Clustering and Dynamic Invariant Detection

Arno Pol

July 20, 2015

Master's thesis computing science
Study Line: Algorithmic Data Analysis
Utrecht University

Contents

1 Word of thanks

First I would like to thank Professor Wishnu Prasetya for his continued support during this project. His help was invaluable to my research. His detailed feedback helped improve this thesis, and his insightful comments and questions helped improve my understanding of software testing. Without his help and support I could never have completed this project.

I want to thank Professor Ad Feelders for his support with and insights on the data mining portion of this research. His feedback, and his insightful courses over the years helped me understand the field of Data Mining. Furthermore I would like to thank professor Michael Ernst with his support on Daikon, without his insight into interning issues debugging my software would have taken much more time.

2 Introduction

Computers are everywhere, from our cellphones, to our car computers and our desktops. Each of these systems can contain a vast array of programs, and these programs are built with a large amount of code. Software failures in these systems can lead to anything from benign program crashes to missile defense system failure. [20] Detecting these software failures is thus critically important to maintaining a modern automated society.

There are many tools and methods to validate software. In this thesis I will discuss invariant detection. Invariant detection is the automated generation of necessary conditions for software validation. This automation aims to save the programmer time and effort in detecting software failures.

This thesis describes a successful attempt at improving the output of dynamic invariant detection. I will examine a method of clustering the input to Daikon, a dynamic invariant detection tool. This technique leads to the detection of more correct invariants, at a minimal cost in invalid invariants. Furthermore I present a novel technique to quickly identify buggy programs without programmer intervention: "statistical kills".

This thesis is organized as follows: The introduction section contains information about invariant detection (section 2.1), the invariant detection tool Daikon (section 2.2), and prior research into clustering Daikon inputs by Dodoo, Lin, and Ernst[7] (section 2.3). This is followed by a section containing an outline of this thesis' research area, problem statement and research questions in chapter 3 . Chapter 4 contains an explanation of how my research was conducted, which samples were used, which quality measures were applied, and justifications for my methods.

Chapter 5 contains information about the results obtained trough the procedures described in chapter 4. Chapter 6 explains features of these results. Chapter 7 contains my conclusion about the research questions posed in section 3.1. Chapter 8 contains information about the new insights obtained trough my research. Finally chapter 9 contains information about avenues for future research I found interesting.

2.1 Invariant detection

An invariant of a set of executions is a property that holds at a certain point in a program during those executions. [11] Thus invariant detection is the detection of these properties. These properties take the form of logical statements about the variables at a certain program point.

In this thesis, I will define the term program point, as either an entry, or an exit point of a function in a program. An example of a logical statement about the variables at a program point, or invariant would be $x > 0$ or $abs(x) > y$ at myFunction::ENTER . Algorithm 1 shows an example of a program. Figure 1 shows an example of a set of valid invariants of this program. The set of invariants in figure 1 fully specify the program in algorithm 1.

Algorithm 1 Array copy program

Input: An array X
Output: An array A

```
1: function COPY(X)
2:     A ← array[]
3:     B ← 0
4:     while B < size(X) do
5:         A ← Append(A, X[B])
6:         B ← B + 1
7:     end while
8:     return A
9: end function
```

Ideally, an invariant detector would come to the following invariants.

Figure 1: Invariants for array copy program

Point name	Invariant
Copy::ENTER	$size(X) >= 0$
Copy::EXIT	$B \equiv size(X)$
Copy::EXIT	$X \equiv A$

These invariants describe which conditions must be satisfied in a normal execution of this program. Instrumenting a program with such invariants will reveal if the behavior of functions changes between revisions of a program. Given a sufficiently complete set of invariants and test executions, a change in the code resulting in abnormal behavior will result in an invariant being invalidated. If a set of inferred invariants is intuitive enough the programmer can review the

invariants against his expectations of the program state. This could lead to early discovery of critical bugs in program code.

In the example above, an array copy program is only valid if the resulting copy of the input array contains the same elements as the output. Therefore, by default, any invalid copy program will invalidate one of the invariants.

2.1.1 Types of invariant detection

There are two main types of invariant detection, static and dynamic invariant detection. The difference lies in the information used to infer invariants. Static invariant detection attempts to infer invariants directly from the program text. Whereas dynamic invariant detection attempts to infer invariants from the data generated by program executions.

Static invariant detection[12][18] is typically done using a symbolic execution algorithm. In symbolic execution an interpreter steps through a program, and determines value ranges for each variable at a program point. Unknown values returned by for example external methods are assigned a symbol.[14] Thus static invariant detection has no knowledge about the contents of a program's input variables or data returned from external functions. When evaluating a program it has less information about the values variables are likely to have. This relative lack of information results in very generic invariants.

Meanwhile dynamic invariant detection has it's own weakness, in the fact that it infers invariants from data gathered from program executions. Invariants inferred from this data will reflect the test case this state was computed from. For instance, we could run a set of executions of the copy algorithm in section 2.1 with as input an array containing the numbers 1 through 20. Then if we use this data generated from these executions for dynamic invariant detection, our invariant detector might infer that B always equals 20 once the function exits, and perhaps even invariants about the contents of X and A.

Thus here we see the main difference, and the main trade-off in invariant detection. The trade-off between soundness, and completeness. Where static analysis is always sound, it is usually incomplete. On the other hand, while dynamic analysis is often unsound, it is more complete.

2.2 Daikon

As explained in section 2.1.1, dynamic invariant detection seeks to extract invariants from program state. One of the most well-known tools to do this is

Daikon. Daikon was developed as a PhD project at Washington University. Daikon includes tools to automatically infer invariants for a variety of programming languages.

2.2.1 How Daikon infers invariants

Once one of the Daikon front-ends like Chicory is started with a program as its arguments it outputs a program trace. This program trace contains the variable, method and class definitions involved in this trace, as well as function invocations. Each invocation consists of a list of values of variables at a function's entry and exit point. Each function may have only one entry point, but multiple exit points. Each function invocation has a nonce, an incremental number, unique to each entry and exit pair.

Once this logging phase is completed Daikon begins its analysis of the program log file. Daikon reads the log file, and infers all derived values. A derived value is a value gained by applying a function to a variable, or two variables. These derived variables allow Daikon to quickly identify invariants that involve multiple variables and functions, invariants such as $x[y-1] == null$ or $max(y) >= x$. These derived variables do not need to appear in the Chicory output, or even the program source, they are simply helper variables for invariant detection.

Then Daikon proceeds to infer invariants in order of complexity, that is Daikon infers first, second and third order invariants for all supported functions. First order invariants are simply invariants over one variable. For instance, $x >= 0$. Second order invariants, are invariants over two variables, for instance $x + y >= 0$. And third order invariants are invariants over three variables. These invariants are generated one by one, and checked against the samples for the program point over which the invariant is inferred. If an invariant is invalidated by any of the samples for the program point, it is discarded. From these invariants that are compatible with our dataset, only invariants that are statistically justified are reported. [3][10]

Invariants are statistically justified when the probability of an invariant appearing in a random input is smaller than a user-specified threshold. The methods that compute this probability differ per invariant type, and can be found in Michael D. Ernst's thesis Dynamically Discovering Likely Program Invariants[9].

2.3 Prior work

Daikon does a good job at detecting invariants. Some invariant types however still require a lot of user involvement, for instance implication invariants. Implication invariants are implications of the form $A \Rightarrow B$, where A and B are

individual invariants, and the postcondition B is a logical consequent of the precondition A. In[7] Dodoo, Lin, and Ernst investigated the use of clustering to split data trace files into separate clusters to automate the detection of implication invariants. Their research resulted in the conclusion that cluster analysis to infer implication invariants is a useful technique.

