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

Contents

Dave
Dave
Dave
Dave
Dave

Dave

Summary?

Dave

Figures

List of Tables

Chapter 1

Introduction

Test suites are an important part in every major software project [**?**]. They check a specified set of behaviours are met by a software program. Meeting the behaviours increases 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 [**?**, **?**]. With tests being added throughout, it is desirable to ensure that any new test case is not replicating the behaviour of any of the previous. Even with careful planning, it is difficult to avoid redundant test cases. The goals of the project are to create a tool to identify these test cases within a test suite. After the tools creation, its capability to identify redundant tests are explored through experiments.

The execution of a test case leaves a trail of data. This trail contains information ranging from low level to high level run time data. A low level example would be machine code while a high level could be the method execution data. A variety of previous research [**?**, **?**, **?**, **?**, **?**, **?**, **?**] 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 1,000 test cases. For benchmarks with a large number of test cases storing every statement execution could be costly, therefore our project investigates using method execution data to identify redundant test cases. One of the issues that previous studies reported was a high level of false positives. Method execution data gives different ways to explore this issue which are explored further on.

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. A basic visualisation of the idea is shown in Figure 1.1, the spectra are three different tests where colours represent method executions. Intuitively, it is clear that Test 1 is different from Tests 2 and 3 as the method executions is different between them however, there are similarities between Tests 2 and 3 which could identify some level of redundancy between the test cases.

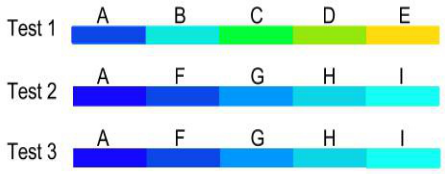

Figure 1.1: A spectra where each colour represents different method execution details. We see similarities between Tests 2 and 3. In contrast, Test 1 is different from 2 and 3.

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, dependent on the project. Therefore 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.

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. 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, he is currently writing a language called Whiley. The language contains an extended static checking tool to eliminate run time exceptions through formal verification techniques. In the main compiler module alone, there are roughly 20,000 tests. Relocating some these tests into another suite would result in allowing him to increase development speed due to a reduction in the time taken to run a large test suite. Increasing the frequency of test suite execution also allows for bugs to be traced back to code changes easier and reduces the time spent debugging. Therefore, by redistributing the test suite, not only does the time taken to run a test suite decrease, there is incentive for developers to execute the suite more often and in turn helping them reduce time taken to debug.

The contributions of the report is split into two parts:

- Create a tool for identifying redundant 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. ~~It also examines previous research in this area of interest.~~ The design and implementation of the tool is then examined in Chapter 3. Chapter 4 investigates the different techniques and conducts then discusses some experiments. Finally, Chapter 5 concludes the report and identifies future work.

Chapter 2

Background

This chapter ~~firstly~~ explores the current research conducted in this area~~, it~~ . It then covers the key ideas ~~that are~~ 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 Redudant Tests

Testing is a critical part to any software engineering process, not only to stop incidents stemming from the ~~the~~ product, but also partially related to the increase in popularity of agile methodologies [**?**]. Many of these methodologies ~~employ~~ use test driven development and continuous integration~~resulting~~ . This results in testing becoming more important throughout the development process. For these reasons there have been research papers that examine the different approaches ~~that can be~~ taken to identify redundant test cases and reduce the size of test suites [**?**, **?**, **?**, **?**, **?**, **?**, **?**].

Whether programmatically reducing a test suite's size is worth the trade off in the ability to locate bugs is unclear. Wong et al. [**?**, **?**] 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 test suite reduction did not severely impact fault detection capability. In contrast to this finding, Rothermel et al. [**?**, **?**] further expanded on 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 ~~uncertainty created by these conflicting studies~~ conflicting studies create uncertainty. This motivates our research to decouple the removal of tests from the framework 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 [**?**] 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 ~~that are executed by the~~ 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 ~~respect to~~ every other test. It was calculated by measuring the percentage of a test case that is also a subset of the other test case. The second metric examined each test case with ~~respect to every~~ the test suite as a whole. It was calculated by measuring the percentage of a test case that is covered by the test suite 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. However, 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.

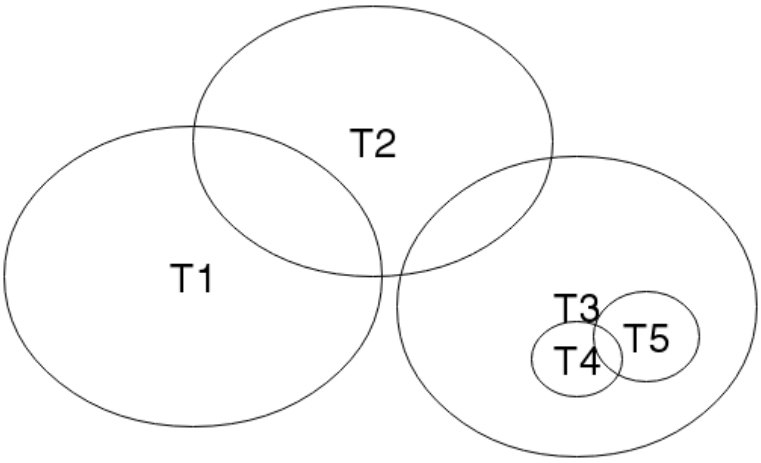

Figure 2.1: The coverage of each test is shown by a circle. It shows that T4 and T5 are redundant as T3 already covers those statements.

Zhang, Marinov, Zhang and Khurshid [**?**] 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 and would continue until all the test requirements had been satisfied. So 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 [**?**] where essential test cases are selected 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, showing 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 [**?**] examine 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, Manhattan distance, unigram cosine similarity and bigram cosine similarity. They each were measuring

how closely related two tests were based on the sequence of commands and procedures loaded. Their findings were similar to Maurer et al. [**?**] 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 and 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 [**?**].

Taking into 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 [**?**]. 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 and 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 and et al. [**?**] describe a call tree as the overall tree of every method execution. 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 represent the frequency. Finally, the bottom left (call tree) creates a new path for every new method execution regardless of the uniqueness, giving us the most information about the context if the order is retained.

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 [**?**]. 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 case spectras. There are a variety of different implementations of edit distance metrics. Cohen, Ravikumar and Fienberg [**?**] explore different edit distance metrics for name matching. They concluded that Monge and Elkan [**?**] 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. One of the other techniques they explored was the Levenshtein distance [**?**] metric by itself. This metric is the minimal number of operations that can be

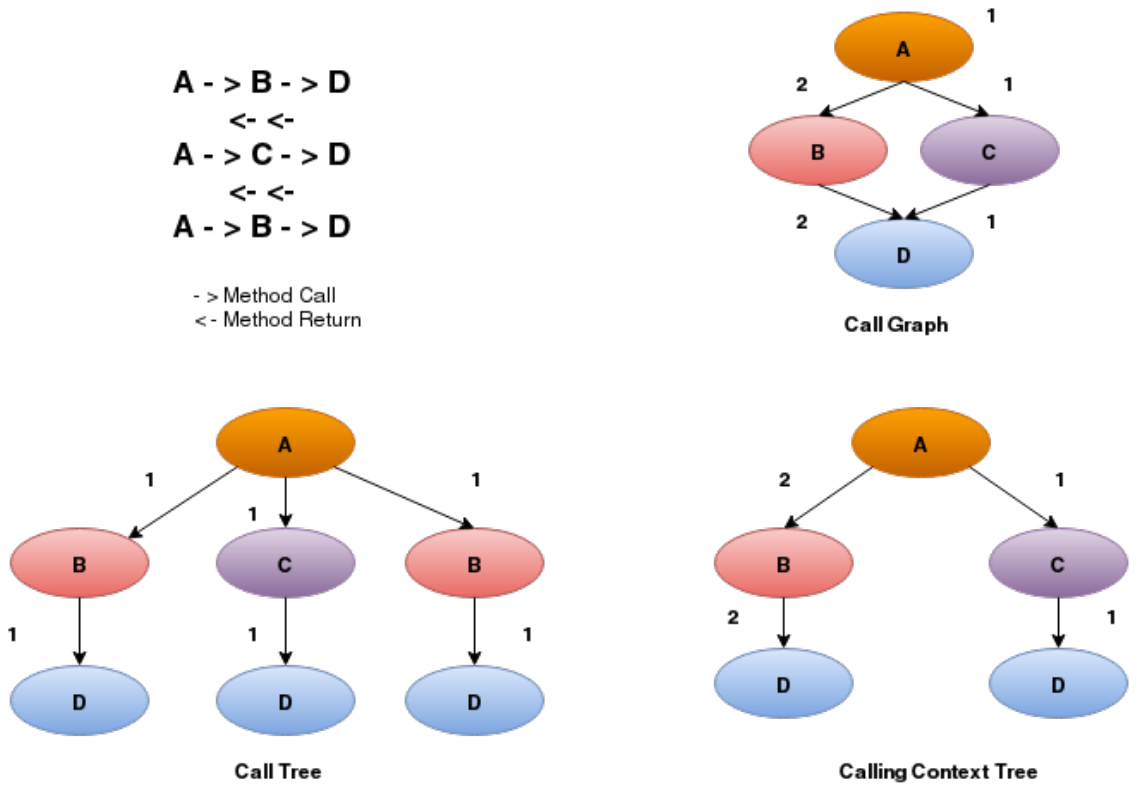

Figure 2.2: A diagram showing the three different variations of Call Graph information. Where each have a varying level of information that is stored.

done to make one test's spectrum equal to 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 spectra. 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.

An example of Levenshtein is shown in Table 2.1, where 'kitchen' is being changed into 'kitten'. The example 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.

