Automated platform testing using input generation and code coverage

Authors

Per Heed
Alexander Westrup

Examiner from Lund University, Faculty of Engineering

Per Runeson

Advisor from Sony Ericsson Mobile Communication

Erik André

Lund, 2009

Abstract

When using a Java platform it is important to test all aspects of it thoroughly. In this thesis an attempt to test the stability and overall function is presented.

The method is to test the platform by running a large number of Java applications on the platform and see if anything goes wrong. To stress the platform as much as possible it is important to not only start the application but to attempt to explore it as well. This makes sure that as much of the platform as possible is tested.

The focus of the thesis is to compare different ways to generate input to the application and try to find the most efficient method. Four different input generation methods are evaluated, random, adaptive and both with a constant startup sequence and compared to a manual reference.

To compare the input generators performance, ten games were selected and run a number of times. During these runs the code coverage of the games was calculated. Factorial design was applied to the code coverage values to determine if there were any statistically significant differences between the input generators.

The results show that the startup sequence give good code coverage values by quickly going through the menus and start the game. The adaptivity gives somewhat better code coverage in some games than simply random but requires code coverage to function which decreases the performance of the phone.

From the results it can be concluded that no input generator is the best for all situations. However in the test environment used in the thesis, the random with startup sequence was deemed the best because it reaches the second highest code coverage values and does not require to be run in debug mode, which causes a big performance loss.

Acknowledgments

We would like to thank Erik André for answering all our questions and supporting us through the process, Per Lilja for giving us the opportunity to do this thesis and everyone else at Sony Ericsson who helped us with technical questions.

Also a big thank you to Per Runeson for showing great interest in our work and giving valuable input throughout the thesis.

Table of contents

1 INTRODUCTION 1

2 BACKGROUND 2
2.1 MIDLETS, JAVA PROGRAMS ON A MOBILE PHONE 2
2.2 COMMERCIAL JAVA PROGRAMS 2
2.3 TESTING A JAVA PLATFORM 3
2.4 CREATING TEST CASES 3
2.4.1 BLACK BOX 3
2.4.2 WHITE BOX 4
2.5 EVALUATING TESTS 4
2.5.1 CODE COVERAGE 5
2.5.2 MUTATION TESTING 5
2.6 COMMUNICATION BETWEEN COMPUTER AND PHONE 6

3 PROBLEM DESCRIPTION 7
3.1 QUESTIONS 7
3.2 METHOD 8

4 DETAILED METHOD 9
4.1 INPUT GENERATORS 9
4.2 TERMINATION TIME TEST 9
4.3 INPUT GENERATOR PERFORMANCE TEST 10
4.4 MUTATION TESTING 10

5 SETUP 12
5.1 PROGRAM FLOW OVERVIEW 12
5.2 CHOOSING THE MIDLETS 12
5.3 PREPARING THE MIDLETS 14
5.4 THE PHONE 16
5.5 INPUT GENERATOR PARAMETERS 16

6 RESULTS 18
6.1 TERMINATION TEST 18
6.2 INPUT GENERATOR PERFORMANCE TEST 19
6.2.1 CODE COVERAGE GRAPHS 19
6.2.2 AVERAGE RUN TIME GRAPH 21
6.2.3 BOX PLOTS 21
6.2.4 GRAPH ANALYSIS 24
6.2.5 FACTORIAL DESIGN 25
6.3 MUTATION TESTING 28

7 DISCUSSION 29

8 CONCLUSIONS AND FUTURE WORK 31

REFERENCES 32

APPENDIX A: CODE COVERAGE END VALUES 34

1 Introduction

Sony Ericsson, as other companies, has a Java platform implemented in their mobile phones. This is used to run various applications with games as the most common type.

When a new version of a platform is released it is important to thoroughly test it since a lot of the phone's functionality requires a working Java platform. Today a lot of the testing is fully or partly automated, such as function testing and performance testing. But another important aspect is stability which is currently tested by manually running games and other applications. This is of course not ideal as it is expensive and rather slow; running applications takes a large amount of time. This is why it would be good to automate this process and be able to run long over night tests.

There is much literature written about how to test the functionality of Java platforms and to see that everything works according to specifications [5],[6]. However there is very little written about how to test quality attributes such as stability.

This report describes an approach to stress testing a Java platform on a mobile phone using third party applications running automatically. The focus is on how to run the applications in a way that tests the most of the Java platform in as little time as possible. To achieve this, different problems have been examined including input generation, variation across applications and coverage measures.

In this report it is first described what has been done in this field and a few key theories that are applied and what potential problems follow them, see Section 2. Then the main questions that this report will try to answer are presented followed by a detailed description of the approach taken to answer them, see Section 3 and Section 4. In Section 5 it is described more in detail about the specific test setup used. At the end of the report all results from the tests run are presented as well as a factorial design analysis, see Section 6. In Section 7 there is a discussion about the results and a recommendation is given to Sony Ericsson. Lastly, in Section 8, it is presented which conclusions can be made and where it would be interesting to continue on the work done in this thesis.

2 Background

2.1 MIDlets, Java programs on a mobile phone

Many mobile phones can today run third party applications, for example games. To do this in a smooth way, the phones have a Java platform able to run Java programs made in the Java Micro Edition (Java ME) version. For a more elaborate description of Java ME refer to Suns webpage [1]. This Java version has been specifically made with mobile devices in mind. As with other versions of Java, applications created will be able to run on every mobile phone that has implemented a Java ME runtime environment.

A Java ME environment is made up of two different parts. The first part contains the basic components to make a virtual machine possible and is called Connected Limited Device Configuration (CLDC) [2]. On top of the CLDC it is possible to add a profile which in the case for most mobile devices with displays is the Mobile Information Device Profile (MIDP) [3]. Applications that conform to the MIDP specification are called MIDlets.

Figure 1 shows the different layers of the phone. It starts with the MIDlet layer, second is a Java ME layer and third a native layer, down to the hardware. The purpose of the native layer is to connect the high level Java with the hardware.

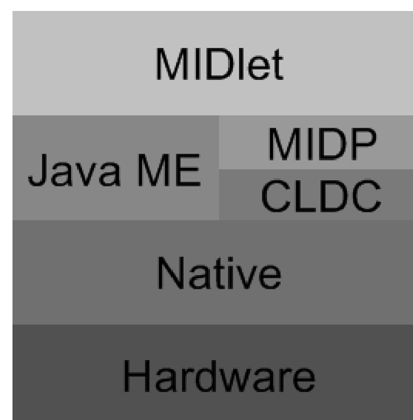

Figure 1: Layers of the phone.

2.2 Commercial Java programs

When compiling a Java program the compiler does not generate machine code; instead it generates bytecode that a virtual machine can understand. In the bytecode a lot of information can be preserved to enable debugging and code analysis, for example finding bottlenecks. The information preserved is for example references to source lines, unused variables and the same name of classes, methods and variables as in the source code. All this information makes it very easy to decompile a Java program, reverse engineer bytecode to source code.

If a Java program is commercial and is sold to a number of different customers it is often not wanted that anyone except the owner are able to decompile the program and do their own modifications and potentially call it their own.

To prevent decompiling of commercial Java programs there are two actions to be taken. First and most easily done is to compile the code without debug information; this will for example remove source line references and unused variables. Secondly an obfuscator can be used on the compiled code. An obfuscator takes bytecode and alters as much information as possible to reduce readability without altering the behaviour of the program. It can also change names of classes, methods and variables and the structure of the code in ways that are not accepted by the standard Java compiler but are accepted by the virtual machine. So if you try to decompile the obfuscated code you will get code that is not possible to recompile without some work. [4]

2.3 Testing a Java platform

There has already been work done on how to test different aspects of a Java platform on a mobile phone. A technique that is close to this thesis work is to create specific MIDlets which stress tests a certain function of a Java ME platform to make sure it works properly [5]. This technique is aimed at testing one function at the time whereas the method presented in this thesis is focused on testing the entire platform at once. However both methods use MIDlets to test functions of the layers below, see Figure 1. When studying this test technique one problem emerges, the problem of not knowing what is actually being tested, the test MIDlet or the platform.

There has also been work done in testing the performance of a Java ME platform by a similar approach and that is to produce benchmark MIDlets, for example JBenchmark [6] that focuses on graphics and gaming. These MIDlets use functions that are performance critical and do time measurements of how long a function or a sequence of functions take to execute. This approach is mainly usable to compare different platforms or compare versions of a platform to see if the performance has increased or decreased.

2.4 Creating test cases

When creating tests for a program it is quickly discovered that it is not possibly to test every possible use case, for example a method taking a 32-bit number as argument has 2^{32} possible inputs. Not being able to test everything means there is a need to choose wisely how to create the test cases. There are two different view points for test cases, black box [7](pp. 61-88) and white box [7](pp. 97-127).

2.4.1 Black box

When doing black box testing the tests are created in a manner where the actual structure of the tested software is ignored, only the input and output is considered. When constructing these types of tests a specification is used to find out what the possible inputs are and what the expected outputs should be. When selecting which of the possible inputs should be used for the tests there are a number of different methods. The easiest way is to randomly select a number of possible inputs; there are many thoughts on if this is a good or bad way to choose inputs, for example [8]. A more structured way to choose inputs is to create equivalence classes within the possible inputs. These classes are groups of inputs for which the same behaviour from the software is expected which means only one input from each class needs to be

tested. A method that is similar to equivalence classes is boundary value analysis. In this method the inputs that are in the boundary of the equivalence classes are tested as these values more often cause problems for the software than inputs in the middle of a class.

When using COTS (Commercial of-the-shelf) components, these need testing as well and most of the time the source code is not provided so this testing will always be black box testing. When testing COTS, it is often good to test the most commonly used features the most. How the program is used in a live environment is called its operational profile, this is created by watching the product being used or with knowledge about how the product is used. It has been proposed that using it to create test cases is an efficient way to get good reliability in the software by only running a rather small amount of tests [9].

2.4.2 White box

The difference between black box and white box testing is that in white box testing the internal structure of the program is known, that means that the source code is available. This has a lot of benefits and results in several new ways of testing it.

The goal with white box testing is to be certain that the internal components are working as they should. This can for example result in test cases being created to make sure all branches are executed. When testing with a white box approach it is needed to have some sort of framework that tells you which elements to focus on, what test data to choose and when you have tested enough. Such a framework is called test adequacy criteria.

Black box testing can be started as soon as something primitive is created as long as the input-output is the same. White box testing often starts a lot later in the process because of the close relationship with the actual source code. It is very important to notice that it is not a question of choosing one of the approaches when you start to test, but instead use both because they complement each other and can be used at different times.

There are some different test evaluation methods you can apply when testing with a white box approach, two of the main techniques are code coverage, see Section 2.5.1, and mutation testing, see Section 2.5.2.

2.5 Evaluating tests

To automatically test a program it is not enough to just write tests for it. It is also needed to make sure the tests can be trusted by evaluating how thoroughly the tests actually test the program. For example, all parts of the program need to be tested and all possible use scenarios should be tested. Since even small programs can be used in an almost endless number of ways it is not possible to test every possible scenario [10]. Instead there should be test cases that cover them good enough. To know if the test cases do cover them well enough there are a number of different techniques that can be used depending on the situation of which two are described below.

2.5.1 Code coverage

A common way to evaluate tests is to measure code coverage [11]. Code coverage means that the tests are run and the percentage of source code executed during the run is measured. There are many different elements in the code that can be covered, methods, source lines, branches or statements are some common elements. If the coverage percentage is high, it indicates that the test suite exercises most of the program but if it is low, there are parts of the program that are not tested thoroughly enough or not at all.

On the surface, code coverage seems to be all that is needed to evaluate the tests but many studies have shown that a good code coverage value does not mean the tests are good enough [10],[12]. This is due to the fact that just because some code has been run without it crashing does not mean it will not crash the next time it is run. There might be specific input values or states of a class that make the code crash.

Figure 2 shows the structure of a code snippet with an if-statement and two branches. Assume that the if-statement always fires the left branch in the executed test cases, then the code coverage when counting the branches will be 50%. A statement code coverage will reach 4/5, 80% with the same execution.