In their paper, Dodoo, Lin, and Ernst[7] explain that Daikon uses algorithm 2 to infer implication invariants. Its inputs are two lists of invariants, gained by splitting a dataset using a splitting condition and then deducing invariants from them.

Algorithm 2 Create-Implications as per Dodoo, Lin, and Ernst[7]

Input: Two sets of invariants S_1 and S_2
Output:A list of implication invariants

```
1: function CREATE-IMPLICATIONS( S1,S2 )
2:     for s1 ∈ S1 do
3:         if ∃s2 ∈ S2 such that s1 ⇒ ¬s2 and s2 ⇒ ¬s1 then
4:             { s1 and s2 are mutually exclusive }
5:             for s' ∈ (S1 \ S2) do
6:                 output "s1 ⇒ s'"
7:             end for
8:         end if
9:     end for
10: end function
```

If a dataset is split in two using a Boolean condition, one of the generated invariant sets computed from this data contains the splitting condition and the other the negation of it. The above algorithm finds all such splitting conditions, then sees which invariants were gained by applying them. It then outputs an implication for each invariant gained from such a precondition.

The rationale behind using a clustering algorithm as the splitting predicate in the above method is that programs tend to follow different paths depending on input variables. These different paths contain different operations, and thus clustering will divide the data in clusters for different program paths taken. The division between clusters will then represent the conditionals in the program source.

2.3.1 Goal functions

In [7] Dodoo, Lin, and Ernst applied two test suites. One of them is a computer generated test suite, from which invariants were inferred with Daikon, and ver-

ified with ESC/Java[12]. Their precision and recall functions were constructed as follows:

$$precision = \frac{\text{number of invariants that are correct}}{\text{number of reported invariants}}$$

$$recall = \frac{\text{number of invariants that are correct}}{\text{number of invariants in goal set}}$$

number of invariants in goal set	=	number of correct invariants reported by Daikon
	+	number of additional invariants necessary to validate Daikon-generated invariant set with ESC/Java

These metrics measure if automatically generated implication invariants decrease the effort necessary by the programmer to verify invariants generated by Daikon using ESC/Java. Dodoo, Lin, and Ernst[7] show that this is indeed the case.

3 Research problem

Dodoo, Lin, and Ernst[7] confirmed that clustering is a good way to generate splitting conditions for detecting implication invariants. The problem is then determining an optimal clustering algorithm for this task. Dodoo, Lin, and Ernst[7] recommend investigating a clustering function that is more robust to outliers than the k-means algorithm used in their paper.

Furthermore I believe that clustering by itself has merits, as it exposes invariants that could not ordinarily be inferred. If indeed splitting separates data in groups based upon the program flow, splitting itself should be sufficient to improve Daikon's output. Therefore I also wish to evaluate the effect of clustering as-is on the quality of Daikon's output.

Another problem is that Dodoo, Lin, and Ernst[7] used a rather artificial measure to determine the value of clustering for implication invariants, as described in section 2.3.1. Ideally I would like to use a method of determining the effect of clustering on Daikon's output that involves verification of real world mistakes in code.

3.1 Research question

Thus my research questions are threefold. Can I determine a better clustering function for Daikon log files? Is it possible to determine a better metric for the quality of a clustering? And does clustering alone, without detecting implication invariants, have merit?

3.2 Hypothesis

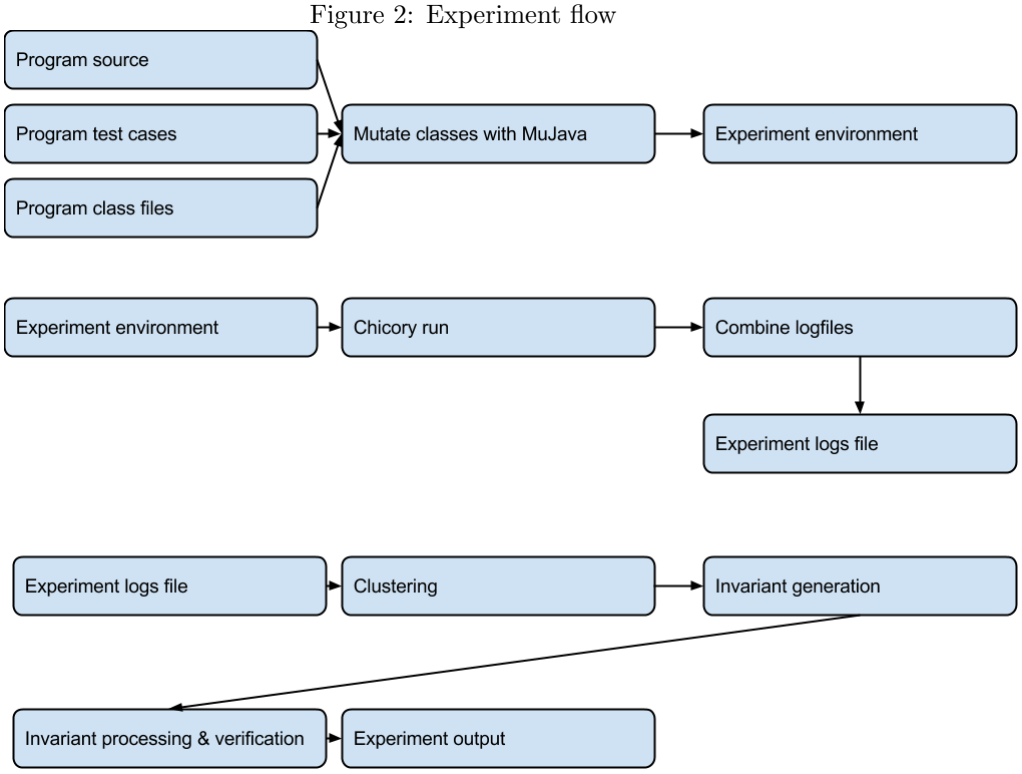
Since Dodoo, Lin, and Ernst[7] are limited in the attributes and distance functions they can evaluate by their choice of tools and algorithm, I should most certainly be able to find a better clustering algorithm for Daikon log files. By introducing mistakes similar to those a programmer makes into the code, and determining which invariants are invalidated, I expect to also be able to learn more about the quality of a clustering algorithm. Because conditional statements are not likely to be the only source of separate clusters in a program's execution trace, I expect simple clustering without implication inference to increase the quantity of useful invariants Daikon infers.

