CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Search-based Test Generation Framework for Android Apps with Support for Multi-objective Generation: STGFA-SMOG

Master's thesis in Computer Science and Engineering

TEKLIT BERIHU GEREZIHER
SELAM WELU GEBREKRSTOS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Search-based Test Generation Framework for Android Apps with Support for Multi-objective Generation: STGFA-SMOG

TEKLIT BERIHU GEREZIHER
SELAM WELU GEBREKRSTOS

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

TEKLIT BERIHU GEREZIHER
SELAM WELU GEBREKRSTOS

Supervisor: Gregory Gay, Computer Science and Engineering
Examiner: Jennifer Horkoff, Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Acknowledgements

This study was supervised by Gregory Gay, Computer Science, and Engineering. We want to thank him for all the support, close follow-ups, and positive energy he gave us. It would haven't been possible to do such a great job without his efforts and guidance. We are thankful that he was always there for us when we needed help, and we believe his professional feedback helped us effectively during our entire journey.

Abstract

Search-based test generation is an automated test generation technique that saves time and energy which developers spend for a manual test generation. In search-based test generation, the test input creation is framed as a search problem, in which search algorithms are used to automatically select the optimal solution based on feedback from one or more fitness functions - scoring functions based on attainment of selected test criteria, such as code coverage or number of crashes detected. The effectiveness of test input generation depends on the selection of measurable test goals and the selection of effective fitness functions. Search-based test generation with support for multi-objective test generation has been applied in other domains such as desktop software. However, automated test case generation for Android apps is not a mature field, and existing approaches do not allow testing to be adapted to specific goals, like identifying excessive CPU usage or maximizing code coverage. In addition, multi-objective search-based test generation can target multiple fitness functions at once, shaping a test suite that has multiple properties of interest (for example, you could both maximize CPU usage and the number of crashes, and you could shorten test case length, all during one test generation session). However, we do not know which combinations of fitness functions are effective at triggering crashes or meeting other testing goals during Android GUI-based testing.

In this study, we developed a search-based test generation framework for Android GUI-based testing with support for multiple fitness functions, largely based on non-functional properties such as maximizing CPU or battery usage. We compared STGFA-SMOG with the Random test generation tool we developed as a baseline in the intermediate and final evaluations to determine whether STGFA-SMOG generates tests more adept at maximizing the properties of interest, and whether it generates tests more effective at triggering crashes. We also assess which configurations of STGFA-SMOG (fitness function combinations) trigger the most crashes, and the impact of adjusting the search budget on fitness and crash detection. Based on the intermediate evaluation STGFA-SMOG produced better fitness values than Random test generation for all search budgets we applied. Furthermore, STGFA-SMOG with both search budgets detected more crashes than Random test generation, and more crashes were detected with a 30 generation search budget than with a 10 generation budget. However, the comparison between the different configurations of fitness functions only demonstrated minor differences between fitness function configurations. The empirical results revealed that the best configuration is dependent on the app-under-test. Ultimately, STGFA-SMOG can enable testers to generate test cases for non-functional properties of interest, such as CPU, memory, battery, and network usage, and is able to trigger crashes in Android applications.

Contents

List of Figures ix

List of Tables xi

1 Introduction 1
- 1.1 Introduction . . . 1
- 1.2 Problem Identification and Motivation . . . 3
- 1.3 Objective of the Research . . . 4
- 1.4 Significance of the Research . . . 5
- 1.5 Thesis Organization . . . 5

2 Background 7
- 2.1 Android Testing . . . 7
- 2.2 Automated Test Generation . . . 8
- 2.3 Search Based Test Generation . . . 9
  - 2.3.1 Hill Climbing . . . 10
  - 2.3.2 Genetic Algorithm . . . 11

3 Related Work 15
- 3.1 Monkey . . . 15
- 3.2 DynoDroid . . . 16
- 3.3 EvoDroid . . . 18
- 3.4 Sapienz . . . 19
- 3.5 GUIRipper . . . 20
- 3.6 SwiftHand . . . 21
- 3.7 Empirical Studies On Test Suite Generation . . . 21
- 3.8 Summary . . . 24

4 Approach 27
- 4.1 Framework . . . 27
- 4.2 STGFA-SMOG Parameters . . . 28
- 4.3 Problem Representation . . . 29
- 4.4 NSGA-II . . . 31
- 4.5 Initial Population . . . 32
- 4.6 Generations of the Algorithm . . . 33
  - 4.6.1 Fitness Evaluation . . . 34
  - 4.6.2 Candidate Selection . . . 38

4.6.3 Variation Operators 38

5 Methodology 43
5.1 Design Science Iterations 44
5.2 Objective-centered Solution 44
5.3 Iteration I: Design of Prototype and Evaluation 45
5.4 Iteration II: Addition of Genetic Algorithm and Evaluation 46
5.5 Iteration III: Addition of More Fitness Functions and Evaluation . . . 47
5.5.1 Addition of More Fitness Functions 47
5.5.2 Intermediate Evaluation 47
5.5.3 Final Evaluation of STGFA-SMOG 49

6 Results 53
6.1 Intermediate Evaluation Results 54
6.2 Final Evaluation Results 62
6.2.1 Crashes Triggered and Running Time Logs 63
6.2.2 Comparison of Techniques and Configurations 66
6.2.3 Effect of Search Budget 73

7 Discussion 75
7.1 RQ1: Framework Design 75
7.2 RQ2: Effectiveness of Test Cases 76
7.3 RQ3: Effectiveness of the Combinations of Fitness Functions 77
7.4 RQ4: Effect of Search Budget 77
7.5 Threats to Validity 78
7.5.1 Internal Validity 78
7.5.2 External Validity 78
7.5.3 Conclusion Validity 79

8 Conclusion 81
8.1 Conclusion 81

A Appendix 1 I

List of Figures

2.1 Genotype and Phenotype representation of the TSP 12
2.2 Main steps of a genetic algorithm [1] 13

4.1 Architecture of STGFA-SMOG 28
4.2 Individual's genotype representation 31

5.1 DSRM process for the STGFA-SMOG study 46

List of Tables

2.1 Crossover example 12

3.1 Android testing tools with their corresponding type 15

5.1 Apps used for intermediate evaluation 48
5.2 Apps used for final evaluation 50

6.1 Median values of STGFA-SMOG fitness values with 10 generation search budget and Random generation for each app 58
6.2 Median values of STGFA-SMOG fitness values with 10 generation search budget and Random generation for each app 58
6.3 Median values of STGFA-SMOG fitness values with 15 generation search budget and Random generation for each app 58
6.4 Median values of STGFA-SMOG fitness values with 15 generation search budget and Random generation for each app 59
6.5 P-Values for Mann-Whitney rank-sum test for the fitness values of STGFA-SMOG with 10 generation versus Random test data generation (baseline) 59
6.6 Results of Vargha-Delaney A Measure for the fitness values of 10G STGFA-SMOG versus Random test data generation (baseline). Note that large effect sizes are bolded and effect size is only measured when statistical significant difference is found using the Mann-Whitney Wilcoxon test. 59
6.7 P-Values for Mann-Whitney rank-sum test for the fitness values of 15G STGFA-SMOG versus Random test data generation (baseline) . 60
6.8 Results of Vargha-Delaney A Measure for the fitness values of 15G STGFA-SMOG versus Random test data generation (baseline) 60
6.9 P-Values for Mann-Whitney rank-sum test for the fitness values of STGFA-SMOG 15 versus 10 generation, values in the table are bolded for P-value less than 0.05 and 'NA' in the Network column shows these apps operates offline 61
6.10 Results of Vargha-Delaney A Measure for the fitness values of STGFA-SMOG 15 versus 10 generation, note that effect size is only measured in cases where differences were detected through the Mann-Whitney Wilcoxon test and values are bolded for those who have high effect-size. 61
6.11 Unique crashes detected by STGFA-SMOG with 30 generation across all trials 64

6.12 Average number of times the crashes are triggered by STGFA-SMOG with 30 generation per trial 65
6.13 Unique crashes detected by STGFA-SMOG with 10 generation across all trials 65
6.14 Average number of times the crashes are triggered by STGFA-SMOG with 10 generation per trial 65
6.15 Unique crashes detected by Random test generation across all trials . 65
6.16 Average number of times the crashes are triggered by Random test generation per trial, (where Random-10 is Random test generation run for 10 generation, and Random-30 is Random test generation run for 30 generation) 66
6.17 Time taken by STGFA-SMOG with 10 generation 66
6.18 Time taken by STGFA-SMOG with 30 generation 66
6.19 P-Values for Mann-Whitney rank-sum test of the comparison of the different combination of the fitness functions of the STGFA-SMOG as well as Random test generation by the number of crashes detected in 10 generation in the **scoutantblokish** app. 68
6.20 Results of Vargha-Delaney A Measure of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 10 generation in the **scoutantblokish** app. Note that large effect sizes are bolded and effect size is only measured when statistical significant difference is found using the Mann-Whitney Wilcoxon test. 68
6.21 P-Values for Mann-Whitney rank-sum test of the comparison of the different combination of the fitness functions of the STGFA-SMOG as well as Random test generation by the number of crashes detected in 10 generation in the **dib2calc** app 69
6.22 Results of Vargha-Delaney A Measure of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 10 generation in the **dib2calc** app. Note that large effect sizes are bolded and effect size is only measured when statistical significant difference is found using the Mann-Whitney Wilcoxon test. 69
6.23 P-Values for Mann-Whitney rank-sum test of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 30 generation in the **scoutantblokish** app. 71
6.24 Results of Vargha-Delaney A Measure of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 30 generation in the **scoutantblokish** app. Note that large effect sizes are bolded and effect size is only measured when statistical significant difference is found using the Mann-Whitney Wilcoxon test. 71

6.25 P-Values for Mann-Whitney rank-sum test of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 30 generation in the **dib2calc** app. 72
6.26 Results of Vargha-Delaney A Measure of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 30 generation in the **dib2calc** app. Note that large effect sizes are bolded and effect size is only measured when statistical significant difference is found using the Mann-Whitney Wilcoxon test. 72
6.27 P-Values for Mann-Whitney rank-sum test of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 30 generation in the **palmcalc** app. 72
6.28 P-Values for Mann-Whitney rank-sum test of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 30 generation in the **sourceforgesolitaire** app. 73
6.29 Results of Vargha-Delaney A Measure of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 30 generation in the **sourceforgesolitaire** app. Note that large effect sizes are bolded and effect size is only measured when statistical significant difference is found using the Mann-Whitney Wilcoxon test. 73
6.30 P-Values for Mann-Whitney rank-sum test of the crashes detected by STGFA-SMOG with 30 generation versus 10 generation 74
6.31 Results of Vargha-Delaney A Measure of the crashes detected by STGFA-SMOG with 30 generation versus 10 generation 74

1
Introduction

1.1 Introduction

Mobile devices are becoming popular because of their portability, accessibility, and connection ability to the Internet. As of August 20, 2020 statista [2] published that there are around 3.5 billion smartphone users all over the world. Due to the popularity of smartphones, developers are engaged in developing applications and pushing them to Google play store, and/or App Store. Based on AppBrain statistics there are around 3 million Android apps on Google play as of November 23, 2020 [3]. These huge numbers of Android apps need to be properly verified to ensure that they are reliable and crash free. Testing is one of the prominent techniques of software verification which is widely used these days. However, software testing is a time-consuming, tedious and costly activity in the software development life cycle [4], and becomes more expensive as software complexity increases.

The majority of this cost comes from the manual work done by developers to create test cases and assess their corresponding expected outcomes. A number of research projects have been conducted to identify a way to minimize the cost of software testing. One approach of minimizing the cost of software testing is automating the process of test generation [4]. For example, with search-based test generation, researchers and practitioners were able to automatically generate test cases that detect faults and maximize code coverage over a system-under-test (SUT). In search-based test generation, the test input creation is framed as a search problem, in which search algorithms are used to automatically select the optimal solution based on specified test criteria from the space of possible test cases. Search algorithms systematically select the optimal test cases from the space of possible test cases based on well-defined test goals and fitness functions – scoring functions that provide feedback and shape the test suites to meet high-level goals of the testers [5].

Search-based test generation is a type of automated test generation, in which test cases are generated using search algorithms guided by fitness functions that compute the approximate fitness of the test cases to a specified test goal [4]. In this approach, test cases are generated semi-randomly, tested against the SUT, and the fitness functions compute the closeness of the test result to a specified test goal. If the test cases generated meet the objective of the test, these test cases are returned as solutions. However, if the test goal is not attained, the search algorithms select the best test cases that give the closest fitness to the test goal and generate new test cases based on the selected test cases. This process continues until the test goal is reached or the search budget is exhausted. The effectiveness of generating test cases to attain the test goal depends on two things, the selection of measurable test goals and selection of effective fitness functions [6]. That is, the effectiveness of the test case generation relies on the approximation functions that compute the closeness to the test goal. Fitness functions shape the test suites with properties that testers desire from the generated suites. Many fitness functions have been applied in domains such as desktop software [6], but we do not know yet which will be effective for triggering crashes or meeting other testing goals when testing Android apps.

Based on different studies search-based testing most likely has better acceptance than a random generation in producing effective solutions within a given time and the guidance toward the test goal helps here a lot that is why we are applying search-based test generation for Android apps in this study. As stated by Shamshiri [7], while random search relies on encountering solutions by chance, guided searches aim to find solutions more directly by using a problem-specific “fitness function” and with a good fitness function, guided search-based approaches are capable of finding suitable solutions in extremely large or infinite search spaces.

In addition, automated test case generation for Android apps is not a mature field, and existing approaches are not sophisticated enough to ensure that faults are discovered and existing approaches do not allow testing to be adapted to specific goals, like identifying excessive CPU usage or maximizing coverage of the source code. A search-based test generation is a flexible approach that enables testers and developers to guide the generation in some way and this guidance allow easy adaptation of the test generation framework to different goals, as well as fine-tuning of the search-based approach. In particular, multi-objective generation can target multiple fitness functions at once, shaping a test suite that has multiple properties of interest (for example, you could both maximize CPU usage and the number of crashes, and you could shorten test case length, all during one test generation session).

In this study, we developed a search-based test generation framework for testing Android apps with support for multiple fitness functions that generate test cases for Android apps, run them, and collect information about how good the test cases are. We focused on the goal of discovering fatal crashes, as those are a direct indication of a fault in the Android app. We do not know which fitness functions lead to the discovery of more crashes, so we conducted empirical research to compare our framework to a Random test generation baseline, and we compare different configurations of the framework (different combinations of fitness functions) to discover whether certain combinations of fitness functions can produce more crashes than others.

Search-based test generation framework for Android apps with support for multi-objective generation (STGFA-SMOG) is a framework that generates test cases with multifaceted properties for Android GUI-based testing and supports the addition of new fitness functions over time. STGFA-SMOG allows the creation of more complex test cases and customization of the framework for specialized testing goals such as memory usage, CPU usage, code coverage, crashes, etc. STGFA-SMOG generates test cases by optimizing different combinations of fitness functions that allow testers to meet different testing goals. The empirical study shows that STGFA-SMOG outperformed the Random test generation we developed as a baseline in revealing more crashes and generating test cases with higher fitness values. However, we did not find a persistent pattern that shows one configuration outperforms the other configurations in all the app-under-test (AUT), where AUT is an Android application that is being tested using STGFA-SMOG. Rather we observed that the best configuration that triggers the most crashes differs from AUT to AUT. So, the best configuration is dependent on the app-under-test. As future work, we recommend extending the scope of experiments to look at a larger number of apps, a wider variety of fitness function combinations, and additional search budgets.

1.2 Problem Identification and Motivation

In search-based test generation, in order to generate test cases that are effective at detecting faults in a software, we need to define test goals that improve the likelihood of fault detection and corresponding fitness functions that represent those goals. It is obvious that the number of faults that exist in a piece of software is not known before it is tested. As a result, test goals are designed with the aim of increasing the probability of revealing faults. It is important to carefully choose test goals and fitness functions so as to maximize the probability of revealing faults. Each fitness function imbues a test case with certain properties related to that function. Single-objective test generation is often ineffective at fault detection, as the test suites are focused entirely on that one goal. Multi-objective generation focuses on targeting multiple fitness functions concurrently, efficiently generating multi-faceted test suites [6, 8]. For multi-objective generation to be effective, we need to understand the effect of choosing different combinations of fitness functions on the effectiveness of the generated test suites. An empirical goal of this project is to guide how to choose fitness functions and which of them to combine for effective multi-objective test generation.

A motivation for this thesis work is the research work done by Salahirad et. al [6]. In their work, they studied the effect of the combination of fitness functions in search-based unit test generation. To understand how effective test cases are generated when we use different combinations of fitness functions, they assessed the EvoSuite test generation framework. Their study mainly focused on unit test generation for java based applications using single-objective generation and multi-objective generation. As far as we know, there is no research study done that shows the effect of the different multi-objective configurations of fitness functions on the effectiveness of generated test cases to system testing of Android applications. However, to carry out this study, there is a lack of publicly available multi-objective test generation tools for Android GUI-based testing. There is only one multi-objective test generation tool for Android apps [9] that supports only three fitness functions and it is an outdated tool at this time. So, we do not know how can a search-based test generation framework for Android GUI-based testing best be designed to support multi-objective generation and the addition of fitness functions over time.

Search-based test generation tools iterate over the search process to find the optimal solutions for the problem at hand until the allocated search budget is finished. The search budget allocated can be a time budget or a maximum number of iterations or generations allowed. It is also important to know whether an increased search budget improves the effectiveness of the generated test cases. There are no research studies found which investigated the effect of an increased search budget on the effectiveness of test cases generated for Android GUI-based testing. Since the random generation of test cases for Android GUI-based testing is the most applied approach in Android apps testing, it is important to compare the effectiveness of the test cases generated by the search-based test generation to the test cases generated by the random test generator prototype.