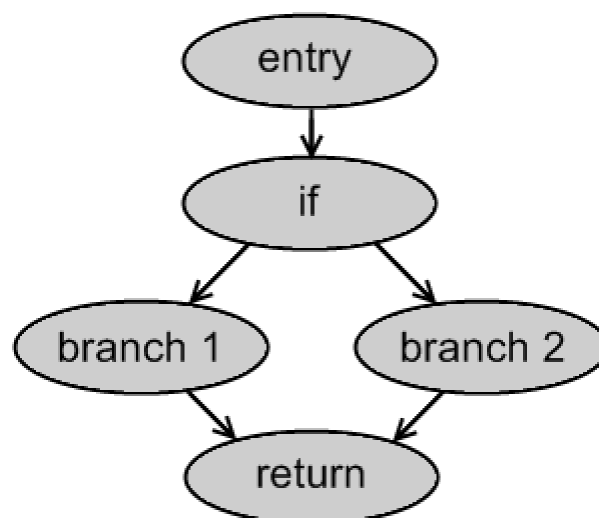

Figure 2: Code structure.

2.5.2 Mutation testing

Mutation testing is an evaluation method that injects faults in the source code, called mutants. These mutants can be anything from changed variables to changed operators, for example “>” to “<=”.

A large amount of variants of the source code is produced with one or more mutations in each variant. Then the test cases are run to see which mutations can be found and which can not. If all mutations are detected the test cases are good. If most of the mutations go by undetected, more or better test cases should be written. [7](pp. 116-118)

There are many things to think about when mutating code, for instance if the behaviour is actually changed, for example changing x/y to x*y when y=1 or y=-1 does not alter the behaviour. If there is a mutation that does not change the behaviour of the code the results will show it as if the tests failed to detect it, when in fact it is impossible to detect the mutation. This will give an incorrect rating of the test cases. It is also important to make sure the code compiles after the source code has been mutated since it is needed to run the code through the tests. There are some mutation tools which uses compiled code, for example Jumble [13].

2.6 Communication between computer and phone

Controlling a phone from a computer can be done in a few different ways but the easiest is done by connecting the phone to a computer with an USB cable. Then an Attention (AT) connection is established between the program running on the computer and the phone. When this is done AT commands can be sent to the phone [14], these can be anything from “Start MIDlet X” to “Press key 1”. When controlling a phone in this way it is important to realize that commands can be sent very fast from the computer but the phone might need a while to process the command. If commands are sent faster than they can be processed they will be put in a queue until the queue overflows.

To be able to access the phone’s file system to transfer and delete files for example, an Object Exchange (OBEX) connection is needed [15]. OBEX is a protocol used by many different portable devices to exchange data in a simple way.

When running a MIDlet on the phone it is done according to Figure 3. First the MIDlet is transferred using the OBEX connection ①. Once the MIDlet is on the phone it is installed and started by sending AT-commands to the phone ②. When the MIDlet is running, input can be given with AT-commands simulating key presses, also debug info is sent from the phone to the computer ③.

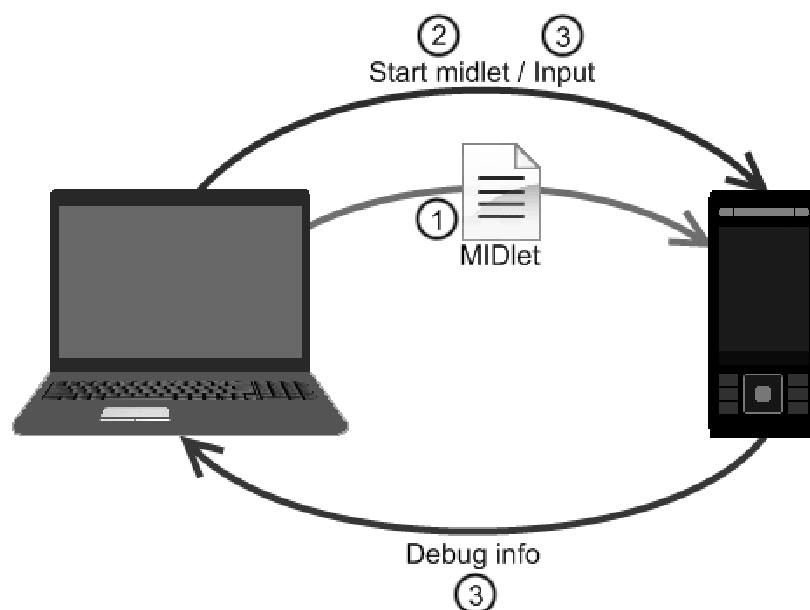

Figure 3: Communication between computer and phone.

3 Problem description

3.1 Questions

The problems involved in this master thesis are about how to use third party applications with no access to source code to test the stability of a platform. The testing should be as effective as possible and test as much of the platform as possible in as little time as possible.

There are three main issues to investigate.

RQ1. How can we give good input, in the form of key presses, to the applications running on the platform?

Is it good enough to give random input or do we need to implement some sort of intelligence? An option that would probably be successful is to write specific instructions on how to control each application. In this thesis we want to enable controlling applications in a general way and will not investigate this option.

RQ2. How do we know when to quit an application?

Each application only has a certain amount of code and can only exercise the platform to a certain extent. After a while it will be a waste of time to try to find more to explore as it will be too little new things found each second.

RQ3. Is code coverage a good way to measure fault detection capabilities in this specific case?

As discussed in 2.5.1 code coverage might not be a very good way to measure how good a test suite is. It was decided to use code coverage to measure how good the input generators are, so we need to investigate what a good code coverage value actually means in this particular case.

3.2 Method

To answer the questions in Section 3.1 the following steps are taken.

1. Develop a prototype test tool. An overview of the program can be seen in Section 5.1.
2. Select a number of commercial MIDlet games with as different control systems as possible. The criteria used can be found in Section 5.2.
3. Develop a set of input generators with different characteristics. The input generators are described in detail in Section 4.1.
4. Run tests to see when the increase of code coverage is very low or stops to determine when to terminate the application. The test is described in Section 4.2.
5. Run tests to determine the performance, described in Table 1, for each input generator. The test is described in Section 4.3.
6. Run tests to validate the use of code coverage by comparing code coverage with mutation testing. The test is described in Section 4.4.

When evaluating which input generator should be used there will be a number of criteria used described in Table 1. Since it is also important to see which benefits and drawbacks that can be observed compared to the current manual solution some criteria that only differs between manual and automated input generation will be used in the evaluation as well, these are cost, scalability and applicability.

Table 1: Description of the evaluation criteria.

Criteria	Description
Performance	Average code coverage value an input generator is able to reach
Portability	The amount of work needed to make the input generator work with a different phone and possibly different types of games.
Run speed	The input generation method might require additional information from the phone that will decrease the performance of the phone and thus reduce the run speed of the application.
Cost	Cost to run a test.
Scalability	How the cost scales when running multiple tests at the same time.
Applicability	The type of applications the input generation method can handle.

4 Detailed method

4.1 Input generators

For the tests five different input generators were used, described below. Exact settings for the different input generators are described under Section 5.5.

Manual
This is not an automated input generator but instead a person running the same programs as the automated variants. The person tries to achieve as high code coverage by exploring as much of the MIDlet as possible. This approach will of course not be used in the final solution but will be used as a reference value to what is used at the moment.

Random
This input generator looks at all the keys available and selects the next key to push randomly.

Startup random
This input generator has a predefined startup sequence that is used to try to get the game to start by clicking the specific keys defined in Section 5.5. After the startup sequence is done the input generation is performed in the same way as for the random input generator.

Adaptive
This input generator is programmed to have some adaptive behaviour, based on feedback from the debug information. It does this by having a number of predefined key sets where a key set is a subset of all the keys to choose from. When the input generator selects the next key to push, it selects a random key from within the current key set. When the program is running the generator observes code coverage change and if it is below a threshold it changes key set. Key sets also have a predefined minimum number of key presses before the generator should change key set to prevent the input generator from changing key sets too often. A maximum value is also defined to prevent unwanted behaviour.

Startup adaptive
This input generator works in the same way as the adaptive input generator but has a predefined startup sequence. The startup sequence is exactly the same as the startup random input generator uses.

4.2 Termination time test

The first test runs were made to determine how long it is reasonable to run the applications and still get some increase in code coverage. The run time for each application was set to 20 minutes as it was thought to be long enough. All four input generators were run on all ten applications. This test corresponds to step 4 in Section 3.2. The result from this test is presented in Section 6.1.

4.3 Input generator performance test

This test was divided into two test batches. First all input generators were run on all applications to get a good overview of the performance of the input generators.
Having dedicated two weeks to the first test batch and wanting as much confidence as possible in the results, it was decided to run each input generator 20 times on each application. The time each application should be run was decided by the termination time test described in Section 4.2.

To see if there were any statistically significant difference between the different input generators the method of factorial design [16] was used on the end code coverage values, see Section 6.2.5.

After analysing the results from the first test batch, see Section 6.2.5, it was decided that there was a need to get more confident results on the difference between the two best performing input generators, startup adaptive and startup random. For this second test batch, one week was dedicated and on this time an additional 30 runs were done.

To have a reference value to the current practice with manual testing there were also four runs with manual input generation done for each application. The manual input generation was done by the authors.

This test corresponds to step 5 in Section 3.2. The results from this test are presented in Section 6.2.

4.4 Mutation testing

To validate if the code coverage values are a good measure of how good an input generator tests the platform, mutation testing was used on the application. This makes sure that the applications are explored in a good way and, in the end if it uses the application to its full potential to exercise the platform.

As before there was no access to the application source code and the mutations had to be done on bytecode level. To do mutation testing you need to know if the mutation was detected or not. The way this is normally done is by looking at the output, but the way it was done in this thesis was by looking at exceptions thrown by the MIDlet. First the application was run a number of times without any mutations to see which exceptions were to be expected and not caused by a mutation, then 50 variants of the application were created with one mutation in each. All mutated variants were run for the time decided in the termination test, described in Section 4.2, and the exceptions thrown were recorded.

The mutations were done on bytecode level and the possible mutations are described below.

Mutations of conditional instructions were done according to Table 2.

Table 2: Mutations of conditional instructions

Before mutation	After mutation
IFEQ	IFNE
IFNE	IFEQ
IFNULL	IFNONNULL
IFNONNULL	IFNULL

Mutations of calculation instructions for integer, long, float and double values were done according to Table 3.

Table 3: Mutations of calculation instructions

Before mutation	After mutation
Add (+)	Sub (-)
Sub (-)	Add (+)
Mul (*)	Div (/)
Div (/)	Mul (*)

This test corresponds to step 6 in Section 3.2. The results from this test are described in Section 6.3.

5 Setup

5.1 Program flow overview

For step 1 in Section 3.2 a test tool was developed and it consists of several distinguished states. The transitions between the different states are shown in Figure 4. To be able to give an overview of the tool the different states are described below.

- **Modify the MIDlet(s):** Prepares the MIDlets so it will be able to generate the information needed, see Section 5.3. For the mutation testing the mutations were done in this state as well.
- **Transfer, install and start MIDlet:** Makes the phone ready to run the MIDlet and start it.
- **Generate input:** Generate a new input to the phone by using an input generator, see Section 4.1.
- **Calculate Code Coverage:** Reads the received debug information from the phone and calculates code coverage values.
- **Save information:** Saves different kinds of information for report and re-creation purposes.
- **Stop, uninstall and delete MIDlet:** Stops the MIDlet on the phone and makes the phone go back into original state ready to receive a new MIDlet.
- **Print report:** Print different kinds of reports into text files.