4 Methodology

4.1 Process overview

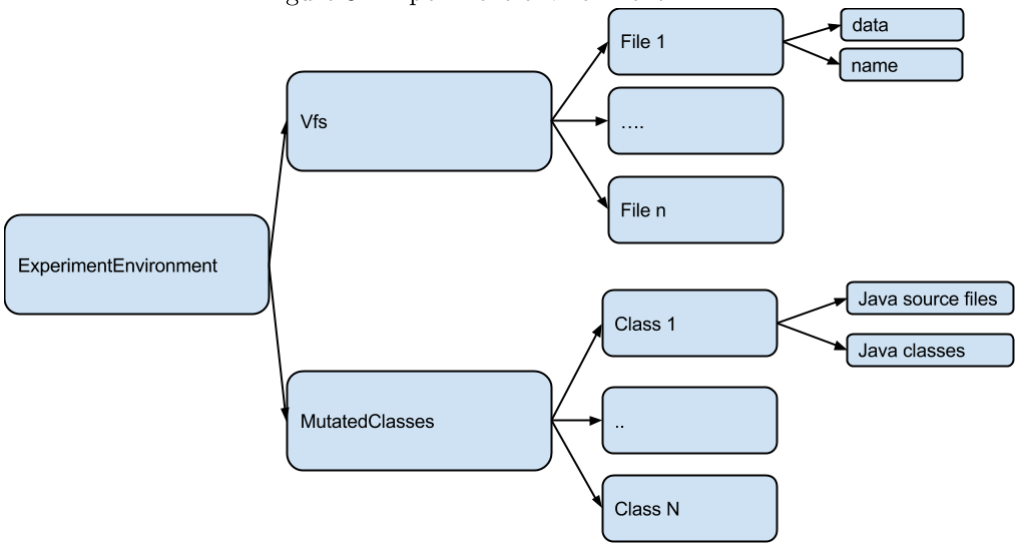
The setup for my experiment is quite involved, and fully automated. In figure 2 you can see a general outline of the process.

Figure 2: Experiment flow



First the source that is to be evaluated is compiled. The source code and binary class files are then processed using MuJava, generating the necessary mutated class files for the experiment. These mutated class files, including all other dependencies are then packaged in an experiment environment. An experiment environment is simply a large binary archive of all the files necessary to later reproduce a set of generated Daikon log files.

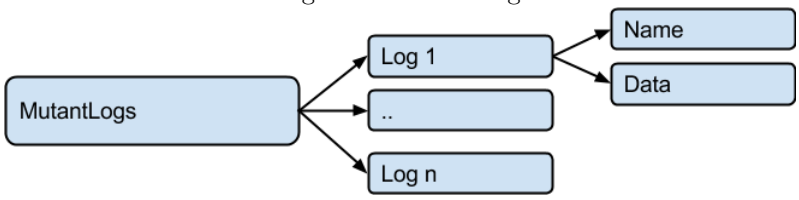
Figure 3: Experiment environment



In figure 3 you can see the structure of an experiment environment file, as generated by my software. It contains a virtual file system (Vfs) and a set of mutated classes. This file system contains one or more folders containing all Java source files and miscellaneous files necessary to run the program used in the experiment.

This file system is later extracted to a temporary folder, and one of its classes is replaced by one of the mutants from the list of mutated classes. This list of mutated classes contains mutants for all classes in the project source folder. These mutants are generated according to the process specified in section 4.2. Then a log file is generated by running this program with Chicory. The log files for all mutants are then combined into a MutantLogs file, the structure of which is shown in figure 4.

Figure 4: mutantLogs



A MutantLogs file is a collection of named, GZipped Daikon trace files. The name of each log file contains the class name, the mutation that was applied to it, and where this mutation was applied.

The clustering program loads the log environment, applies a given clustering algorithm to the logs, generates invariants for a base case, and determines the strength of the invariants against the generated mutants. Once a set of invariants has been tested against a certain split of the data, the result is computed and passed on to the graphical user interface.

4.2 Mutating the source

The first step in the experiment process is to generate a set of mutants. For this I will use a mutation testing engine. Mutation testing[17] is a method to test the effectiveness of test cases. In mutation testing the source code of a program is modified in order to introduce flaws. After mutants are generated, test cases are run against the mutants to see if the mutations are detected. A set of test cases is said to be strong if the majority of mutations are detected.

The tool I employed for mutation testing is MuJava[17] (Mutation System for Java). MuJava is the product of a collaboration between the Korea Advanced Institute of Science and Technology in South Korea, and George Mason University in the USA.

MuJava had two major shortcomings for my project, only one of which could be mitigated. The first major flaw was that not every mutation leads to an actual change in program executions. Since the typical Java program contains quite a bit of unused and unreferenced methods, modifying these will have no effect on the actual runs of the program.

The second major flaw was the large number of generated mutants resulting from applying all the available mutation operators in MuJava. As quite a bit of time is taken per run of the program, and often multiple runs are necessary to produce good log files, I sought to reduce the number of operators applied to the program source. Fortunately, my supervisor Professor, Wishnu Prasetya found a reference to the minimal set of operators necessary for mutation testing in Java programs. Figure 5 contains information from Introduction to Software Testing[1] combined with which tests are actually available in MuJava.

Figure 5: Sufficient mutations

Ammann & Offutt	Description	Relevant MuJava operators
ABS	Absolute value insertion (insert abs() instruction)	None
AOR	Arithmetic operator replacement (Replace + - * / ** and %). Arithmetic operators are also replaced by their left and right operand.	AORU AORB
ROR	Relational operator replacement (Replace < <= > >= = ! =) and replace relational statements by true or false.	ROR
COR	Conditional operator replacement (replace & && | ||) and insert true false as left or right operand.	COR
SOR	Shift operator replacement replace << >> and replace shift statements by the left operand.	SOR
LOR	Logical operator replacement replace bitwise operators. Logical operators are also replaced by their left and right operand.	LOR
ASR	Assignment operator replacement replace all the special assignment operators like *= with another.	ASRS
UOI	Unary operator insertion (insert unary operator + - ! ∼) before expressions of the correct type.	AOIU
UOD	Delete unary operator (+ - ! ∼) .	AODU
SVR	Scalar variable replacement replace order of appearance of variables in multiplication .	None
BSR	Bomb statement replacement, insert throw statements .	Irrelevant and detrimental to this research.

As you might notice there I have failed to find an absolute value insertion operator in MuJava. I have disabled the bomb station inserting operation as well, since throw statements can easily be detected by other methods than invariant testing.

Each of the operators shown in the table above might generate multiple mutants. For instance example the AOR operation, applied to the expression $a = b*c$ will generate the following mutated expressions: $a = b$, $a = c$, $a = b + c$, $a = b - c$, $a = b/c$, $a = b**c$ and $a = b\%c$. So even with a strongly reduced set of mutation operators, there is still quite a large set of possible mutants for each line of code. Automating the running of each of these mutants however, makes it feasible to apply this to small and medium sized Java projects.

4.3 Test Subjects

In this thesis I'll use two of the test suites also used in[6]. Namely StackAr and QueueAr, two projects that are shipped with the Daikon invariant detector.

I chose these two test suites for their ubiquitous usage in papers about Daikon, and their good code coverage. The first of those properties will allow future

researchers to easily and quickly evaluate my results, while the second hopefully allows for a large number of mutants to influence the trace files generated by Daikon.

4.3.1 StackAr

StackAr contains 462 lines of code. 73% of it's statements are covered by running StackArTester. In the class MyInteger only the constructor is covered. Thus any mutations of MyInteger are unlikely to result in changes to Daikon trace files generated from StackAr.

Figure 6: StackAr coverage graph

Name	Statement	Branch	Loop	Term	?-Operator
StackAr	73.0 %	58.3 %	42.9 %	62.5 %	0.0 %
DataStructures	73.0 %	58.3 %	42.9 %	62.5 %	0.0 %
Comparable	-	-	-	-	-
Hashable	-	-	-	-	-
MyInteger	12.5 %	0.0 %	-	0.0 %	0.0 %
Overflow	-	-	-	-	-
StackAr	63.6 %	50.0 %	0.0 %	41.7 %	-
StackArTester	81.2 %	75.0 %	60.0 %	100.0 %	-
Underflow	-	-	-	-	-

4.3.2 QueueAr

QueueAr contains 544 lines of code. 55.3% of it's statements are covered by running QueueArTester. The problems with coverage of the MyInteger class are exactly the same as with StackAr.