Therefore, to address the problems discussed above we developed a search-based test generation framework with support for multi-objective generation for Android apps. The framework is developed with support for multi-objective generation through balancing different combinations of fitness functions. Based on the developed framework, we investigated the differences in the effectiveness of generated test cases when we vary the combination of fitness functions for GUI-based testing of Android apps. The framework is also run with different search budgets to generate test cases and compared the effectiveness of the test cases in causing crashes with the respect to the search budgets allocated.

1.3 Objective of the Research

The objective of this research is to develop a search-based test generation framework with support for multi-objective test generation and the addition of new fitness functions over time. The goal of this research is to create a framework that can be used in practice to carry out Android GUI-based testing. Moreover, the framework can be used to perform empirical studies on test generation for Android GUI-based testing, especially related to non-functional aspects of execution such as memory consumption or energy efficiency. With the empirical study, this research attempted to address research questions such as "whether the framework created can successfully identify crashes in Android apps (i.e., check whether the framework is more effective than a random generation baseline), and see if any particular combination of fitness functions from the set implemented is more effective at triggering crashes". The framework is designed to support several fitness functions, including ones based on coverage, crash detection, and performance aspects such as CPU, memory, network, or energy usage. It is also designed to easily add additional fitness functions in the future, and allow users to select fitness functions to be used during test case generation.

The framework is used to test different Android apps by varying the combination of fitness functions and evaluate their effectiveness in triggering crashes. An empirical assessment is carried out to compare the effectiveness of the test cases generated and find out which combination of fitness functions generated the most effective test cases based on the results obtained. STGFA-SMOG was run on different Android apps to generate test cases by varying the combination of fitness functions and keeping the search budget the same. The result data collected in this process are empirically assessed to know which combination fitness functions generated effective test cases in terms of causing crashes. A Similar process is carried out by varying the search budget to understand its effect on the effectiveness of the test cases generated.

1.4 Significance of the Research

The output of this research is a search-based test generation framework with support for multi-objective generation and an empirical study on the effectiveness of test cases generated by different combinations of fitness functions and search budgets. Accordingly, this research study has two main contributions. The first is, the result of this study presents insights on the effect of using different combinations of fitness functions and balancing them concurrently, and different search budgets on the effectiveness of the generated test cases for GUI-based testing of Android applications. This will benefit both researchers and practitioners with advice on how to select fitness functions in order to generate effective test suites in this domain. The second contribution is the framework developed also enable the creation of and experimentation with new fitness functions or existing one targeting the Android apps. The practitioners can use the tool to generate test cases for Android GUI-based testing by specifying an appropriate search budget for their purpose. The researchers can also use the tool as starting prototype to design an advanced search-based test generation approach for Android apps and conduct further study.

1.5 Thesis Organization

This introductory chapter is followed by seven other chapters, each of which discusses a particular focus of the study. Chapter 2 presents the literature review of Android apps testing, automated test generation, search-based test generation, and genetic algorithms to provide a theoretical background that supports this study. In chapter 3, a review of related works to this study from the industrial practice and academic research are discussed in detail. Chapter 4 deals with the approach used to develop the framework in more detail. Chapter 5 is dedicated to present the methodology used in this study. It discusses the design science research methodology, the iterations of the framework development, tools used, and other configuration details. The experimental results obtained when running STGFA-SMOG on different Android apps are presented in Chapter 6 where as Chapter 7 discusses the research findings based on the empirical analysis performed and threats and limitations the research. Chapter 8, which is the last chapter, presents the final conclusions based the research findings.

2
Background

2.1 Android Testing

Android is an open-source operating system for smartphone and mobile devices which runs on the Linux kernel. Developers develop mobile apps using Java programming language codes that can control mobile devices with Google-enabled Java libraries. The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language [10].

As the number of mobile users is increasing every year markets are competing on providing products with better performance. The performance of mobile devices relies on how the application runs on the device. The time taken and accuracy of the operation intended to be done on the mobile app is the basic concern here. To deal with this concern testers test the applications aiming production of high-quality applications including testing the security holes, reliability of the app, and its functionality. In other words, Android app testing aims at testing the functionality, usability, and compatibility of apps running on Android devices [11]. The process to test an Android app is: the first step is the installation of the app which we want to test, running it followed by the application of test input. For example, if we are testing the GUI of an app, we interact (manually or through automation) by touching, swiping, or applying other actions to the GUI. After the input is applied, the execution behavior is observed and collected from all sorts of perspectives, from the changes to the information displayed to the screen to the amount of data downloaded. The reaction to the tests will be seen and studied. Running the test cases provides lots of information on how our app is responding to the tests.

Android testing could be conducted at different levels based on the level of abstraction where the test is conducted.
Unit Testing:- this level of testing is at the lower level which might be testing a function or a method in the code whereas unit is to mean a fraction of the code which has a single return value. According to [12], a unit test is an automated piece of code that invokes a unit of work in the system and a unit of work can span a single method, a whole class, or multiple classes working together to achieve one single logical purpose that can be verified. For example, it could be a function that calculates the sum of two integer values.

Integration Testing:- testing in which it tests the interaction between components in the system or in the app. These components can be as simple as two classes that interact, or could be two entire subsystems that interact. For example, a developer may have a component that fetches data from and writes to a database and a second component that transforms that data and sends it to be written. Tests of the interactions between those components would be considered part of integration testing.

System Testing:- When we test the system in its entirety, we are doing what is called "system testing" [12]. Testers use this type of testing to capture system-level bugs while users can test a system as acceptance testing to check the performance of the software or targeted system. System testing can take place through a programmable interface (e.g., an API) or through a user interface (e.g., the GUI). We particularly are focusing on GUI testing of Android apps in this thesis because most of the time users interact with Android devices through graphical user interfaces. Making sure that the GUI of these apps is working properly is crucial since end-users interact with it to perform different activities. In addition, crashing an app is a bad experience to the end-users that owner of the app probably gets many negative feedback and reviews which may lead face a great loss of the market. User Interfaces of an app can be tested either manually or automatically by generating events.

Additionally, apps can be tested on their performance to check whether they consume resources beyond expected or if they are efficient. Compatibility test is also done to check if the apps can operate in different versions of the OS and other platforms. This tells testing is not only about the functional requirement but also non-functional requirement including the time and memory usage.

Android applications have 4 components [13]. Those are:- **Activities** which represent user interfaces and constitute the visible part of Android applications. It represents a single screen with a user interface that performs actions. Sometimes a single screen can have several activities and one of the activities would be selected as an activity to be considered during the launch of the App. In Java most commonly an activity is implemented as a subclass of Activity class. **Broadcast Receivers** wait to receive event messages, for example, it could be incoming calls or text messages, from other components or the system while **Content Providers** act as the standard interface to share structured data between applications. The fourth component **Services** are the ones that execute tasks in the background. For example, we play music in the background while we are on other pages. Android Service Components can open numerous opportunities for malicious actions because their processing is hidden to the device user.

2.2 Automated Test Generation

Tests can be created either manually or automatically. Nowadays, software in the market is increasing and complexity to address the needs of users which makes thorough manual testing difficult and impractical. It is almost impossible to ensure the quality of complex software which needs large volumes of manual input to test thoroughly of inputs manually and it is also unrealistic economically because it is a lot of work for human beings creating every test case and with every constraint. In addition, software might be tester-dependent in case of manual testing, if the tester is skilled he/she can easily manage to address the corner cases and possible holes and for junior testers, this may not be the case. Automated test case generation can reduce the human burden. Generating test data by hand is tedious, expensive, and error–prone [8]. It can also be dependent on the individual's skills if there is no standard tool or framework to handle these issues. Automated testing can cover a lot of cases within a short period of time and can generate better-updated test cases. It can also help with improving the reliability and quality of the product. But this does not mean the use of manual testing is negligible. In many industries manual testing is being used even today because it is helpful in many scenarios including some activities which need special attention, for example automating software installation is difficult.

2.3 Search Based Test Generation

A simple approach to test generation in Android would be to randomly generate test cases, like the monkey tool included in the Android development tools does. But in some cases, random generation is not smart enough to exercise the deeper features of the software. A complex piece of software may have a near-infinite number of possible inputs. Random input generation can only cover a small portion of that space, and may apply input that is redundant to that already applied. Some critical features of software may be skipped while we generate randomly and that could result in failure of the software in the market. So, random generation is not smart enough to ensure adequate testing of complex apps.

The metaheuristic search technique involves the transformation of test criteria to fitness functions in which solutions of the search can be compared and contrasted with respect to the overall search goal and the application of a strategy (the "metaheuristic") to sample from the space of possible inputs that incorporates feedback from those fitness functions [5]. As an example, evolutionary algorithms use simulated evolution as a search strategy to evolve candidate solutions, using operators inspired by genetics and natural selection. In the genetic algorithm which is a well-known evolutionary algorithm, the search is driven with the exchange of information between solutions to breed new solutions while evolution uses mutation. Parents recombined and mutated iteratively to give birth to new generations. For example, if we have two parents they crossover and create two new offspring. Many mechanisms could be used in order to select which to select for breeding with aim of producing new offspring.

Fitness functions is the measure of quality of generated test suites according the goals of the test generation process. For example, we might use coverage of the code as a fitness function, and attempt to maximize the percentage of the code executed by the test cases. Individuals which have good fitness will be selected as the basis for the next generation of the search process. According to McMinn [1], the use of metaheuristic search algorithms to generate test input, was firstly published by two American researchers, Webb Miller and David Spooner in 1976. Their approach aimed to generate float-type test input. They used the cost function to select better input. The inputs which are close to the average line will be selected and the far ones will be rejected. The new thing they came up with was they used a numerical maximization method unlike what was being used before which is symbolic execution. A couple of years after researchers applied the genetic algorithm to the problem interest in the topics started to grow and became a popular research area for many researchers [1].

How we structure our fitness function is also important here to solve the optimization problem because this guides our search toward the desired result. For meta-heuristic searches, guidance can be provided in the form of a problem-specific fitness function, which scores different points in the search space concerning their 'goodness' or their suitability for solving the problem at hand [1]. It is the way we structure what we are looking for that determines the ease to come up with required solutions. We can derive the fitness functions from the given problem keeping in mind how we can connect it with the format of the solution so that we can use the fitness information on different optimization algorithms including genetic algorithms & hill climbing.

2.3.1 Hill Climbing

Hill climbing is a metaheuristic search algorithm that finds a better solution in which it randomly chooses an arbitrary solution from the search space as a starting point and makes incremental changes to the solution to come up with some somehow better solution. When we make incremental changes, we "explore the neighborhood" of the current solution, attempting to identify a small change that improves solution quality. This process continues iteratively until time runs out. The last solution may not be the absolute best (global optimal maximum) but it is sufficiently good considering the time allotted and it is local optimal. There are several hill climbing algorithms, and each differ in how the new solution is selected. For example, "steepest ascent" evaluates all neighbors and selects the one that makes the largest improvement to the score, while "random ascent" evaluates neighbors at random until it identifies the first that is an improvement over the current solution. A common example here is the Travelling Salesman Problem. Travelling Salesman Problem is about a traveler needs to visit all the cities from a given list, where distances between all the cities are known and each city should be visited just once. The problem is concerned on what is the shortest possible route that he visits each city exactly once and returns to the origin city. The first solution will be generated randomly that includes all the cities to be visited in the list. After then making changes to the solution will continue with changing orders (swapping the order) of the cities and other operations to find the local optimal solution. The iteration of evaluating the neighbors and altering the solution will stop where none of the direct neighbors of the current solution are better than the current solution. This does not show the solution is the best, it is limited to local optimal not globally because it considers only one solution at a time and moves only in the local neighborhood of those solutions [1].

Hill climbing is claimed to be highly dependent on the first solution randomly selected. It can turn out a solution that is far from the global optimal. To overcome the issue of local optimal, most hill climbing algorithms incorporate restarts when stuck in a local optimal.

2.3.2 Genetic Algorithm

Search-based approaches can use local search or global search, and that hill climbing is an example of a local search, an algorithm improve solutions by making changes to them, as explained in the above and genetic algorithm is an example of a global search where solutions are sampled from the full search space. According to McMinn [1], genetic algorithms are derived from Darwinian evolution and the concept of survival of the fittest in which the survived ones can keep their generation for the future. Each point in the search space currently under consideration is referred to as an 'individual' or a 'chromosome'. The current set of individuals which is under consideration for the next generation are collectively referred to as the current 'population'. An individual is characterized by a set of parameters (variables) known as genes. Genes are joined into a string to form a Chromosome (solution). Unlike the Hill Climbing algorithm, genetic algorithms maintain a population of solutions rather than just one current solution [1]. In addition, the genetic algorithm provides a decision to use different types of representation of genotype, mapping to phenotype. This needs a careful and smart way to choose which representation is better for the specific problem provided. Genotype is the actual encoded structure of the solution, appearing in the way that can be manipulated easily by the algorithm. Phenotype is a decoded structure that is the representation of a genotype after being altered in some way. In other words, the genotype is the solution's manipulable representation, while the phenotype is an evaluable solution to the problem [14].

For example, let's take the TSP (Travelling Salesman Problem) and the genotype in this case is the genetic information of an individual of the population that is the way how we represent the problem in our code. To be specific, it could be either strings to represent city names or list of city numbers that represent the city. The phenotype of the TSP is the sequence of cities as they are visited as the information obtained based on the genotype.

The first step in the genetic algorithm is the initialization of the population in a random fashion or with some other way such as heuristics to seed the population. From the population, parents are selected based on their fitness (best fitted will be a candidate). When a new population is formed, the gene is divided into four sections. The first section contains some of the best solutions from the previous population, brought over unchanged. The second section is created by performing crossover on some of the best solutions. A crossover takes two parents and combines

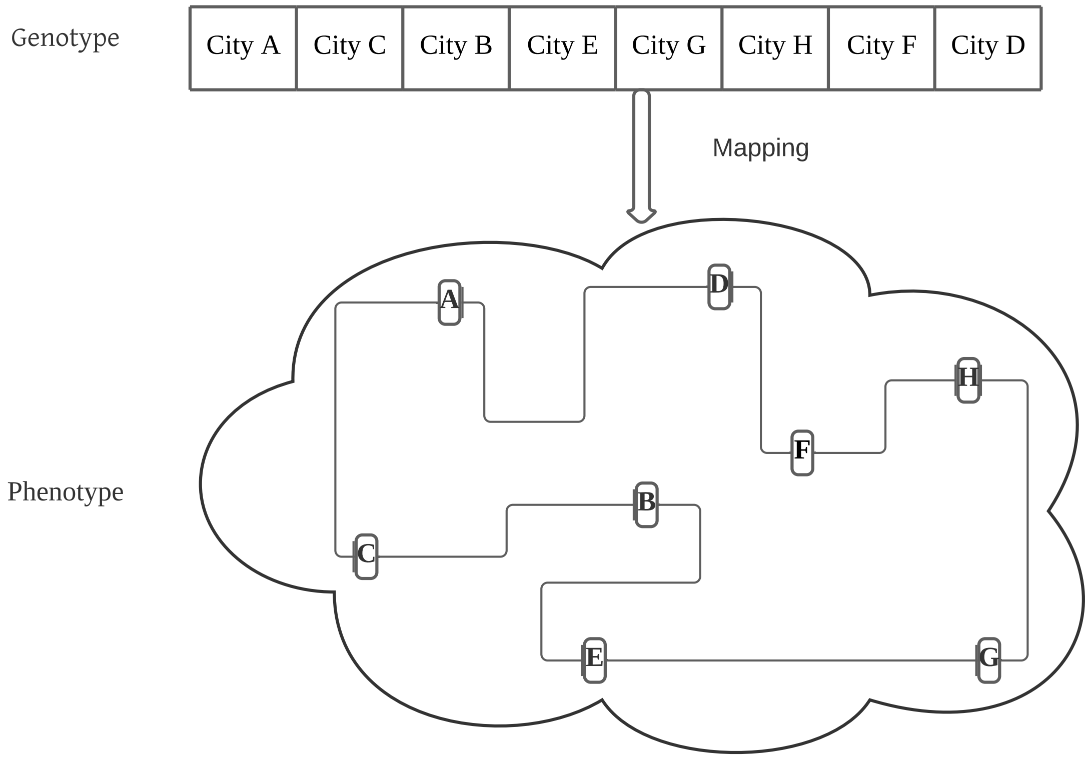

Figure 2.1: Genotype and Phenotype representation of the TSP

the segments to form new offspring.
For example let's take the following two parents;

Parent 1: [0,1,1,0,0,1]
Parent 2: [1,0,1,0,0,1,1]

The parents are first to be split into segments as parent1: [011],[001] and parent2: [101][011] and the result of the crossover will be half of the segment of parent 2 is appended to the end of half segment of parent 1 and the vice versa.

Table 2.1: Crossover example

Parent 1	[0 1 1] [0 0 1]
Parent 2	[1 0 1] [0 1 1]
Offspring 1	[0 1 1] [0 1 1]
Offspring 2	[1 0 1] [0 0 1]

For more segments in a given parent, for example for two-point crossover 2 indexes will be generated randomly on the parent. If we have [1001] and [0010] & the generated indexes are 1 to 2 then [00] from the first gene and [01] from the second is selected then these two are the ones that will exchange. As a result, the offspring will be [1011] and [0000].

The third section is created by mutating some of the best solutions and the fourth section is created completely at random to introduce additional diversity into the population. Mutation is a process where a small incremental change is made to one of the existing best solutions to create a new member of the population. It is similar to what would be done in a local search. The summary of what is being done in the process to find an optimal solution is drawn as follows.