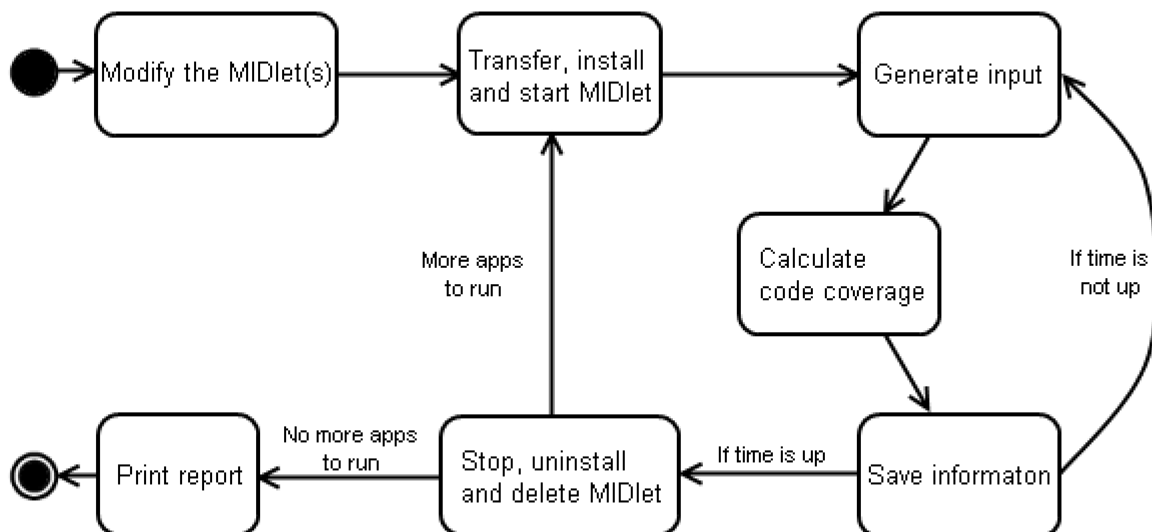

Figure 4: Program flow

5.2 Choosing the MIDlets

For step 2 in Section 3.2 it was decided only to use games for the tests, for two main reasons. Firstly games are by far the most commonly downloaded type of MIDlet. Secondly games are rather easy to control as they usually do not need to connect to the internet and log in, they do not need text input and the menus are rather similar.

Before choosing the games an important issue is to determine how many games that should be included in the test suite. Variables taken into account were the number of ways to control the game, different types of games and that the total test time should be reasonably long. It all landed in ten different games.

However there are a lot of different games to choose from, around 2000 available to Sony Ericsson for use in this thesis, so it needs to be narrowed down to a lot fewer. There are many ways to figure out which games should be run. Randomly selecting is the easiest way and could prove to be good enough but it was decided to use a different method to make sure games with different characteristics would be represented.

To select the specific games the ideal would be to analyze all 2000 games and write down their characteristics and then take decisions based on these. As this was not feasible due to the number of games, the games were chosen on more subjective grounds. We looked at what groups of games we could find and then chose one to represent each group. The games chosen are presented in Table 4, and below is also a short description of the characteristics of each game.

Table 4: List of the games chosen as well as a link to where the game is described in more detail (accessed 18 dec 2008).

Game	***Website***
3D Need For Drift	www.falconmobile.com/games.html
NHL 5 on 5 2007	www.1up.com/do/gameOverview?cId=3156264
Pro Golf 2007	www.gameloft.com/mobile-games/pro-golf-2007-feat-vijay-singh/
Karpov 2	www.microforuminternational.com/games_Board_karpov3dadvancedChess.html
Tetris Pop	www.eamobile.com/Web/mobile-game/tetris-pop
Indiana Jones and the Kingdom of the Crystal Skull	http://us.thqwireless.com/category.html?id=602
Cooking Mama	www.eamobile.com/Web/UK/en/mobile-games/cooking-mama
Virtua Fighter Mobile 3D	www.southend.se/games/index.php
Prehistoric Tribes	www.pocketgamer.co.uk/review.asp?c=6424&sec=0
AMA IQ Booster	www.pocketgamer.co.uk/r/Mobile/AMA+IQ+Booster/review.asp?c=3453

- 3D Need For Drift is a car game where the car auto accelerates and you only have to steer the car.
- NHL represents a classic hockey or football game where you can control all players and you pass and shoot the puck or ball. You can move your player in any direction.
- Pro Golf is a golf game where you aim with the joystick and then press a key twice to determine the power of the stroke.
- Many games goes under board game category, this is represented by the chess game Karpov 2.

- Tetris Pop is a classic Tetris game where the blocks fall automatically and you only have to rotate and move the block.
- Indiana Jones is an adventure, platform game where you move your player horizontally and can make him jump by pressing upwards.
- Cooking Mama represents games with many smaller games added into it and the controls are different depending on the mini game.
- Virtua Fighter is a 3D fighting game where you steer your player in three dimensions and try to defeat your opponent by using different attacks.
- Prehistoric Tribes represents strategy games where you control a lot of units and can move them anywhere and give them orders.
- IQ Booster is a sort of quiz game where you are given a question and different answer alternatives.

5.3 Preparing the MIDlets

In this thesis it was decided to use code coverage as the method to determine how good the tests perform. A problem to calculate code coverage on commercial Java programs without access to the source code is that there is no debug information available. Normally when you test Java programs you compile them with a debug flag, which adds line number instructions to the bytecode that are used as reference to source code lines. The tool used to calculate code coverage in this thesis needs these instructions to work.

This led to an extension of the test tool to be able to inject bytecode into class files inside a jar file. As it is not possible to know which bytecode lines corresponds to a source code line it was decided to measure bytecode coverage rather than source code coverage. When the MIDlet is running on the phone, the Java VM sends an event to the test tool at every line number instruction. Because each event takes a while to prepare and send you want to have as few line number instructions as possible while still able to get an accurate result.

The bytecode injection is done according the following steps.

- A line number instruction with a unique number is added at the start of every possible path the program can take.
- The weight of the line numbers is calculated by counting the number of bytecode instructions the program is certain to execute after each line number instruction.
- All the weights are saved in a file along with the total amount of line number instructions and the total amount of bytecode instructions in the program.
- When the program is run it identifies which line number instruction was run and looks up its weight in the file.
- The weights of the line number instructions run so far are summed up and divided with the total number of bytecode instructions to get the current bytecode coverage.

An example of how the injection works is shown in Figure 5. The method "example" is injected with a label and a line number at the start of the method, a line number after the existing label and a label and line number after the if instruction. The code has been modified to be a good example and does not have any useful functionality. Injected rows are marked with a "!" at the start.

Original bytecode	Bytecode after injection

<pre>// access flags 17
public final example()V
 FRAME FULL [] []
  ALOAD 0
  GETFIELD h.f : Z
  IFNE L0
  RETURN
 L0 (6)
  RETURN
  MAXSTACK = 3
  MAXLOCALS = 3</pre>
	<pre> // access flags 17
	  public final example()V
	! L0 (0)
	! LINENUMBER 1 L0
	   FRAME FULL [] []
	    ALOAD 0
	    GETFIELD h.f : Z
	    IFNE L2
	! L1 (5)
	! LINENUMBER 2 L1
	    RETURN
	   L2 (6)
	! LINENUMBER 3 L2
	    RETURN
	    MAXSTACK = 3
	    MAXLOCALS = 3</pre>

Figure 5: Bytecode injection example.

The total number of bytecode instructions for his method is six (6) and the weights for the injected line number instructions are listed in Table 5.

Table 5: Line number weights for the method "example".

Line number	Weight
1	4
2	1
3	1

If line numbers 1 and 3 have been run the code coverage value would be:

$$(4+1)/6 = 0.8\overline{333}$$

The code coverage values presented in Section 6 are not actual source code coverage but bytecode coverage of the generated source line references. This measure does not take into account if an exception is thrown in the middle of a statement and always assumes it was successfully run.

5.4 The phone

All tests are done on physical phones of the model Sony Ericsson C905, shown in Figure 6. The phone used has the following key setup:

- Soft key 1 and Soft key 2.
- Joystick up, down, left, right and middle key.
- C-key.
- Numpad: 0-9, * and #.
- Green, red and switch window key.
- Volume up and down.
- Camera keys.

Figure 6: Key mapping on the C905

5.5 Input generator parameters

All input generators have access to all keys except the green, red, switch window and camera keys. These are not used since they can not (to our knowledge) be used by an application.

The time between each key press is 5 seconds. This time corresponds to approximately one key press every second when running in normal mode, but the MIDlets are run in debug mode, which makes the phone send events when certain bytecode instructions are encountered. Debug mode reduces performance and the time set is to compensate for it.

To allow the MIDlet to fully load before the generator starts pushing keys there is a 20 second pause after sending the start command to the phone.

Key sets
Each key set has specific keys available and also a minimum number of times it must press these keys before it is allowed to change to another key set. There is also a maximum amount of times allowed to press keys from the same key set. When it is time to change to another key set the generator pick one of the other two at random. The key sets used are described in Table 6.

Table 6: Properties for the key sets

	Joystick	Soft keys	Numpad
Keys	Joystick left, right, up, down and middle key	Soft key 1 and Soft key 2	0-9, *, #
Min presses	10	2	5
Max presses	50	2	30

The code coverage threshold for changing key set is set to 0.2% increase over 20 seconds.

Startup sequence
The startup sequence is a static sequence that is the same for every game. The sequence is derived from our best knowledge with the purpose of getting through the menus often found in MIDlet games and gets the game started, and it is defined as:

1. Soft key 1
2. Joystick middle key
3. Joystick middle key
4. Soft key 1
5. Numpad 5
6. Joystick middle key
7. Soft key 1
8. Joystick middle key
9. Joystick middle key

6 Results

6.1 Termination test

In the termination test described in Section 4.2, 20 minute runs were done on all games and the average of all these runs is shown in Figure 7.
When looking at the results the increase in code coverage after 10 minutes is very slow. To have a safety buffer it was decided to set the time for each run in the input generator performance test to 15 minutes.

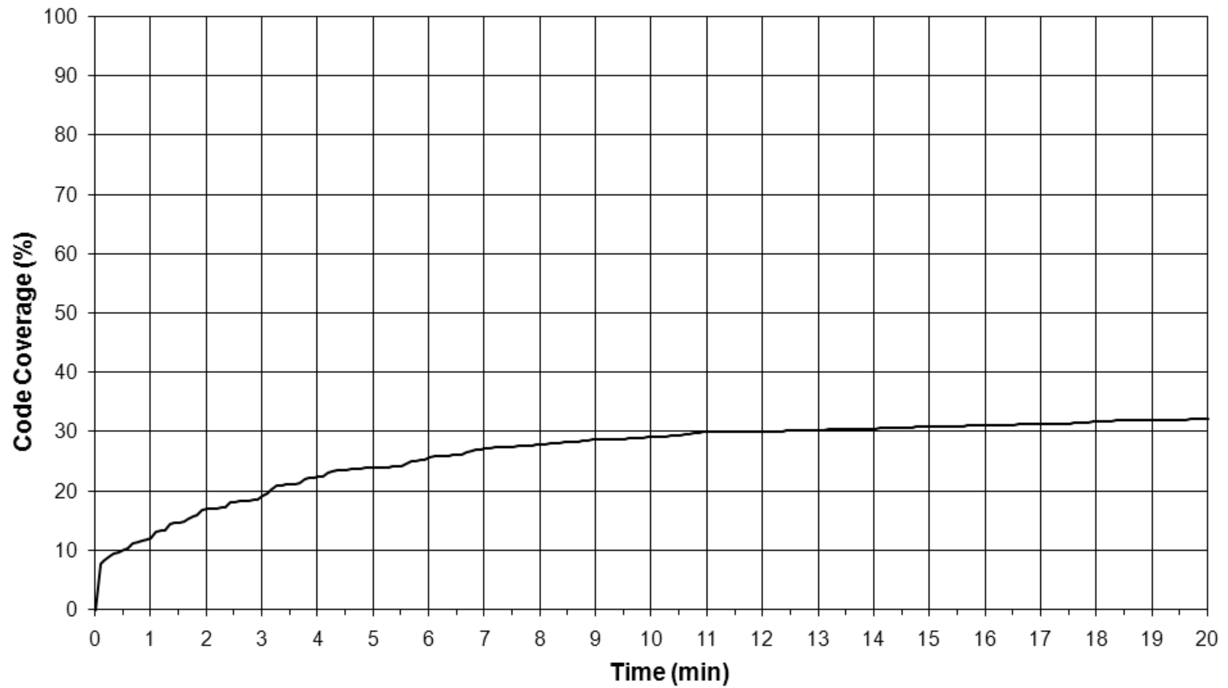

Figure 7: 20 min test, average of all apps on all generators

6.2 Input generator performance test

These results come from the test described in Section 4.3, 20 runs on each application for the random and adaptive input generators, 50 runs for the startup random and startup adaptive input generators and 4 runs for the manual input generation. Each run is set to last 15 minutes. All code coverage end values for the runs in this test suite can be found in Appendix A.