Figure 7: QueueAr coverage graph

Name	Statement	Branch	Loop	Term	?-Operator
QueueAr	55.3 %	50.0 %	44.4 %	63.6 %	0.0 %
DataStructures	55.3 %	50.0 %	44.4 %	63.6 %	0.0 %
Comparable	-	-	-	-	-
Hashable	-	-	-	-	-
MyInteger	12.5 %	0.0 %	-	0.0 %	0.0 %
Overflow	-	-	-	-	-
QueueAr	27.4 %	60.0 %	0.0 %	50.0 %	-
QueueArTester	95.0 %	50.0 %	66.7 %	100.0 %	-
Underflow	-	-	-	-	-

4.4 Testing the test set

In order to determine how well errors in the test suites described in section 4.3 are reflected in the invariants generated from them. I will Run Daikon on a log file generated by a correct version of the StackAr program, and evaluate the invariants generated by Daikon on this program against a different set of logs for this program and it's mutants .

Figure 8: Initial output for StackAr, no clustering

Class	Method	Mutation	Invariants	Invalidated invariants
-	-	no_mutation	629	13
-	-	no_mutation	629	9
-	-	no_mutation	629	0
StackAr	void_makeEmpty()	AORB_7	629	41
StackAr	boolean_isFull()	ROR_14	629	14
StackAr	boolean_isFull()	ROR_13	629	71
StackAr	boolean_isFull()	ROR_12	629	76
StackAr	boolean_isFull()	ROR_11	629	73
StackAr	boolean_isFull()	ROR_10	629	76
StackAr	boolean_isFull()	ROR_9	629	14
StackAr	boolean_isFull()	ROR_8	629	10
StackAr	boolean_isFull()	AORB_4	629	12
StackAr	boolean_isFull()	AORB_3	629	63
StackAr	boolean_isFull()	AORB_2	629	15
StackAr	boolean_isFull()	AORB_1	629	5
StackAr	boolean_isEmpty()	ROR_6	629	56
StackAr	boolean_isEmpty()	ROR_4	629	13
StackAr	boolean_isEmpty()	ROR_2	629	54
StackAr	StackAr()	AOIU_1	629	5
MyInteger	int_hash(int)	ROR_29	629	12
MyInteger	int_hash(int)	ROR_28	629	15
MyInteger	int_hash(int)	ROR_27	629	14
MyInteger	int_hash(int)	ROR_26	629	14
MyInteger	int_hash(int)	ROR_25	629	14
MyInteger	int_hash(int)	ROR_24	629	14
MyInteger	int_hash(int)	ROR_23	629	12
MyInteger	int_hash(int)	AOIU_4	629	11
MyInteger	int_hash(int)	AODU_2	629	15
MyInteger	int_hash(int)	AORB_8	629	14
MyInteger	int_hash(int)	AORB_7	629	10
MyInteger	int_hash(int)	AORB_6	629	11
MyInteger	int_hash(int)	AORB_5	629	14
MyInteger	int_hash(int)	AORB_4	629	16
MyInteger	int_hash(int)	AORB_3	629	15
MyInteger	int_hash(int)	AORB_2	629	8
MyInteger	int_hash(int)	AORB_1	629	11
MyInteger	boolean_equals(java.lang.Object)	COR_2	629	17
MyInteger	boolean_equals(java.lang.Object)	COR_1	629	13
MyInteger	boolean_equals(java.lang.Object)	ROR_22	629	14
MyInteger	boolean_equals(java.lang.Object)	ROR_21	629	13
MyInteger	boolean_equals(java.lang.Object)	ROR_20	629	12
MyInteger	boolean_equals(java.lang.Object)	ROR_19	629	6
MyInteger	boolean_equals(java.lang.Object)	ROR_18	629	10
MyInteger	boolean_equals(java.lang.Object)	ROR_17	629	10
MyInteger	boolean_equals(java.lang.Object)	ROR_16	629	8
MyInteger	int_compareTo(DataStructures.Comparable)	ROR_14	629	8
MyInteger	boolean_equals(java.lang.Object)	ROR_15	629	13
MyInteger	int_compareTo(DataStructures.Comparable)	ROR_13	629	15
MyInteger	int_compareTo(DataStructures.Comparable)	ROR_12	629	14
MyInteger	int_compareTo(DataStructures.Comparable)	ROR_11	629	13
MyInteger	int_compareTo(DataStructures.Comparable)	ROR_10	629	14
MyInteger	int_compareTo(DataStructures.Comparable)	ROR_9	629	7
MyInteger	int_compareTo(DataStructures.Comparable)	ROR_8	629	5
MyInteger	int_compareTo(DataStructures.Comparable)	ROR_7	629	17
MyInteger	int_compareTo(DataStructures.Comparable)	ROR_6	629	17
MyInteger	int_compareTo(DataStructures.Comparable)	ROR_5	629	8
MyInteger	int_compareTo(DataStructures.Comparable)	ROR_4	629	14
MyInteger	int_compareTo(DataStructures.Comparable)	ROR_3	629	15
MyInteger	int_compareTo(DataStructures.Comparable)	ROR_2	629	11
MyInteger	int_compareTo(DataStructures.Comparable)	AODU_1	629	15
MyInteger	java.lang.String_toString()	AOIU_3	629	15
MyInteger	int_intValue()	AOIU_2	629	12
MyInteger	MyInteger(int)	AOIU_1	629	11

In figure 8, for entries containing an original program, marked by a value of no_mutation in the operation column, a maximum of 13 invariants are invalidated. There are quite a few mutants that invalidate a similar or lower number of invariants. Looking at the names of the mutants, it becomes obvious why. The majority of mutations is applied to the class MyInteger, which had very poor coverage. Thus, by far not every mutation of the source code leads to an actual change in the log files generated.

In my experiment I will remove all mutants that do not lead to a change in program behavior. This ensures that I can accurately measure how many mutants are recognized by my algorithm.

Another problem is that the number of invalidated invariants even in the case of a correct program is high.

There are several possibilities to reduce the number of these false positive invariants. By far the easiest approach to reduce the number of false positives in data mining is to gather more data. For this thesis, I will run multiple runs of each program, then combine the log files. These combined log files contain more information about the program, and do thus yield more accurate invariants.

Another approach commonly taken is to have a human agent verify the invariant sets. This is laborious, and in this case unnecessary. Not least of all because what I am attempting to measure is the increase in the number of distinct invariants - and the number of false positives generated through clustering.

4.5 Clustering algorithms and distance functions

The next step in my experiment is clustering. This section describes which clustering functions I applied to the input data. As these functions are applied to huge datasets, both speed and accuracy are important. Below you will see information about the functions I used in this experiment.

4.5.1 Clustering functions

The design of the software used in this thesis allows for interchangeability of clustering and distance functions. This means that I have the freedom to define and use any arbitrary clustering function. Furthermore, unlike applied in Dodoo, Lin, and Ernst[7], I also will attempt to cluster on attributes of the data that are not numeric - for instance strings and arrays. Below I will explain the clustering algorithms I used, and the reasons for using them.

4.5.1.1 K-medoids

K-medoids clustering is a clustering algorithm first described by Kaufman and Rousseeuw [13]. Whereas k-means, as used by Dodoo, Lin, and Ernst[7], constructs a center point from all samples in a cluster, K-medoids picks one of the data-points as the center of a cluster. This makes it less vulnerable to outliers, as points far from the center have no influence at all on the location of the cluster's center.