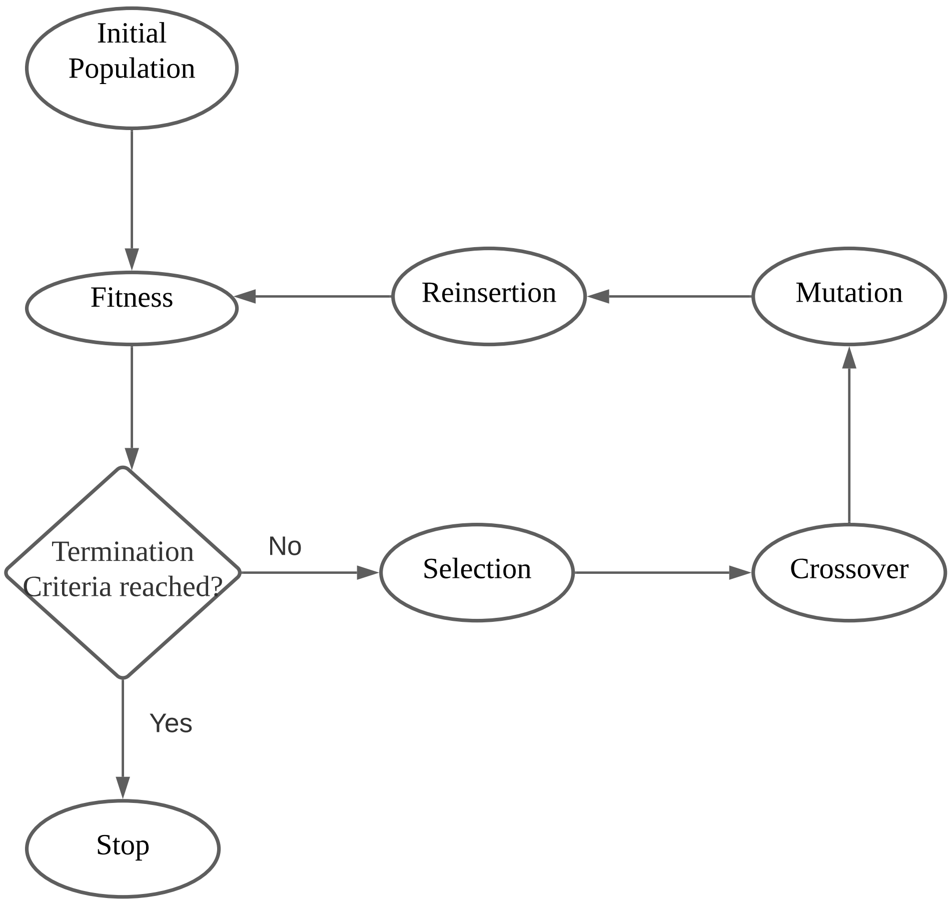

Figure 2.2: Main steps of a genetic algorithm [1]

3
Related Work

This chapter discusses different related works of Android test generation tools and compare them with our framework (STGFA-SMOG).

Table 3.1: Android testing tools with their corresponding type

Tools	**Type**	**Generate UI events**	**Generate system events**
Monkey	Random	Yes	Yes
Dynodroid	Random	Yes	Yes
EvoDroid	Search-based	Yes	No
Sapienz	Search-based	Yes	Yes
GUIRipper	Model-based	Yes	No
SwiftHand	Model-based	Yes	No
STGFA-SMOG	Search-based	Yes	Yes

3.1 Monkey

A Monkey is a testing tool for Android apps that randomly generate events from predefined distribution and send them to the app [15]. Unlike Monkey, STGFA-SMOG applies search-based technique in generating the test cases because we believe it can help us in ending up with an effective test than total randomness. The tool Monkey is known for its good code coverage, ease, and portability. Portability is in the sense that the tool operates in different version of Android devices. In other words, the compatibility of Monkey on different versions of Android devices makes it popular in industries, state-of-practice. The ease to use comes from the random generation approach, testers should not worry about which test data or test case to use since they can generate randomly from some given distribution and these random events simply trigger on random coordinates in the Android screen. Monkey runs on both emulator and real physical Android devices. It sends a pseudo-random stream of user events into the system, which acts as a stress test on the application software you are developing.

Monkey watches the system under test and looks for three conditions, which it treats specially: [16] The first is if you have constrained the Monkey to run in one or more specific packages, it watches for attempts to navigate to any other packages and blocks them while the second option is if your application crashes or receives any sort of unhandled exception, the Monkey will stop and report the error. The third possible scenario is if your application generates an application not responding error, the Monkey will stop and report the error. Depending on the verbosity level testers have selected, testers will also see reports on the progress of the Monkey and the events being generated. According wetzlmaier [17], without knowledge about these existing defects, the Monkey is likely running into the same failure over and over again creating from the viewpoint of the developers “false alarms”. Furthermore, if these defects are easily detectable by the Monkey, they will dominate the results of the test runs and mask other defects that are harder to find.

Although Monkey generates different types of events such as **APPSWITCH** (switch to a different app) and **FLIP flip**[15] (open keyboard), we did not apply these events, as we did not find their corresponding ADB command, in our tool to interact with the system under test. Monkey does not have a guidance while generating these events. This probably ends up in the difficulty to address the complex features of the system and may not be easy to create test cases that visit the corner cases and other behaviors of the system with a random generation used in Monkey.

3.2 DynoDroid

DynoDroid is an input generation system for Android apps that enables the random generation of both system events and UI events. The main principle underlying DynoDroid is an observe-select-execute cycle, in which it first observes which events are relevant to the app in the current state, then selects one of those events, and finally executes the selected event to yield a new state in which it repeats this process [18]. DynoDroid focuses on User Interface through the generation of UI events such as click, text box, and others similarly to what we are doing here in our study. Sometimes it could be needed for human knowledge and intelligence to generate events. For example, if it is needed to fill in a password and username in an app that needs login credentials it is better to be human-assisted, in other words, a human can easily remember and can easily recognize the password format to come up with an effective test case than random generation. In such cases, DynoDroid enables the tester or user to pause the random generation by the system and generate arbitrary events manually for the text boxes in which the overall system thereby combines the benefits of both the automated and manual approaches. Many significant features of Android are controlled with system events such as managing the volume of a speaker, notifications of battery alert, and connecting to other devices. It is difficult to cover all the system events because of the complexity and diversity of the events, it seems impractical to include the permutation of these large number of system events. This is found challenging for Aravind [18] to achieve their goals of Robustness, Efficiency, Automation, Versatility, and Black box (does the system forgo the need for app sources and the ability to decompile app binaries?)

DynoDroid works in Observe-select and Execute cycle in which Observer computes which are relevant events, events which registered a listener and provide them to the selector which selects an event from the list of events and the executor executes the event in the current given emulator state. Executer uses ADB (Android Debug Bridge) to send the events to the Emulator where the app under test is running. As mentioned in the above paragraph DynoDroid allows switching between automation and human. The executor listens to commands from a console and starts in human mode, in which it does not trigger any events and instead allows the human to exercise the app uninterrupted in the emulator until a **RESUME** command is sent from the console. **RESUME** command switches the generation of events to the machine mode and starts to generate automatically until the goal is achieved or until **PAUSE** command is received from the console. **STOP** command finally stops the DynoDroid. After then the Observer will continue to compute the relevant event and returns with a possible small set of events, the minimum number of relevant events.

According to Aravind [18], DynoDroid found to outperform Monkey on managing time to reach the peak of code coverage that DynoDroid covers lots of code faster than Monkey though it is slower in generating events which turned out 5X slower than Monkey. The reason for this delay is said to be the calling of view hierarchy after each event generated. But on the other side, reaching comparable code coverage with Monkey with a small number of inputs seems inspiring. In addition, DynoDroid is claimed to have a good automation degree in which the study by Aravind [18] comes out with the ratio of coverage achieved by the intersection of DynoDroid and Human to the total coverage achieved by Human varies from 8% to 100%.

In addition to the code coverage, DynoDroid is also concerned about the issues of fault detection. The FATAL EXCEPTION is believed to be severe and causes application crashes that DynoDroid specifically addressed only this type of exception. What they did was mining the Android emulator logs for any unhandled exceptions that were thrown from code in packages of the app under test and finally they checked for false positives manually [18]. When it comes to our study though we are also focusing on FATAL EXCEPTION similar to DynoDroid we are including fitness functions unlike measuring fault detection and code coverage as final performance evaluation of the framework. We are adding multiple test objectives which are measured with the fitness functions because it helps the STGFA-SMOG to see different criterion to ensure the quality of the application. We can use these criterion as final performance evaluation of our framework also. Unlike DynoDroid which uses random generation of test cases STGFA-SMOG use fitness functions to guide the generation as it uses search-based generation. This approach helps to save time used to find solution which are close to the test objective. In addition it is smart way to address different test objectives using the optimization algorithm, this makes STGFA-SMOG more realistic because users and testers in the real world needs a software to address different functional and nonfunctional requirements. Objective of a test is not always straight and single-objective that leads us to consider the multi-objective test case generation.

3.3 EvoDroid

EvoDroid is the first evolutionary testing framework targeted at Android. In other words, it is a tool that applied an evolutionary technique for system testing of Android apps, and it is believed that it solved the issue of applying evolutionary approaches in system testing. This is done by analyzing the ADF (application development framework) and its constraint on the way apps can be built [19]. ADF allows the programmers to extend the base functionality of the platform using a well-defined API and provides a container to manage the lifecycle of components comprising an app and facilitates communication among them. In addition, it is good to have ADF in order to have consistency among developers and for common consensus. EvoDroid is aimed at finding a set of tests that maximizes the code coverage. According to Riyadh [19], the focus of EvoDroid is on generating test cases that maximize code coverage, not on whether the test cases have passed or failed. Code coverage can be expressed as visiting the unique paths from the start node to the leaf nodes. Reaching high code coverage of apps such as ERS, a simple app used in EvoDroid requires trying out a variety of inputs which are User Interface events.

EvoDroid starts from the root node and finds a test case that makes the path until the leaf node. For example, if we have node “A” as the root node and J,K,L are leaf nodes the aim is to find a test case that actually reaches the leaf nodes. Each test is also considered as individual and its genes are app input which are specifically events in the Android case. Similar to other evolutionary based tools such as Sapienz, EvoDroid goes through mutation and cross-over to generate new offspring from their parents and to represent the better solutions throughout generations. Unlike any prior approach, EvoDroid takes each path in the CGM (Call Graph Model), breaks it into segments, and runs the evolutionary search for each segment separately [19]. CGM is the graphical representation of sequences in an app and it can be viewed as the model for function and method calls in the app. The evolutionary process continues until all the paths and their segments are covered or maybe a specified criterion is fulfilled (number of test cases, time limit). Code coverage is the focus area of EvoDroid as its fitness function and seems acceptable to have a single fitness function as the first to introduce an evolutionary approach for system-level testing. In addition, segments can be skipped if they are already covered and this seems helpful to save time and resources. For example, if a segment in a given path is already visited with some path it can be skipped in the later possible visit. Let’s say we have paths A-B-E-F and A-C-E-F-G, E-F is a segment and is repeated in both paths. In such cases, we have the ability to skip it once it has already been covered.

STGFA-SMOG is different from EvoDroid on the number of fitness function we applied. We added fitness functions such us memory usage, cpu usage, fault detection and length of test sequences on top of the code coverage to ensure the quality of the application considering different testing goals. STGFA-SMOG is flexible where developers can add new fitness functions over time and allow testers to choose different combinations of fitness functions to generate test cases for different purposes.

3.4 Sapienz

Sapienz is another search-based test generation approach for testing Android apps[8]. During the generation of test cases, Sapienz includes randomly generated atomic events in addition to the mutated solutions and solutions which are passed through crossover for evaluation of their fitness. This is important to address the diversity of the test cases and to control speedy convergence of test cases towards the fitness function. The Sapienz approach focused on the fault revealing and shortest possible test sequences because the longer the test oracle the less realistic to happen in reality. Therefore it is good to save time, the energy of developers, and memory by minimizing non-applicable long test sequences. Sapienz uses three fitness functions, aimed at maximizing code coverage, maximizing the number of crashes, and minimizing the length of the test sequence.

Sapienz uses a multi-objective (pareto-optimal) genetic algorithm to optimize the three factors. Pareto optimality is an approach to handle and balance the scores of the three fitness functions. In multi-objective search increasing on a given objective can lead to a reduction of the other objective. For example, when people need to buy a car they are concerned about the cost and quality of the car. So, they cannot reach maximum quality with very low cost, increasing in quality increases the money to spend, people want to pay a little price, at the same time they want a new model car so they have to optimize meaning they have to manage the trade-offs and end up with optimal solutions. Specifically to this context, it is not mandatory to sacrifice the length of the test sequence for the sake of maximizing fault revelation neither for maximizing code coverage. With Pareto multiobjective approach these constraints can be balanced in order to select a solution that best covers each function without harming attainment of the other functions.

In other definition, pareto optimality is a formal way to manage these trade-offs in which solution X dominates another solution Y, if for all objectives X is not worse than Y and there is one objective X is strictly better than Y. A solution belongs to the Pareto set if there is no other solution that can improve at least one of the objectives without degrading any other objective [20]. The set of solutions that cannot be dominated by others are considered as equally viable, which is named as Pareto front. Most previous frameworks such as Monkey & Dynodroid were based on random testing which didn't apply genetic algorithm & pareto Optimality while Sapienz ended up with better performance (larger effect size) than Dynodroid and Monkey [9]. As we see Sapienz focuses on the 3 fitness functions, length of the test sequence, fault detection, code coverage, and our thesis improves this by adding new fitness functions & allowing the tester to choose from different fitness functions and making it easy to implement more functions.

The search-based approach for generating the test and the usage of the Pareto algorithm to solve the optimization problem is similar to what we are developing and studying in this paper. The difference between Sapienz and the framework we are developing is the number and selection of fitness functions. Our framework, STGFA-SMOG, allows a tester to choose fitness functions of his/her choice to see which combination turns out with effective tests. This allows testers to conduct tests for different test goals by choosing fitness functions from a larger selection so they can tune test generation towards different goals. It will also allow them to implement new fitness functions & will be available open-source, which the current version of Sapienz is not. Providing it as open-source is helpful for the coming researchers to get the detailed implementation of our framework and they can reuse and modify it for different purposes of study. Sapienz takes the status of the AUT into account while generating the test case which is out of the scope of our study, STGFA-SMOG. For example, if the AUT is on the page which has a form to be filled, the Sapienz uses this status to generate the corresponding events. Though this is what we did not include in our study STGFA-SMOG is more flexible than Sapienz in that testers can select combinations of fitness functions of their choice.

3.5 GUIRipper

GUIRipper is a tool that dynamically builds a model of the AUT by crawling it from a starting state. When visiting a new state, it keeps a list of events that can be generated on the current state of the activity and systematically triggers them [21]. It uses model-based exploration, where Android testing generates a model of the GUI of the app to generate events and systematically explore the behavior of the app. These models are usually finite state machines where activities are represented as states and events as transitions. For example, if we have simply a login page the event click can be represented as a transition from state active home page to active manager page. But in the first version of this research GUIRipper, the model was EFG(Event flow Graph) which is a stateless model. The fact that Android applications are state sensitive drives the research to focus on the state machines which has dynamic abstraction and end up with MobiGUITAR tool and left the first version of GUIRipper unfeasible. MobiGUITAR produces several types of artifacts (such as crash reports, finite-state-machine models, GUI sequences, and executable JUnit test cases) that provide information useful for debugging. As an experiment by Amalfitano [22], MobiGUITAR outperforms Monkey and DynoDroid on fault detection.

Our focus area in this study, STGFA-SMOG, is in search-based exploration not model-based exploration but these are also promising Android GUI testing tools while all tools are aimed to ensure the quality of Android applications specifically the graphical user interface section since we interact with apps with this section mainly from clicking to swiping. Model-based tools generate tests from the model of the app and use this model as a guide for the generation as we have the fitness function to guide our search-based exploration. In addition, model-based test generation tools can use tester-defined stopping criteria such as 80% of code coverage whereas a maximum number of iterations are used as a stopping criterion in STGFA-SMOG.

3.6 SwiftHand

SwiftHand [23] is another model-based test generation tool that aims to maximize the coverage of the AUT. Similarly to the other tools such as MobiGUITAR, it uses a dynamic finite state machine model of the app. Automatic exploration algorithms will occasionally need to restart the app, in order to explore additional states reachable from the initial state and one of the main characteristics of SwiftHand is to optimize the exploration strategy to minimize this restart of the app while crawling. It uses machine learning to learn the UI model of the app under test and use this information to generate the test suits. SwiftHand generates only touching and scrolling UI events and can not generate system events [23]. But our framework STGFA-SMOG generates both UI events and system events such as adjusting volume. This makes our tool STGFA-SMOG all rounded and realistic to include the important system events as well as the UI events such as click and swiping. We are also guiding the exploration of the test suits in STGFA-SMOG with the fitness function by measuring the fitness value to choose the better test cases and keep them for the next generation while SwiftHand uses the learned model to generate user inputs that visit unexplored states of the app.

3.7 Empirical Studies On Test Suite Generation

In the study by Wang [24], one of the research questions was " how to efficiently combine multiple test generation tools on applicable industrial apps to achieve better coverage and fault detection than applying these tools individually?" The state-of-the-art Sapienz and the state of practice Monkey are among the evaluated and studied Android testing tools. They were able to see the difference in code coverage achieved and the number of crashes identified on different apps by several Android generation tools. To answer the research question, they come up to measure and analyze the statistics of rank-1 method and activity coverage plus rank-1 unique crashes achieved by each test generation tool on the apps. In rank-n, where "n" is the number of tools that cover the given method or activity, it could also be the number of tools that detect a given specific fault. As a result, rank-1 is to means the crash is unique and detected by one generation tool and the same is true when it comes to coverage, the given method or activity is covered with one of the generation tools only. If test generation tool A's method/activity or unique crash statistics is 'a/b' and tool B's method/activity or unique crash statistics is 'c/d', by running both tool A and tool B (i.e., combining tool A and tool B) we could achieve at least max(a+d,b +c) percent coverage of methods/activities or unique crashes that are covered or triggered by all the six test generation tools used in the experiment. Combination of Sapienz and Monkey is claimed to have the better coverage which is 90% of all covered methods by all the tools on these apps while Monkey with Sapienz and/or Stoat [25](UI test generation tool for Android apps, with model-based evolutionary testing) is a good combination for revealing more faults. Based on this analysis they provide suggestions for combining multiple tools for better coverage and/or fault detection than applying these tools individually.