6.2.1 Code coverage graphs

Figure 8-Figure 10 shows the code coverage over time for all four input generators and the manual reference value. If an input generator manages to quit the application before the 15 minutes are over it is considered to keep the end value for the remainder of the time.

Figure 8 shows the average for the entire test. It can be seen that the two input generators with startup sequence performs a lot better than the other two in the beginning. When looking at the end of the graph, all input generators land in an interval of around ten percentage points. It can also be seen that the startup adaptive performs best, followed by startup random, adaptive on third place and random last. As seen in the graph the manual input generation performs much better than the automated input generators.
Figure 9 shows the normal case for the average code coverage for a single game. The two input generators with a startup sequence perform better than the ones without.
Figure 10 shows a special case where the two input generators using adaptivity performs better than the random input generators.

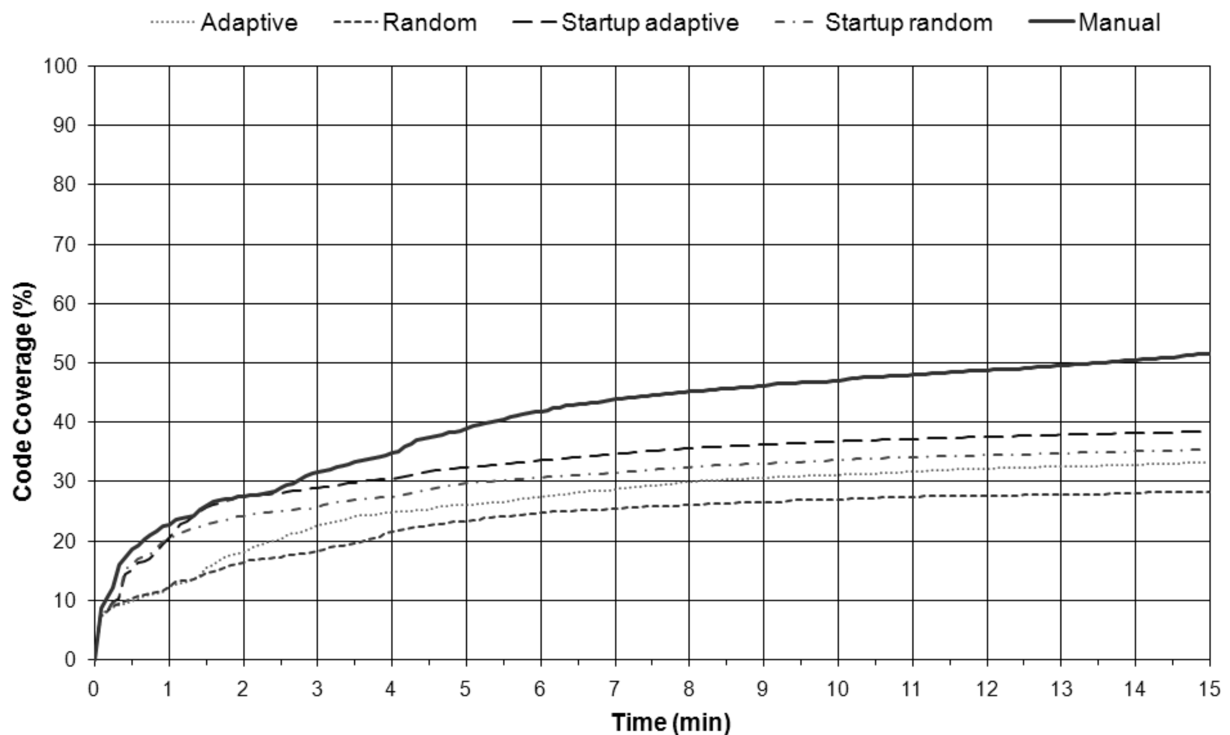

Figure 8: Average for all runs for all input generators.

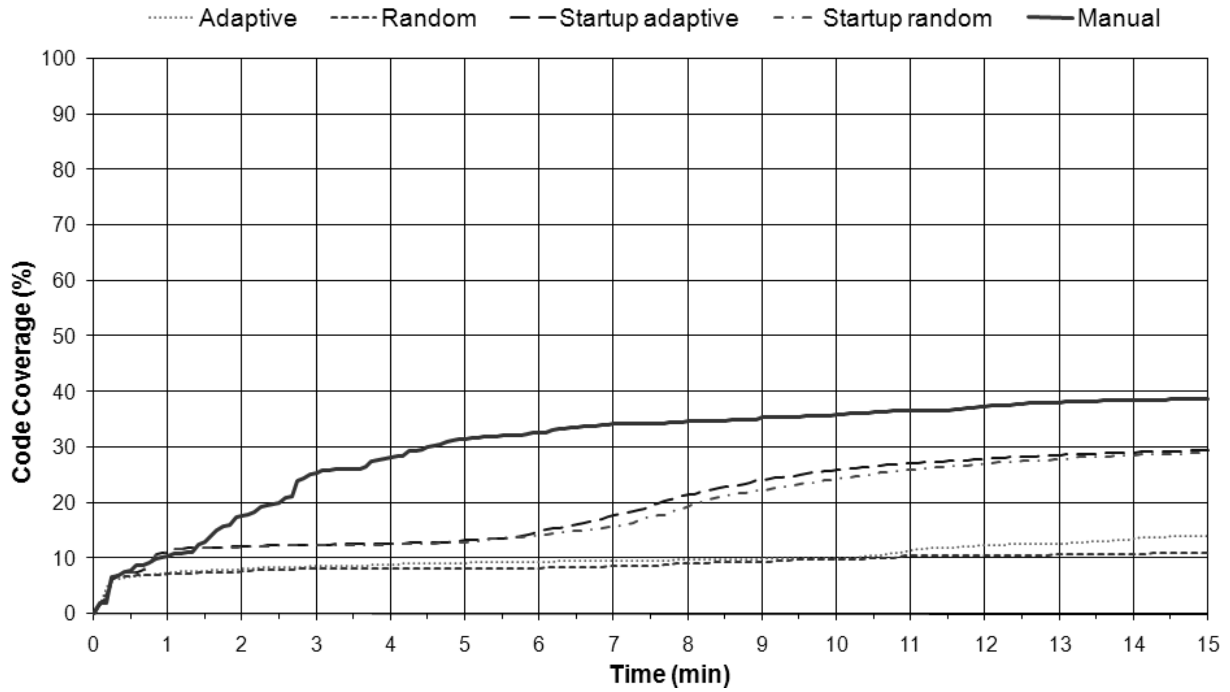

Figure 9: Average for Prehistoric Tribes

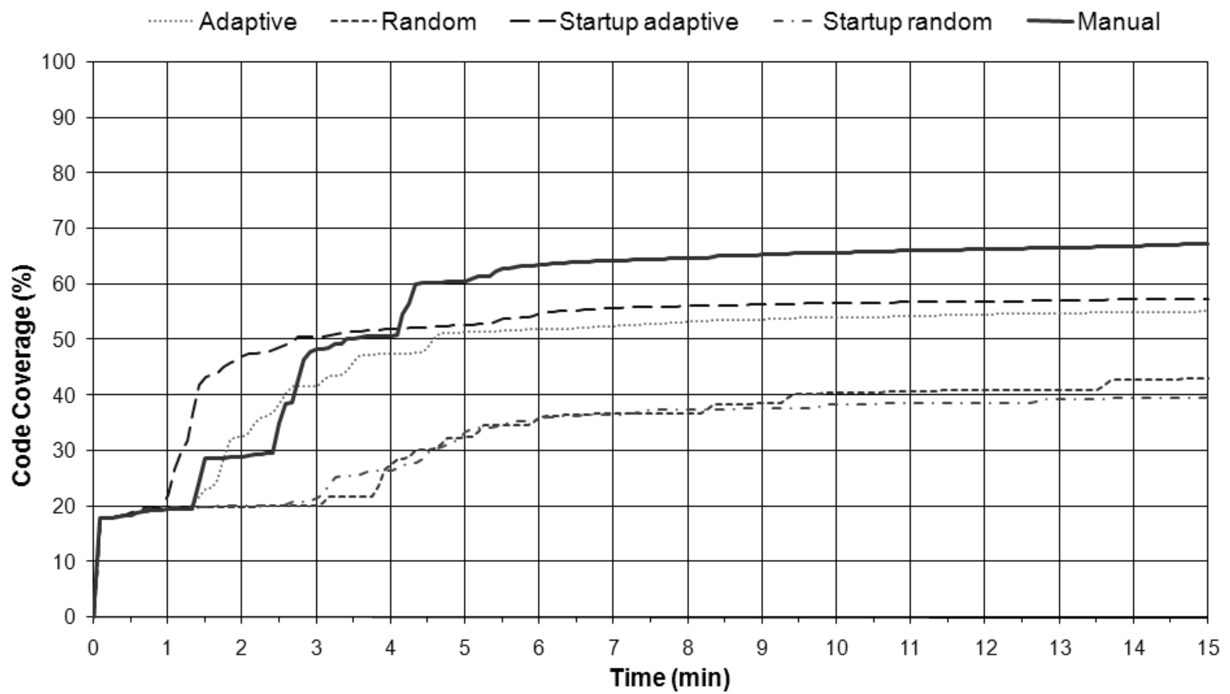

Figure 10: Average for Virtua Fighter Mobile 3D

6.2.2 Average run time graph

Figure 11 shows the average run time for each input generator for every game. All runs were set to last 15 minutes but the input generators sometimes managed to quit the game which results in shorter runs. For example it can be seen that the input generators often tends to quit Indiana Jones early but they almost always run NHL for the full 15 minutes.

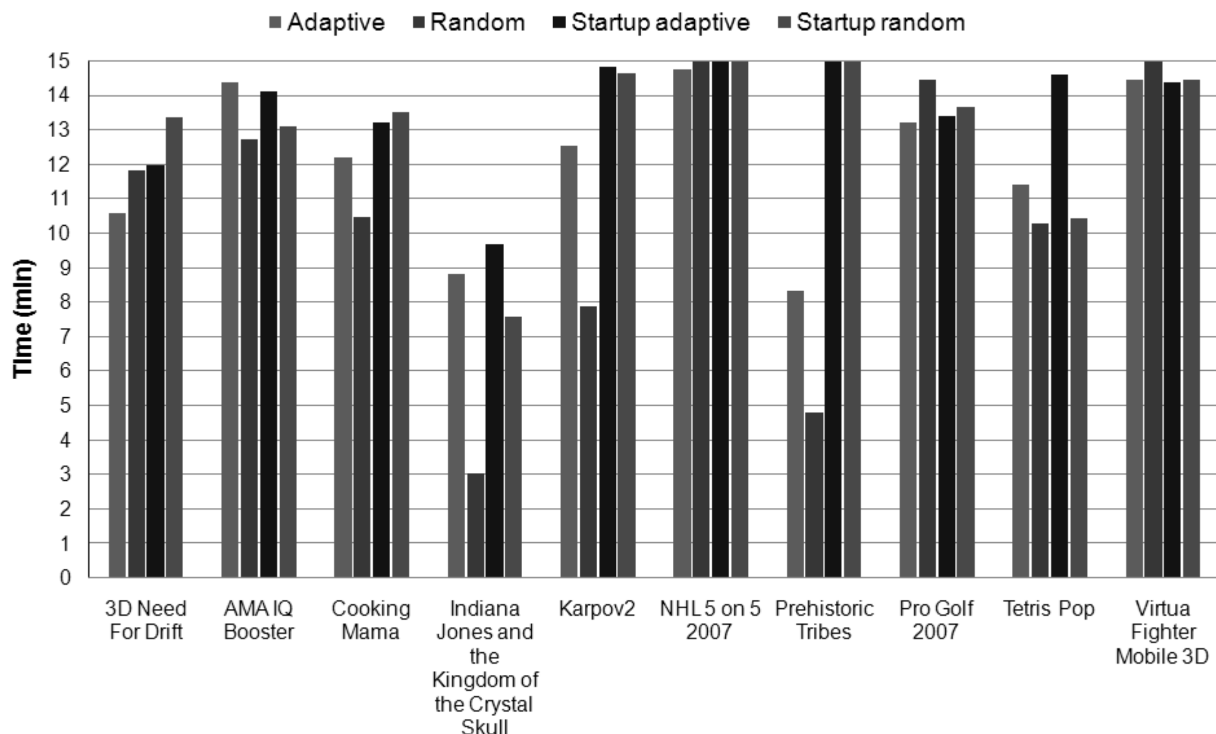

Figure 11: Average run time.