Due to the fact that centers are chosen from the dataset, and not constructed, any distance metric can be applied. As there is no need to average any values in any distance metrics value space.

This allows us the freedom to experiment with different distance metrics without having to restructure our clustering algorithm. Below you will see the variant of k-medoids used to generate the result in this thesis.

Algorithm 3 K-medoids clustering

Input:a set of value tuples at function exit points L
Input:a parameter for the number of clusters K
Input:a parameter for the number of restarts R
Output:a set of sets of function executions Outputmeds

```
1: function CLUSTER-DATA( L,K,R )
2:     Outputmeds ← ∅
3:     for Restart ∈ {1, ..., R} do
4:         Meds ← RandomMedoids(L, K)
5:         for Counter ∈ {1, ..., sqrt(#L)} do
6:             Changed ← False
7:             Bestmeds ← Meds
8:             for Medoid ∈ Meds do
9:                 for Iteration ∈ {1, ..., sqrt(#L)} do
10:                    Newmedoid ← RandomFrom(L)
11:                    Newmeds ← (Meds \ {Medoid}) ∪ {Newmedoid}
12:                    if Distance(Newmeds, L) < Distance(Bestmeds, L) then
13:                        Bestmeds ← Newmeds
14:                        Changed ← True
15:                    end if
16:                end for
17:            end for
18:            Meds ← Bestmeds
19:            if ¬Changed then
20:                break
21:            end if
22:        end for
23:        if Distance(Meds, L) < Distance(Outputmeds, L) then
24:            Outputmeds ← Meds
25:        end if
26:    end for
27:    return Outputmeds
28: end function
```

As can be seen above a set of runs of k-medoids were completed. For each medoid, only a subset of data points is tried as a replacement. This was done to reduce the computation time necessary for my algorithm. Better results could be had by evaluating every data point for every medoid, but at a massive cost in program runtime. More restarts will lead to a better clustering. I chose 3 restarts as it was the number of restarts that reduced the computation time necessary for my experiment to about a day per program.

4.5.1.2 Random clustering

Another clustering algorithm I applied was random clustering, where items are randomly assigned to an output cluster. Random clustering was used both as a

benchmark, and because it was the optimal clustering solution Dodoo, Lin, and Ernst[7] found for the problem of detecting implication invariants.

4.5.2 Distance functions

In my experiment various distance metrics were applied, and their effect on the output evaluated. Below I will discuss the various distance metrics, their origins and merits.

4.5.2.1 Naive distance

The naive distance metric simply compares each entry in the value lists for two execution states at a program point, and counts the number of values that differ. Below you can see a recursive definition of this function. This function operates on recursive sets of values. In this notation, the Java array containing values 1 through 3 would be described as {1,{2,3}} . All following distance function definitions will follow this model of an array.

Figure 9: Naive distance function

$d(\emptyset, \emptyset) = 0$
$d(\{v, X\}, \{v, Y\}) = 0 + d(X, Y)$
$d(\{a, X\}, \{b, Y\}) = 1 + d(X, Y)$

4.5.2.2 Semi-naive distance

The semi-naive distance metric ignores array-type values, and computes a normalized hamming distance between String values. For numeric values the percentage difference between the values is computed. Any other types are simply compared, and one is counted if they differ, zero if they do not differ. All these values are summed up, and returned as the Semi-naive distance metric. Below you can see the semi-naive distance function.

Figure 10: Semi-naive distance function

$d(\emptyset, \emptyset) = 0$
$d(\{\{a\}, X\}, \{\{b\}, Y\}) = 0 + d(X, Y)$
$d(\{"a", X\}, \{"b", Y\}) = Hamming(a, b) + d(X, Y)$
$d(\{Num(a), X\}, \{Num(b), Y\}) = (Max(a, b) - Min(a, b))/Max(a, b) + d(X, Y)$
$d(\{v, X\}, \{v, Y\}) = 0 + d(X, Y)$
$d(\{a, X\}, \{b, Y\}) = 1 + d(X, Y)$

4.5.2.3 SumDistance

Almost the same as the Semi-naive distance metric, however the SumDistance metric computes distance values for array values too. See below.

Figure 11: SumDistance distance function

$d(\emptyset, \emptyset) = 0$
$d(\{\{a\}, X\}, \{\{b\}, Y\}) = d(a, b)/Max(\#a, \#b) + d(X, Y)$
$d(\{"a", X\}, \{"b", Y\}) = Hamming(a, b) + d(X, Y)$
$d(\{Num(a), X\}, \{Num(b), Y\}) = (Max(a, b) - Min(a, b))/Max(a, b) + d(X, Y)$
$d(\{v, X\}, \{v, Y\}) = 0 + d(X, Y)$
$d(\{a, X\}, \{b, Y\}) = 1 + d(X, Y)$

4.6 Statistical kills

As explained in the section on code coverage, a number of false positives were generated for a correct program. I was determined however to find a way to automatically deduce if a program has indeed been mutated, or not. Preferably without the need for human filtering of the invariant sets. This led me to the notion of a statistical kill.

A statistical kill is an instance of a program where the number of invalidated invariants sufficiently deviates from the average number of invariants invalidated on a run of a correct program to warrant marking the program as buggy. Algorithm 4 shows the algorithm I applied to determine the difference between a mutant and a correct program.

Algorithm 4 Mutant detection algorithm

Input:a set of log files of a correct program L
Input:a log file that might or might not belong to a mutant M
Output:a boolean B

```
1:  function IsMutant( L,M )
2:      l ← PickRandomElement(L)
3:      N ← Cluster(l)
4:      I ← Daikon(N)
5:      E ← []
6:      for t ∈ L do
7:          if t ≠ l then
8:              E ← Append(E, InvariantsInvalidated(I, t))
9:          end if
10:     end for
11:     return InvariantsInvalidated(I, M) > Max(E)
12: end function
```

This algorithm does the following: It takes as input a set of execution logs of a correct program. It picks from this set, at random, one log and generates a set of invariants for it. These invariants are then tested against all the other log files of the correct program. The number of failed invariants is recorded, and stored.

The maximum of the number of failed invariants is then computed. Once a program invalidates a number of invariants, above this number, it is considered a mutant. This is equivalent to looking up the recall value for 100% precision in a precision-recall graph.

4.7 Quality measures

As described in section 2.1, invariants represent information about the values that one or more variables at a program point are likely to have. Increasing the number of invariants found by Daikon, therefore means that more information about a program's variables was captured in its invariants. However optimizing the output of Daikon is not as simple as maximizing the number of invariants created. This section discusses the quality measures chosen for this experiment, and why they were chosen.

4.7.1 Distinct invariants gained

The number of distinct invariants, is simply the size of the set of all invariants from all clusters. Thus the number of invariants gained is the number of invariants generated with clustering minus the number generated without clustering. This measure is based upon the idea that an invariant represents a constraint on, and thus information about a program's variables. Therefore more information is better.

However, this quality measure alone is not sufficient. Invariants are only useful insofar they help the programmer identify program faults. Thus an overly specific invariant set, might become tailored to the test case and not the program.

In this case many of our invariants will be invalidated by a correct program, and therefore they will provide no useful information to the programmer. This is analogous to the data-mining concept over-fitting, where a model of the data represents the training data in such great detail that it is no longer useful on different datasets. To avoid over-fitting, the next quality measure is essential.