An example of Monge & Elkan is shown in Table 2.2. The comparison of the strings 'paul johnson' and 'johson paule' are being compared. 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 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

A Wilcoxon Signed Rank test is used to infer whether there are any significant differences between the techniques implemented in this report. The Wilcoxon Signed Rank test was first proposed by Frank Wilcoxon in 1945 [**?**]. 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 [**?**]. The test does not assume that the data follows a normal distribution which is ideal for the data presented in our research and is used when two nominal variables and one measure variable is being measured [**?**].

2.2 Performance Evaluation

Throughout the report different techniques are designed and implemented to identify redundant test cases. To evaluate the techniques, experiments are conducted and must adhere to a standard procedure. 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 will be examined in reference to research papers that explore the issues.

Andy Georges et al. [**?**] and Steve Blackburn et al. [**?**] 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 the experiment cannot be reproduced. 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 representation of the results. A set of parameters that are of interest are explored in the paper, these 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 that the analysis 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.

- **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.

- **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 the 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 around the School of Engineering and Computer Science. 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, if a user logs on while a process is being conducted, the application is paused until the machine returns to an idle state.

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 ~~can be~~ is split into several different sections and ~~each section will be discussed in this chapter~~ the chapter will discuss each. The first objective of the tool was to trace the test data. We explored two different frameworks and compared the usability of each. After the tool was able to trace the data, we had to determine what spectras we were interested in. The three different spectra types are then discussed. The data traced from each test case was then compared to that from every other test to determine the level of redundancy. The comparison was done with edit distance metrics. Throughout the testing of the tool, one of the main impediments was the time taken to analyse the large amount of data. We examine how breaking the analysis into a series of pipeline stages decreases the time taken.

3.2 Tracing

A fundamental requirement for this project was to trace ~~which methods were~~ 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 ~~called~~ Aspect Oriented Programming (AOP). AOP can be used to add code to an existing program without modifying the source code~~, this code added is known as an *aspect*~~. AspectJ uses an aspect to identify where and what is added. This process can be achieved through the following methods:

- **Compile time** – The classes are compiled with the aspect woven into them. When the program is executed, the methods have the code from the aspect woven into it already. 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 modification 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). JDI is similar to using an observer pattern. ~~A~~ It selects a list of classes to observe~~are selected~~. When a method within a selected class is called, the listener class is notified and is given 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 we explore tracing 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~~and resulted in the extra information being more important~~. This meant that the tracing performance was less of an issue.

To distinguish what code to weave, and where in the existing code to weave it, AspectJ uses a *point cut* [**?**]. ~~A point cut was made to record~~ Our point cut recorded every execution. ~~A simplified version of the aspect is shown in Figure 3.1~~ Figure 3.1 shows a simplified version. There are two point cuts within the aspect~~, the~~ . The first point cut ~~is going to be~~ 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 point cut ~~is used to trace~~ traces everything apart from the methods with a Test annotation attached~~and passed~~ . It then passes the information to the static service. The next stage was to weave the aspect into the benchmark. Using compile time weaving would have meant that each benchmark ~~needed~~ 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.

```
pointcut testMethod(): execution(@org.junit.Test * *(..));
before(): testMethod(){
    StaticService.addMethodCall("New Test:" + thisJoinPointStaticPart
            .getSignature().getDeclaringTypeName());
}

pointcut traceMethods() : ( !within(@org.junit.Test *) && execution(* *(..))&&);
before(): traceMethods(){
    StaticService.addMethodCall(New Test:" + thisJoinPointStaticPart
            .getSignature().getDeclaringTypeName();
}
```

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

Dave

3.3 Filtering By Spectra

The idea of a 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~~ . 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 applied to this trace information to filter out data to match the spectras characteristic. The tool ensures that the traced information is able to be filtered by the spectra.

We will examine the spectra characteristics in relation to the example shown in Figure 3.2. The different spectra's take up different levels of resources, these are are – time taken and memory.

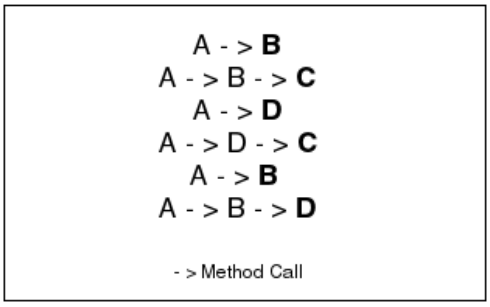

Figure 3.2: A simple representation of a call tree of three that is traced from a test. Each letter represents a method call and the bold letter is the active method in each line of the call tree.

3.3.1 Unique Method Calls

The ”unique method call” spectra filters out the repetition of methods. The data from applying the filter only has each method represented a single time. The motivation of this spectra 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 3.2, the method calls are ”B,C,D,C,B,D”. When we use a unique method call spectra to filter this trace information, the data will become ”B,C,D”. This shows that we are filtering out a large portion of data while retaining an adequate representation of the original trace information. We can explore how this effects the Whiley Compiler benchmark. There are 494 unique methods called for a particular test and 80,000 method executions. Utilising the unique method call spectra, the tool will only use 494 method executions when comparing the test with another, rather than the 80,~~000,~~ 000. This is $^{6}/_{1000}$ of the original amount.

3.3.2 All Method Calls

The ”all method calls” spectra filters out information regarding the call tree. The motivation behind this approach is to increase our confidence ~~with~~ in the results, but this increases the time taken and memory used to analyse the information.

Examining Figure 3.2, the output of filtering with this spectra would be ”B,C,D,C,B,D”. This shows a loss of the context data. We can explore how this effects the Whiley Compiler benchmark. When using an ”all method calls” spectra, 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 [**?**]. The motivation behind the approach is to increase our confidence in the results by taking into account the *context* of a call. The consequence of increasing the amount of data is ~~an~~ a further increase in the time taken and memory used to analyse the trace data ~~but a further increase in the confidence in~~ in comparison to the other two ~~spectras~~spectrums.

We can filter the Figure 3.2 using a K depth of two. This would lead to the filtered data returning ’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 calling context spectra would examine all 80,000 method executions, ~~with~~ each containing K depth of method calls.

To retrieve the calling context in Java, the current stack trace of a method call is examined~~and~~. 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.

Dave

3.4 Analysis Metrics

The data traced needs to be analysed to produce a quantitative value. The value produced will be used to determine the level of similarity between two test cases. Edit distance metrics were discussed in Section 2.1.3, with two different edit distance metrics considered. They were: Monge & Elkan [**?**] which splits the strings into sections and compares the sections and Levenshtein [**?**] 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 ~~test cases trace information was repreesnted~~ 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 3.3 can be used to show the difference between 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. This direct matching was preferred as we were interested in the order of the method calls as well as what methods were called.

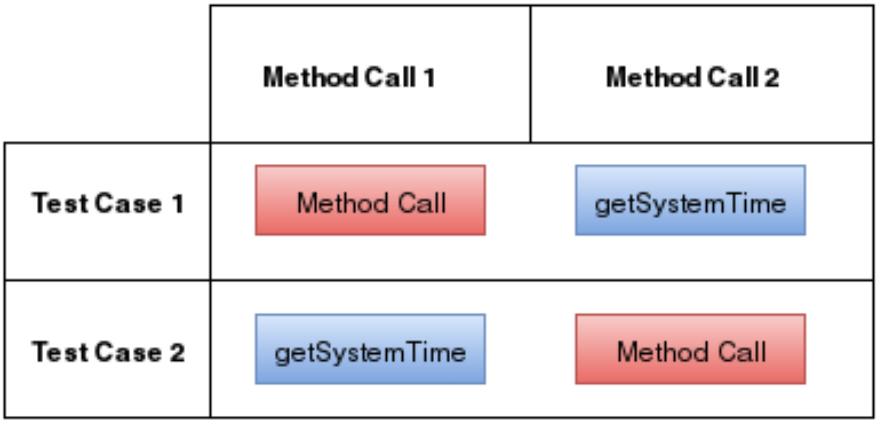

Figure 3.3: Trace information used to show the difference between Monge & Elkan and Levenshtein metric implementations. The Monge & Elkan metric will disregard ordering and measure these test cases as being the same. The Levenshtein metric examines one to one and will measure the test cases as having no similarities.

Dave

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 ~~of~~ the available information (call tree) in some cases resulted in the analysis taking several days to complete. This is not considered an acceptable time frame. This issue arises when each of the spectrum of the test cases contain tens of thousands of method calls. ~~To get around the~~ A pipeline approach was implemented to get around this issue of lengthy analysis~~, a pipeline approach was implemented. The idea is visualized in Figure 3.4~~. Figure 3.4 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 ~~within~~ larger the redundancy limit specified, it will ~~then be kept~~ be stored and analysed during the next stage. The goal is that each subsequent stage should soundly eliminate pairs from consideration that the following stages would have removed. ~~This goal can be achieved by using heuristics~~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 3.4: 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. ~~By utilising different spectra filters, this allows for~~ The spectra filter characteristics examine the trace information ~~to be examined with different characteristics~~in a different perspective. The particular filter of interest is the ”unique method calls” spectra. 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 only. We see a pipeline set up in Figure 3.5. The first stage uses a ”unique method calls” spectra and pipelines the output into a ”call tree” of depth thee. The three test cases at the top are the trace information input into the pipeline. The remainder of this section explores this example.

Pipeline Stage 1 The tool first filters the trace information with a ”unique method call” spectra. 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 examined are test cases 1 and 3. Looking at the test cases, when filtered with a call tree of depth three there are no similarities between the two method calls. The output of the pipeline as a whole is there are no redundant test cases in the trace information.