An empirical study on the search algorithms conducted by Gordon Fraser and his team [26] has addressed the influence of using multi-objective optimization, rather than the generally common approach of optimizing simply for code coverage. In their study, the simple genetic algorithm they used for line coverage uses mutation, and crossover which is similar to NSGA-II. The only difference is the NSGA-II used in the study additionally uses a fitness function minimizing the number of events executed in the test suite and maximizing the number of crashes produced by the test suite. In other words, the former optimizes only line coverage while the latter optimizes line coverage, the number of events, crashes. NSGA-II as used in Sapienz selects individuals for reproduction randomly whereas the simple genetic algorithm uses standard fitness proportionate selection. Finally, the number of unique crashes and code coverage is measured to determine the difference between the multi-objective and single-objective search approaches. Mean values of code coverage for each app are calculated and it turned out that NSGA-II achieves better coverage for the apps **easy_xkcd** and **redreader**. On the overall result, the study has shown NSGA-II achieves about 1% more line coverage on average. Based on the statics of the number of unique crashes on each app there is only one statistically significant difference in mean crashes, i.e., for the **markor** app where NSGA-II discovers about 0.2 crashes more than the simple genetic algorithm on average. Overall NSGA-II discovers about 0.24 crashes more than the simple genetic algorithm. In conclusion, the researchers believe the NSGA-II triggers a few more unique crashes than the simple genetic algorithm because it keeps test suites containing crashes in its population.

Another empirical study on Sapienz [27] is conducted to answer the research questions such as the contribution of the Algorithm (NSGA-II) and the use of motif genes on the effectiveness of Sapienz which was measured with statement coverage in this specific context. NSGA-II with and without motif genes was implemented to show if motif genes make a difference. The same is done to the random generation, random search with and without motif genes is compared for checking whether motif genes contribute to major gains in cases where an evolutionary algorithm is not applied. In both cases, NSGA-II and Random Search improve their effectiveness for test cases with motif genes based on the statics of the results. The statistical analysis also shows the overall statement coverage by NSGA-II turns out better than the other algorithms though the statistical significance is not observed on the comparison with random search, an algorithm that was second for statement coverage. In other words, evolutionary algorithms do not have a significant contribution to the effectiveness of test generation for Android. In broadway, [28] took the metric code coverage to investigate whether the different level of coverage has different fault detection ability, for example, if statement coverage finds more crash than method or vice versa. This study is also conducted with Sapienz that ended up that the joint usage of several granularities of code coverage metric leads to discovering more bugs though there is no difference among one another. A study on code coverage by a team in the University of California [23] aimed to reach better coverage quickly by learning and exploring an abstraction of the model of GUI of an android app was one of the interesting researches regarding android testing with code coverage metrics. This study was initiated with the draw back of random testing tools such as Monkey that generates random inputs regardless of the structure of the GUI including test inputs that touches random coordinates on the screen and this approach is considered as a problem to reach high code coverage quickly. SwiftHand was used to generate the sequences test inputs while machine learning was used for learning the model of the GUI.

Zeng et al. [29] asked the question "Are we really there yet in an industrial case?" in terms of the applicability of automated test generation for Android apps in an industrial setting. In their study, they investigated the first industrial applicability of the Monkey tool and its limitations in an industrial setting. They found that Monkey achieves low line coverage and low activity coverage for the particular AUT they used. Based on their study they identified two limitations for Monkey achieving low code coverage; (1) lack of knowledge of the location of the widgets on a screen, and (2) lack of knowledge about the context or state of the app. To address these limitations they developed an enhanced version of Monkey by incorporating widget awareness and state awareness with guided exploration and achieved better code coverage than the original Monkey. Another previous work related to Zeng et al. study is the empirical study done by Choudhary et al. [21] where the researchers asked the question "Are we there yet?" in terms of the applicability of automated test generation tools for Android apps. Their case study was mainly focused on publicly available test generation tools applied to open-source Android apps as the AUT. They compared the existing test input generation tools for Android in terms of ease of use, ability to work on multiple platforms, code coverage, and ability to detect faults. The study result showed that Monkey outperforms the other existing tools considered in the study in terms of the comparison criteria specified.

Different evolutionary algorithms have been used to generate unit test suites for code coverage. An empirical study carried out by Campos et al. [30] investigated the influence of the specific flavor of evolutionary algorithms in the whole test suite generation. They empirically evaluated different algorithms (such as monotonic genetic algorithm (GA), standard GA, steady-state GA, many-objective sorting algorithm (MOSA), DynaMOSA, $1 + (\lambda, \lambda)$ GA, $\mu + \lambda$ EA, Random search, and Random testing) using the EvoSuite tool applied on open-source Java classes. Their study result showed that $\mu + \lambda$ EA performed better than other complex search algorithms. Generally, search algorithms that use test suite archives performed better than random search and random testing, and many-objective search algorithms achieved better branch coverage than single-objective search algorithms. Hence, they concluded that the choice of the specific flavor of EA has a significant influence on the effectiveness of the whole test suite generation. Vogel et al. [31] asked the question "Does diversity improve the test suite generation for mobile applications?" in terms of code coverage, test suite size, and fault detection. To answer this question they carry out the fitness landscape analysis of Sapienz in terms of genotypic diversity of solutions. Based on their empirical analysis they found that a decrease in the improvement of the test suites generation and degradation of diversity in all solutions in the search space over generations. To alleviate these limitations, they developed an extended version of Sapienz by incorporating algorithms that maintain the diversity of the population in the search space throughout the generations. However, even though the new version of Sapienz revealed more faults than Sapienz the results do not show statistically significant differences in terms of fault detection and code coverage. Hence, the diversity promotion approach integrated with Sapienz does not results in the expected effect.

It is important to understand the underlying structure of the fitness landscape in order to better understand the performance of search algorithms. Hence, Albunian et al. [32] carried out an empirical investigation to understand the cause and effect of the fitness landscape in unit test generation using search-based testing on Java classes. In their study, they studied the two main properties of the fitness landscape (i.e. **ruggedness** and **neutrality**), and their causes and effect. Based on their study, they found that ruggedness property provides informative landscapes and results in better performance of the search while the neutrality property provides landscapes harder to cover and results in low performance in the search. Their study result showed that the fitness landscape is highly dominated by the neutrality property. The main causes for the neutrality dominated fitness landscape are (1) accessibility of the methods, (2) failing to satisfy preconditions of the method calls, and (3) boolean comparisons in the code. Finally, to improve the difficulties of the fitness landscape they suggested incorporating inter-procedural distance information and testability transformations. Another empirical study done by Yasin et al. [33] investigated the impact of event sequence length in terms of code coverage and fault detection. Their study results showed that long events sequence has a small positive effect compared to short events sequence on code coverage and fault detection. They also compared different test input generation tools for Android apps in terms of method coverage, activity coverage, and fault detection, and Sapienz outperformed all the other tools considered under the study. They also found that most of the tools studied were able to identify faults triggered by user events but not by system events.

3.8 Summary

To summarize, through a review of related works, this study identified that multi-objective generations created to test Java-based desktop applications use a different combination of fitness functions to guide the search process and generate effective test cases that cause crashes to happen. This logic also applies to multi-objective generations used in Android GUI-based testing. Salahirad et. al [6] carried out a research study on EvoSuite [34], a multi-objective test generator for unit testing to investigate which combination of fitness functions available in EvoSuite generates effective test suites in terms of crashes they cause to happen. This research work motivated us to carry out a similar investigation on multi-objective generation tools for Android GUI-based testing. However, no mature test generation tools with support for multi-objective generation for Android GUI-based testing are publicly available so far. Furthermore, the tools developed for Java-based desktop applications cannot be used for this study as they are developed for white-box testing.

Moreover, the review of related works allowed us to understand the trending approaches used in Android GUI-based test generation. It also helped to understand which part of Android apps testing is extensively studied and which part is immature and still open for investigation. From the literature review, it is identified that most of the existing Android GUI-based test generation tools are not search-based tools. For example, the state-of-the-art [18] and state-of-practice [16] tools use random fuzzing approach to generate test cases and they do not use fitness functions.

The empirical study [26] gives an insight into implementing different search algorithms in a framework aimed to better understand how search algorithms influence the effectiveness of search-based testing for Android apps. It shows how various search algorithms affect the quality of test suites. In addition, it brings an understanding of how the output data of the experiment has to be managed and how the statistic should be analyzed. An interesting point from the empirical study [24] presented above is that combining different Android test generation tools on apps has a positive impact on the quality of test suites. In general terms, the study shows how Android UI test generation tools could be improved to reveal more crashes that end up with a better quality of tests for industrial apps.

Hence, in this research, a tool called STGFA-SMOG is developed that allows practitioners to choose different combinations of fitness functions easily and carry out Android GUI-based testing. STGFA-SMOG is structured and modularized in a way that it can be easy to understand and modify (e.g. adding new testing goals) by researchers in the future. Many companies still use manual-based testing and unable to use automatic tools. The reason could be the lack of flexibility of the automatic test generation tools to configure, modify and understand in a way that the company needs to apply them. STGFA-SMOG is flexible and allows researchers or practitioners to choose the fitness functions they need through a single interface without modifying the internal code. The tool is designed to be easy for adding new fitness functions over time, which is not applied in the latest search-based test generation tool Sapienz and the implementation of the framework is organized considering the possible future researches. Modularizing and structuring the fitness functions based on their functionality helps in the ease of adding new fitness functions in a separate module without restructuring what is already done. This minimizes the time and energy wastage used to understand complex function calls and to identify where the appropriate place to add the new feature or new fitness function for future researchers.

4 Approach

4.1 Framework

To investigate the effect of varying different combinations of fitness functions on the automated generation of effective and fault-detecting test cases, a framework called STGFA-SMOG is proposed in this research. The search-based test generation approach is used to develop the framework with support for a multi-objective generation. Multi-objective generation is the generation of test cases or test suites that can achieve multiple objectives simultaneously. The framework is initially developed to support four fitness functions. In the subsequent iterations of the development, more fitness functions are added such as code coverage, battery usage, and network usage. STGFA-SMOG supports testing Android applications available in the APK format. In search-based test generation, the initial step is to define the genotype representation of the solution space of the problem domain.

The framework starts by generating semi-random atomic events which are evolved through generations until the allocated resource is consumed (i.e. maximum number of generations). The individuals in the population represent individual test cases, each containing a random number of a sequence of atomic events that ranges between 20 up to 50 atomic events. Every generation's population is encoded to contain 20 individuals by default. Since it is user-selectable, the user can change the value to the intended number of individual test cases the population should represent. The overall architecture of the framework is depicted in Figure 4.1. When a user selects a code coverage objective in combination with the crash and test case length objectives in the fitness function configuration, the AUT is instrumented using ACVTool[1] before test case generation is started. In the instrumentation, probes are inserted into the bytecode of the AUT so that ACVTool can track their call and measure code coverage at the classes, methods, and instructions level. If the code coverage objective is not selected, the AUT is not instrumented and the test cases are run over the original AUT.

As shown in the figure, the *event generator* module generates atomic events, and the *test case* module organizes them as individuals or test cases and sends them to the *fitness evaluator* module. The *fitness evaluator* part of the genetic algo-

[1]https://github.com/pilgun/acvtool

rithm sends the test cases to the *test runner* module. The *test runner* module in turn sends the events one by one to the AUT running on the emulator via Android Debug Bridge (ADB), where ADB is a command-line tool that is used to communicate with Android-based devices, and each test case's fitness is computed by the fitness functions. After the test cases are executed and evaluated for their fitness, STGFA-SMOG uses the NSGA-II algorithm to select the best candidate solutions or individuals and pass them to the *variation operator* module. The variation operator performs crossover, mutation, reproduction and introduces randomly generated new individuals to generate offspring and drive the population.

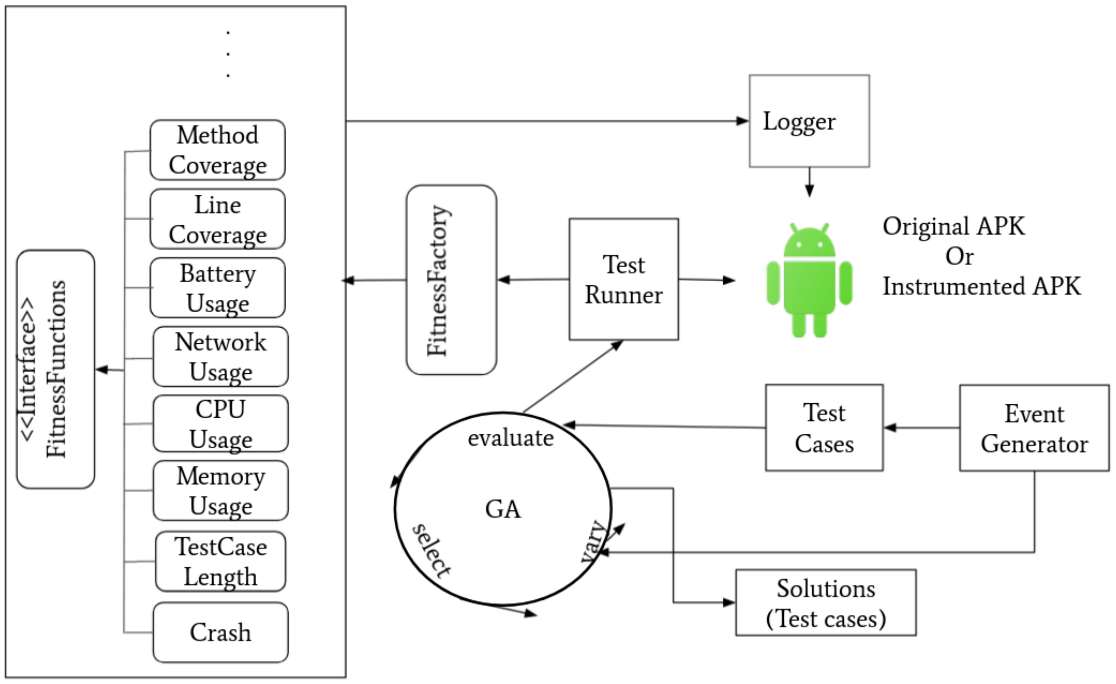

Figure 4.1: Architecture of STGFA-SMOG

4.2 STGFA-SMOG Parameters

STGFA-SMOG has different parameters (see Appendix A) that need to be set by the user. The parameters can be seen as three parts. The first part contains the parameters used to set different genetic algorithm parameters. Using $SEQUENCE\_LENGTH\_MIN$ and $SEQUENCE\_LENGTH\_MAX$ user can set the minimum and the maximum number of atomic events a test case can contain respectively. The $POPULATION\_SIZE$ and $OFFSPRING\_SIZE$ parameters are used to set the number of test cases (individuals) that a population and offspring in each generation should contain. The maximum number of generations the genetic algorithm to evolve is set using the $NGENERATION$ parameter. There is no evidence that shows the genetic algorithm parameters are constrained by the search budget.

The percentage composition of the new offspring to be created using crossover, mutation, and reproduction is specified using $CXPB, MUTPB$, and $REPROPB$ respectively. The remaining percentage (i.e. $1-(CXPB+MUTPB+REPROPB)$) of the offspring is made of new randomly generated test cases. There are other two important parameters that allow the user to configure the fitness functions to combine and specify their corresponding fitness weights. The $FITNESS\_FUNCS$ parameter as shown in Appendix A is a Python list that allows users to specify which combination of fitness functions to use. According to the combination of fitness functions specified, the $FITNESS\_WEIGHTS$ parameter allows the user to specify their corresponding fitness weights. If the fitness function is a maximization function, its fitness weight is 1.0. Otherwise, its fitness weight is -1.0.

The second part contains the parameters used to specify after how many atomic events execution of a test case (i.e. interval), the CPU, memory, network, and battery usage of the AUT is retrieved to evaluate the corresponding test case's fitness values. By default, the CPU, memory, network, and battery usage of the AUT is retrieved after every three atomic events of a test case are executed. The third part contains the parameter used to specify the number of top best test cases (solutions) that ever exist throughout the whole generations that the STGFA-SMOG to output at the end of the generation.

4.3 Problem Representation

In search-based test generation, the first step is to define a representation of the valid candidate solutions for the problem at hand in a way the search algorithm can manipulate and use them [1]. In STGFA-SMOG, the candidate solutions called individuals are encoded to contain a sequence of atomic events (e.g. click, swipe, and input text). In the context of software testing, the sequence of atomic events sent to test the AUT is called a test case. The representation of the individuals or candidate solutions generated by STGFA-SMOG is shown in Figure 4.2. As shown in the figure, an individual or a test case is made of a sequence of multiple atomic events, and an atomic event is composed of an action and parameter values. An action can be **tap, keyevent, or swipe** and their corresponding parameter values can be ***20 30, KEYCODE_BACK, and "admin"*** respectively. A solution, which is a test case T, is represented as a set of atomic events e_i. So, given a test case T with length $|T| = n$, T will contain $T = \{e_1, e_2, e_3, \ldots, e_n\}$ where e_i is an atomic event. STGFA-SMOG generates a set of these test cases which is called population in evolutionary algorithms. Each individual consists of one chromosome which is a test case in STGFA-SMOG. Each chromosome contains multiple genes called atomic events, which are randomly generated and ordered to create the sequence. Individuals encoded as test cases allow the population to evolve many times with minimal computational resources compared to individuals encoded as test suites.

Each event in a test case represents an atomic event that has an action type $T(e_i) \in A$, where A is a finite set of Android UI events or system events. Currently, seven types of android UI events and system events are represented in STGFA-SMOG, and they are discussed in the next paragraphs. There is no standard categorization of Android UI and system events. We classified the Android events applied in our framework into seven categories based on their similarity in functionality.