4.7.2 Number of false positives

The number of invariants that fail on a correct program on average. A good algorithm will minimize the input required by a programmer. Therefore I will seek to minimize the number of invariants a programmer has to manually filter out. A bigger number of invalid invariants will also lead to worse results when deriving anything from these invariants, for example implication invariants.

4.7.3 Number of implication invariants

The number of implication invariants, generated using the method described in section 2.2. This should give an idea of the number of implications that can be inferred from this clustering. As implications are inferred from mutually exclusive invariants, it also gives a good idea of the separation between clusters. Or in other words how well a clustering algorithm is actually separating the data into distinct groups.

4.7.4 Mutants detected

The percentage of mutants detected using the statistical kill algorithm specified in the previous section. This gives us an idea of the noise generated by false positives. If there are too many false positives in the generated invariant set, quickly determining the difference between a mutant and a correct program becomes difficult. Thus, if this number deviates too far from 100% this is a good indication that false positives are starting to become problematic, or that my statistical kill algorithm is inaccurate.

4.7.4.1 Verifying actual mutants

To verify the the above quality measure I needed information about which mutations influence the program. In order to determine the mutants that have an influence on the program path, I extracted the generated mutants from a log environment file. I removed all the mutant class files, and kept only the source files. I then filtered out all the files that contained mutants for functions that were not covered.

I then proceeded to work my way trough the remaining few source files,seeing if the mutations applied had any effect on the program's execution. After this

I removed all those mutants that did not influence the program from this experiment's environment files. This leaves a clearly defined set of mutants that introduce bugs that change program behavior.

4.7.5 Aggregate quality

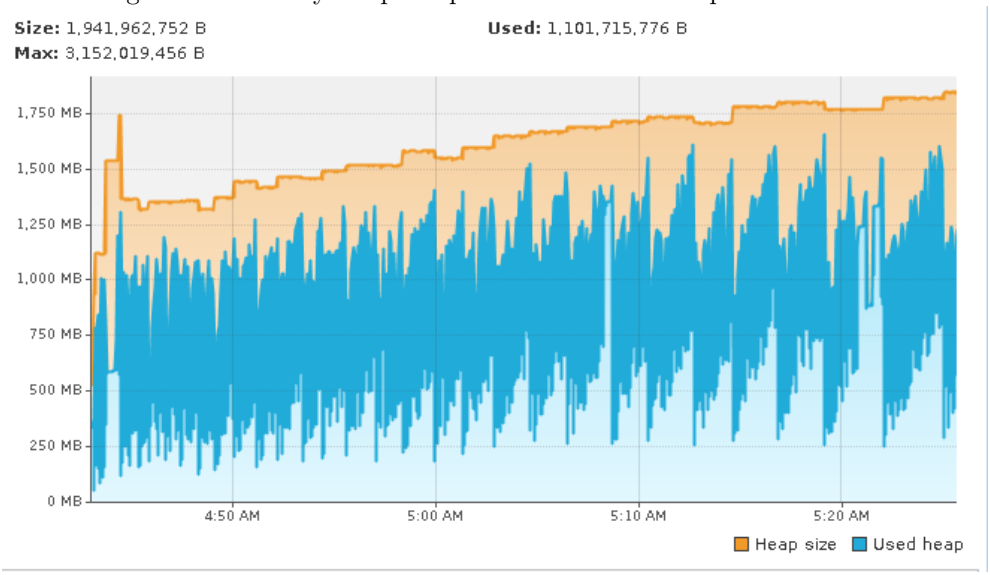
As described in section 2.3 a good clustering ideally reflects different branches in the program text. When a clustering closely reflects branch statements in the code, it will group over properties that appear time and time again in it's program logs. Hence the number of false positives will be low. When this same clustering captures a lot of information about these branches, the number of invariants gained by assigning items in a dataset to clusters will be high. Hence the aggregate quality of a clustering is the total number of unique invariants gained by clustering over Daikon without clustering, divided by the number of false positives.

4.8 Visual Debugging

During development I ran into a series of issues with my code. As in any sufficiently complex project debugging turned out to be far from trivial. Most of these issues were found in the code that read log files and converted them to an internal representation that was suitable for quick manipulation. Another big area of potential errors was memory management. This section discusses one of the tools I used to explore one of these memory management problems.

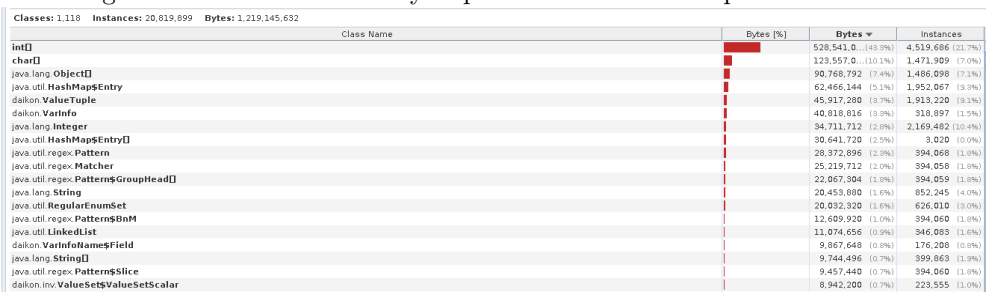
VisualVM is a visual performance analysis tool for Java. It allows the user to monitor information about the program, in real time. Using VisualVM to graph the memory of my software, I get figure 12. This graph contains information about the program's memory usage and the objects that occupy its memory space.

Figure 12: Memory Graph of partial execution of experiment.



Looking at figure 12, it appears that the memory usage is linearly increasing with execution time. Since the system memory available per task is fairly limited, such an increase is problematic. Below we'll see which objects occupy the majority of the program's memory space.

Figure 13: Classes in memory of partial execution of experiment.



All these structures have something in common, namely that they are objects generated by reading and manipulating log files. After dealing with an interning problem in my code, with the help of Professor Michael Ernst, and going over the temporary solution to the memory management issues, I quickly figured out the origin of the poor memory management.

Java has an interning system, which means that it has a set of functions that can ensure that only one copy of the same data is stored in memory. Daikon

makes heavy use of interning, to reduce its memory footprint. To store interned objects Daikon uses a WeakHashMap in its interning system. This class is not thread safe, thus multi-threaded access wreaked havoc on its internals, causing it to never forget objects that were allocated!

Thus, to run Daikon's thread-unsafe code in a multi-threaded manner, I had to launch a different JVM instance for each thread. This due to the fact that each JVM instance gets its own interning system. I implemented a quick MapReduce[4] algorithm, as an elegant way to delegate computation to different JVM instances.

5 Results

5.1 QueueAr

First I will examine the effects of different clustering algorithms on the log file produced for 10 runs of the program QueueAr, included in the Daikon source folder. The QueueAr test suite was modified to use a random number generator seeded with a static value, and executions were logged in random order. This to diminish the effect of system time on the internal state of the program.