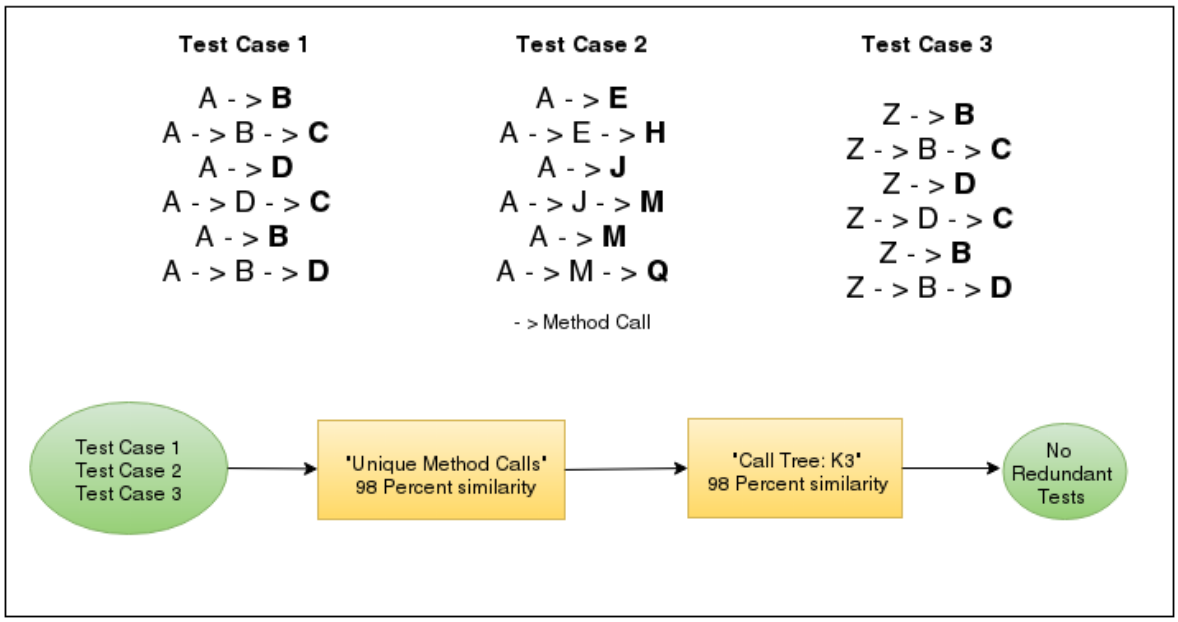

Figure 3.5: A two stage pipeline that analyses the trace information shown. The first stage uses a ”unique method calls” filter with a 98% similarity and the second stage filters with a call tree of K Depth three with a 98% similarity. After analysing the data, no test cases are returned as being redundant.

~~This example contains a pipeline of length two. When using a pipeline of length three,~~ An alternative heuristic to use is the ”all method calls” spectra~~may be used as the second stage heuristic. The ”all method calls” spectra would not be appropriate in the last pipeline as there is limited sense in using a subset of data.~~

~~The list spectra may also be used as another type of heuristic. The~~ . The motivation behind the spectra is to be used as the second stage in a three stage pipeline. In comparison to the ”unique method calls” filter, the ”all method calls” ~~increases~~ decreases the amount of comparisons that the final pipeline ~~should and with more confidence of the results however,~~ ~~increases~~ does. It increases the confidence in the results but also, increases the time taken. This approach would not be used in the last pipeline as there is limited sense to use a subset of information retrieved from the test cases.

3.5.1 Saving to the Network

To execute the analysis on a grid computing system, the test data had to accessible on the School of Engineering’s local network. To achieve this, the trace data was saved to the local network. Not only did it allow for the analysis to be executed on the grid system, it also it allowed for the data to be reanalysed without having to re ~~execute~~ trace the test suite.

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 due to knowing the statement path, parameters are irrelevant as you know the path the method will take, and parameters do not add any more information. When tracing at the method level rather than at the statement level, parameters become crucial to determine the degree of redundancy between test cases. This is because parameters give insight into the execution paths that will be taken and act as a proxy to the statement information without storing the same amount of the data.

There were two variations of parameters that were considered to trace. Firstly, primitive types only. ~~By~~ Primitives only represented a small number of parameters when running test experiments on the benchmarks~~, the use of primitives showed that only a limited number of parameters were collected~~. The parameters collected had a limited effect on the number of false positives identified and time taken. The second approach ~~is~~ was to use reflection. Reflection is the ability to examine and modify objects at run time [**?**]. Since AspectJ gives direct access to the object's within the method parameters, reflection can be used. By using reflection, we retrieve the the fields of the objects ~~are retrieved and returned~~ and return 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 Apaches commons language library.

The most common use case of the tool is expected to use parameters therefore it was decided to optimize this through storing the data with parameters. The optimization involved saving the parameters with the trace information. If parameters value is set to false, the parameters have to be split off rather than being concatenated on. This means that setting the parameters to false would increase the set up time.

3.7 Weighting

Dave

Maurer et al. [**?**] and Robinson et al. [**?**] 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 3.6. The figure shows test cases with different proportions of setup and tear down in relation in the exercise stage. The figure also shows how the size of the test case may lead to different proportions.

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 ~~removed completely~~ completely removed. The other variation that was considered, and used, was completely removing the most used method calls for each test case. This involved determining the top 20% method calls~~, and removing every method execution~~. The method executions to these calls were then removed at the start of the pipeline stage. During initial experiments, the first variation ~~was found to have~~ 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, therefore it was difficult to find a solution that could be applied on every benchmark. This technique is referred to as weighting throughout the remainder of the report.

Deciding the scope of the weighting techniques calculation was important. The two options considered were per test case or test suite. The issue discovered with calculating per test suite was when the benchmark contained a majority of large test cases. The test cases had their setup and teardown methods removed~~, but at~~. At the same time these setup and ~~tear down~~ teardown methods were often the same and did not always take up 20% of the method calls. This meant that the weighting technique per test suite was removing exercise method calls. This was occurring in the per test case variation as well, but to a lesser extent. The removal of some exercise method calls had some impact on representing the test cases correctly but overall was negligible.

Figure 3.6: 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

Chapter 4

Results and Discussion

The following chapter reports on the outcome of several experiments that explore the use of our tool on a realistic benchmark suite. The key 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 (tests identified & types identified).

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

4.1 Experimental Method

The experimental method is key to producing believable and reproducible results. Before the experiments could be conducted, 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 several experiments. The experiments consisted of using the tool with the different techniques discussed in Chapter 3, each being executed 30 times per benchmark. The experiments examined: Pipeline length, K depth, Parameters and Weighting. The issue of running a large number of tests was solved by using a grid computing system. The performance time was the time taken to analyse the individual stages as well as the total time. After the results had been gathered, a Wilcoxon signed rank test [**?**] was used to determine whether the results were significantly different or not. The significance level used was 95% with a null hypothesis of the two samples median being equal. Overall, seven different settings were run per benchmark with each setting being run 30 times. For every settings experimented on, they all consisted of a first pipeline stage using a "unique method calls" spectra filter.

The goals of this paper as mentioned in Section 1 are the creation of a tool to identify redundant test cases and experimenting with different techniques on realistic benchmarks.

4.2 Environmental Methodologies

As previously discussed in Section 2.2 there are a variety of challenges that present themselves when using Java to evaluate performance. These challenges are discussed throughout the following section.

4.2.1 Grid Computing

A grid computing system was used to execute the data analysis, with a total of 150 jobs queued on the grid at a single time. When using a grid system it is important to ensure that all the machines used are the same for every experiment. The grid allowed for particular types of machines to be specified – 8GB DDR3 RAM, Linux 4.0.5 64 bit system and an Intel Core i7-3770 CPU running @ 3.40GHz.

Dave

4.2.2 Measuring Time

The time taken is an important variable in evaluating the performance of a tool. The notation of CPU time was used 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 tool may be paused and the RAM used by the tool could potentially be moved into virtual memory. This is dependent on the amount of resources that the user needs. When the user logs out, the tool will restart and any information lost will need to be recalculated by the tool. The issues were limited by running the tests overnight when users were unlikely to log in.

When measuring the time taken, we had to decide what should be measured. Ideally, the start up cost of the tool would be measured 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 being run 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. This would produce unreliable results and the start up cost was not measured as a result.

Heap Size

The heap is the location that the JVM uses to store objects that are produced during the execution of an application. The amount of heap that was allocated to the JVM was 6GB for every benchmark on every run.

4.2.3 Software Environment

A concurrent-mark-sweep garbage collection strategy (default) was used. This approach uses multiple threads to scan the heap, mark unused objects and recycle them [**?**]. 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 lose credibility. They were located by looking at popular Java frameworks, Github repositories and David Pearce's personal projects. For a benchmark to be considered, 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 in order to fully evaluate the tool. Information on the benchmarks are shown in Tables 4.1 , 4.2, 4.3 and 4.4. The information shown is a description of each benchmark, the authors, types of tests, number

of tests traced, lines of code and version used. An important thing to note is that the number of tests is only the amount that were traced. A range of test types were chosen, David J Pearce's projects contain end to end tests which run through a whole module at once, while the rest attempt to test units of code at a time. Having a range of test types and sizes gives a wider evaluation view and insight into the potential situations where different settings may not be helpful in achieving the aim.

Some of the benchmarks produced up to 100,000 method calls per test, each with parameter information. This required a large amount of memory to store the details when the tests were being traced. Whiley and Ant were the benchmarks that did not have all of 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" [**?**] .	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	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	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.

Dave

4.4 Experiment I - Pipeline Length Comparison

4.4.1 Motivation

The pipeline length would be expected to impact each factor of interest. Selecting a pipeline too small or too large will have varying negative effects on these. This experiment explores this trade off and the length of the pipelines that should be used with the benchmarks.