Android system events: These types of events represent the events that perform system-wide operations such as home button press keyevent, back button press keyevent, increase, decrease, or mute volume keyevent, and end call keyevent etc. Some of the android system events represented in our framework are listed as follows.

KEYCODE_HOME, KEYCODE_BACK, KEYCODE_CALL,
KEYCODE_ENDCALL, KEYCODE_VOLUME_DOWN,
KEYCODE_VOLUME_MUTE, KEYCODE_MUTE,
KEYCODE_VOLUME_UP

Example of android system event: adb shell input keyevent KEYCODE_HOME

Major Navigation events: These events are events that are used to navigate major navigation areas of android devices such as menu button press keyevent, keyevent to select system-defined functions, keyevent that put the device to sleep, and directional pad center keyevent. The major navigation events represented in the framework are:

KEYCODE_MENU, KEYCODE_SOFT_RIGHT,
KEYCODE_DPAD_CENTER

Major navigation events are rarely used events and are not recommended to send many such events during testing [16]. So, such atomic events are rarely sent to the AUT in our framework as well.
Example of major navigation event: adb shell input keyevent KEYCODE_MENU

Navigation events: These event types allow you to automatically navigate around the UI of the AUT. They are keyevent used to move DPad down, up, right, and left. The most common navigation events represented in our candidate solutions are:

KEYCODE_DPAD_UP, KEYCODE_DPAD_DOWN,
KEYCODE_DPAD_LEFT, KEYCODE_DPAD_RIGHT

Example of navigation event: adb shell input keyevent KEYCODE_DPAD_UP

Input text events: This type of event is a command that allows you to automatically enter text to text fields in the UI forms of the AUT. The source of the text input can be a keyboard or touchscreen and the default one is touchscreen. This text input event is represented to contain randomly generated characters with a length ranging from 5 to 10 characters.
Example of input text event: adb shell input text username

Tap event: This type of event allows you to automatically perform click or long-press action on the UI of the AUT running on an emulator or real device. This type of event requires the coordinate points that lie in the application display area of the emulator or real device where the AUT is running on. In STGFA-SMOG, the screen size of the emulator or real device is first determined and the coordinate points are generated accordingly.

Example syntax of tap event: adb shell input tap <x> <y>

Swipe event: Swipe event allows to automatically press down over an element of the UI and swipe horizontally or vertically. This type of event requires coordinate points where the swipe event begins and ends. Similar to the case of tap, the starting and ending coordinate points of swipe events are generated based on the screen size of the emulator or physical device.
Example syntax of swipe event: adb shell input swipe <x1> <y1> <x2> <y2>

Drag and drop event: This event allow users to move data from one view to another view using automated drag and drop gesture. Similar to the swipe event, this event also takes x and y coordinates where the drag begins and the drop ends. Syntax of drag and drop event: adb shell input draganddrop <x1> <y1> <x2> <y2>

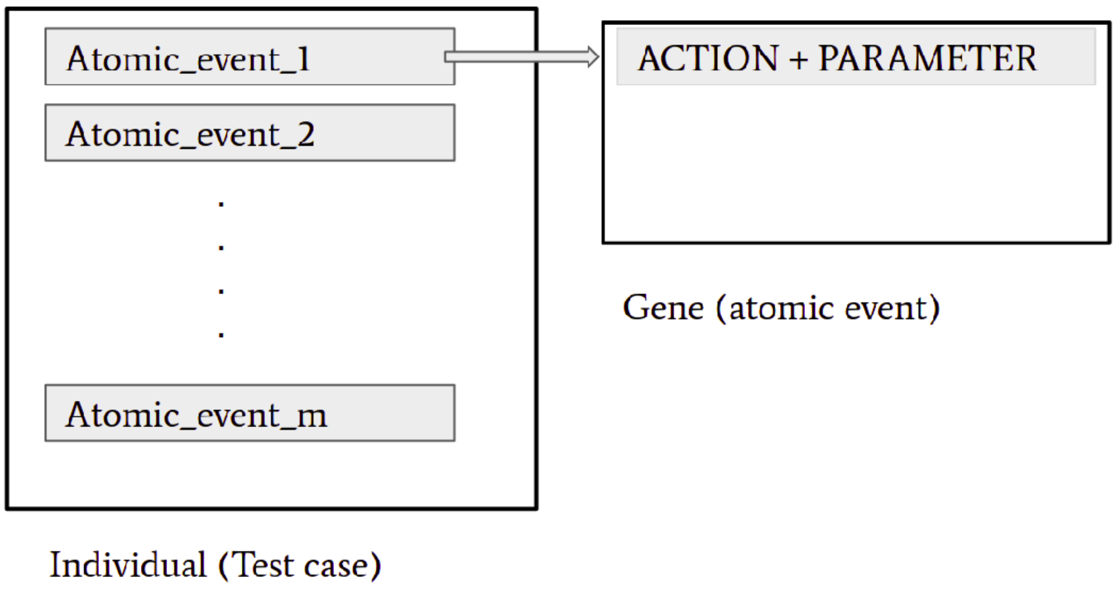

Figure 4.2: Individual's genotype representation

4.4 NSGA-II

To ensure that this paper is self-contained about the NSGA-II adopted to STGFA-SMOG, this section briefly discusses NSGA-II; for more details please reference [35]. NSGA-II is a multi-objective evolutionary algorithm (EA) that uses non-dominated sorting and crowded-comparison operator ($\prec_n$) instead of sharing parameter (σ_{share}) to find Pareto-optimal solutions to a multi-objective problem [35]. A Pareto-optimal solution is one where no fitness function can be further optimized without negatively affecting another fitness function value. For a given population the NSGA-II algorithm sorts the individuals in the population into non-domination levels (or non-domination fronts) using the non-domination strategy. Non-domination strategy is the count of how many individuals dominate an individual I and individuals with count zero are ranked in the first non-domination level, which means they are the best individuals in the current generation [35]. Individuals dominated only by one individual are ranked in the second non-domination level. If individuals have the same non-domination count, NSGA-II calculates the crowding distance of the individuals, and their crowding-distance is compared during the selection process. That is, an individual with a higher crowding distance (less dense region) is selected first. Crowding-distance is the measure of the crowdedness of a solution by other solutions. An individual with a smaller value of this distance is somehow more crowded by other solutions. That is, the individual is surrounded by more other solutions.

NSGA-II algorithm is good at preventing the loss of elitist solutions which were found in the previous generation and maintaining the diversity of the Pareto-optimal solutions. To maintain elitist solutions, NSGA-II combines the parent population and the offspring population and performs a fast non-dominated sorting on the combined population. This strategy reduces the loss of elitist solutions which were found in the previous generation (i.e. parent population). The crowded-comparison operator guides the selection process in each generation toward a uniformly spread-out Pareto-optimal front. When two individuals have the same non-domination rank (non-domination level), the operator compares the two individual's crowding distance and selects the one with a higher crowding distance (i.e. less dense region). This strategy enabled the NSGA-II algorithm to achieve diversified Pareto-optimal solutions for a given multi-objective problem.

In NSGA-II algorithm, every individual I in a population P has two attributes:
1. non-domination rank - I_{rank}
2. crowding distance - $I_{distance}$

The NSGA-II algorithm selection strategy is stated as follows. When two individuals are with different non-domination ranks, the crowded-comparison operator selects the individual with a lower non-domination rank. However, if two individuals have the same non-domination rank, the crowded-comparison operator compares their crowding distance and favours the individual with higher crowding distance.

Hence, due to the fact that NSGA-II has better computational complexity, handles the loss of elitist solutions, and provides diversified Pareto-optimal solutions, we adopted it in the STGFA-SMOG implementation. Moreover, it is also a widely used algorithm in search-based test generation research studies (such as in References [9, 8, 26]) this time.

4.5 Initial Population

After the individuals' representation is defined as a sequence of atomic events collectively called a test case, the initial population P_0 is created by randomly generating a sequence of atomic events which denote an individual or a test case. The population is defined to contain 20 randomly generated individuals. The initial population is run on the AUT and each individual's fitness value is computed against the number of crashes they triggered, CPU used by the AUT, memory used by the AUT, and their test case length. Once the individual's fitness is evaluated, variation operators (i.e. crossover, mutation, reproduction, and addition of new randomly generated individuals) are used to create offspring population of the same size as the original population. The new offspring population created is then handed in to the next generation, which is discussed in the next section.

4.6 Generations of the Algorithm

STGFA-SMOG iterates over a number of generations, in which the number of generations is monitored by specifying the maximum number of generations. In the first generation, the new offspring population generated from the initial population is run on the AUT and each individual's fitness value with respect to each objective function is calculated.

To select the best candidates from the population, the selection operator available in the NSGA-II algorithm [35] is used. In each generation, the parent and the offspring are combined, which is called the $\mu + \lambda$ genetic algorithm, after the fitness of the offspring population is evaluated. Through non-dominated sorting, the selection operator ranks all individuals into different Pareto front levels. To preserve diversity and avoid conflict of individuals having the same Pareto front level, the selection operator calculates each individual's crowding distance. Crowding distance computes an estimate of the density of candidate solutions surrounding a particular solution or individual in the population. The individuals for the next generation are selected from the Pareto front according to the order of their front level. The next population is created by selecting the test cases with higher non-domination rank (i.e. smaller front level value), and if test cases have the same non-domination rank, the selection operator favors the test case with greater crowding distance (i.e less dense test case). By using the Pareto-optimal approach, STGFA-SMOG selects shorter test cases without sacrificing longer test cases, when they are the only ones that detect faults, or when they are necessary to achieve higher CPU usage, memory usage, battery usage, network usage, or code coverage. However, STGFA-SMOG progressively replaces the longer test cases with shorter test cases when they are equally good.

In each generation, the parent population and the offspring population are combined so that the diversity of the Pareto-optimal solutions are maintained. The same procedure as described in the above paragraph continues in each generation until the maximum number of generations is reached. When the maximum number of generation is reached, STGFA-SMOG outputs the population of the final generation to a file, which include the best test cases discovered during the test generation process, and log history of the test cases. A user can set any positive integer as a maximum number of generations. The larger the number of generation search budgets used the longer the computational time it takes when generating the test cases.

4.6.1 Fitness Evaluation

To guide the search process, eight fitness functions are implemented. The fitness evaluation is recorded as a tuple for each of the objectives and the framework supports the tester to select different combinations of fitness functions from the available ones as well as add new fitness functions over time. The minimum and maximum configuration required on the number of fitness functions to be combined is *one* and *three* fitness functions respectively. NSGA-II is not efficient (struggles) when applied to more than three combination of fitness functions [36, 37, 38]. The factory method design patterns is used to implement the support of user-selectable fitness functions and to support the addition of fitness functions at a later time.

In STGFA-SMOG the fitness functions are implemented as separate classes. However, the framework does not know which fitness functions to use in advance. The fitness functions are user-specified according to the objective of the user. So, which of the fitness function classes to instantiate is not known until the framework is run by specifying the intended fitness functions. To solve this challenge, the factory method design pattern is used that allows to dynamically instantiate the appropriate fitness function classes based on the user input of the fitness functions specified. Factory method is a creational design pattern that provides an interface for creating an object but it defers instantiation to the subclasses to decide [39]. The factory method enables to call all fitness functions from one interface and the interface lets the subclasses decide which fitness function classes to be instantiated corresponding to the fitness functions selected by the user.

The fitness functions that STGFA-SMOG supports are listed below.

- Number of crashes – maximized
- Length of test cases – minimized
- CPU usage – maximized
- Memory usage – maximized
- Network usage – maximized
- Battery usage – maximized
- Line coverage – maximized
- Method coverage – maximized
- Class coverage – maximized

To calculate the number of crashes triggered by each test case, the crash fitness function counts the number of "AndroidRuntime: FATAL EXCEPTION" types of crashes, which are Java and Android specific exceptions, generated by the AUT. Native crashes or exceptions generated by native codes (i.e. codes implemented in C/C++) are not handled by the framework and it is out of the scope of this study. STGFA-SMOG continuously monitors and checks if the AUT crashes. When the AUT crashes, STGFA-SMOG restarts it and continues running the atomic events. The CPU and memory usage of the AUT is calculated by querying the CPU and memory used by the AUT after every three (default configuration) atomic events of a test case are executed (see Algorithm 2) and the values obtained are accumulated. However, this is user-selectable and the user can configure it to the required interval.

After the sequence of atomic events available in one test case are finished, the CPU and memory fitness value of that corresponding test case is calculated by taking the average of the accumulated CPU usage and memory usage (see Line 31&34 in Algorithm 2 respectively). In this manner, the fitness value of each test case available in the population is calculated. The test case length fitness function computes the fitness value of each test case by counting the number of atomic events each test case contains. Network usage is the amount of data an Android application sends and receives while running. Battery usage measures the amount of battery on a device or emulator the app consumes. Network and battery usage are computed in a similar procedure as CPU and memory usage are computed. Algorithm 2 is part of Algorithm 1 and fits to Line 8&12.

For the code coverage measures, a third-party Android code coverage tool in black-box testing called ACVTool [40] is used. ACVTool instruments the bytecode of the APK file by inserting probes to track code coverage at method and instruction levels. So, the method and instruction coverage implemented in STGFA-SMOG are at the bytecode level not at the source code level. There is a difference in measuring the statement and method coverage of an Android app at the source code and at the bytecode levels. Because the instructions and methods within the bytecode may not exactly correspond to the statements and methods within the original source code [40]. This is because a single statement within the original source code may correspond to several instructions within the bytecode as cited by Pilgun et. al [40]. Moreover, the compiler may optimize the bytecode so that the number of methods is different, or the control flow structure of the app is altered [41, 42].

The STGFA-SMOG developed generates test cases for Android GUI-based testing. The test cases can be used to trigger crashes, identify excessive usage of CPU, memory, battery, and network. STGFA-SMOG is run to generate test cases in order to trigger crashes in the AUT. It can be run repeatedly until more crashes are identified in the AUT. The test cases generated in this process are reusable and can be used to test if the crash found is fixed. After the crashes found in the AUT are fixed, the test cases can also be used for regression testing. The test cases can be reused to test if the AUT is working as expected after changes are made to fix the crashes found. It can be run every time the Android app is updated or less often depending on the interest of the developer. A small sample of data is provided in Tables 6.17 and 6.18 that show how long STGFA-SMOG runs for each fitness function configuration in two search budgets.

Algorithm 1: Overall algorithm of STGFA-SMOG

```
1  Input: AUT $A$, selected fitness functions $F \leftarrow \{f_1, f_2, \ldots, f_n\}$, crossover
     probability $cp$, mutation probability $mp$, reproduction probability $rp$, diversity
     probability $dp$, population size $p_{size}$, max generation $g_{max}$
2  Output Pareto Front $PF$, Test Report $TR$
3  generation $g \leftarrow 0$;
4  $P_g \leftarrow \emptyset$;
5  boot emulator $E$;
6  install $A$;
7  initialize population $P_0$;                              semi-random events
8  evaluate $P_0$ with $F$ and update $(PF, TR)$;            (e.g. see Algorithm 2)
9  while $g \leq g_{max}$ do
10     $g \leftarrow g + 1$;
11     $OF \leftarrow testcaseVariation(P, cp, mp, rp, dp)$;
12     evaluate $OF$ with $F$ and update $(PF, TR)$;         (e.g. see Algorithm 2)
13     $NDF \leftarrow \emptyset$;              initialize non-dominated Pareto fronts (NDF)
14     $NDF \leftarrow sortNonDominated(P \cup OF, p_{size})$;
15     $P_g \leftarrow \emptyset$;
16     for each front level $\mathcal{F}$ in $NDF$ do
17         if $|P_g| + |\mathcal{F}| \geq p_{size}$ then
18             break;
19         else
20             compute crowding distance for $\mathcal{F}$;
21             $P_g \leftarrow P_g \cup \mathcal{F}$;
22         end
23     end
24     $sortCrowedDistance(\mathcal{F}, \prec_n)$;
25     $P_g \leftarrow P_g \cup \mathcal{F}[1 : (p_{size} - |P_g|)]$
26 end
27 return $(P_g, PF, TR)$
```

Algorithm 2: CPU, Memory, Network, and Battery usage computation

```
Input: Test case $T$
Output: CPU usage $total-cpu-used$ and memory usage
  $total-memory-used$
$cpu-info-counter \leftarrow 0$;
$mem-info-counter \leftarrow 0$;
$net-info-counter \leftarrow 0$;
$batt-info-counter \leftarrow 0$;
$total-cpu-used \leftarrow 0$;
$total-memory-used \leftarrow 0$;
$total-network-used \leftarrow 0.0$;
$total-battery-used \leftarrow 0.0$;
for $index, atomicevent\ in\ enumerate(T)$ do
    $run(atomicevent)$;            run atomic event via ADB
    if $(index+1)\%3 = 0$ then
        $total-cpu-used \leftarrow total-cpu-used + getFitnessValue()$;
        $cpu-info-counter \leftarrow cpu-info-counter + 1$;
    end
    if $(index+1)\%3 = 0$ then
        $total-memory-used \leftarrow total-memory-used + getFitnessValue()$;
        $mem-info-counter \leftarrow mem-info-counter + 1$;
    end
    if $(index+1)\%3 = 0$ then
        $total-network-used \leftarrow total-network-used + getFitnessValue()$;
        $net-info-counter \leftarrow net-info-counter + 1$;
    end
    if $(index+1)\%3 = 0$ then
        $total-battery-used \leftarrow total-battery-used + getFitnessValue()$;
        $batt-info-counter \leftarrow batt-info-counter + 1$;
    end
end
if $cpu-info-counter \neq 0$ then
    $total-cpu-used \leftarrow total-cpu-used/cpu-info-counter$
end
if $mem-info-counter \neq 0$ then
    $total-memory-used \leftarrow total-memory-used/mem-info-counter$
end
if $net-info-counter \neq 0$ then
    $total-network-used \leftarrow total-network-used/net-info-counter$
end
if $batt-info-counter \neq 0$ then
    $total-battery-used \leftarrow total-battery-used/batt-info-counter$
end
return $total-memory-used, total-cpu-used$
```

4.6.2 Candidate Selection

To select the best test cases, STGFA-SMOG combines the evaluated offspring and the parent population. The combined population is sorted based on the non-dominated sorting strategy into non-domination levels. The test cases that belong in the first non-domination level F_1 are first selected and added to the next generation's (let g_1) population (i.e. P_1). If the size of the P_1 is not full according to the specified size of a population, the test cases in the second non-domination level F_2 are selected and added to P_1 (see Line 21 in Algorithm 1).