Algorithm	No. clusters	Distance Metric	% Detected	Invariants added	Number of false positives	Number of implication invariants	Quality
Random split	1	-	100.0	0	1.0	0	0.0
Random split	3	-	88.24	20	18.3	1664	1.09
Random split	5	-	17.65	140	106.6	19474	1.31
Random split	7	-	0.0	297	246.7	69064	1.20
Random split	9	-	0.0	351	288.3	91840	1.22
Random split	11	-	0.0	434	381.2	150434	1.14
Random split	13	-	0.0	402	390.7	175328	1.03
Random split	15	-	0.0	553	507.7	245254	1.09
Random split	17	-	0.0	751	641.5	320390	1.17
Random split	19	-	0.0	722	662.0	374940	1.09
K medoids	1	SumDistance	100.0	0	1.0	0	0.0
K medoids	3	SumDistance	58.82	0	1.0	0	0.0
K medoids	5	SumDistance	58.82	-2	1.0	0	-2.0
K medoids	7	SumDistance	58.82	-2	1.0	0	-2.0
K medoids	9	SumDistance	58.82	-2	1.0	0	-2.0
K medoids	11	SumDistance	58.82	-2	1.0	0	-2.0
K medoids	13	SumDistance	58.82	-2	1.0	0	-2.0
K medoids	15	SumDistance	58.82	-2	1.0	0	-2.0
K medoids	17	SumDistance	58.82	-2	1.0	0	-2.0
K medoids	19	SumDistance	58.82	-2	1.0	0	-2.0
K medoids	1	Naive Sum Distance	100.0	0	1.0	0	0.0
K medoids	3	Naive Sum Distance	58.82	579	19.4	28900	29.85
K medoids	5	Naive Sum Distance	47.06	957	36.3	82124	26.36
K medoids	7	Naive Sum Distance	41.18	1311	80.3	185888	16.33
K medoids	9	Naive Sum Distance	41.18	1528	74.0	299314	20.65
K medoids	11	Naive Sum Distance	61.76	1794	65.3	511390	27.47
K medoids	13	Naive Sum Distance	44.12	1915	124.4	722384	15.39
K medoids	15	Naive Sum Distance	41.18	2013	122.8	1001608	16.39
K medoids	17	Naive Sum Distance	41.18	2050	139.9	1313582	14.65
K medoids	19	Naive Sum Distance	41.18	2278	157.4	1693116	14.47
K medoids	1	Semi-Naive Sum Distance	100.0	0	1.0	0	0.0
K medoids	3	Semi-Naive Sum Distance	41.18	49	14.7	1760	3.33
K medoids	5	Semi-Naive Sum Distance	41.18	110	17.0	5650	6.47
K medoids	7	Semi-Naive Sum Distance	41.18	187	48.1	19146	3.89
K medoids	9	Semi-Naive Sum Distance	14.71	244	70.4	44546	3.47
K medoids	11	Semi-Naive Sum Distance	29.41	264	48.4	61758	5.45
K medoids	13	Semi-Naive Sum Distance	2.94	308	87.3	143058	3.53
K medoids	15	Semi-Naive Sum Distance	2.94	235	74.3	116262	3.16
K medoids	17	Semi-Naive Sum Distance	2.94	296	101.7	196050	2.91
K medoids	19	Semi-Naive Sum Distance	2.94	258	43.5	226370	5.93

Figure 14: Output for QueueAr
Column 1: the clustering algorithm applied.
Column 2: the number of clusters generated.
Column 3: the distance metric used.
Column 4: the percentage of mutants detected using algorithm 4.
Column 5: the number of additional distinct invariants generated.
Column 6: the average number of false positive invariants generated.
Column 7: the number of implication invariants generated.
Column 8: the aggregate quality metric as described in section 4.7.5.

What we see here, is that my initial tests, namely random clustering and SumDistance produce suboptimal results. Random clustering predictably massively increased the number of false positive invariants. It also massively increased the number of invariants generated and the number of implications inferred from this. This happened because when randomly dividing any set of values over enough clusters, the range of values within each cluster will eventually drastically differ from the original range of values. K-medoids with SumDistance as distance function on the other hand clusters too strongly, and thus ends up assigning all samples to one cluster. Therefore it does not provide us with any extra invariants.

5.2 StackAr

Next I will look at the set of logs generated for 10 runs of StackAr.

Algorithm	No. clusters	Distance Metric	% Detected	Invariants added	Number of false positives	Number of implication invariants	Quality
Random split	1	-	100.0	0	2.0	0	0.0
Random split	3	-	77.78	32	34.4	3810	0.93
Random split	5	-	77.78	57	55.4	5362	1.03
Random split	7	-	0.0	100	107.9	22364	0.93
Random split	9	-	0.0	151	166.7	65314	0.91
Random split	11	-	0.0	172	185.7	126620	0.93
Random split	13	-	0.0	192	214.4	183126	0.90
Random split	15	-	0.0	238	258.5	285816	0.92
Random split	17	-	0.0	309	329.4	416316	0.94
Random split	19	-	0.0	367	387.5	507270	0.95
K medoids	1	Naive Sum Distance	100.0	0	2.0	0	0.0
K medoids	3	Naive Sum Distance	88.89	269	28.6	12684	9.41
K medoids	5	Naive Sum Distance	88.89	668	50.6	66084	13.20
K medoids	7	Naive Sum Distance	88.89	1068	95.6	176422	11.17
K medoids	9	Naive Sum Distance	77.78	1470	143.9	329548	10.22
K medoids	11	Naive Sum Distance	22.22	1618	138.7	492124	11.67
K medoids	13	Naive Sum Distance	66.67	1869	179.6	782328	10.41
K medoids	15	Naive Sum Distance	77.78	1951	183.9	1013388	10.61
K medoids	17	Naive Sum Distance	44.44	2214	230.3	1397668	9.61
K medoids	19	Naive Sum Distance	77.78	2391	214.4	1805192	11.15
K medoids	1	Semi-Naive Sum Distance	100.0	0	2.0	0	0.0
K medoids	3	Semi-Naive Sum Distance	88.89	201	24.4	13000	8.24
K medoids	5	Semi-Naive Sum Distance	0.0	298	87.3	34458	3.42
K medoids	7	Semi-Naive Sum Distance	88.89	440	84.5	109978	5.21
K medoids	9	Semi-Naive Sum Distance	22.22	430	131.3	123758	3.27
K medoids	11	Semi-Naive Sum Distance	0.0	416	110.4	116168	3.77
K medoids	13	Semi-Naive Sum Distance	44.44	453	113.8	334658	3.98
K medoids	15	Semi-Naive Sum Distance	0.0	492	152.8	312096	3.22
K medoids	17	Semi-Naive Sum Distance	22.22	405	109.4	413016	3.70
K medoids	19	Semi-Naive Sum Distance	0.0	563	170.1	584722	3.31
K medoids	1	SumDistance	100.0	0	2.0	0	0.0
K medoids	3	SumDistance	88.89	10	4.7	314	2.13
K medoids	5	SumDistance	88.89	56	17.1	5486	3.27
K medoids	7	SumDistance	88.89	22	13.1	3358	1.68
K medoids	9	SumDistance	88.89	20	14.1	2960	1.42
K medoids	11	SumDistance	88.89	2	2.0	0	1.0
K medoids	13	SumDistance	88.89	8	2.9	0	2.76
K medoids	15	SumDistance	88.89	6	2.3	0	2.61
K medoids	17	SumDistance	88.89	10	2.7	0	3.70
K medoids	19	SumDistance	88.89	12	2.0	0	6.0

Figure 15: Output for StackAr

Column 1: the clustering algorithm applied.
Column 2: the number of clusters generated.
Column 3: the distance metric used.
Column 4: the percentage of mutants detected using algorithm 4.
Column 5: the number of additional distinct invariants generated.
Column 6: the average number of false positive invariants generated.
Column 7: the number of implication invariants generated.
Column 8: the aggregate quality metric as described in section 4.7.5.