Hypothesis 1 *Increasing the length of the pipeline improves the factors of interest*

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 and performance. This is because each stage will perform a decreased number of comparisons than the previous. This is reflected in hypothesis 1.

4.4.2 Settings

There were two different pipeline lengths tested, two and three respectively. For both length's, the final pipeline stage was the same. It is noted that a single pipeline comparison was left out of the time taken graph due to the amount of memory and time taken to finish analysing, however we could infer the number of comparisons that it would have been performed with a single pipeline.

4.4.3 Results

The table for comparing the pipeline of length two and three is shown in Table 4.5. It shows that there was 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 length 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 as they all produced the exact same number of redundant tests in both pipeline lengths.

A chart showing how each benchmark reacted to the change in the pipeline length is shown in Figure 4.1. It shows the number of test case comparisons that the final stage of the pipeline had to conduct.

	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 an significant increase and a '-' represents an significant decrease.

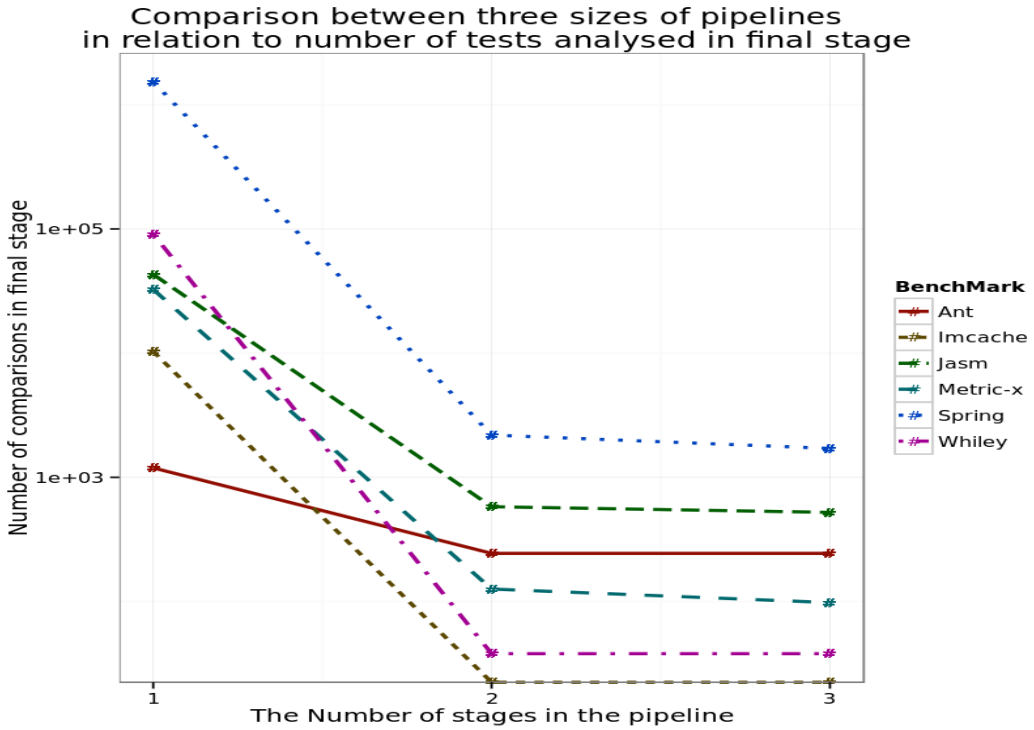

Figure 4.1: A figure showing the effect that the different pipeline lengths have on the number of comparisons during the final stage (Most computationally heavy).

4.4.4 Discussion

Inspecting Figure 4.1, it becomes clear that the pipeline length does improve the factors of interest, but the length does not scale in every benchmark. This is shown by every two or three stage pipeline being a significant improvement on a single stage, however the majority of the benchmark's perform better with a two stage over a three. These results imply that the benchmarks that took less time using a two stage, were spending more time on the second stage than they were saving from the reduced number of comparisons in the third stage. There are two scenarios proposed that would cause more time to be spent on the second stage than was saved in the third stage. Firstly, the tool was looking for a different level of similarity between benchmarks, this was dependent on how each benchmark reacted to different similarity levels. The benchmarks with unit tests needed to have a higher level of similarity compared to benchmarks with end to end tests. This implies that each benchmark also has its own optimal settings for the second pipeline. The settings were chosen by experimenting with different options, therefore an inefficient second pipeline would have contributed to the result. The second reason is that there may be little difference between the outcome from the second and third stages. An example is shown in Figure 4.2. Examining the first stage in the pipelines, it shows a large reduction in the number of comparisons. 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. This leads us to accept the first hypothesis, that increasing the pipeline length improves the factors of interest, however the optimal length is different for each 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 while using the same final stage settings, this could imply two things. Firstly, the second pipeline stage is more specific than the third. 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 identifying a higher similarity than the final stage. This would cause a different output between the two lengths. Secondly, there is a bug in the code.

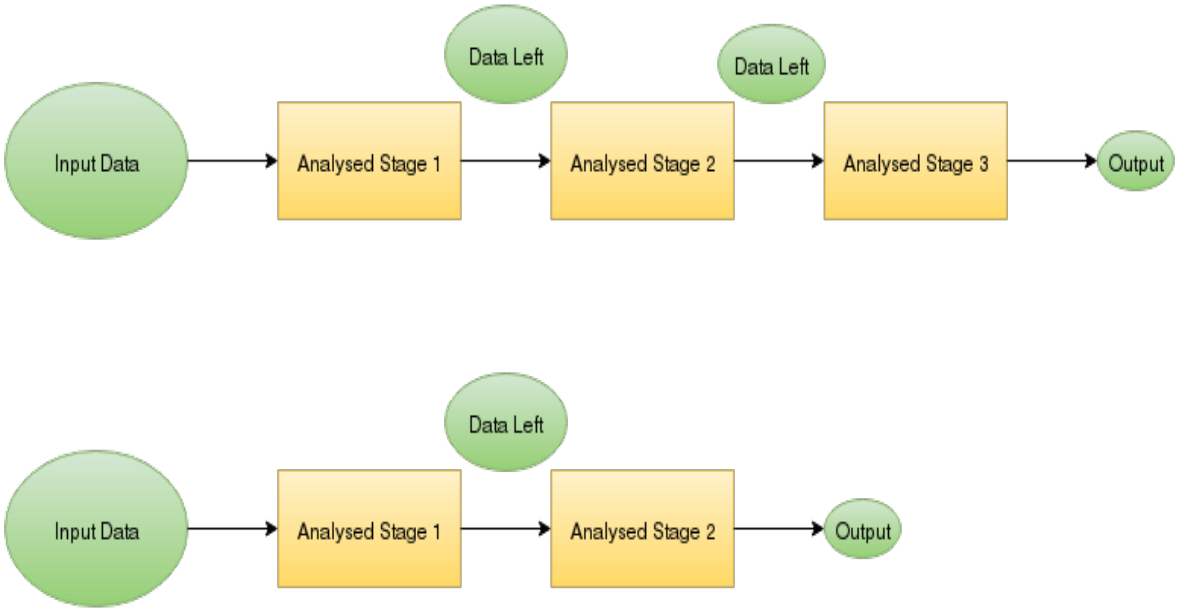

Figure 4.2: The circle size represents the data that is left in the pipeline. The figure shows that the difference that is between a two stage pipeline and a three stage only has limited room for a reduction in test cases identified. This shows the time taken to execute the extra stage costs more than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

Hypothesis 2 *The precision improvement from increasing K Depth outweighs the performance decrease.*

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. This should be reflected through a reduction in the number of tests identified. The extra data used causes extra analysis, this implies an increase in the time taken as K depth increases. It is expected that the precision increase is worth the performance decrease. This is reflected in hypothesis 2.

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. There were significant tests considered to perform the test for three samples, however we felt the graphs were enough to convey the results.

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 albeit by a limited amount. Figure 4.4 shows the differences in time taken. Overall, for every benchmark there wasn't a large amount of change however it shows that as the K depth increases, the time taken increases.

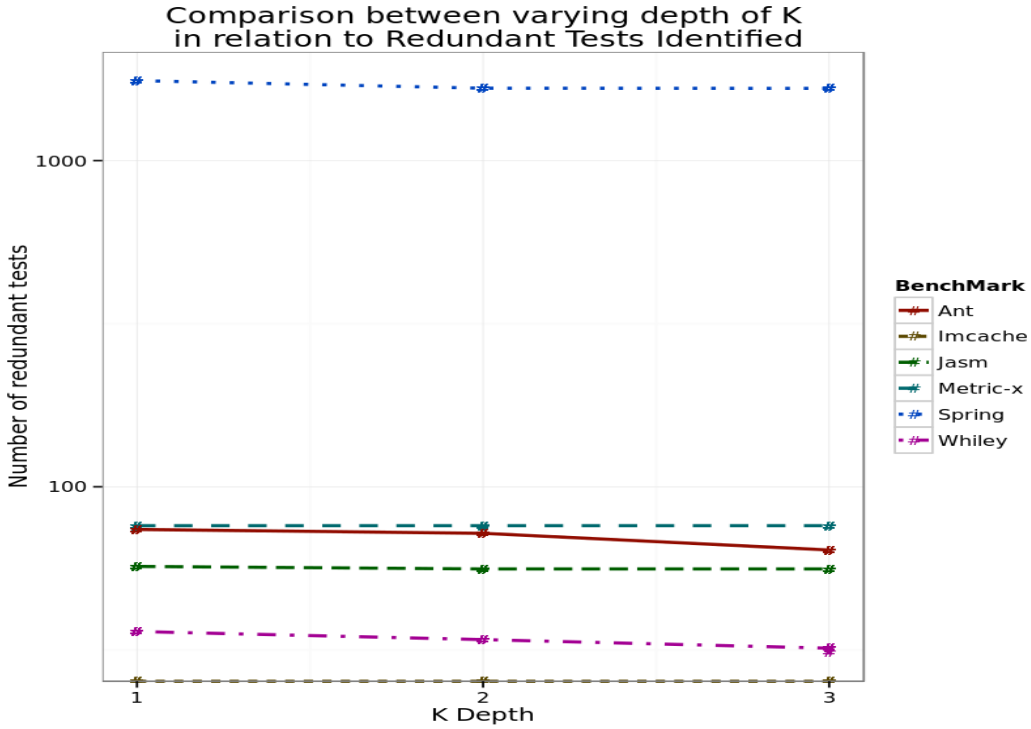

Figure 4.3: A figure showing the effect that a change in the depth of the call tree has on the number of redundant tests are identified.

