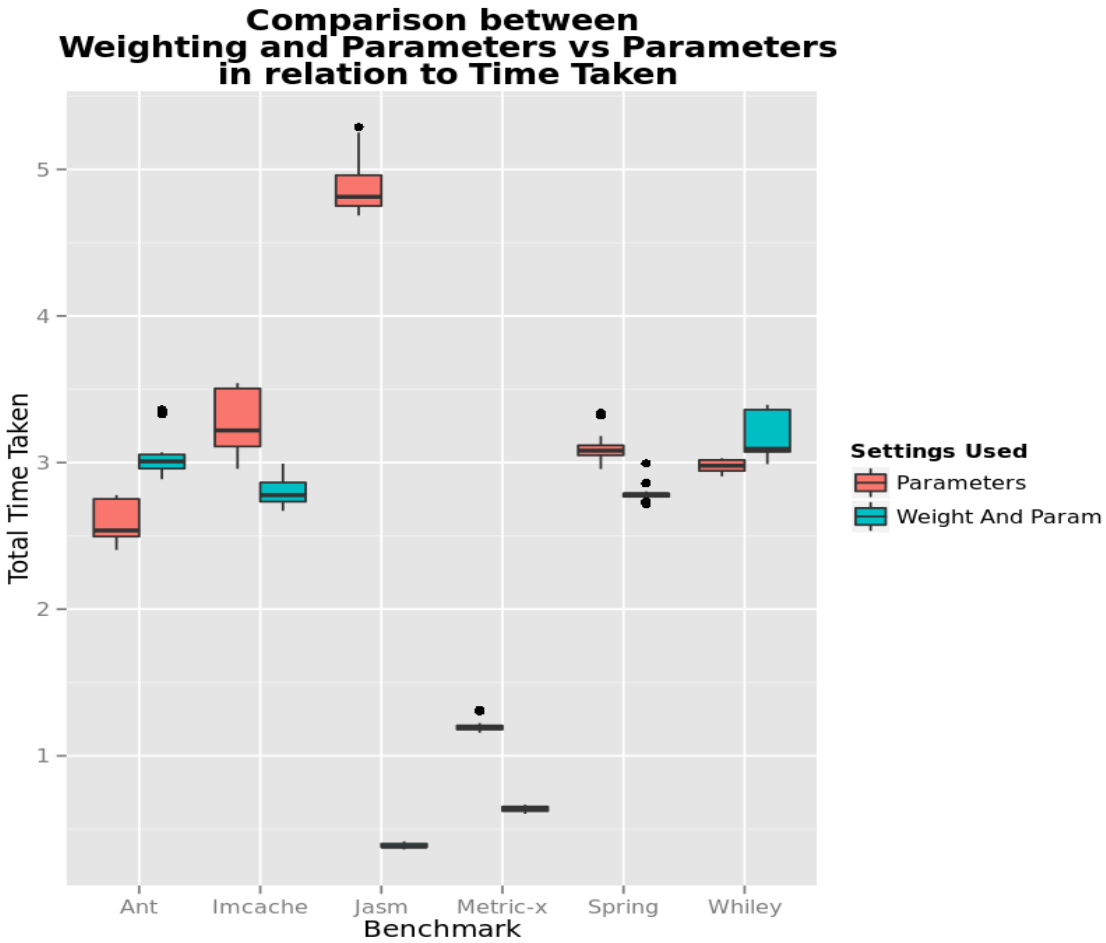

Figure D.1: A figure showing the relationship that using weighting and parameters combined vs parameters alone has on the total time taken to analyse the data.

Bibliography

[1] The anatomy of an aspectj. Online. `https://eclipse.org/aspectj/doc/released/progguide/language-anatomy.html` - Accessed at: 2015.07.01.

[2] Different types of weaving in aspectj. Online. `https://docs.oracle.com/javase/tutorial/reflect/` - Accessed at: 2015.10.01.

[3] How agile methods, continuous integration, and test-driven enhance design and development of complex systems. Online. `http://www.ibm.com/developerworks/rational/library/continuous-integration-agile-development/continuous-integration-agile-development-pdf.pdf` - Accessed at: 2015.10.08.

[4] Introduction to the spring framework. Online. `http://docs.spring.io/spring/docs/current/spring-framework-reference/html/overview.html` - Accessed at: 2015.10.08.

[5] *Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study* (September 2007), no. MSR-TR-2007-09, IEEE Computer Society.

[6] BECK, K. *Extreme Programming Explained: Embrace Change*. An Alan R. Apt Book Series. Addison-Wesley, 2000.