What is interesting in the above table is that the ability of the statistical kill algorithm appears to be highly correlated with the quality metric I selected earlier. Furthermore the downwards trend of the number of false positives above 9 clusters, when clustering with a semi-naïve distance function was also remarkable. Very noticeable is the difference between random splitting, and actual clustering functions.

6 Discussion

Daikon with clustering showed a clear increase in the number of valid invariants inferred from a program. Furthermore optimizing my distance metric led to a massive decrease in the number of false positives generated, and an increase in accuracy of mutant detection. Unlike Dodoo, Lin, and Ernst[7] who found only a small difference between random clustering and their clustering functions, the difference between k-medoids and random clustering for me is massive.

This difference between clustering and random splitting I noticed might be in large part due to the measurement procedure I applied, which is completely different from that of Dodoo, Lin, and Ernst[7]. Where they looked at the number of additional invariants a human would have to infer to verify the invariant set generated by Daikon, I looked at the invariants I could gain by clustering Daikon inputs.

Random clustering performs rather horribly, always coming dead last in my tests. The biggest improvements over random clustering for QueueAr were found with a Naive distance function, and K=3. Where there were 19.4 false positives per cluster on average, however the number of generated invariants increased by 579.

For StackAr the most impressive results were produced by applying k-medoids with a naive sum distance function, and k=5 . My technique for statistical kills performed amicably, returning a realistic number of killed runs for each and every clustering run. For all of the base cases with k=1 the number of detected mutants using my statistical kill algorithm equaled the number of mutations that actually had an effect on the program's execution path.

7 Conclusion

In this thesis I examined methods of clustering Daikon log files, and their advantages and disadvantages. I evaluated a clustering function that is less sensitive to outliers than k-means, namely k-medoids, with success. Furthermore trough

tweaking the distance function large gains in accuracy were had. Thus to the first research question, Can I determine a better clustering function for Daikon log files, the answer is a resounding yes.

The answer to the second research question, namely if it is possible to determine a better metric for the quality of invariants gained by clustering trace files, is not so clear. Plain Daikon without clustering invalidated enough invariants to allow identification of all mutants created by MuJava. Due to my choice of performance metric I can however say that the number of mutants left undetected as a result of clustering is smaller than the number of mutants potentially detected as a result of clustering Daikon's input.

To the third research question, does clustering alone, without detecting implication invariants, have merit, the answer is yes. Valid, new invariants were gained through the application of clustering operations to log files. Tough, as it turns out, even a small change in a program often invalidates a huge number of invariants. So many in fact that the notion of statistical kills, or simply determining if a run was bad by the number of invariants invalidated, works well. This shows that there is in practice no need to manually remove all false positive invariants.

While evaluation of clustering functions is rather memory and runtime expensive, once a good clustering function is found using it is quick. For my experiment I used a computer running Gentoo, with two Opteron 6272 processors, each having 16 cores, and 64Gb ram. Generating output for the biggest set of logs, namely that of QueueAr takes roughly a day. But when a good clustering function is chosen, it is quick to cluster a log file, and then evaluate each cluster in parallel, taking only a modest amount of extra time over a regular Daikon run.

8 Contributions

I verified the work of Dodoo, Lin, and Ernst[7] insofar that clustering does gain us useful invariants, and therefore information about a program. Through a smart choice of the clustering algorithm I examined, I managed to make clustering for Daikon more robust to outliers. Furthermore this clustering algorithm allows for easy evaluation of the effect of different distance functions on clustering log files. I examined the invariants invalidated by mutation testing, instead of using ESC/Java as a benchmark, and found that the set of invariants normally inferred by Daikon is rather complete. This quality of Daikon's output allowed me to create an easy algorithm for determining if a program is defective, namely the statistical kill algorithm.

9 Future work

The obvious next step to add a complete implementation of Dodoo, Lin, and Ernst[7] to my framework, as to evaluate the precision and recall metrics for implication invariants. Furthermore since the number of clustering algorithms tested in this thesis is extremely limited, and the distance metrics can certainly be improved, two other avenues of research would be developing more sensible distance metrics and clustering functions for program log files.

The notion of a statistical kill was described, however this notion could be improved upon by for instance cross-validating the generated invariant set. To verify the quality of a clustering metric, some notion of the information gain from invariants inferred from a clustering needs to be developed.

Since my framework does not ensure complete code coverage, it would be beneficial to include a testing tool, such as T3[16] in my software, and to evaluate the results of guaranteeing coverage. It would also be interesting to see the effect of clustering on other correctness properties, such as algebraic specifications[8]. Furthermore it would be interesting to see how this research applies to different programming languages for which mutation testing is convenient, such as JavaScript. [15].

Due to the problems I've experienced with multi-threading, it would certainly be interesting to see how much speed could be gained by optimizing Daikon for multi-threaded use. Certainly invariants for different entry and exit pairs could be computed in parallel. This would help Daikon scale to much bigger programs, and some measure of thread safety would certainly help future developers who want to link to Daikon in their own projects. With the price of machines capable of running a massive number of threads in parallel as low as they are today, massive performance gains could be achieved. If Java's networking libraries are used, even more massive log files could be processed by distributing the load over several machines.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[2] James H. Andrews. Testing using log file analysis: Tools, methods, and issues. In *In Proceedings of the 1998 International Conference on Automated Software Engineering (ASE'98*, pages 157–166. IEEE Computer Society, 1998.

[3] Dana Angluin. Computational learning theory: Survey and selected bibliography. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 351–369, New York, NY, USA, 1992. ACM.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[5] William Dickinson, David Leon, and Andy Podgurski. Pursuing failure: The distribution of program failures in a profile space. In *In Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 246–255. ACM Press, 2001.

[6] Nii Dodoo. Selecting predicates for conditional invariant detection using cluster analysis. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, September 2002.

[7] Nii Dodoo, Lee Lin, and Michael D. Ernst. Selecting, refining, and evaluating predicates for program analysis. Technical Report MIT-LCS-TR-914, MIT Laboratory for Computer Science, Cambridge, MA, July 21, 2003.

[8] A. Elyasov, I. S. W. B. Prasetya, and J. Hage. Guided algebraic specification mining for failure simplification. In *Testing Software and Systems*, pages 223–238. Springer, 2013.

[9] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[10] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 449–458, New York, NY, USA, 2000. ACM.

[11] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[12] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. *SIGPLAN Not.*, 37(5):234–245, May 2002.

[13] L. Kaufman and P.J. Rousseeuw. Clustering by means of medoids. In Y. Dodge, editor, *Statistical Data Analysis Based on the L1-Norm and Related Methods*, pages 405–416. North-Holland, 1987.

[14] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[15] I. S. W. B. Prasetya, A. Elyasov, A.and Middelkoop, and J. Hage. Fittest log format (version 1.1). *Department of Information and Computing Sciences, Utrecht University, Tech. Rep. UU-CS-2012-014*, 2012.

[16] I. S. W. B. Prasetya, T. E. J. Vos, and A. Baars. Trace-based reflexive testing of oo programs with t2. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 151–160. IEEE, 2008.

[17] Yu seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava : An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15:97–133, 2005.

[18] Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] Anand Yeolekar. Improving dynamic inference with variable dependence graph. In Borzoo Bonakdarpour and ScottA. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 301–306. Springer International Publishing, 2014.

[20] M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Security and Privacy*, 7(2):87–90, March 2009.