6.2.3 Box plots

Figure 12-Figure 21 shows the box plots [17] for each of the ten games tested. These box plots are created out of the tests end values found in Appendix A. The box plots shows the median value with the black line inside the box, the grey box shows 25% of the values above and below the median and the black lines outside the grey box show the maximum and minimum values. This means that if the box is big, the end results of the input generator have a big spread and if the box is small the input generator mostly gets to the same result.

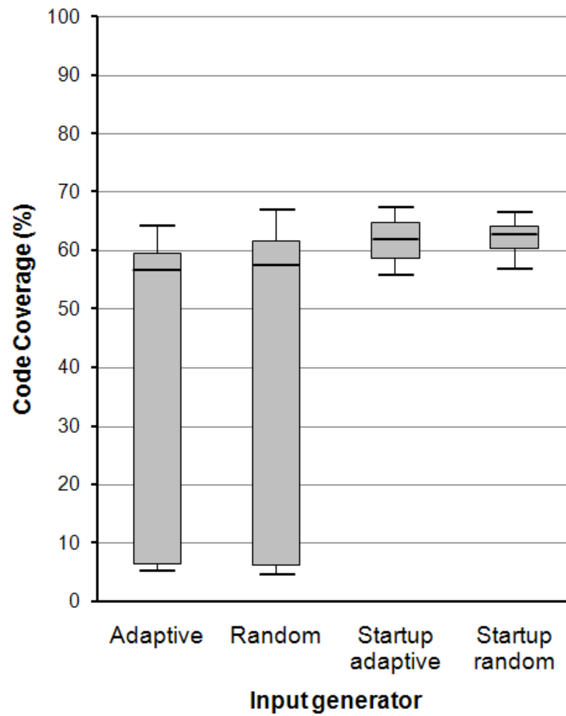

Figure 12: 3D Need For Drift, adaptive and random has a large spread and lower medians.

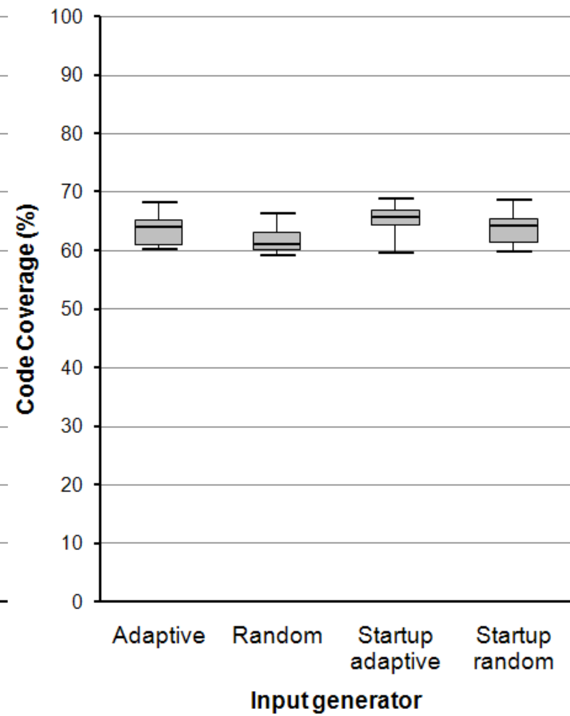

Figure 13: AMA IQ Booster, all four has small spread and similair median.

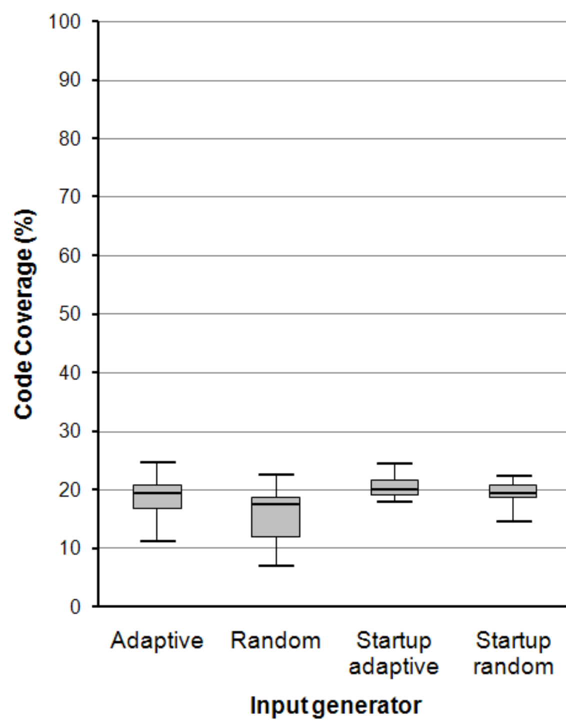

Figure 14: Cooking Mama, all four rather similar.

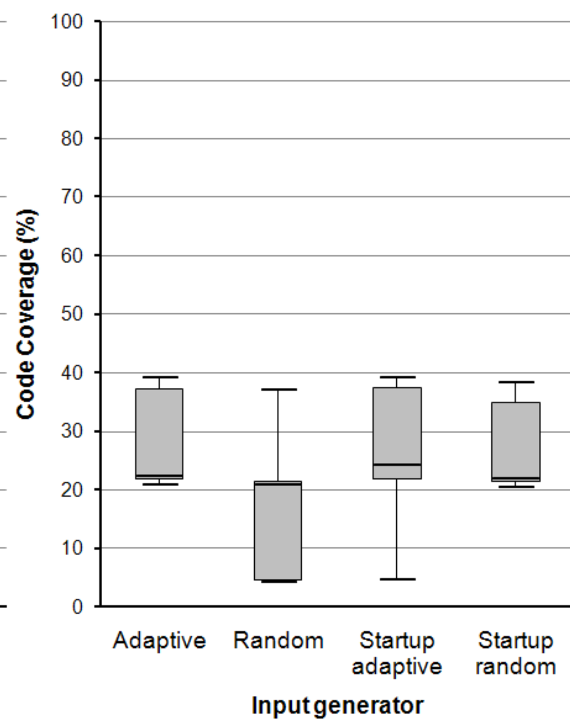

Figure 15: Indiana Jones and the Kingdom of the Crystal Skull, all four has similar medians, random has many runs with lower end values

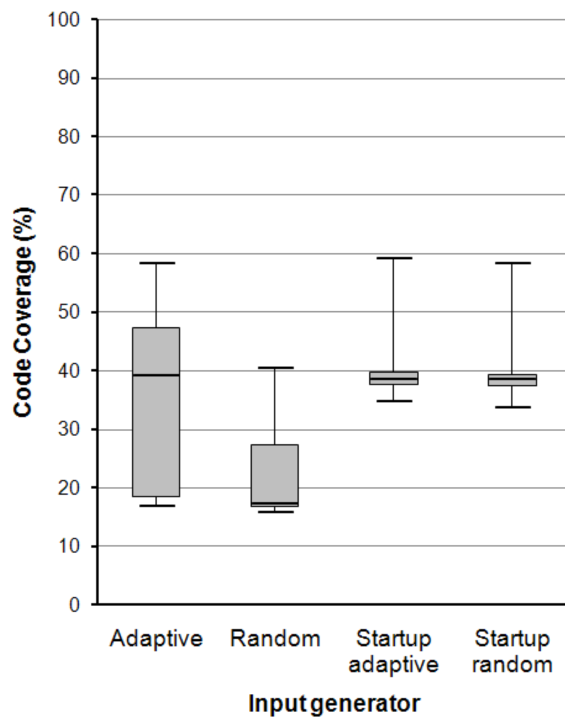

Figure 16: Karpov 2, random is not as good as the others.

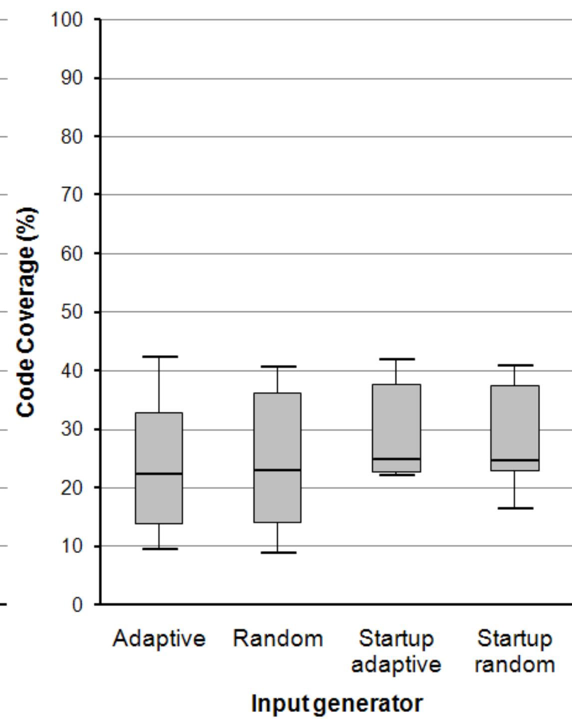

Figure 17: NHL 5 on 5 2007, all four are similar.

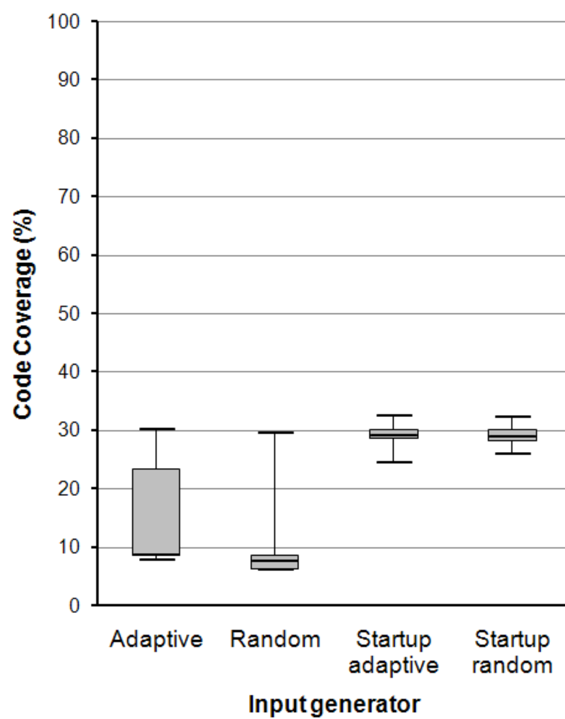

Figure 18: Prehistoric Tribes, both with startup are much better than the two without.

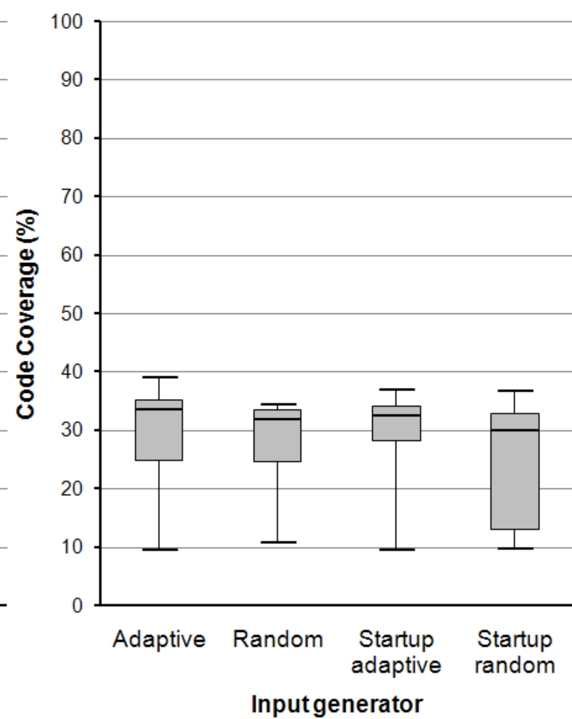

Figure 19: Pro Golf 2007, all four are similar.