When P_1 is not full but cannot accommodate all the test cases in the current non-domination level (let F_l) (see Line 17 in Algorithm 1), the solutions in F_l are sorted in descending order of their crowding distance and if the P_1 is left with k solutions to be filled, the first k solutions from F_l are added to P_1. Note that, every time the test cases in a non-domination level F_i are selected and added to P_1, their corresponding crowding distance is computed (see Line 20 in Algorithm 1). The selection process continues in a similar way until P_1 contains the required number of individuals set by the user. The best solutions selected are then subjected to variation operations (see section 4.6.3) in order to create new offspring population.

In each generation, the parent population and the offspring population are combined to select the best candidate test cases. Combining parent population and offspring population enables to reduce the loss of elitist test cases found in the previous generation and maintain the diversity of the Pareto-optimal solutions. The same procedure as described in the above paragraphs is applied in each generation to select the best test cases for the next population until the maximum number of generations is reached.

4.6.3 Variation Operators

Variation operators are used to generate new offspring from the parent individuals through mutation, crossover, reproduction, or creation of randomly generated new individuals. The variation operation algorithm used in our framework is shown in Algorithm 3. The variation operators implemented in STGFA-SMOG works at the test case level because individuals are encoded as single test cases. STGFA-SMOG applied fine-grained mutation, crossover, and reproduction operators. A randomly generated new individuals are also introduced in addition to the individuals obtained through crossover, mutation, and reproduction. This approach allows to maintain the diversity of the population.

STGFA-SMOG allows the user to set the percentage composition of the new population to be created through crossover, mutation, reproduction, and addition of new randomly generated individuals. STGFA-SMOG's default configuration sets $30\%, 30\%, 15\%$ of the new population is obtained by performing crossover, mutation, and reproduction respectively on the top-scoring individuals. The remaining 25% of the new population is made up of new randomly generated individuals. There is no standard way that guides a user on how to set these variation operator's percentage compositions. A user can configure the variation operator's percentage composition based on its interest. However, to know the globally better configuration for these variation operator's percentage composition, we need further investigation with different configurations.

As shown in Algorithm 3, to perform the variation operations STGFA-SMOG generates a random number between 0 and 1. If the random number generated is less than the crossover probability (see Line 6 in Algorithm 3), two individuals are selected at random from the top-scoring individuals and the two individuals are mated using a uniform crossover operation. With uniform crossover, for each atomic event in the two individuals, a random number between 0 and 1 is generated and if the number is below the independent probability specified, the atomic event at the specific index in the two individuals are swapped. This process continues until all atomic events in the individuals are traversed. Finally, a random number between 0 and 1 is generated, and the first child obtained through the uniform crossover is appended to the offspring population if the result is less than 0.5 and the second child is discarded. Otherwise, the second child is appended to the offspring population, and the first child is discarded. The crossover operator performs a uniform crossover operation to achieve inter-individual variations across the population.

When the randomly generated number is greater than the crossover probability and below the sum of crossover probability and mutation probability (see Line 11 in Algorithm 3), one individual is selected at random from the top-scoring individuals and the order of the atomic events in the selected individual are shuffled at random. The shuffling function selects two indices or atomic events at random and generates a random number between 0 and 1. When the random number generated is less than the independent probability *indp* specified, the position of the two atomic events is swapped. Otherwise, the position of the atomic events is kept as it is. This operation mutates the order of the atomic events in the individual. After this operation, the action type of each atomic event in the individual is extracted and each atomic event is recreated by randomly generating new parameter values corresponding to their action type. For example, if the action type extracted is **tap**, it takes coordinate points $< x >, < y >$ on the device or emulator screen as parameter values. So, its parameter values are changed by randomly generating new coordinate points. After the mutation operation is done, the child is appended to the offspring population. The mutation operation is applied to introduce a variation in the internal nature of an individual.

The randomly generated number could be greater than the sum of crossover probability and mutation probability and less than the sum of crossover, mutation, and reproduction probability (see Line 22 in Algorithm 3). In this case, the variation operator selects one individual at random from the top-scoring individuals and append it without making any changes. Reproduction operation is used to leave some top-scoring parent individuals unchanged and pass them to the offspring population of the next generation. Finally, when the randomly generated number is greater than the sum of the crossover, mutation, and reproduction probability (see Line 25 in Algorithm 3), the variation operator randomly generates a new individual and appends it to the offspring population.

During evolutionary computation maintaining diversity is also helpful to widen the possibility of getting an optimal solution to the problem at hand. To achieve population diversity we introduced randomly generated new individuals into the offspring. This helps the solution space from being dominated by the type of events that appear in the parent individuals. To perform crossover, mutation, reproduction, and introduce a new individual, user-configurable probabilities for each operator are defined and the operations are performed accordingly as described in the above paragraphs.

Algorithm 3: Test case variation operator

```
Input: Population $P$, crossover probability $cp$, mutation probability $mp$,
  reproduction probability $rp$
Output: Offspring $OF$
$OF \leftarrow \emptyset$;
for $i$ in $range(0, |P|)$ do
    generate $r \sim \cup(0, 1)$;
    if $r < cp$ then
        randomly select parent individuals $x_1, x_2$;
        $x'_1, x'_2 \leftarrow uniformCrossover(x_1, x_2)$;
        $OF \leftarrow OF \cup x'_1$;
    end
    if $r < cp + mp$ then
        randomly select individual $x$;
        $shuffleOrderOfAtomicEvents(x)$;
        for $i$ in $range(0, |x|)$ do
            atomic_event $e \leftarrow x[i]$;
            action $a \leftarrow extractAction(e)$;
            new atomic event $e' \leftarrow generateEvent(a)$;
            $x[i] \leftarrow e'$;
        end
        $OF \leftarrow OF \cup x$;
    end
    if $r < cp + mp + rp$ then
        $OF \leftarrow OF \cup$ (randomly selected individual $x$);
    end
    else
        randomly generate new individual $x$;
        $OF \leftarrow OF \cup x$;
    end
end
return $OF$
```

The most related search-based test data generation tools to STGFA-SMOG are the Sapienz [9] and EvoSuite [34]. Sapienz is a test data generation tool that generates test suites for Android GUI-based testing. The STGFA-SMOG architecture is similar to the Sapienz's architecture in the genetic algorithm module. However, unlike STGFA-SMOG, Spaienz contains a sophisticated modules such as the *Motifcore* that generates test suites based on the current context of the AUT. Moreover, Sapienz also contains a module that perform multilevel instrumentation on the AUT depending on whether black-box testing, gray-box testing, or white-box testing is needed to be performed. Evosuite is another test suite generation tool for Java-based desktop applications unit testing. To compare STGFA-SMOG's architecture with EvoSuite, no document is found that explicitly discusses or shows the EvoSuite architecture.

5 Methodology

In this study, the design science research methodology [43] is applied to answer the research questions formulated. This project has a natural series of cycles, i.e. literature review, design of a prototype test generator, the addition of a genetic algorithm, a series of cycles where additional fitness functions are added, and intermediate and final evaluations of the framework. STGFA-SMOG is evolved through these cycles to assess the difference in the effectiveness of the test cases generated. That is, STGFA-SMOG is evaluated by assessing the effectiveness of the test cases in terms of the number of crashes they caused when we vary the combination of fitness functions. These series of cycles map to the design science research process.

To empirically evaluate the effect of combining different fitness functions on the fault triggering effectiveness of the test cases generated, the following research questions are formulated and attempted to answer them in this study.

RQ1 : How can a search-based test generation framework for Android GUI-based testing best be designed to support multi-objective generation with support for the inclusion of additional fitness functions over time?

RQ2 : In general, how effective are the generated test suites at causing crashes in the assessed apps, in comparison to random test generation?

RQ3 : Which combinations of fitness functions is the most effective at causing the assessed apps to crash?

RQ4 : Does an increased search budget (the number of generations) improve the effectiveness of the resulting test cases?

The STGFA-SMOG tool is available at:
`https://github.com/TeklitB/STGFA-SMOG`
The Random test generation tool is available at:
`https://github.com/TeklitB/Random-Test-Generation`
The experimental data are published on Zenodo and available at:
`https://doi.org/10.5281/zenodo.6387568`

5.1 Design Science Iterations

Design science research is a research methodology through which researchers create, and evaluate artifacts intended to solve real organizational problems [44]. It involves a rigorous process to design and create an artifact that solves an identified real-world problem and contributes knowledge to science, evaluates the artifact created and communicates the obtained results to the intended audiences [44]. In this research, the design science research methodology (DSRM) formulated by Peffers et. al [43] is followed. The DSRM process model followed in this research study consists of six activities as shown in Fig 5.1.

To answer the research questions formulated above the research work is divided into three iterations. After each iteration, we carried out informal and formal evaluations and decided how to evolve the framework by overcoming the limitations identified in the evaluation. *Iteration I* is to develop the test generator prototype with a random fuzzing approach and an initial set of fitness functions and perform an informal evaluation to make sure if the tool is working as expected and fix if errors are found. In *Iteration II*, a genetic algorithm is added to the prototype developed in iteration I and integrated with two fitness functions. Moreover, the framework is also designed to support for users to select different combination fitness functions and support the addition of fitness functions over time, and carried out an informal evaluation. In *Iteration III*, more fitness functions are added to the framework in three sub-iterations and performed an intermediate evaluation to check if the framework works as expected, and fix if there are errors. Finally, we carried out the final evaluation of the framework developed and compared the effectiveness of test cases.

5.2 Objective-centered Solution

Multi-objective generation focuses on targeting multiple fitness functions concurrently in order to generate multi-faceted test cases. Test generation tools that support multi-objective generation use a combination of fitness functions to guide the search process. For multi-objective generation to be effective, we need to understand the effect of choosing different combinations of fitness functions on the effectiveness of the generated test cases to cause crashes on the AUT. Many fitness functions have been applied to the Java-based desktop applications [34] and the effect of varying different combinations of fitness functions on the test cases' effectiveness in terms of causing crashes was investigated [6]. However, it is still open which combination of fitness functions generates test cases that are effective at revealing crashes in Android applications. Therefore, to come up with an objective-centered solution for this research gap, we adopted the DSRM process model shown in Figure 5.1 from Peffers et. al [43] work.

According to the DSRM process model, the first activity performed in this research was problem identification and motivation. Based on the problem identified in *activity 1*, the objective of this research study was defined in activity two. The main objectives of the STGFA-SMOG study are to design a framework that supports multi-objective test case generation for Android GUI-based testing and to assess the effectiveness of the test cases generated in terms of causing crashes when we vary the combination of fitness functions. To achieve the objectives defined in *activity 2*, a prototype of a Random test generation with an initial set of fitness functions was designed first, a genetic algorithm was added to the prototype, and more fitness functions are added over time in *activity 3*.

In *activity 4*, five Android apps were selected randomly and the initial Random test generation prototype was demonstrated to generate test cases for Android GUI-based testing on the selected apps. After the prototype was changed from a random generator to a search-based multi-objective generator, it was also demonstrated to generate test cases by varying combinations of the initial set of fitness functions, and test results were collected for the intermediate evaluation. A similar procedure was applied when more fitness functions were added in the later iterations of the research work. After all fitness functions proposed in this research are implemented, ten Android apps were selected using the random sampling technique and STGFA-SMOG was demonstrated on these apps by varying the combination of fitness functions. The test results obtained during the demonstration were collected for the final empirical evaluation.

In *activity 5* of the research process, using the test results collected the effectiveness of the test cases generated in causing crashes were evaluated empirically. The empirical evaluation was to gain insights on which combination of fitness functions generates effective test cases and draw conclusions. In the last step of the research process (*activity 6*), documentation of the research work was reported, presented to the intended audience at Chalmers University of Technology, and a publication will be prepared for a conference or journal.

The research process for the STGFA-SMOG study iterates over *activity 3, 4, and 5* until the desired scope of the framework is achieved. Based on the limitations or feedback obtained in activity 4 or 5, the research process gets back to activity 3 (i.e. design and development) to improve the limitations of STGFA-SMOG, to add genetic algorithm, and fitness functions to the framework.

5.3 Iteration I: Design of Prototype and Evaluation

In this iteration, an initial implementation of the artifact is developed with the aim to understand how can a search-based test generation framework best be designed to support multi-objective generation and the addition of new fitness functions at a later time. As a first step towards the development of the prototype, a random fuzzing test generator with four initial fitness functions is developed. The random fuzzing test generator generates UI events and system events of Android apps as streams of events called tests cases to the AUT and the fitness functions score the

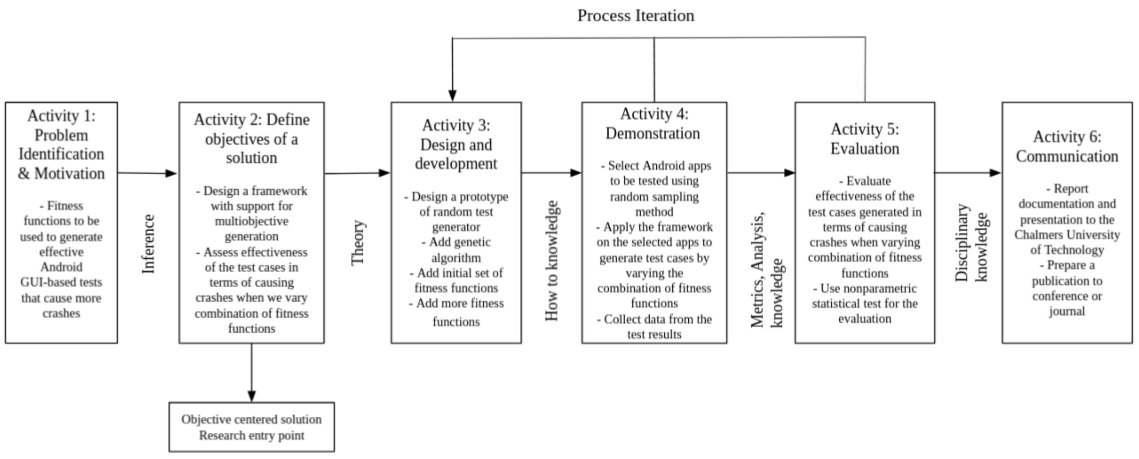
Process Iteration
Activity 1: Problem Identification & Motivation
- Fitness functions to be used to generate effective Android GUI-based tests that cause more crashes
Inference
Activity 2: Define objectives of a solution
- Design a framework with support for multiobjective generation
- Assess effectiveness of the test cases in terms of causing crashes when we vary combination of fitness functions
Objective centered solution
Research entry point
Theory
Activity 3: Design and development
- Design a prototype of random test generator
- Add genetic algorithm
- Add initial set of fitness functions
- Add more fitness functions
How to knowledge
Activity 4: Demonstration
- Select Android apps to be tested using random sampling method
- Apply the framework on the selected apps to generate test cases by varying the combination of fitness functions
- Collect data from the test results
Metrics, Analysis, knowledge
Activity 5: Evaluation
- Evaluate effectiveness of the test cases generated in terms of causing crashes when varying combination of fitness functions
- Use nonparametric statistical test for the evaluation
Disciplinary knowledge
Activity 6: Communication
- Report documentation and presentation to the Chalmers University of Technology
- Prepare a publication to conference or journal

Figure 5.1: DSRM process for the STGFA-SMOG study

test cases' fitness. No percentage distribution of the events that are sent to the AUT is defined. The events percentage distribution is completely random. The random fuzzing test generator continuously generates a stream of user events and system events until the maximum number of test cases to be generated is reached. The maximum number of test cases that the random generator tool should generate is user-defined. After the Random test generation is developed, it is demonstrated on randomly selected apps, and the fitness values of each test case are collected and evaluated. The main purpose of the evaluation is to check if the prototype is working as expected, fitness values are scored for each test case, and fix errors if found.

5.4 Iteration II: Addition of Genetic Algorithm and Evaluation

The final target of this research work is to develop a search-based test generation framework with support for a multi-objective generation. Hence, in this iteration, a genetic algorithm called NSGA-II with two fitness functions (i.e. crash and test case size) is added to the initial prototype. For the NSGA-II algorithm implementation, a framework called DEAP [45] that provides a flexible implementation of different evolutionary algorithms is used. The framework to support user selectable fitness functions and the addition of new fitness functions over time, the fitness functions component of the framework is implemented by following the factory method design pattern. At this iteration, the framework supports two fitness functions, i.e. maximizing the number of crashes, and minimizing the length of test cases.

5.5 Iteration III: Addition of More Fitness Functions and Evaluation

5.5.1 Addition of More Fitness Functions

After the genetic algorithm is added in the previous iteration, more additional fitness functions are added to the prototype in this iteration. STGFA-SMOG is designed to be customizable to specialized testing goals such as crashes, code coverage, CPU usage, memory usage, network usage, and battery usage. That is, it allows for single-objective or multi-objective test generation of user-specified goals. In this iteration, six fitness functions (i.e. CPU usage, memory usage, battery usage, network usage, method coverage, and line coverage) are added in three sub-iterations. In the first sub-iterations, CPU usage and memory usage fitness functions are implemented. In the second sub-iterations of this iteration, network usage and battery usage fitness functions are added. In the last sub-iterations, code coverage fitness functions (method coverage and line coverage) are implemented using a third-party tool called ACVTool. In each sub-iteration, the framework was applied to test subjects, evaluated their fitness values compared to the baseline, and get back to the design iteration to improve the framework based on the evaluation results and add fitness functions.