4.5.4 Discussion

The result shown in Figure 4.3 shows several things. Firstly, 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. Secondly, 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 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

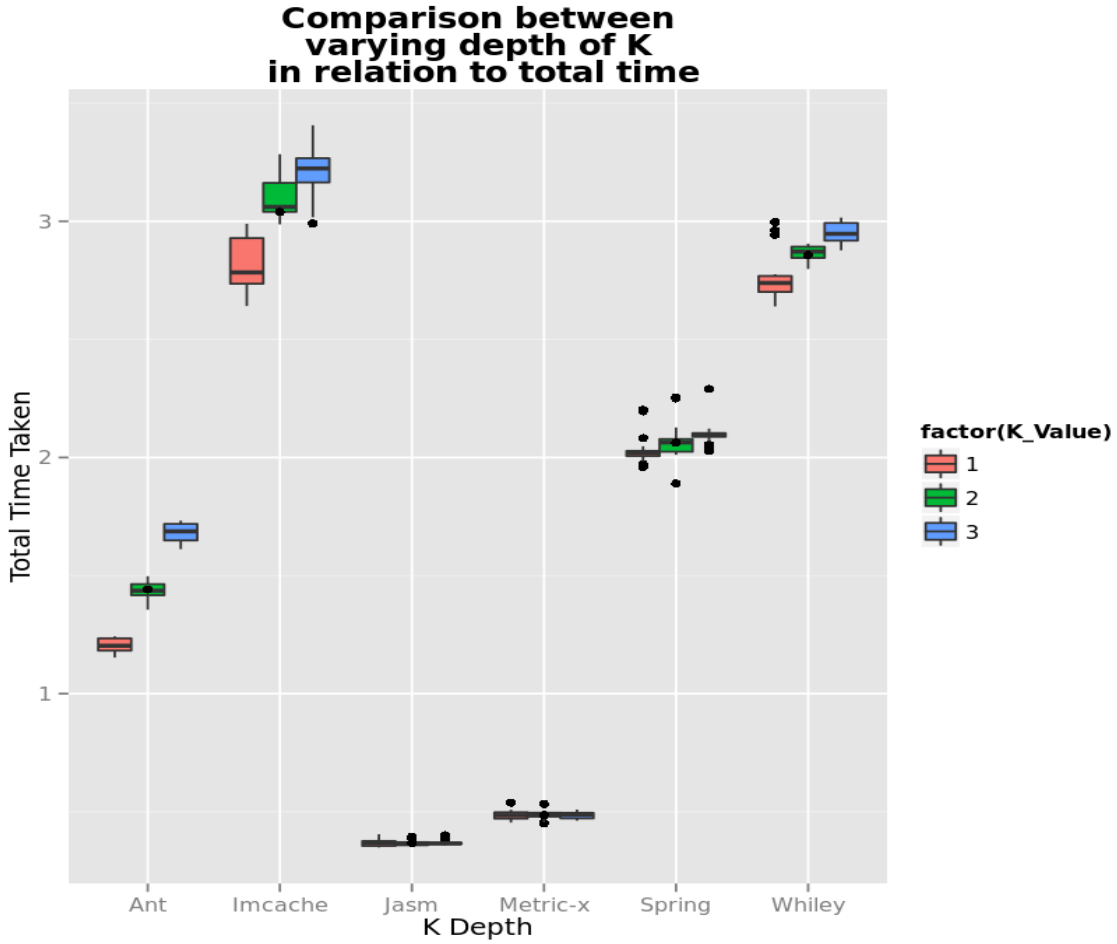

Figure 4.4: A figure showing the relationship that using a different K Depth has on the total time taken to analyse the data.

non-redundant tests. To confirm this, Figure 4.5 can be examined. Inspecting the figure, it shows that the heuristic pipeline removes a large portion of the comparisons. It is not enough to confirm the heuristic can reduce comparisons needed, but to inspect the number of tests 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 heuristic has a good accuracy of identifying redundant tests.

There is only a limited change in the number of redundant tests identified, however we need to also 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. The differences are shown in Figure 4.4. It shows that Ant, Imcache and Whiley have larger differences than the rest, however in the perspective of time these equate to around half a minute difference. Although increasing the depth has an smaller effect than expected on precision, the impact on time taken isn't substantial. Taking this into account, I can accept the hypothesis with the results implying that for some projects the calling context does not have to be particularly deep.

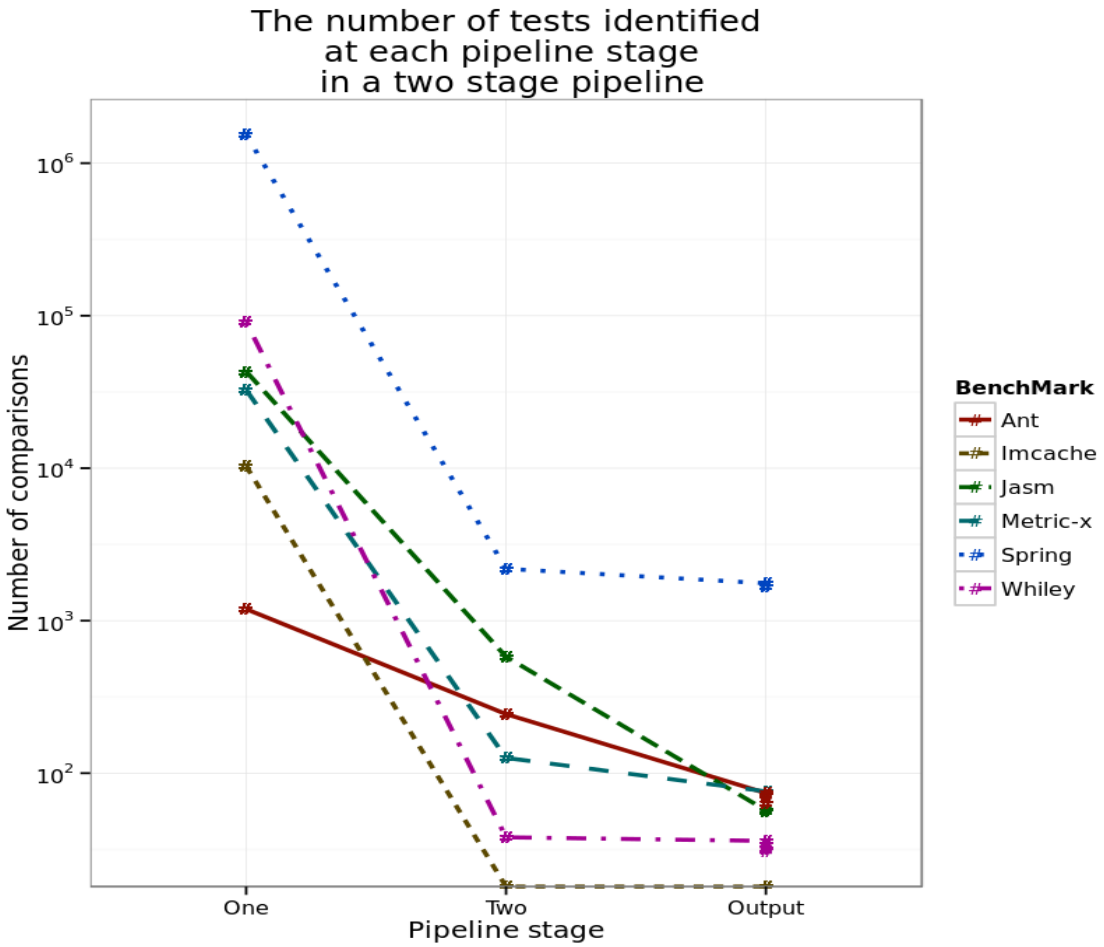

Figure 4.5: A figure showing the the number of comparisons that each stage performs. The pipeline contains two stages and is from Experiment II. The graph how the first stage heuristic is able to reduce a large number of the comparisons needed to do for the subsequent stage.

4.6 Experiment III - Parameter Comparison

4.6.1 Motivation

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

Hypothesis 3 *Utilising parameters increases the time taken in comparison to a two stage pipeline.*

Hypothesis 4 *Utilising parameters increases the precision in comparison to a two stage pipeline.*

Two factors have to be taken into account when discussing the impact of parameters on the time taken. Firstly, it is intuitive that analysing the extra data generated increases the time. The amount of increase is interesting, parameters only add extra computation when the method execution of the given K 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 parameter won't be taken into account and therefore 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 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 be increased. This is reflected in the hypothesises that parameters increase the precision and the total time taken in comparison to the pipeline of length two in Experiment I

4.6.2 Settings

The use of parameters is compared directly to the pipeline of length two used in Experiment I.

4.6.3 Results

The information for comparing parameters and no parameters is shown in Table 4.6. 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 for 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, therefore no difference between the two. Examining Appendix B.1 – Ant, Imcache, Jasm and Spring appear to 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 significant Table 4.6 matches what is expected. For every benchmark, using parameters significantly increases the time taken. Examining Appendix B.1 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 accumulate effect is related to the setup methods. If the set up methods are a large majority of 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.