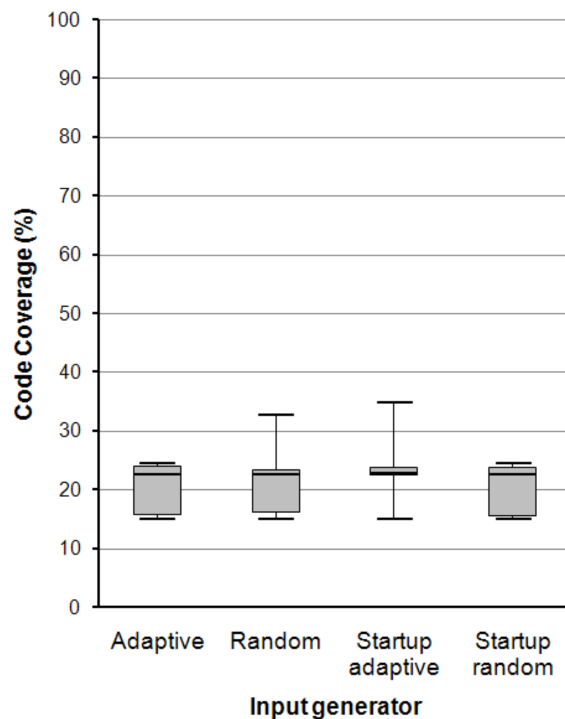

Figure 20: Tetris Pop, all four are similar but the startup adaptive has a very small deviation.

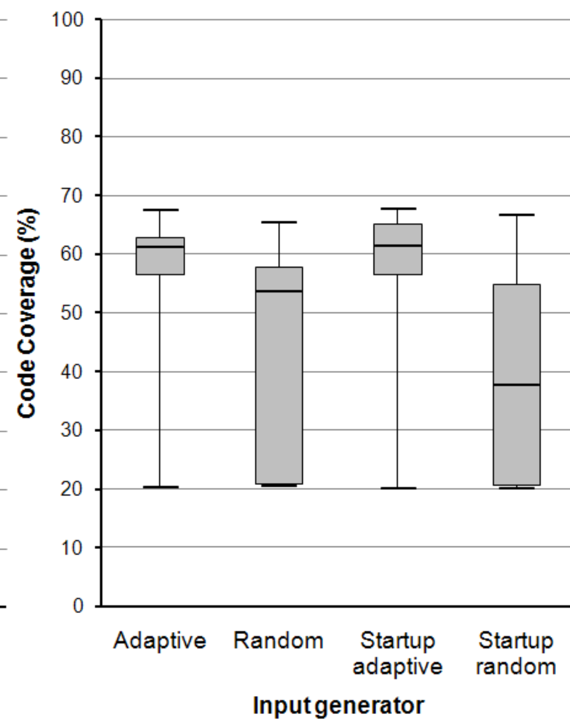

Figure 21: Virtua Fighter Mobile 3D, the two with adaptive behaviour are much better than the two random.

6.2.4 Graph analysis

From the graphs it can be seen that there are different reasons for low average code coverage values.

In Prehistoric Tribes for example the reason the adaptive and random input generators give low average code coverage, see Figure 9 and Figure 18, is because the average run time is low, see Figure 11. Low average run time is caused by poor input generation that quits the application early rather than explores the application as much as possible.

Another reason for low code coverage values is when the input generator manages to keep the application running but is unable to explore it in a good way. For example this is the case for startup random and random input generators in Virtua Fighter, see Figure 10, Figure 11 and Figure 21.

6.2.5 Factorial design

Doing factorial design [16] over all the end values from the input generator performance test, described in Section 4.3, gives the following results. The actual calculations were made automatically and only the results are presented here.

First the calculations were made on the initial 20 runs with each of the four automated input generators, the values can be found in Appendix A. The results from these calculations are presented in Table 7.

Table 7: All four input generators, ten applications and 20 runs on each application.

Source of Variation	Sum of Squares	Degrees of Freedom	Mean Square	F_0	P-Value
Application	171224.74705	9	19024.97189	178.52880	<< 0.01
Input Generator	11343.00636	3	3781.00212	35.48062	< 0.01
Interaction	14650.13915	27	542.59775	5.09169	< 0.01
Error	80989.61259	760	106.56528		
Total	278207.50515	799			

Lookup in the F-table gives:

$F_{0.01,9,760} = 2.41$
$F_{0.01,3,760} = 3.78$
$F_{0.01,27,760} = 1.79$

As all F_0 values are larger than the F-values gotten from the F-table, it can be concluded that there are, with more than 99% probability, significant differences between all three factors. This means that the code coverage values depend on which applications are run, which input generator is used and also the combination of application and input generator. Further calculations needs to be made to determine in what way they make an impact. Since the interaction is significant there is a need to compare the input generators for each application individually to get a proper result.

The comparison between all the input generators is presented in Table 8. The values are the average code coverage value of the first input generator for the specific application subtracted by the average code coverage value of the second input generator. If the absolute value of the result is above a threshold there is, with 95% probability, a significant difference. The threshold is:

$T_{0.05} = 8.37914$

Table 8: Comparison of all input generators over all applications. Yellow colour means no significant difference, blue colour means the input generator presented on top is significantly better and orange colour means the input generator presented on the bottom is significantly better.

Games \ Input generators	Startup adaptive vs Startup random	Startup adaptive vs Adaptive	Startup adaptive vs Random	Startup random vs Adaptive	Startup random vs Random	Adaptive vs Random
3D Need for Drift	-0.79498	18.48753	20.0617	19.28251	20.85668	1.57418
AMA IQ Booster	1.54873	2.44268	4.02541	0.89395	2.47668	1.58273
Cooking Mama	0.96042	2.27392	4.79169	1.3135	3.83127	2.51777
Indiana Jones …	3.07457	1.9579	14.55424	-1.11666	11.47968	12.59634
Karpov 2	1.66703	4.13777	18.64421	2.47074	16.97717	14.50643
NHL 5 on 5 2007	2.36675	6.69706	6.1166	4.33032	3.74985	-0.58046
Prehistoric Tribes	0.47035	15.36457	18.61958	14.89422	18.14923	3.255
Pro Golf 2007	4.5725	0.06999	0.89066	-4.50252	-3.68184	0.82067
Tetris Pop	1.20505	2.97884	2.81123	1.77379	1.60618	-0.16761
Virtua Fighter Mobile 3D	16.00768	0.32459	12.60406	-15.68309	-3.40362	12.27947

No difference | First is better | Second is better

In Table 8 four different cases can be observed.

Case 1: For 3D Need For Drift and Prehistoric Tribes the significant difference is between the two input generators with the startup sequence and the two without.

Case 2: For Indiana Jones and Karpov 2 the random input generator is significantly worse than the other three input generators.

Case 3: For Virtua Fighter Mobile 3D the two input generators with adaptive behaviour are significantly better than the two random.

Case 4: For the rest of the games there is no significant difference between the input generators at all.

To be able to make a better distinction between the two highest performing input generation methods an additional 30 runs on each of the ten applications were made. The results from calculations with these additional runs are presented in Table 9.

Table 9: Two input generators with startup sequence, ten applications and 50 runs on each application.

Source of Variation	Sum of Squares	Degrees of Freedom	Mean Square	F_0	P-Value
Application	234694.25332	9	26077.13926	477.60789	<< 0.01
Input Generator	2347.12281	1	2347.12281	42.98801	< 0.01
Interaction	6320.37493	9	702.26388	12.86210	< 0.01
Error	53507.48355	980	54.59947		
Total	296869.23461	999			

Lookup in the F-table gives:

$F_{0.01,9,980} = 2.41$
$F_{0.01,1,980} = 6.63$
$F_{0.01,9,980} = 2.41$

As all F_0 values are larger than the F-values gotten from the F-table, it can be concluded that there are, with more than 99% probability, significant differences between all three factors. This means that the code coverage values depend on which applications are run, which input generator is used and also the combination of application and input generator. Further calculations needs to be made to determine in what way they make an impact. Since the interaction is significant there is a need to compare the input generators for each application individually to get a proper result.

The comparison between the two startup input generators is presented in Table 9. The values are the average code coverage value of the first input generator for the specific application subtracted by the average code coverage value of the second input generator. If the absolute value of the result is above a threshold there is, with 95% probability, a significant different. The threshold is:

$T_{0.05} = 2.89460$

Table 10: Comparison of two input generators over all applications. Yellow colour means no significant difference, blue colour means the input generator presented on top is significantly better and orange colour means the input generator presented on the bottom is significantly better. Notice that in this table startup adaptive always is better then startup random and therefore there are no orange markers.

Games \ Input generators	Startup adaptive vs Startup random
3D Need for Drift	-0.55839
AMA IQ Booster	1.70335
Cooking Mama	0.75501
Indiana Jones …	2.97028
Karpov 2	1.07657
NHL 5 on 5 2007	0.99872
Prehistoric Tribes	0.36425
Pro Golf 2007	2.41152
Tetris Pop	3.13913
Virtua Fighter Mobile 3D	17.7802

No difference   First is better   Second is better (no occurrence)

In Table 10 it can be seen that the startup adaptive performs better in three games and that there is no significant difference in the other seven. This means that the startup adaptive input generator is proved significantly better than startup random on two additional games compared to Table 8.

6.3 Mutation testing

These results come from the mutation testing described in Section 4.4. From the 50 variants made, only three can be definitely said to be found by the tests. Code coverage values achieved from the test runs are around 50-60 percent, which means at most 50-60 percent of the mutations can be found. Hence the conclusions that can be made from this test are none. It is too hard to determine if a mutation has been found simply by looking at exceptions when it is not known if the mutation can produce an exception. As seen in Section 4.4 normal calculations are mutated, this could mean an “Index out of bounds” exception will be produced but it could also mean that a character in the game moves to the left instead of to the right and since this does not throw any exception it will not be detected.

7 Discussion

To answer RQ1 in Section 3.1 the criteria presented in Section 3.2 needs to be discussed.

Looking at the data you can see that there is a small difference between all the input generators. Manual proves to give best performance by far, however the problem with manual is as stated before that it is very costly and does not scale up in a good way, double the test time and you also double the cost.

An important aspect when deciding which input generator to choose is what performance loss you get when running in debug mode to get code coverage calculations. Manual and random can be used without debug mode but adaptive input generation requires code coverage. In our case the debug mode reduced the performance in a significant way but this could very well differ depending on the specific implementation and amount of information gathered in the debug mode.

When looking at the automated input generators we can see that the startup sequence is the most important to get good code coverage value. The startup sequence has a big flaw however that does not show in the results of these tests but that was noticed when running other games. In one game tested the soft key usage was switched compared to all other games which for our startup sequence meant that the game was terminated as quickly as possible. This means that you have to check if the startup sequence works for your selected applications in the intended manner.

When grouping the two input generators with startup sequence and the ones without you can also see that the adaptive approach gives slightly higher code coverage values.

When looking at which input generator is easiest to port to different types of mobile devices different problems can be observed. The fully random input generator is easy to port, all you need to do is define all the keys you wish to be pressed and you are done. The startup sequence will require you to find out if there is a sequence that starts most applications and if so which that could be. When you move on to the adaptive input generation you also need to define the key sets.

To sum it all up Table 11 presents all the pros and cons with each input generation technique. For each criteria the position for each input generator is presented in parenthesis, manual has no position as it is only included as reference.

Table 11: Criteria table, position in parenthesis.