5.5.2 Intermediate Evaluation

The aim of the intermediate evaluation was to see if STGFA-SMOG is functioning as intended so that we can fix errors and make adjustments. An informal experiment similar in structure to the intermediate evaluation (see section 5.5.3), but on a smaller scale was carried out for each sub-iteration. The test subjects used, the configurations applied, and the data collected in the intermediate evaluation are discussed in the following sub-subsections.

Test Subjects

A small set of Android applications were chosen from the F-Droid store and the tool is applied to generate test cases by varying the combination of fitness functions added. The motivation for choosing the F-Droid Android app repository is because it provides free access to free and open-source Android apps and it is a commonly used repository in other research studies [9, 46, 47]. A total of 5 Android apps were collected for the intermediate evaluation of the framework. The test subjects were selected at random from a collection of open-source apps. Apps that require authentication and failed to be successfully instrumented by the code coverage tool are discarded and replaced by other randomly selected apps. This process was repeated until we got 5 apps that do not require authentication and are successfully instrumented by the ACVTool. The Android apps used for the intermediate evaluation are shown in Table 5.1. We believe the size of the app does not have an impact on the test generation and is not included in the table.

Table 5.1: Apps used for intermediate evaluation

App Name	Version	Domain	Description
traficparis	2021.04	Navigation	Simple, responsive map for your trek
blokish	3.2	Game	Board game
billthefarmer	1.36	Currency	Simple currency conversion
rpncalc	1.0.3	Science	A simple, modern calculator that uses RPN
uaraven	1.4.1	Health	Food additives reference

Experiment Configurations

The general configuration of the fitness function combination used to generate test cases is (number of crashes) + (test case size) + (new fitness function). For the intermediate evaluation, the following combination of fitness functions were applied:

Sub-Iteration 1:
- (number of crashes) + (test suite size) + (CPU usage)
- (number of crashes) + (test suite size) + (memory usage)

Sub-Iteration 2:
- (number of crashes) + (test suite size) + (network usage)
- (number of crashes) + (test suite size) + (battery usage)

Sub-Iteration 3:
- (number of crashes) + (test suite size) + (line coverage)
- (number of crashes) + (test suite size) + (method coverage)

In the experiment, number of generations were used as a search budget. Two search budgets of values 10 and 15 generations were used. In each search budget, the population size was set to 10 individuals. The STGFA-SMOG's crossover, mutation, reproduction, and addition of random new individual probability were set to 0.3, 0.3, 0.15, and 0.25 respectively. The minimum and maximum length of the test cases were set to 20 and 50 respectively. In each search budget, five trials were performed for each app. Additional experimentation with other search budgets and population sizes was carried out to identify settings for the final evaluation.

In the intermediate evaluation, the focus was on the fitness values of each new fitness function not on the number of crashes. Because this iteration aimed to see if the genetic algorithm is working as expected, the fitness values are improved when the test cases are evolved, and better fitness values than the Random test generation are obtained at the end of the evolution. The initially developed Random test generation prototype is used as a baseline to compare with STGFA-SMOG. So, the final fitness values between the Random test generation baseline and STGFA-SMOG were compared. The STGFA-SMOG is expected to outperform the Random test generation prototype (i.e. it should produce better fitness function values than the Random test generation). Random generation is the process that the random generator prototype generates one population, executes it against the AUT, and prints it to a file, rather than evolving that population.

Test Generation Data Collection

To perform the intermediate evaluation, multiple trials were performed (5 trials) for each configuration and search budget and collected the following data.

- Number of crashes detected for each test case and app
- Code coverage for each test case and app
- Final fitness values for each test case and app

Based on the collected data the resulting test cases between the STGFA-SMOG and Random test generation, and search budgets were compared based on their final fitness values per app. The fitness values of the resulting test cases at different search budgets and for different combinations of fitness functions were evaluated. Mann-Whitney U-test and Vargha-Delaney nonparametric test were used to measure the statistical difference and effect size respectively.

5.5.3 Final Evaluation of STGFA-SMOG

The STGFA-SMOG is applied on a randomly selected open-source Android apps from the F-Droid[1] which is an online store of free and open-source Android applications. Test results are collected and analysed empirically to find out which combination of fitness functions generate test cases that cause more crashes.

Test Subjects

Free open-source Android apps were collected as test generation targets from the F-Droid and other databases. A total of around 10 Android apps are collected for the final evaluation of the framework. Apps with varying complexity (i.e. different types and ranges of functionality available) and from different product domains/app types are selected. The test generation targets are selected at random from a collection of open-source apps. The Android applications are collected in APK format. After the Android apps are collected, they are analysed to identify Apps on which STGFA-SMOG is not applicable without manual intervention. For example, some applications may require login credentials and without successful login, the functionalities of the applications are not available to the user. In such a case, STGFA-SMOG will not be applicable and cannot achieve the desired goals of the test. Furthermore, when the apps are instrumented with the code coverage tool, some of the apps did not work as in the original app and the apps that failed to successfully install on the target emulator are discarded from the data set. Apps that do not fulfill any of those criteria are discarded and a new random app was selected instead. This process was repeated until we obtained ten apps that fulfilled all the criteria discussed previously. The selected apps are listed in Table 5.2.

[1]https://www.f-droid.org/

Table 5.2: Apps used for final evaluation

App Name	**Version**	**Domain**	**Description**
scoutantblokish	3.4	Game	Board game involving four players
sourceforge solitaire	3.4.1	Game	Solitaire Card Games
palmcalc	3.0.4	Scientific	Retro scientific calculator and converters
dib22alc	0.12.62	Science	The crazy calculator (RPN mode)
mousepounce	1.2.1	Game	Play Egyptian Rat Screw against humans or cats.
xskat	1.6	Game	Play famous German card game Skat
bmicalculator	4.0.2	Health	Body Mass Index calculator
simpleaccounting	1.6.1	Money	A very simple way to store your balance
salomaxcurrencies	1.5.1	Money	An exchange rates currency converter for Android
converteurofranc	2.9 (18)	Money	Inflation calculator for USA, UK and France

Experiment Configurations

To conduct the experimentation, a combination of crashes, CPU usage, memory usage, test case length, network usage, and battery usage fitness functions were used. In the experiment, number of generations is used as a search budget. Two search budgets of values 10 and 30 generations were used. In each search budget, the population size was set to 20 individuals. The STGFA-SMOG's crossover, mutation, reproduction, and addition of random new individual probability were set to 0.3, 0.3, 0.15, and 0.25 respectively. The minimum and maximum sequence length of test cases were set to 20 and 50 respectively. The configuration values could be changed following the intermediate evaluation conducted to identify settings for the final evaluation. The initially developed random fuzzing test generation prototype is used as a baseline to compare with STGFA-SMOG.

The main goal of the evaluation was to see if certain configurations of fitness functions can be found that result in more crashes. So, test cases were generated targeting different combinations of three fitness functions from the pool of fitness functions implemented. Generally, the following configuration format of fitness function combinations is used: (number of crashes) + (test suite size) + (one of the other functions). Hence, the following configurations of fitness functions were used for the final evaluation:

- (number of crashes) + (test suite size) + (CPU usage)
- (number of crashes) + (test suite size) + (memory usage)
- (number of crashes) + (test suite size) + (network usage)
- (number of crashes) + (test suite size) + (battery usage)

For each app, five trials were performed in each search budget and for each fitness function combination. The experiment was conducted on a PC with a five-core 1.60GH CPU and 8GB RAM on Ubuntu 20.04 operating system. Google Pixel 4 XL emulator with Android API 28 was used to run the Android apps.

Test Generation Data Collection

To evaluate the fault-triggering effectiveness of the test cases, the STGFA-SMOG was run on the Android apps and generated test cases by varying the combination of the fitness functions according to the configurations specified in section 5.5.3. In an attempt to answer the research questions formulated, multiple trials were performed (5 trials) for each configuration and search budget and collect the following data.

- Number of crashes detected for each test case and app
- Final fitness values for each test case and app

Based on the collected data the resulting test cases' effectiveness was compared and evaluated. The Effectiveness of resulting test cases at different search budgets and for different combinations of fitness functions were evaluated. The data collected are drawn from unknown distribution and do not follow normality. This is from the fact that outputs of randomized algorithms vary from one run to another run and the probability distributions of search algorithms are often strongly departing from normality [48]. Arcuri and Andrea [48] recommended that the non-parameter test is the appropriate statistical test to empirically assess experimental data obtained from search algorithms applied to software engineering problems. So, to analyze the data without any assumptions on their distribution, a non-parametric test was used in this research. The most commonly used (as in the References [49, 27, 50]) and recommended by A. Arcuri and L. Briand [48] non-parametric test types to measure the statistical difference and effect size are Mann-Whitney U-test [51] and Vargha-Delaney [52] respectively. Hence, Mann-Whitney U-test and Vargha-Delaney were used to measure the statistical difference and effect size between the pair of the combination of fitness functions, the framework, Random test generation, and search budgets.

6 Results

This chapter presents the intermediate and final evaluation results of the research work. The main interest of this research is to understand which combination of the fitness functions of the STGFA-SMOG reveals more crashes than others. To achieve this goal the STGFA-SMOG framework is evolved through different iteration. Section 6.1 shows the intermediate evaluation results that are performed to assess whether STGFA-SMOG is working as expected and to find errors. Based on the intermediate evaluation results obtained, STGFA-SMOG is improved when evolved through the different iteration. Section 6.2 presents the final evaluation results of the comparison of the different configurations of fitness functions within STGFA-SMOG, Random test generation, and search budgets in terms of the number of crashes revealed. The Mann-Whitney Wilcoxon test is applied to check whether there is a statistically significant difference in the distributions of fitness values or crashes between different configurations of STGFA-SMOG and Random test generation or not and an effect size test, Vargha-Delaney, is used to examine the magnitude of the difference when statistically significant differences are found.

To determine whether a distribution of results for one technique is different from another using the Mann-Whitney Wilcoxon test, we adopted the most common α value which is $\alpha = 0.05$. The corresponding confidence level for the chosen α value is 95%. If the p-value of the test result is less than 0.05, we reject the null hypothesis as the test shows the presence of a statistical difference between the distributions. When a statistically significant difference is found, it is important to determine which technique is better. As a result, the Vargha-Delaney test is conducted to see the effect size and compare the performance of the variables. In this test, effect size less than 0.5 indicates better performance of the second technique from the first technique in each hypothesis while values greater than 0.5 show better performance of the first technique from the second technique. The effect size of one technique or variable over the other technique or variable can be small, medium, or large. According to Vargha-Delaney's [52] interpretation, a small effect is $0.56 <= A_{12} < 0.64$, medium is $0.64 <= A_{12} < 0.71$, and large is $A_{12} >= 0.71$ where A_{12} is the effect size.

Note:
The column names of the tables in Sections 6.1 and 6.2 are short notations for the combination of the fitness functions as follows.
CLB - denotes the combination of Crash, test case Length, and Battery fitness functions.
CLC - denotes the combination of Crash, test case Length, and CPU fitness functions.
CLM - denotes the combination of Crash, test case Length, and Memory fitness functions.
CLN - denotes the combination of Crash, test case Length, and Network fitness functions.
CLLC - denotes the combination of Crash, test case Length, and Line Coverage fitness functions.
CLMC - denotes the combination of Crash, test case Length, and Method Coverage fitness functions.

The value 'NA' in the table entries indicates that the two distributions compared have the same values, which are all zeros. For example, some apps do not use network data, they operate offline and the comparison of the Network usage fitness values between STGFA-SMOG and Random test generation gives 'NA' such as in Table 6.5, 6.7, and 6.9. Because the network usage fitness values are all zeros both in STGFA-SMOG and Random test generation and the Mann-Whitney Wilcoxon test R function gives 'NA'. Effect size is measured only when a statistically significant difference is found using the Mann-Whitney Wilcoxon test. Moreover, the table entries in bold font represent p-values < 0.05 and large effect sizes.

6.1 Intermediate Evaluation Results

This section depicts the intermediate evaluation results for STGFA-SMOG and Random test generation. The intermediate evaluation is mainly intended to compare the fitness values of the different combinations of fitness functions in STGFA-SMOG and Random test generation with the goal of examining whether search-based test generation produces consistently higher fitness values than random generation. Moreover, the search budgets are compared to investigate whether a large search budget (i.e. 15 generations) produces higher fitness values than a small search budget (i.e. 10 generations).

The general hypothesis of the intermediate evaluation is that different techniques: Random test generation, STGFA-SMOG with different configurations of the fitness functions, and using different search budgets for STGFA-SMOG make a difference in performance. To statistically test the experimental data collected, the following null and alternative hypotheses are formulated.

Null hypotheses:

- $H0_1$: The observations of results for both STGFA-SMOG with a 10 generation search budget and Random test generation are drawn from the same distribution.

- $H0_2$: The observations of results for both STGFA-SMOG with a 15 generation search budget and Random test generation are drawn from the same distribution.

- $H0_3$: The observations of results for STGFA-SMOG with both 15 and 10 generation search budgets are drawn from the same distribution.

Alternative hypotheses:

- H_1: Test cases generated using STGFA-SMOG with a 10 generation search budget have a different distribution of results than test cases generated using Random test generation.

- H_2: Test cases generated using STGFA-SMOG with a 15 generation search budget have a different distribution of results than test cases generated using Random test generation.

- H_3: Test cases generated using STGFA-SMOG with a 15 generation search budget have a different distribution of results than est cases generated using STGFA-SMOG with the 10 generation search budget.

In Table 6.1, 6.2, 6.3, and 6.4 the columns: **CLC** compares performance between STGFA-SMOG optimizing the [**(crash)** + **(test case length)** + **(CPU usage)**] fitness function combination and random generation. The values listed are the median fitness values for CPU usage, **CLM** compares performance between STGFA-SMOG optimizing the [**(crash)** + **(test case length)** + **(Memory usage)**] fitness function combination and random generation. The values listed are the median fitness values for Memory usage, **CLB** compares performance between STGFA-SMOG optimizing the [**(crash)** + **(test case length)** + **(Battery usage)**] fitness function combination and random generation. The values listed are the median fitness values for Battery usage. Similarly, **CLN** compares performance between STGFA-SMOG optimizing the [**(crash)** + **(test case length)** + **(Network usage)**] fitness function combination and random generation. The values listed are the median fitness values for Network usage, **CLLC** compares performance between STGFA-SMOG optimizing the [**(crash)** + **(test case length)** + **(Line Coverage)**] fitness function combination and random generation. The values listed are the median fitness values for Line Coverage, and **CLMC** compares performance between STGFA-SMOG optimizing the [**(crash)** + **(test case length)** + **(Method Coverage)**] fitness function combination and random generation. The values listed are the median fitness values for Method Coverage. Table 6.1 and 6.2 compares STGFA-SMOG with 10 generation search budget and Random test generation, and

Table 6.3 and 6.4 compares STGFA-SMOG with 15 generation search budget and Random test generation.

In Table 6.5, and 6.7 the columns: **CLC** shows the p-value from the Mann-Whitney Wilcoxon test when comparing the distribution of fitness values for the CPU usage fitness function between random generation and STGFA-SMOG optimizing the [**(crash) + (test case length) + (CPU usage)**] fitness function combination, **CLM** shows the p-value from the Mann-Whitney Wilcoxon test when comparing the distribution of fitness values for the Memory usage fitness function between random generation and STGFA-SMOG optimizing the [**(crash) + (test case length) + (Memory usage)**] fitness function combination, **CLB** shows the p-value from the Mann-Whitney Wilcoxon test when comparing the distribution of fitness values for the Battery usage fitness function between random generation and STGFA-SMOG optimizing the [**(crash) + (test case length) + (Battery usage)**] fitness function combination, **CLN** shows the p-value from the Mann-Whitney Wilcoxon test when comparing the distribution of fitness values for the Network usage fitness function between random generation and STGFA-SMOG optimizing the [**(crash) + (test case length) + (Network usage)**] fitness function combination, **CLLC** shows the p-value from the Mann-Whitney Wilcoxon test when comparing the distribution of fitness values for the Line coverage fitness function between random generation and STGFA-SMOG optimizing the [**(crash) + (test case length) + (Line Coverage)**] fitness function combinations, and **CLMC** shows the p-value from the Mann-Whitney Wilcoxon test when comparing the distribution of fitness values for the Method Coverage fitness function between random generation and STGFA-SMOG optimizing the [**(crash) + (test case length) + (Method Coverage)**] fitness function combinations. Similarly, the columns in Table 6.9 shows p-values from the Mann-Whitney Wilcoxon test as described in the previous sentences but the comparison is between STGFA-SMOG with 10 and 15 generation search budgets.

In Table 6.6, and 6.8 the columns: **CLC** shows the effect size values from the Vargha-Delaney test when comparing the distribution of fitness values for the CPU usage fitness function between random generation and STGFA-SMOG optimizing the [**(crash) + (test case length) + (CPU Usage**], **CLM** shows the effect size values from the Vargha-Delaney test when comparing the distribution of fitness values for the Memory usage fitness function between random generation and STGFA-SMOG optimizing the [**(crash) + (test case length) + (Memory Usage**], **CLB** shows the effect size values from the Vargha-Delaney test when comparing the distribution of fitness values for the Battery usage fitness function between random generation and STGFA-SMOG optimizing the [**(crash) + (test case length) + (Battery Usage**], **CLN** shows the effect size values when comparing the distribution of fitness values from the Vargha-Delaney test for the Network usage fitness function between random generation and STGFA-SMOG optimizing the [**(crash) + (test case length) + (Network Usage**], **CLLC** shows the effect size values from the Vargha-Delaney test when comparing the distribution of fitness values for the Line Coverage fitness function between random generation and STGFA-SMOG optimizing the [**(crash) + (test case length) + (Line Coverage**], **CLMC** shows the effect size values from the Vargha-Delaney test when comparing the distribution of fitness values for the Method Coverage fitness function between random generation and the STGFA-SMOG optimizing the [**(crash) + (test case length) + (Method Coverage**]. The columns in Table 6.10 also shows similar effect size values from the Vargha-Delaney test as described in the previous sentences but the comparison is between STGFA-SMOG with 10 and 15 generation search budgets.