[7] BLACKBURN, S. M., MCKINLEY, K. S., GARNER, R., HOFFMANN, C., KHAN, A. M., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., ET AL. Wake up and smell the coffee: evaluation methodology for the 21st century. *Communications of the ACM 51*, 8 (2008), 83–89.

[8] COHEN, W., RAVIKUMAR, P., AND FIENBERG, S. A comparison of string metrics for matching names and records. In *Kdd workshop on data cleaning and object consolidation* (2003), vol. 3, pp. 73–78.

[9] GEORGES, A., BUYTAERT, D., AND EECKHOUT, L. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices 42*, 10 (2007), 57–76.

[10] GIBBONS, J., AND CHAKRABORTI, S. Nonparametric statistical inference. In *International Encyclopedia of Statistical Science*, M. Lovric, Ed. Springer Berlin Heidelberg, 2011, pp. 977–979.

[11] GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K. Gprof: A call graph execution profiler. In *ACM Sigplan Notices* (1982), vol. 17, ACM, pp. 120–126.

[12] HARROLD, M. J., GUPTA, R., AND SOFFA, M. L. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM) 2*, 3 (1993), 270–285.

[13] JEFFREY, D., AND GUPTA, N. Test suite reduction with selective redundancy. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on* (2005), IEEE, pp. 549–558.

[14] KOOCHAKZADEH, N., GAROUSI, V., AND MAURER, F. Test redundancy measurement based on coverage information: evaluations and lessons learned. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on* (2009), IEEE, pp. 220–229.

[15] LEVENSHTEIN, V. I. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady* (1966), vol. 10, pp. 707–710.

[16] LI, N., FRANCIS, P., AND ROBINSON, B. Static detection of redundant test cases: An initial study. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on* (2008), IEEE, pp. 303–304.

[17] MCDONALD, J. H. *Handbook of biological statistics*, vol. 2. Sparky House Publishing Baltimore, MD, 2009.

[18] MONGE, A., AND ELKAN, C. An efficient domain-independent algorithm for detecting approximately duplicate database records.

[19] NAVARRO, G. A guided tour to approximate string matching. *ACM computing surveys (CSUR) 33*, 1 (2001), 31–88.

[20] NELSON, S. F. Profiling initialisation behaviour in java.

[21] ORACLE. Oracle documentation. Online. `https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html` - Accessed 2015.07.06.

[22] PARSONS, D., SUSNJAK, T., AND LANGE, M. Influences on regression testing strategies in agile software development environments. *Software Quality Journal 22*, 4 (Dec. 2014), 717–739.

[23] ROTHERMEL, G., HARROLD, M. J., OSTRIN, J., AND HONG, C. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Software Maintenance, 1998. Proceedings., International Conference on* (1998), IEEE, pp. 34–43.

[24] ROTHERMEL, G., HARROLD, M. J., VON RONNE, J., AND HONG, C. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability 12*, 4 (2002), 219–249.

[25] STACK EXCHANGE. Issue tracking. Online. `http://programmers.stackexchange.com/questions/137611/add-a-unit-test-for-each-new-bug` - Accessed at: 2015.06.29.

[26] STACK EXCHANGE. When to write unit tests. Online. `http://programmers.stackexchange.com/questions/186268/when-to-write-unit-tests-is-there-any-or-list-of-terms-to-take-right\\-decision` - Accessed at: 2015.06.29.

[27] WILCOXON, F. Individual comparisons by ranking methods. *Biometrics bulletin* (1945), 80–83.

[28] WONG, W. E., HORGAN, J. R., LONDON, S., AND MATHUR, A. P. Effect of test set minimization on fault detection effectiveness. In *Software Engineering, 1995. ICSE 1995. 17th International Conference on* (1995), IEEE, pp. 41–41.

[29] WONG, W. E., HORGAN, J. R., MATHUR, A. P., AND PASQUINI, A. Test set size minimization and fault detection effectiveness: A case study in a space application. *Journal of Systems and Software 48*, 2 (1999), 79–89.

[30] YOURDON, E., AND CONSTANTINE, L. L. *Structured design: Fundamentals of a discipline of computer program and systems design*. Prentice-Hall, Inc., 1979.

[31] ZHANG, L., MARINOV, D., ZHANG, L., AND KHURSHID, S. An empirical study of junit test-suite reduction. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on* (2011), IEEE, pp. 170–179.

[32] ZHUANG, X., AND ET AL. Accurate, efficient, and adaptive calling context profiling, 2006.