	Manual (reference)	Random	Startup random	Adaptive	Startup adaptive
Performance	Very good	Very poor (4)	Medium (2)	Poor (3)	Good (1)
Portability	Good	Good (1)	Medium (2)	Bad (3)	Very bad (4)
Run speed	Fast	Fast (1)	Fast (1)	Slow (3)	Slow (3)
Cost	Expensive	Cheap (1)	Cheap (1)	Cheap (1)	Cheap (1)
Scalability	Bad	Good (1)	Good (1)	Good (1)	Good (1)
Applicability	Everything	Games (1)	Games (1)	Games (1)	Games (1)

In this thesis we have taken a rather easy solution to RQ2 in Section 3.1, when to stop running an application. We said we always stop after 15 minutes. This could of course be done in more advanced ways such as looking at code coverage change and when it is below a certain threshold the application should be stopped. However if time is not extremely critical it should be enough to have a constant run time.

We have not been successful in finding an answer to RQ3 in Section 3.1 in this thesis since the mutation testing did not show what we had hoped.

Our recommendation for Sony Ericsson is to use the startup random input generator. This input generator provides good results on most applications tested and does not require the applications to be run in debug mode.

8 Conclusions and future work

Conclusions

As a conclusion to the questions presented in Section 3.1 you could say that for most games it is most important to make sure they start up in a good way to get good code coverage, in other words a good startup sequence is essential to receive good code coverage values.

Depending on the performance of your system you would want to choose either the random method or the adaptive. If the performance of the device drops a lot when running in debug mode it is probably not worth running adaptive, however if the performance stays more or less the same you should probably use the adaptive input generation.

Future work

If there is the possibility to do modifications on the platform in an easy manner it would be good to do the code coverage and mutation testing measurements on the platform code rather than the MIDlet code as in this thesis.

An area that would probably work well to improve the input generation would be to implement some sort of real AI, like Genetic Algorithms. This would save data from each run of a certain application and for each additional run it would learn how to run that application better and better. This approach was not included in our thesis as we did not have time to implement it.

Another area not covered in this thesis that would be interesting to explore is to run non-game applications with features like internet access and need for text input and see how input generation performs in such environments.

References

[1] http://java.sun.com/javame/index.jsp. [Accessed: Jan 19, 2009]

[2] http://java.sun.com/products/cldc/. [Accessed: Jan 19, 2009]

[3] http://java.sun.com/products/midp/. [Accessed: Jan 19, 2009]

[4] Collberg C., Thomborson C., Low D., “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs,” *In Principles of Programming Languages*, 1998, pp. 184-196.

[5] Mazlan, M.A., “Stress Test on J2ME Compatible Mobile Device,” *Innovations in Information Technology*, 2006, pp. 1-5.

[6] http://www.jbenchmark.com/index.jsp. [Accessed Jan 19, 2009]

[7] Burnstein Ilene, *Practical Software Testing: A Process-Oriented Approach.* Springer, 2003.

[8] Chen Tsong Yueh, Yu Yuen Tak, “On the expected number of failures detected by subdomain testing and random testing,” *IEEE Transactions on Software Engineering*, 1996, pp. 109-120.

[9] Musa J.D., “Operational profiles in software-reliability engineering,” *IEEE Software*, 1993, pp. 14-32.

[10] Elaine J. Weyuker, “How to judge testing progress,” *Information and Software Technology*, 2004, vol. 46, no. 5, pp. 323-328.

[11] Mark Fewster, Dorothy Graham, *Software test automation: Effective use of test execution tools*. Addison-Wesley Professional, 1999, p. 509.

[12] Fabio Del Frate, et al, “On the correlation between code coverage and software reliability,” *Sixth International Symposium on Software Reliability Engineering*, 1995, pp. 124-132.

[13] Irvine, S.A., et al, “Jumble Java Byte Code to Measure the Effectiveness of Unit Tests,” *Testing: Academic and Industrial Conference Practice and Research*, 2007, pp. 169-175.

[14] Sony Ericsson Mobile Communications AB, “Sony Ericsson AT Commands Online Reference,” November 2008. [Online]. Available: https://developer.sonyericsson.com/site/global/docstools/misc/p_misc.jsp [Accessed: Jan 19, 2009]

[15] Object Exchange Protocol (IrOBEX). ver. 1.4, Infrared Data Association (IrDA), Walnut Creek, CA [Online]. Available: http://www.irda.org [Accessed: Jan 19, 2009]

[16] Douglas C. Montgomery, *Design and Analysis of Experiments,* 5th Ed. New York: Wiley, 2001, pp. 170-183, 642-646.

[17] Blom G., et al, *Sannolikhetsteori och statistikteori med tillämpningar.* Studentlitteratur, 2005.

Appendix A: Code coverage end values

Below are all the end code coverage values from the input generator performance tests used in the calculations and graphs. For the two input generators with startup sequence there are 30 additional runs for each application and to show which values are used in the factorial design calculations those runs are presented under “Additional runs” in each table.

Manual

3D Need for drift			
71,08119	74,23033	76,18451	77,01941

AMA IQ Booster			
67,098	70,04352	73,50781	77,02067

Cooking Mama			
44,77504	44,87366	51,48673	51,87563

Indiana Jones and the Kingdom of the Crystal Skull			
38,81674	40,01268	40,23919	40,45965

Karpov 2			
58,23718	58,66781	62,51311	62,78916

NHL 5 on 5 2007			
21,04316	29,74848	34,62748	50,26211

Prehistoric Tribes			
36,39433	37,45374	40,20019	40,52483

Pro Golf 2007			
44,47576	46,34849	48,19574	48,57316

Tetris Pop			
30,78299	32,62182	32,70382	34,00219

Virtua Fighter Mobile 3D			
66,88321	66,9702	67,0137	68,33703

Random

3D Need for drift									
59.26738	62.47391	5.73993	6.56178	61.74859	4.59716	64.58203	49.64256	6.46264	4.62847
55.93822	53.75444	5.82864	67.03193	64.07326	5.41901	59.41348	61.61814	61.45638	59.13171

AMA IQ Booster									
64.95492	66.36939	59.29315	65.94777	60.19468	60.96021	60.02176	60.60465	61.71602	59.70312
61.0865	61.1001	59.95959	63.52102	60.51916	64.5197	59.71672	60.71151	63.06637	61.12342

Cooking Mama									
7.00197	14.63288	20.10532	22.58755	12.13018	18.66324	11.06955	7.0243	19.84667	17.53935
17.3142	11.20539	12.48	11.17562	18.55904	18.39716	14.65334	22.02188	18.57207	17.97477

Indiana Jones and the Kingdom of the Crystal Skull									
21.29741	22.65946	4.19787	4.49082	4.6086	4.49384	20.78099	4.55726	21.69002	37.10739
21.37896	4.52102	20.4669	21.91955	21.17359	4.6237	21.4756	21.31252	21.536	4.6237

Karpov 2									
16.23696	16.85806	17.27765	16.70899	19.14371	17.40739	16.82769	20.41904	24.75846	17.36046
40.49026	20.12367	39.41092	36.04041	16.60686	15.80633	35.67879	15.8643	34.56909	16.12378

NHL 5 on 5 2007									
25.49748	14.04819	39.83055	24.00212	10.92295	40.07307	36.5274	15.58591	14.17951	23.10405
35.09981	40.62166	36.43421	13.97723	35.92269	22.48769	22.74504	13.08764	9.72518	8.90972

Prehistoric Tribes									
6.31533	14.91072	7.42127	7.87036	6.31858	6.32832	29.5823	6.33157	7.67558	6.29802
6.37377	8.63868	8.03484	29.552	8.57591	27.51109	8.66898	6.33698	7.83032	7.34553

Pro Golf 2007									
34.04516	32.84604	31.70903	31.97974	33.54513	34.45124	31.87942	12.6266	34.01172	32.58329
30.98287	33.21231	33.39225	27.87757	13.22696	14.72705	12.41799	32.02752	33.81266	10.84942

Tetris Pop									
22.70333	22.77042	23.90105	32.6181	15.61514	15.08585	23.18043	32.81937	16.37303	23.91472
22.79154	22.76669	15.75926	16.09845	16.65507	24.10481	16.29849	15.46604	22.80397	22.63002

Virtua Fighter Mobile 3D									
65.3387	54.99075	20.96891	53.8784	56.75736	57.49954	53.98297	51.6315	20.48492	64.5956
58.25837	20.73478	20.57098	20.73108	20.79123	53.54063	20.73755	65.36924	20.78567	59.54192

Startup random

3D Need for drift									
65.57608	64.42549	65.13515	59.59351	61.44855	58.97255	61.50856	62.84179	59.28042	59.29347
63.95585	56.854	62.76351	60.0527	64.14632	62.87831	61.79034	58.57076	64.25851	63.15748
Additional runs									
63.91411	61.30766	64.24546	63.60363	65.16124	61.45899	63.36882	59.41348	64.63421	64.65769
63.23575	66.46838	64.39418	62.20518	61.02327	59.72135	65.17689	63.8828	61.76685	58.05156
62.75047	64.51419	61.85295	59.55698	62.46608	59.2752	60.89543	62.69829	59.46045	64.98643

AMA IQ Booster									
61.30411	67.98788	65.13756	65.94	60.14611	65.92834	65.00738	65.52227	60.08394	64.84612
60.35595	63.556	63.71143	65.31048	63.66286	64.70234	64.92384	64.21271	65.72628	66.55786
Additional runs									
64.26712	60.16554	66.65695	60.09365	65.76708	65.10647	61.44206	64.7004	63.95819	63.70172
62.21536	64.38175	65.3474	62.88373	61.07679	65.17059	66.40242	62.84293	60.14028	60.58716
64.06505	63.49188	64.98407	68.62517	65.68936	65.50284	61.61304	59.9285	60.5347	60.48807

Cooking Mama									
19.52477	19.7983	14.61241	18.92747	20.03833	18.08269	19.26984	21.77254	21.84511	19.44475
17.4612	19.49872	21.66276	21.61252	18.65208	20.27651	17.57471	18.56648	20.06066	20.89799
Additional runs									
14.79104	19.5043	21.33341	20.57609	19.03353	19.53779	22.28611	19.5043	18.75628	18.38041
17.76823	21.8693	20.36768	19.20472	20.01972	18.37297	18.44926	21.43947	18.27621	18.81582
19.61408	19.84295	18.19434	21.29991	19.77969	19.02237	21.74277	21.05244	21.22362	18.51066

Indiana Jones and the Kingdom of the Crystal Skull									
21.12225	21.84706	21.91049	20.5424	33.89104	22.51148	21.46352	37.08323	22.29705	22.75006
20.42764	37.7718	22.30611	38.11005	37.23424	35.05376	21.46654	21.4303	38.31239	20.97729
Additional runs									
21.16453	21.26721	23.86144	21.68398	37.22216	38.29729	21.11923	38.34561	37.75368	20.93501
36.1138	36.31614	21.21285	21.02863	22.25477	21.49674	21.39708	21.4756	21.43936	21.97995
21.36386	23.63192	20.85649	21.96787	21.44842	22.6957	37.80805	24.80068	34.58565	21.37594

Karpov 2									
38.63799	38.14664	38.49997	35.29785	34.39795	38.38403	38.6518	35.44968	39.1404	38.73737
39.60691	58.45249	38.39231	39.01618	38.5469	38.86987	53.4478	37.10319	39.09347	35.38343
Additional runs									
36.62563	39.14868	38.29846	39.34743	35.69536	38.38403	39.69525	39.23977	35.88583	37.99757
39.3916	39.29222	38.91128	38.35919	38.27362	53.76249	39.31983	36.17015	38.8119	34.06117
37.26605	34.65467	39.14868	38.65732	33.77408	39.46337	38.13559	39.38056	38.36195	55.46569

NHL 5 on 5 2007									
23.12841	40.02754	22.41885	38.39449	36.62378	25.35769	40.95526	23.01297	22.78952	22.27694
23.38999	24.93513	22.45274	23.36458	34.75986	24.09955	22.83188	22.54699	24.17156	40.24146
Additional runs									
38.71432	38.22187	25.19036	36.00212	38.68679	25.38205	16.54858	23.18666	22.13397	35.92587
24.62483	40.30606	22.71962	25.07281	22.80646	23.93222	24.4215	38.36696	37.54514	39.4207
22.70797	23.10087	25.37146	20.61848	33.83426	38.35107	22.9812	32.07413	22.39237	39.88774