Reexamining Table 4.6, it shows that every benchmark had a significant decrease in the number of redundant test cases. Looking at Figure 4.6, it confirms that every benchmark reacted with a substantial decrease in redundant tests identified, with Whiley being the least affected. This reiterates the discussion in Section 4.5 that K depth is not enough of

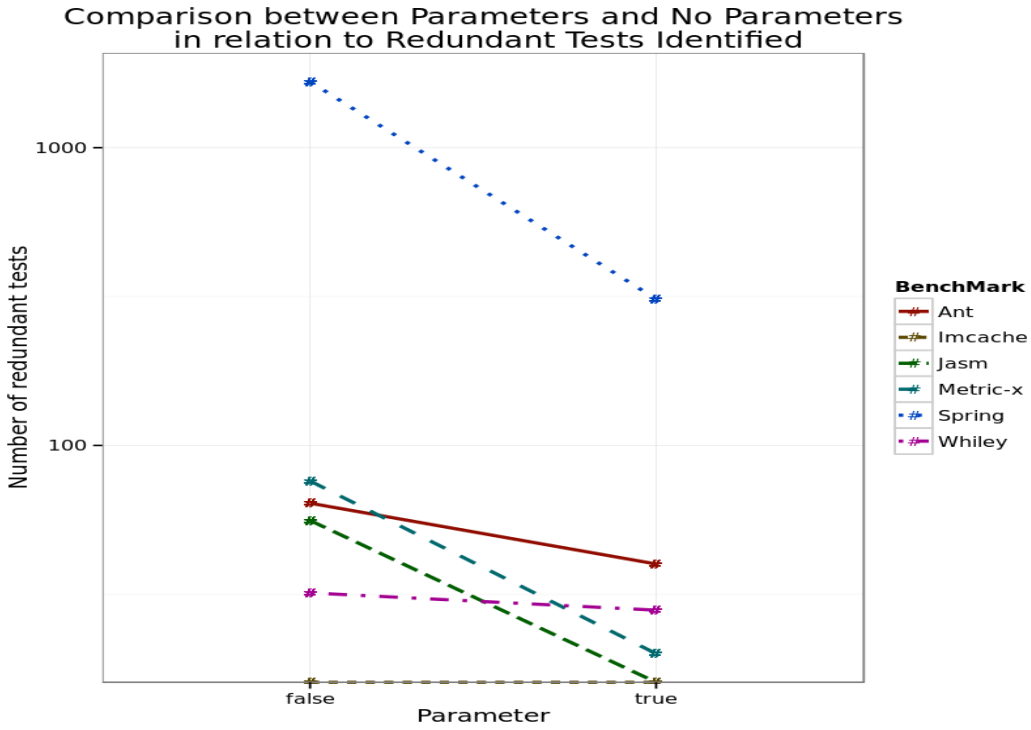

Figure 4.6: A figure showing the effect that using parameters has on the number of redundant tests are identified.

a factor to identify redundant test cases. It is expected that the number of false positives decrease. 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 different array values. The Metric-x coding demonstrate a reduction in the number of the different parameter value tests identified substantially, from 52 to 8. Taking these coding tables into account, it allows us to accept hypothesis 4 that parameters increase the precision.

4.7 Experiment IV - Weighting Comparison

4.7.1 Motivation

Utilising weighting 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, while exploring the impact on performance.

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*

As previously discussed in Section 2, much of the related work encountered difficulties with test cases sharing setup and teardown method calls, resulting 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 the setup and teardown methods represent a large majority

of the test data. The approach of removing a portion of the most executed method calls meant that the amount of data per test case decreases. This should reflect onto the results by showing a decrease in the time taken, as per hypothesis 7. The effect that it has on the number of redundant test cases should be dependent on the benchmark when weighting is used. This is because removing the most common tests should imply a decreased false-positive rate, but also may increase the number of actual redundant tests picked up. The precision is of interest when examining the cause effect of weighting, this is reflected by hypothesis 6.

4.7.2 Settings

The use of weighting is compared directly to the pipeline of length two settings in Experiment I.

4.7.3 Results

There are a mixture of results in regard to 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 increase 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

4.7.4 Discussion

A reduction in the false positive rate was weighting'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 that were identified in [**?**] [**?**]. To confirm this, examining Table 4.8 and 4.9 will give 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 is a similar result. The weighting technique reduces the number of parameter value redundancies identified as well as the limited redundancies, however does not completely remove them. This is interesting and implies that the number of setup and tear down methods that

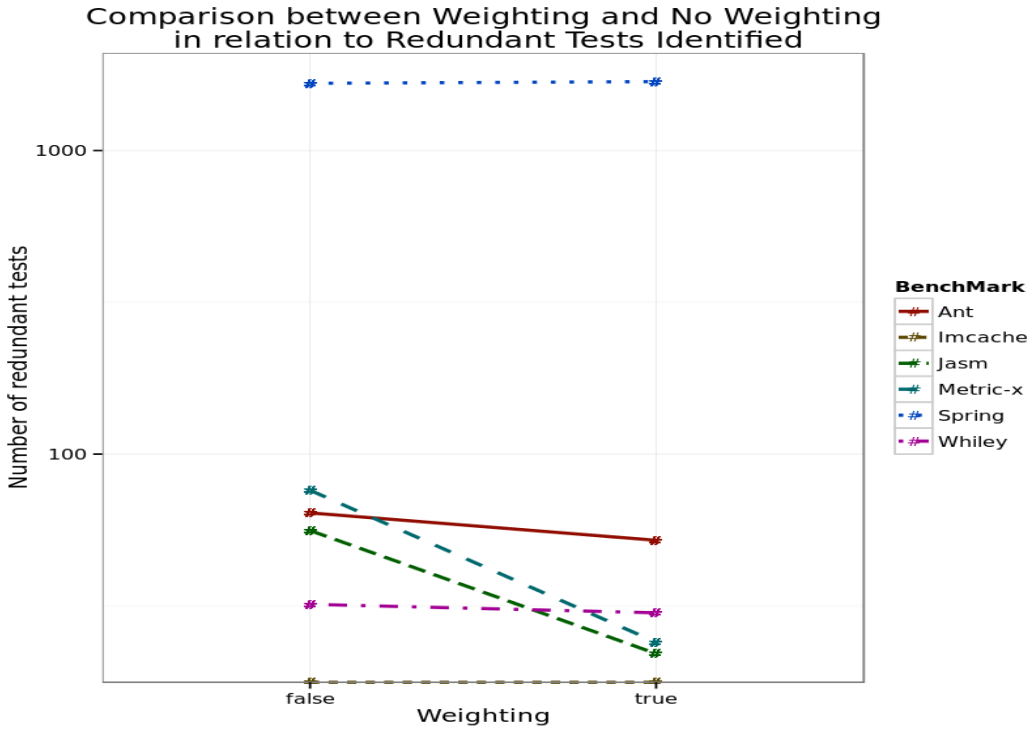

Figure 4.7: A figure showing the effect that using weighting has on the number of redundant tests are identified.

remained in the data analysed was still a large portion of the data. We are able to accept hypothesis 6, although work remains to improve the weighting method to remove redundant tests completely.

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 weighting is calculated once per test case per analysis stage. The weighting calculation has a linear relation with the number of test cases. In comparison, every test case is compared to every other, therefore a n-squared relation with the number of test cases. The less test cases there are, the closer the linear relation is to the n squared and the more impact the linear weight calculation has on the overall time taken. This may explain a relation between size and time taken. The the overall impact on the time taken is shown in Appendix C.1. The figure shows there was no general impact from using weighting on the benchmarks. Taking the discussion into account, it allows us to invalidate hypothesis 7.

4.8 Redundant Test Case Coding

I think discuss some here

	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.

	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.

4.9 Limitations

Dave

A major limitation to the study was volatile memory. For the test suites Whiley and Ant, storing the parameter data and call tree endured a high level of volatile memory usage, limiting the amount of tests that were able to be retrieved. An approach that should have been considered with more thought was the use of a database. Although this would have introduced difficulty reducing the non-deterministic behaviour to a minimum. A major factor would be the difficulty around executing the analysis on a grid system and replicating the conditions each run.

Some of the benchmarks that were chosen contain multiple unit tests within one test case. The tool described throughout this paper would not be able to pick up when a test case subsumes another. The frame work only looks at the total method set, and for each of the singular unit tests, there would need to be a skewed proportion of the conjoined test case in regard to the total method executions. This means that unless one of the unit tests within the conjoined test case was arbitrary larger than the other then it would not be picked up as being redundant, unless there was another conjoined test case similar. This is where the other approaches identified in the related work would be more effective. By looking at the statement information, it is easier to determine when a test is a subset of another.

As discussed in Section 4.4, some benchmarks performed better with a two stage pipeline in comparison to a three stage, and vice versa. This may have been due to not being able to identify the optimal parameters for each benchmark, although there were multiple different parameters tested to explore each benchmark's optimal.

Maybe explore how I decided the optimal ?

Discussed how weighting is applied each stage, but this would be computing redundant information. However, this does not affect the experiments as weighting is ever only executed once per pipeline.

4.10 Summary

Summary?

summarise the experiments

Chapter 5

Future Work and Conclusions

Dave

This chapter will first discuss potential future work which would increase the usefulness of the tool for a wider range of audiences. The experiments are then reflected on and finally concluded.

Finish properly

5.1 Future Work

The tool created throughout the project allows for the trace information from a test suite to be stored and analysed using different techniques. The information we explored using to determine redundant tests was the method calls. Throughout the project, three main areas have been identified for future work.