Prehistoric Tribes									
30.16448	28.946	29.47625	30.06926	28.33784	28.02511	28.68413	28.18634	25.97987	31.17844
29.98701	30.66227	29.82794	29.14295	26.76983	26.32074	29.59528	30.95552	29.61692	28.97847
Additional runs									
28.02511	32.27897	29.00877	30.65902	30.12228	26.30668	27.15182	29.921	27.22757	28.02727
27.15291	29.47733	30.79537	29.51304	28.40602	28.32377	32.36013	30.3149	28.56076	27.7286
28.46553	28.89731	29.49681	28.00671	28.35083	28.77935	28.36381	30.74884	28.3162	30.23699

Pro Golf 2007									
27.71514	32.59762	34.87483	11.01822	17.60303	34.00217	9.86209	12.90369	10.12963	9.96083
33.3397	32.21224	28.49545	31.89534	29.5608	33.52443	33.2139	14.09007	28.51615	13.05179
Additional runs									
31.4192	32.07848	28.42219	30.2997	13.09478	31.53863	35.24588	12.84954	29.5608	29.63087
31.68673	11.14243	33.91299	13.21581	11.20454	9.88916	33.51647	28.09733	32.93363	36.824
11.05007	35.51341	35.17262	31.71062	10.35735	13.80183	31.52749	34.52768	32.33645	32.42563

Tetris Pop									
15.41386	24.25515	24.56701	22.69587	23.31089	22.91579	23.86254	23.94205	22.24859	24.40176
24.05636	23.17173	24.57322	22.62754	15.65117	22.30202	24.22782	24.48128	23.77681	23.99796
Additional runs									
15.07716	15.26849	22.54926	23.15558	16.52337	23.72214	16.33949	23.52086	15.20761	22.86236
15.20389	22.49087	22.51447	15.05976	15.5965	15.47847	22.80272	15.82263	15.37659	23.33325
15.09331	15.09331	24.61795	22.70954	24.29242	23.47862	22.49335	22.70705	15.11567	15.26352

Virtua Fighter Mobile 3D									
20.69221	57.30983	54.51138	20.87636	55.18878	20.5136	20.95502	65.98464	54.51416	54.42254
20.48214	65.98464	20.63576	20.57005	57.53285	54.76217	20.85786	20.74496	66.00962	20.56913
Additional runs									
20.98094	20.54692	54.32352	54.62428	21.00037	20.74033	54.4281	20.16935	20.7792	20.56265
20.48492	20.93096	54.82787	20.81251	54.25689	66.44734	20.62095	55.68573	20.68851	65.96706
64.46141	20.98834	20.70239	66.58708	54.30224	54.73348	20.69684	63.59708	54.63816	54.81029

Adaptive

3D Need for drift									
53.35786	52.57514	62.03298	59.22563	53.34742	63.72887	60.08401	56.37393	6.44959	58.68034
60.32665	56.90357	57.32885	5.56773	64.29764	5.77124	5.96431	5.299	57.49061	6.0478

AMA IQ Booster									
61.23416	60.96021	68.21714	66.19259	60.90192	65.44066	64.16414	64.2982	64.13111	61.00295
64.01065	65.07344	65.58638	61.62664	60.84946	60.92912	61.31771	60.29766	65.15505	65.35517

Cooking Mama									
21.7037	20.14439	18.48461	21.39295	11.26307	20.19277	22.81084	11.19609	19.5992	11.22586
18.31901	19.30148	19.78155	11.22214	18.03989	20.68773	21.24037	13.14428	18.93491	24.62506

Indiana Jones and the Kingdom of the Crystal Skull									
38.10703	21.15849	36.47922	22.09471	37.54832	21.8682	20.79609	21.4605	37.90469	21.68096
22.01015	39.14593	22.49034	22.07659	21.90445	37.98925	37.21007	21.07997	36.20138	21.63566

Karpov 2									
57.56087	57.59675	36.29437	17.61718	38.45581	39.82499	47.07116	58.40004	16.84426	38.85883
17.43775	17.82145	39.23701	39.11003	57.89488	39.6428	18.65511	16.8415	40.27218	48.40446

NHL 5 on 5 2007									
14.85094	13.43712	25.44771	42.38496	38.50252	31.5838	36.35054	14.10326	21.65528	36.5973
9.50808	24.73603	13.22955	13.19884	23.54885	9.63728	22.39661	22.4771	39.16653	18.3606

Prehistoric Tribes									
8.43632	8.37788	8.51531	8.60946	28.25235	23.42712	29.61584	28.0132	8.97522	8.77935
8.65815	8.57808	8.722	8.3649	8.57483	7.9288	8.64841	8.69495	23.5797	30.26837

Pro Golf 2007									
9.66463	34.81113	11.06599	33.83655	35.65354	39.07255	37.30492	32.77438	11.0851	33.30945
35.14077	29.44774	35.80961	32.65495	9.9035	34.28085	9.67737	33.13746	34.33977	35.64717

Tetris Pop									
22.58281	24.22409	23.83147	14.96534	15.0697	23.38046	15.78908	23.82029	15.17034	22.86112
24.12469	15.80896	22.3629	24.57073	24.48873	24.19924	22.26847	23.87496	15.29086	22.31941

Virtua Fighter Mobile 3D									
61.23357	20.7394	62.17472	55.13048	65.07681	60.97261	64.5558	57.52637	20.31001	56.27614
20.77179	66.07996	61.7407	61.57135	65.33592	67.48751	61.06515	60.48769	61.55654	56.68703

Startup adaptive

3D Need for drift									
63.64799	64.3133	59.55698	62.45043	63.18096	56.97923	65.69349	60.61365	57.4593	63.82801
55.73993	55.72688	64.07065	66.71624	58.11417	66.77886	58.18462	58.14809	61.02849	58.37247
Additional runs									
67.49113	63.83584	65.2943	59.28564	61.81121	62.47652	65.29952	62.9905	58.59685	63.14705
56.83312	59.74744	64.90294	56.89835	61.85034	65.70131	60.49624	65.21864	64.81163	65.61783
58.69338	59.73179	61.76946	60.72323	66.16051	56.52526	63.31664	58.20027	65.00209	61.69902

AMA IQ Booster									
66.64529	66.5598	68.37064	65.6641	67.37779	63.01197	67.33116	66.12458	66.61615	66.80462
64.17774	65.41152	66.56563	64.85583	66.07212	65.55724	61.48869	64.65959	67.16406	65.1395
Additional runs									
62.64864	65.82731	65.11619	60.73094	67.07663	66.61421	65.02293	63.49576	66.31888	67.43025
67.09995	65.76902	66.79102	63.8455	67.07663	66.8221	65.26968	64.39535	65.95943	65.00155
68.91467	67.19515	66.62781	61.82482	66.07601	68.05199	60.29572	59.707	61.32937	67.68866

Cooking Mama									
20.00856	22.2675	21.25712	24.43154	18.49019	18.33761	18.90514	19.49686	19.09866	21.83581
23.40069	21.08779	21.5139	19.24752	23.00994	18.64091	19.65316	18.29668	19.22891	20.57981
Additional runs									
20.46816	18.02129	22.03119	18.72093	22.75129	19.24193	19.31264	19.16006	21.71486	19.23635
19.054	22.04607	20.93521	21.51018	21.45436	21.61066	18.90142	21.13803	22.16702	20.72867
20.03461	19.96018	22.07026	18.66138	17.87057	19.24566	19.37219	21.97536	19.00562	18.71162

Indiana Jones and the Kingdom of the Crystal Skull									
37.35504	24.06076	38.8409	21.76854	36.79029	37.02283	21.78062	24.61645	24.46243	36.63928
21.7021	37.82617	37.86241	21.79874	21.01051	37.37618	21.54204	21.39406	38.08891	38.06173
Additional runs									
38.17347	25.38959	39.12479	37.70536	37.23424	21.78666	36.66043	21.51486	21.88029	37.26444
21.99505	38.32749	37.69328	35.33462	37.37618	38.47246	21.40614	39.31807	21.68096	36.76613
24.84598	21.90747	21.08903	22.05847	37.49396	21.24909	4.60256	22.08263	22.71382	20.77495

Karpov 2									
39.75598	52.98404	37.86231	38.59659	37.67736	37.75189	38.5745	37.56142	39.46061	52.97852
51.2284	39.81671	38.92232	51.78049	38.06106	38.55242	40.17004	39.60139	37.11147	38.1494
Additional runs									
36.32474	37.08386	59.1012	40.35499	41.02578	38.72633	39.5241	37.47861	38.40888	38.59383
34.80649	38.49445	35.75885	35.80577	39.5241	53.30702	36.04317	37.61387	36.95412	38.82294
36.26125	39.83327	39.47441	36.84922	38.35919	38.55518	52.77425	38.52206	38.01965	39.06586

NHL 5 on 5 2007									
22.42732	41.24649	41.36405	37.79931	41.8565	22.89012	36.71591	22.27482	22.55335	34.29177
22.10749	24.51258	23.62192	37.36087	25.09081	23.94069	23.04263	40.7551	36.98067	24.28171
Additional runs									
22.53852	23.9947	22.71644	22.61583	22.32777	35.03945	32.91607	38.02701	37.90098	22.23775
22.56712	22.43897	39.09346	34.48133	22.56712	22.65925	22.20387	22.80964	24.17262	22.07572
30.80434	25.10882	38.28329	40.07307	38.62113	24.84512	33.2592	36.52105	38.85306	41.37675

Prehistoric Tribes									
32.65339	29.85824	28.71984	29.66887	28.2069	31.2477	27.53382	28.96656	30.26404	30.9512
28.34866	30.17206	29.79656	28.27508	28.58998	32.2595	28.88973	32.06363	24.5731	29.27281
Additional runs									
29.80954	28.58457	29.48707	29.2674	29.71865	29.20247	31.04751	28.03809	28.91029	28.24803
31.11676	28.54561	31.00855	28.801	27.81517	29.80305	31.22389	29.65047	32.03658	30.0909
25.24619	28.77502	29.07802	27.5652	28.98171	28.83346	28.97738	29.69267	30.86895	29.40483

Pro Golf 2007									
29.10217	34.05631	9.67737	33.94484	9.99586	32.73298	33.88751	32.87789	32.27913	30.63252
34.69966	34.40187	11.06121	32.28072	32.39856	33.92573	31.01471	37.00076	9.95605	34.09134
Additional runs									
10.81757	34.69648	32.7059	30.81884	14.29231	32.55781	32.58966	28.48908	34.01331	36.08988
35.30161	9.84776	33.20912	9.75062	28.55914	9.91624	32.69635	35.24747	11.30964	36.0851
34.48946	34.15663	31.98771	28.14033	9.79999	33.09606	32.2839	10.97522	34.47672	35.71247

Tetris Pop									
23.42271	25.9511	23.19534	23.39165	22.26847	26.37974	22.2051	23.77681	22.758	23.09967
21.94295	34.93775	23.41152	23.91845	26.42323	22.43868	22.25356	22.66481	23.38419	22.75675
Additional runs									
22.39644	22.46478	31.63283	22.02992	15.0374	28.75531	22.89591	24.50861	22.65984	22.35793
24.19178	24.0986	29.8437	22.66854	23.47986	22.16535	23.69977	23.8526	22.63623	22.26723
23.59292	22.24362	22.73315	23.34319	22.77787	22.45111	22.6996	22.66854	22.31817	33.15483

Virtua Fighter Mobile 3D									
56.65001	20.56543	56.57042	66.36406	63.08717	67.00999	21.0781	67.50139	61.77864	20.75791
56.29835	57.42736	66.81288	56.50009	65.34981	66.11605	63.38516	57.89561	55.78105	66.34185
Additional runs									
61.68517	58.20192	66.14381	57.77994	56.59541	61.49454	20.69961	64.42069	58.3611	64.14307
62.40885	61.51675	54.80289	67.61059	63.4851	67.742	62.60133	66.81011	20.15639	56.78234
58.6017	56.45382	55.29336	62.96594	56.43439	56.43716	67.48195	56.37979	66.45382	63.30835