- **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 often grow rapidly if the test cases contain a large number of method calls. An approach to get around this would be to integrate the tool with a database. This would allow the tool to handle arbitrary large suites. The down side would the speed of the tool would decrease as the read/write to disk is slower than to RAM. Giving developers this option would be preferred.

- **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. 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 was a good heuristic to use. On the other hand, it is difficult to judge how good method calls are overall at identifying redundant test cases without any comparison to statement coverage or purposeful redundancy and bugs added. One application could

be to combine the statement and method information together. Method information could be used 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 test cases (Chapter 3)
- Analyse 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 allowed developers to trace information, then run a range of different filters over the data without being constrained to one spectra. The three spectra 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% 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 ones which are end to end tests – 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 a heuristic. Filtering with the ”unique method calls” spectra, the tool was able to remove a large portion of the test case comparisons in the final stage had to perform.

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. An important thing to note is each benchmark had an optimal length of the pipeline, using a length larger caused more time to be taken to analyse than was saved. Pipelining particularly worked well with the use of a heuristic filter.

Call tree depth The experiments showed that as the depth of the call tree increased, there was limited level of reduction in the number of redundant tests identified. This result was unexpected. Although there was a change, it was lower that what was expected. This implies that by increasing the K depth without any other technique, there is limited improvement.

Parameter By using reflection to retrieve the fields of these objects it allowed more insight into the content that the parameter objects were holding and the nature of them, but it also meant more data had to be held and analysed resulting in an increased time taken. There exists this trade off between time taken and confidence of redundancy when using parameters.

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.

5.2.1 Conclusion

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. The output of the tool is adequate in the sense that there is a low number of tests identified and these tests contain limited non-redundant tests.

The results would be expected to build off those achieved in Maurer et al. [**?**] and Robinson et al. [**?**].

Appendices

Appendix A

The effect of pipeline size in regard to the time taken

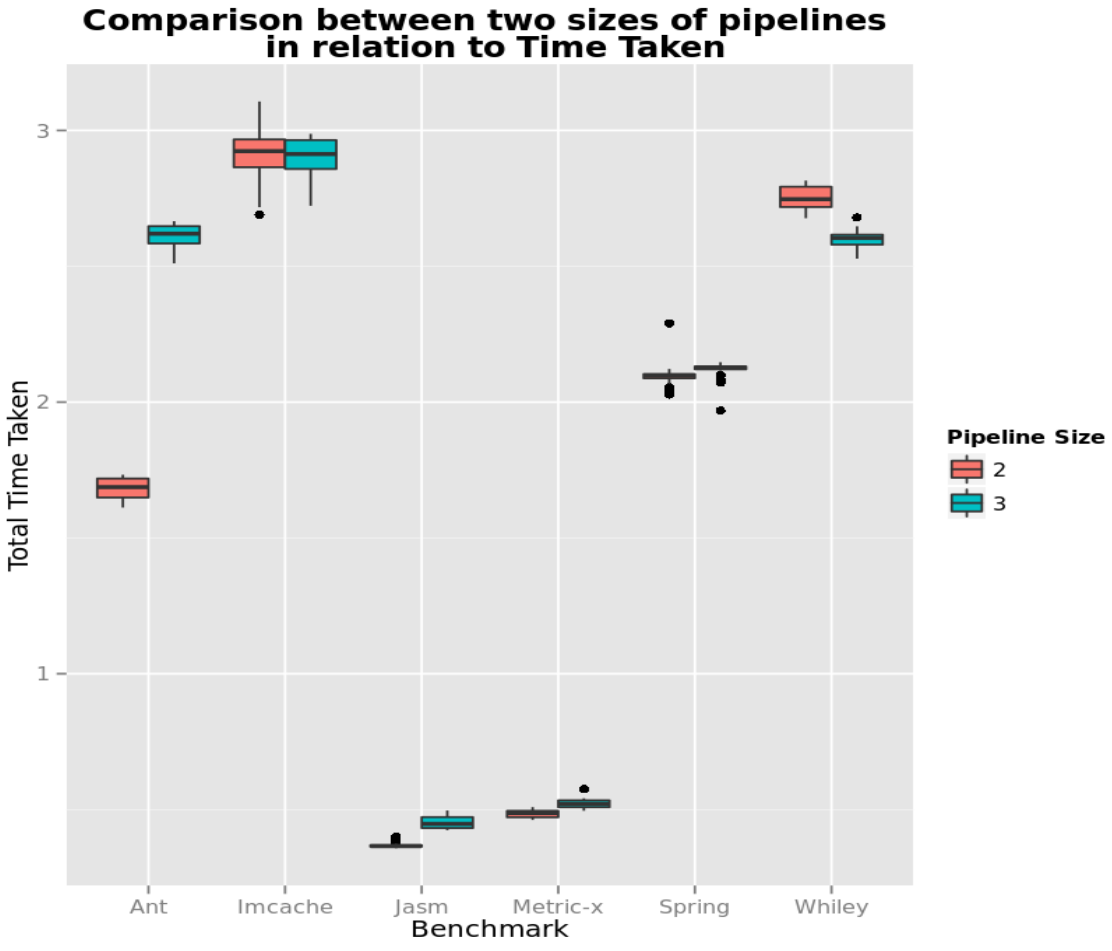

Figure A.1: A figure showing the relationship that using different pipeline sizes has on the total time taken to analyse the data.

Appendix B

The effect of parameters in regard to the time taken

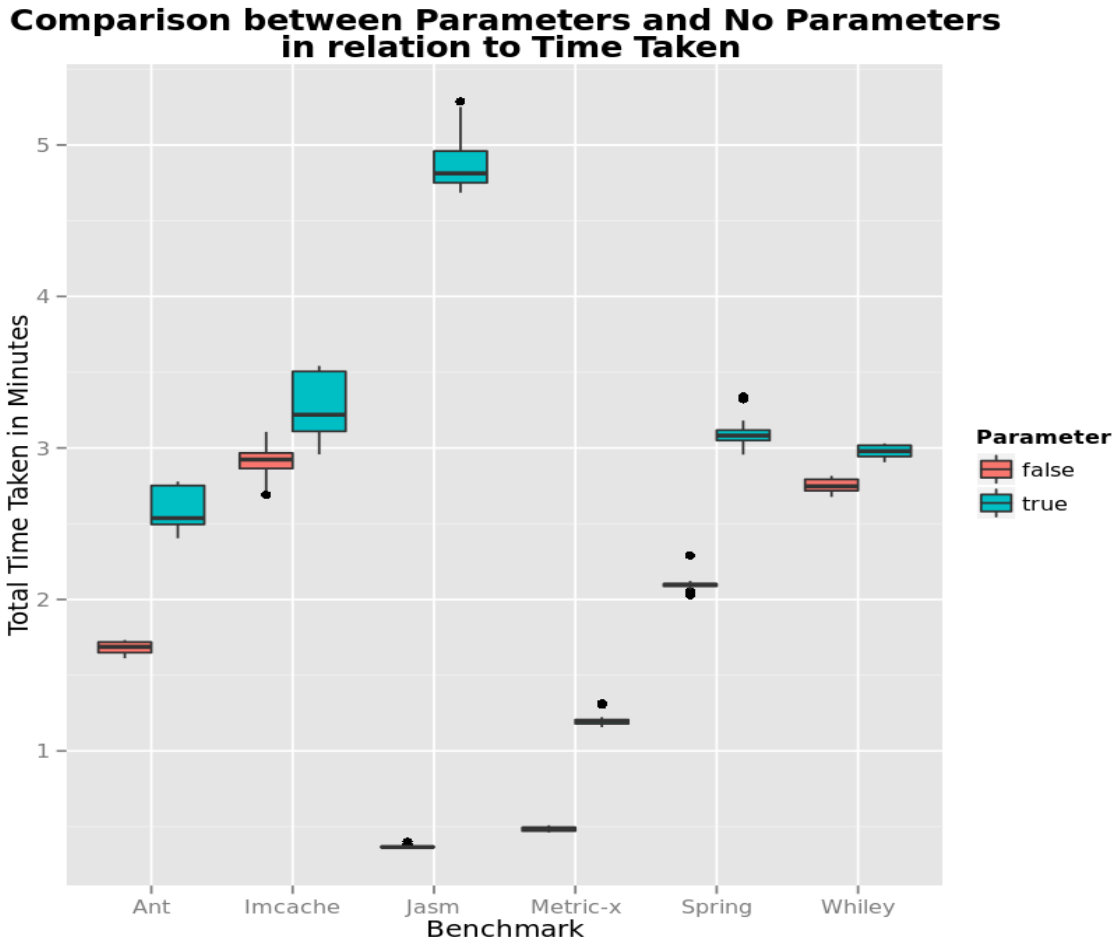

Figure B.1: A figure showing the relationship that using parameters has on the total time taken to analyse the data.

Appendix C

The effect of weighting in regard to the time taken

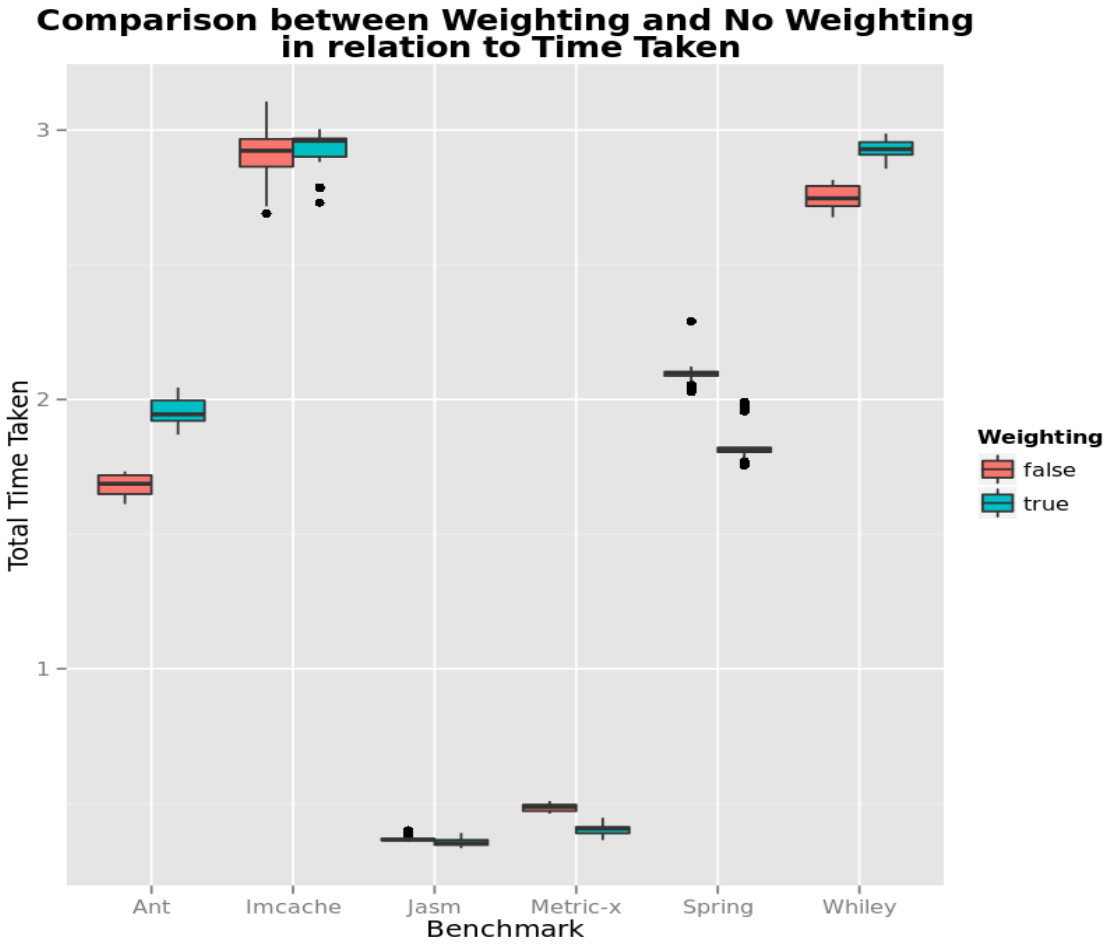

Figure C.1: A figure showing the relationship that using weighting has on the total time taken to analyse the data.

Appendix D

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

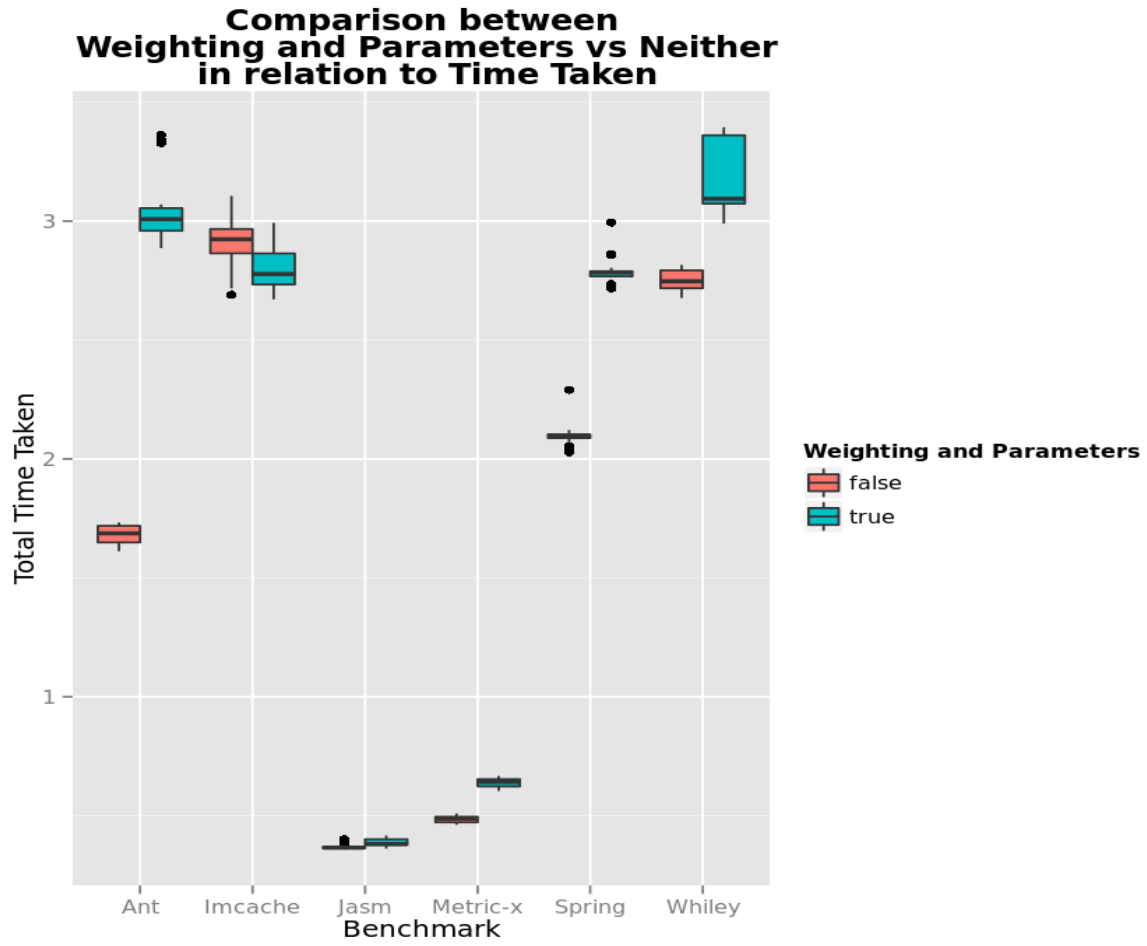

Figure D.1: A figure showing the relationship that using weighting and parameters combined has on the total time taken to analyse the data.

Appendix E

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

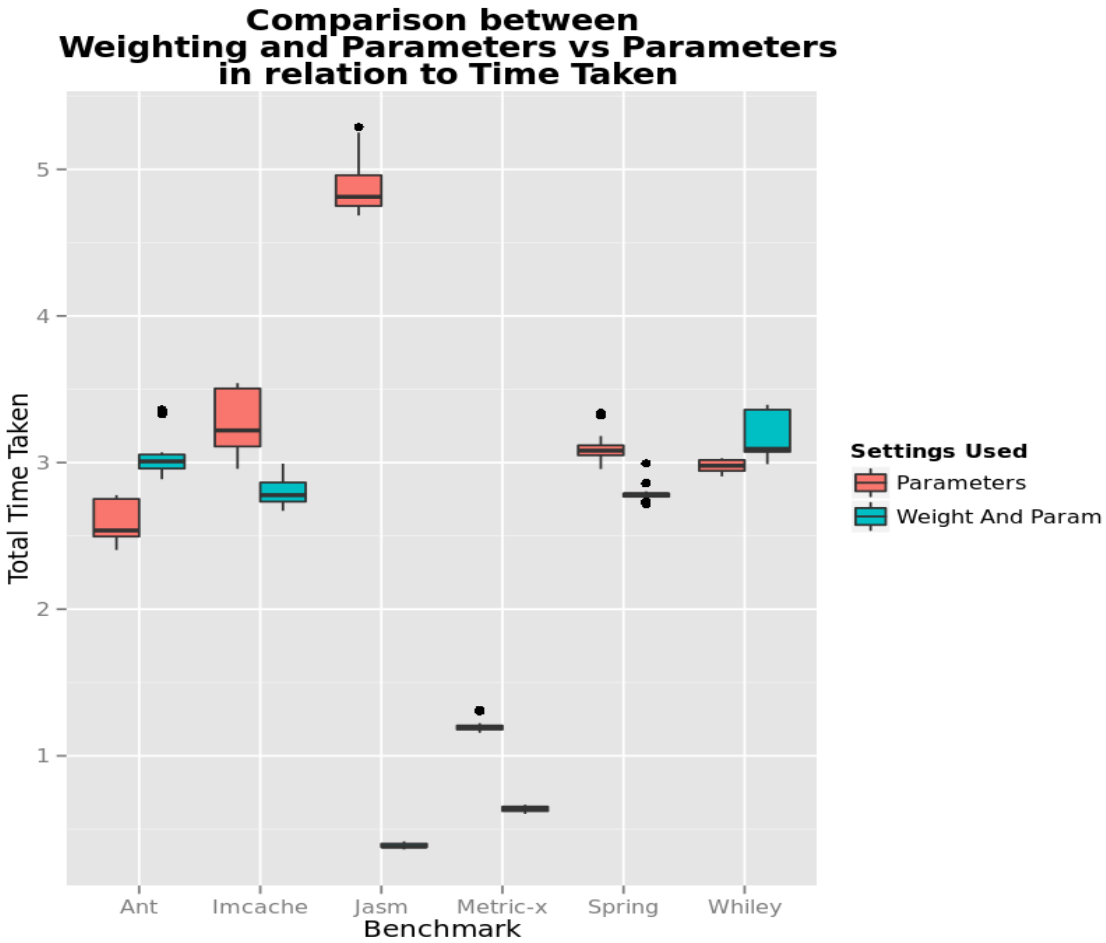

Figure E.1: A figure showing the relationship that using weighting and parameters combined vs parameters alone has on the total time taken to analyse the data.