Table 6.5, 6.7, and 6.9 show the comparison of the distribution of the fitness values between STGFA-SMOG with 10 generation search budget and random generation, STGFA-SMOG with 15 generation search budget and random generation, and STGFA-SMOG with 15 generation and 10 generation search budgets respectively to determine if there is a statistical difference. Similarly, Table 6.6, 6.8, and 6.10 show the comparison of the distribution of the fitness values between STGFA-SMOG with 10 generation search budget and random generation, STGFA-SMOG with 15 generation search budget and random generation, and STGFA-SMOG with 15 generation and 10 generation search budgets respectively to identify which technique is better than the other.

In Table 6.1, and 6.2, the median fitness values of STGFA-SMOG with 10 generation search budget are higher in all configuration of fitness functions for all apps except the **Battery** usage and **Method** coverage median fitness values for the **billthefarmer** app. In the two exception fitness functions (i.e. **Battery** and **Method**) for the **billthefarmer** app, median fitness values of the random generation are higher than median fitness values of the STGFA-SMOG with 10 generation search budget. The reason for this can be the **billthefarmer** app has a small number of widgets and most of the widgets are EditText fields. When most of the widgets are EditText field, it is observed that most of the events generated become input text events as the keyboard is opened for a long time. This makes it hard for the STGFA-SMOG to optimize the fitness values and improve over the generations. The **Network** usage fitness values of STGFA-SMOG with 10 generation search budget and random generation are the same, which are all zero, for **blokish, rpncalc, and uaraven** apps. This is because the three apps do not use network data, they function offline, and no Network usage fitness value is available for these apps.

The results presented in Table 6.3, and 6.4 shows that median values of STGFA-SMOG with 15 generation search budget are higher than median fitness values of random generation in all combination fitness functions for all apps except for **Network** usage which is always zero for the apps that do not use network data. Even though the median value of **Battery** and **Method** coverage for the **billthefarmer** app in STGFA-SMOG with 10 generation search budget are less than the random generation, using a higher search budget (i.e. 15 generation) makes STGFA-SMOG produce higher fitness values in all combination of fitness functions than produced by random generation. This indicates STGFA-SMOG improves its performance when the search budget is increased and produces higher median fitness values from random generation.

Table 6.1: Median values of STGFA-SMOG fitness values with 10 generation search budget and Random generation for each app

	CLC		**CLM**		**CLB**	
	STGFA-SMOG	**Random**	**STGFA-SMOG**	**Random**	**STGFA-SMOG**	**Random**
blokish	26.00	11.50	42610.50	25790.00	6.83e-07	0.00
billthefarmer	24.00	14.00	46204.50	25398.50	**0.00**	**4.40e-07**
traficparis	27.00	18.00	38872.00	28098.00	6.71e-07	0.00
rpncalc	68.00	26.50	136372.50	117998.50	4.66e-07	0.00
uaraven	29.00	15.00	39955.00	25085.50	4.59e-07	0.00

Table 6.2: Median values of STGFA-SMOG fitness values with 10 generation search budget and Random generation for each app

	CLN		**CLLC**		**CLMC**	
	STGFA-SMOG	**Random**	**STGFA-SMOG**	**Random**	**STGFA-SMOG**	**Random**
blokish	0.00	0.00	7.76	6.57	8.53	7.00
billthefarmer	481614.00	37255.00	4.59	4.00	**1.41**	**1.80**
traficparis	2350519.00	124255.40	20.40	18.32	26.26	24.63
rpncalc	0.00	0.00	22.51	20.30	31.80	29.13
uaraven	0.00	0.00	2.19	2.02	2.17	1.96

The p-values shown in Table 6.5 are below 0.05 in most of the fitness functions which shows the presence of statistically significant difference between the distribution of the fitness values of STGFA-SMOG with a 10-generation search budget and random generation except the **Battery** usage fitness function for the **billthefarmer** and **rpncalc**, and **Method** coverage fitness function for the **rpncalc** apps. Based on the statistical analysis there exists a statistically significant difference in the distribution of the fitness values of STGFA-SMOG with 10-generation search budget and random generation except in the distribution of the Battery fitness values in the **billthefarmer** and **rpncalc** apps, and Method coverage fitness values for the **rpncalc** apps. Table 6.6 shows the effect size values for the fitness values of STGFA-SMOG with 10 generation and random generation where statistically significant difference is found in Table 6.5. The effect size values in all the fitness functions are above 0.5 except for the **Battery** fitness function for the **billthefarmer** and **rpncalc** app, **Network** fitness function in the **blokish, rpncalc, and uaraven** apps, and **Method** fitness function for the **billthefarmer** app. The results show that STGFA-SMOG with 10 generation outperforms random generation for the fitness functions **Memory**, **CPU**, and **Line Coverage** in all the 5 apps. Similarly, STGFA-SMOG outperforms random generation for the **Method Coverage** fitness

Table 6.3: Median values of STGFA-SMOG fitness values with 15 generation search budget and Random generation for each app

	CLC		**CLM**		**CLB**	
	STGFA-SMOG	**Random**	**STGFA-SMOG**	**Random**	**STGFA-SMOG**	**Random**
blokish	48.50	10.00	41755.50	24722.50	3.31e-06	1.00e-06
billthefarmer	35.50	23.50	45898.00	25511.50	1.70e-05	8.60e-07
traficparis	29.00	13.50	50705.50	29242.50	0.00	0.00
rpncalc	94.50	52.50	134803.50	120060.50	5.08e-05	0.00
uaraven	42.00	21.50	48234.00	25441.00	1.33e-06	3.10e-07

Table 6.4: Median values of STGFA-SMOG fitness values with 15 generation search budget and Random generation for each app

	CLN		CLLC		CLMC	
	STGFA-SMOG	**Random**	**STGFA-SMOG**	**Random**	**STGFA-SMOG**	**Random**
blokish	0.00	0.00	7.70	6.42	8.16	7.05
billthefarmer	1235955.00	37471.66	4.52	4.03	2.05	1.90
traficparis	1512818.00	135683.00	20.43	18.46	26.81	23.88
rpncalc	0.00	0.00	22.51	20.30	32.39	29.13
uaraven	0.00	0.00	2.22	2.01	2.17	1.96

Table 6.5: P-Values for Mann-Whitney rank-sum test for the fitness values of STGFA-SMOG with 10 generation versus Random test data generation (baseline)

	CLC	CLM	CLB	CLN	CLLC	CLMC
blokish	**0.00**	**$<$ 2.20e-16**	**0.00**	NA	**1.80e-05**	**1.76e-11**
billthefarmer	**5.80e-06**	**$<$ 2.20e-16**	0.58	**$<$ 2.20e-16**	**1.13e-08**	0.05
traficparis	**0.04**	**2.38e-14**	**0.00**	**$<$ 2.20e-16**	**2.21e-06**	**1.52e-06**
rpncalc	**3.73e-10**	**1.10e-11**	0.05	NA	**6.61e-10**	**2.02e-12**
uaraven	**2.55e-08**	**1.11e-15**	**0.00**	NA	**9.85e-13**	**1.92e-11**

Table 6.6: Results of Vargha-Delaney A Measure for the fitness values of 10G STGFA-SMOG versus Random test data generation (baseline). Note that large effect sizes are bolded and effect size is only measured when statistical significant difference is found using the Mann-Whitney Wilcoxon test.

	CLC	CLM	CLB	CLN	CLLC	CLMC
blokish	0.70	**1.00**	0.68	-	**0.75**	**0.89**
billthefarmer	**0.76**	**1.00**	-	**1.00**	**0.83**	-
traficparis	0.62	**0.94**	**0.71**	**1.00**	**0.77**	**0.78**
rpncalc	**0.86**	**0.89**	-	-	**0.86**	**0.90**
uaraven	**0.82**	**0.97**	0.68	-	**0.91**	**0.89**

Table 6.7: P-Values for Mann-Whitney rank-sum test for the fitness values of 15G STGFA-SMOG versus Random test data generation (baseline)

	CLC	CLM	CLB	CLN	CLLC	CLMC
blokish	**1.86e-10**	**2.03e-14**	**0.02**	NA	**2.44e-11**	**2.18e-07**
billthefarmer	**0.03**	**7.48e-16**	**0.00**	**< 2.20e-16**	**1.99e-08**	**5.20e-09**
traficparis	**1.42e-08**	**< 2.20e-16**	**6.11e-11**	**< 2.20e-16**	**1.42e-14**	**< 2.20e-16**
rpncalc	**1.00e-11**	**1.37e-08**	**2.05e-05**	NA	**2.93e-13**	**3.75e-11**
uaraven	**0.01**	**< 2.20e-16**	**0.00**	NA	**3.42e-13**	**2.12e-13**

Table 6.8: Results of Vargha-Delaney A Measure for the fitness values of 15G STGFA-SMOG versus Random test data generation (baseline)

	CLC	CLM	CLB	CLN	CLLC	CLMC
blokish	**0.87**	**0.94**	0.64	-	**0.89**	**0.80**
billthefarmer	0.62	**0.97**	0.70	**1.00**	**0.83**	**0.84**
traficparis	**0.83**	**0.98**	**0.87**	**1.00**	**0.95**	**0.99**
rpncalc	**0.90**	**0.83**	**0.74**	-	**0.92**	**0.87**
uaraven	0.65	**0.99**	0.69	-	**0.92**	**0.92**

function except in the **billthefarmer** app.

STGFA-SMOG with 10 generations outperforms Random test generation for the CPU, Memory, Network, and Line Coverage fitness functions in all apps, often with large effect sizes. It also outperforms Random test generation in 80% of the apps for the Method Coverage fitness function and 60% of the apps for the Battery fitness function.

In Table 6.7 the p-values for all fitness functions in all apps are less than 0.05 except for the **Network** fitness function in the **blokish, rpncalc, and uaraven** apps which is 'NA' as those apps do not use network data. Hence, the p-values show that a statistically significant difference is found between the distribution of the fitness values of STGFA-SMOG with 15 generation search budget and random generation. For all the fitness functions in Table 6.7 where a statistically significant difference is found, their corresponding effect size values are shown in Table 6.8. All of the effect size values in Table 6.8 are greater than 0.05 which implies that STGFA-SMOG with a 15 generation search budget outperforms random generation for all fitness functions in all the five apps. This indicates that increasing the search budget improves the performance of STGFA-SMOG over the random generation.

A statistical difference exists for all the five apps in the STGFA-SMOG with a 15 generation search budget and random generation as in Table 6.7 . This could be due to the improvement in the performance of the STGFA-SMOG framework with higher search budgets.

Table 6.9: P-Values for Mann-Whitney rank-sum test for the fitness values of STGFA-SMOG 15 versus 10 generation, values in the table are bolded for P-value less than 0.05 and 'NA' in the Network column shows these apps operates offline

	CLC	CLM	CLB	CLN	CLLC	CLMC
blokish	**8.04e-07**	0.42	0.11	NA	0.58	**0.02**
billthefarmer	**9.30e-10**	**2.69e-16**	**8.46e-05**	**5.18e-11**	0.076	**5.22e-12**
traficparis	0.11	**0.02**	**0.03**	**8.20e-07**	0.05	**0.00**
rpncalc	**2.81e-14**	0.35	**0.00**	NA	**0.03**	**0.00**
uaraven	**1.12e-08**	**0.04**	0.05	NA	0.10	0.43

Table 6.10: Results of Vargha-Delaney A Measure for the fitness values of STGFA-SMOG 15 versus 10 generation, note that effect size is only measured in cases where differences were detected through the Mann-Whitney Wilcoxon test and values are bolded for those who have high effect-size.

	CLC	CLM	CLB	CLN	CLLC	CLMC
blokish	**0.79**	-	-	-	-	0.63
billthefarmer	**0.86**	**0.98**	**0.72**	**0.88**	-	**0.90**
traficparis	-	0.63	0.63	0.21	-	0.66
rpncalc	**0.94**	-	0.68	-	0.62	0.69
uaraven	**0.83**	0.62	-	-	-	-

As shown in Table 6.9, there exists a statistically significant difference in the distribution of fitness values of STGFA-SMOG with 15 and 10 generation search budgets for the **CPU** fitness function in the **blokish, billthefarmer, rpncalc, and uaraven** apps, **Memory** fitness function in the **billthefarmer, traficparis, and uaraven** apps, **Battery** fitness function in the **billthefarmer, traficparis, and rpncalc** apps, **Network** fitness function in the **billthefarmer and traficparis** apps, **Line Coverage** in the **traficparis and rpncalc** apps, and **Method Coverage** fitness function in the **blokish, billthefarmer, traficparis, and rpncalc** apps. No statistically significant difference is found in the distribution of the fitness values of STGFA-SMOG with 15 and 10 generation search budgets for the remaining fitness function in the corresponding apps. The reason for no statistically significant difference being found can be due to the small difference in the two search budgets used.

In the comparison between STGFA-SMOG with 15 and 10 generation search budgets as shown in Table 6.9 a significant difference is found for many of the fitness functions and apps. We reject the null hypothesis for the **CPU, and Method Coverage** fitness functions for 80% of the apps, **Memory and Battery** fitness functions for 60% of the apps, and **Line Coverage** for 40% of the apps. We also reject the null hypothesis for the **Network** fitness function for 40% of the apps. Based on the results from Table 6.10, STGFA-SMOG with a 15-generation search budget performs better than STGFA-SMOG with 10 generation search budget on all fitness functions in all cases where we rejected the null hypothesis, except for the Network fitness function for the app **traficparis**. In this case, STGFA-SMOG performed better with a 10 generation search budget.

STGFA-SMOG shows improved performance when the search budget is increased from 10 to 15 generations for 80% of apps when targeting the CPU and Method Coverage fitness functions, 60% when targeting Memory and Battery, 50% when targeting Network, and 40% when targeting Line Coverage. The largest differences are seen in the CPU fitness function.

6.2 Final Evaluation Results

In this section, the final evaluation results of STGFA-SMOG and Random test generation are presented. In the final evaluation, the number of crashes detected by the different combinations of the fitness functions is compared to examine whether some combination of the fitness functions uncover more crashes than others. The number of crashes triggered by the different combinations of the fitness functions of STGFA-SMOG is compared against Random test generation, which is the baseline, to assess if STGFA-SMOG detects more crashes than Random test generation. Moreover, the number of crashes revealed by the different combinations of the fitness functions are compared at 30 and 10 generation search budgets to investigate if STGFA-SMOG with a large search budget reveals more crashes than STGFA-SMOG with a small search budget. Unlike the intermediate evaluation, in the final evaluation, we applied only the four fitness function configurations (i.e. **CLB, CLC, CLM, and CLN**) due to time constraints. Because of the third-party tool we used to measure line and method coverage, the time taken by **CLLC and CLMC** fitness function configurations is more than twice the time taken by **CLB** per trial. Hence, we are forced to run our experimentation over the four configurations because of the time limitation and the limited computational resource we have.

Similar to the intermediate evaluation, the Mann-Whitney Wilcoxon test, and the Vargha-Delaney measure are used to find a statistical significance difference and to measure the magnitude of the effect size when a statistically significant difference is found respectively. The final values of the crash fitness function for each test case obtained at the last generation over each trial are merged. Then we feed these fitness values to the Mann-Whitney Wilcoxon test and Vargha-Delaney test R functions.

In the following subsections, we present and describe our experimental results with respect to the research question RQ2, RQ3, and RQ4 (see chapter 5). In subsection 6.2.1 we list the unique crashes revealed across all trials and the average number of times each crash is triggered per trial. Subsection 6.2.2 presents the comparison of the techniques (i.e. STGFA-SMOG vs Random test generation) and the fitness function configurations (i.e. **CLB, CLC, CLM, and CLN**) keeping the search budget constant. In subsection 6.2.3 we compare the effect of the search budget by varying the search budget values, which are 10 generations and 30 generations, and keeping the fitness function configurations constant.

6.2.1 Crashes Triggered and Running Time Logs

The crashes uncovered by STGFA-SMOG with 30 generation and 10 generation search budgets, and Random test generation are depicted in Tables 6.11, 6.13, and 6.15 respectively. In Tables 6.11 and 6.13 we list the unique crashes detected by STGFA-SMOG with 30 generation and 10 generation search budgets respectively across all trials, broken by app and configuration. We also list the average number of times the crashes are triggered by STGFA-SMOG with 30 generation, 10 generation, and Random test generation per trial in Tables 6.12, 6.14, and 6.16 respectively. A test case generated by STGFA-SMOG can trigger more than one crash. However, most of the test cases across all trials revealed one or zero crashes. Hence, we calculated the average number of times each crash is triggered instead of the average number of crashes triggered per trial. Moreover, Tables 6.17 and 6.18 present a sample of running time logs that shows how many hours STGFA-SMOG takes for each fitness function configuration per trial. Even though a higher search budget is not guaranteed to reveal the same crashes that a lower search budget does, in our experiment we observed that all crashes uncovered by the 10 generation search budget are also uncovered by the 30 generation search budget. Based on the stack trace analysis of the exceptions listed in Tables 6.12, 6.14, and 6.15, each exception is the same when they occur every time. This means that each discovered crash is unique. As described in chapter 5, we used ten randomly selected Android apps to experiment with our framework in the final evaluation. Out of the ten Android apps used, crashes are found only in four of the apps: **scoutantblokish, dib2calc, palmcalc, and sourceforgesolitaire**, and only these apps are used in the statistical analysis of the final experimental results.

As shown in Table 6.11, **ActivityNotFoundException** is triggered in **scoutantblokish** app by all fitness function configurations of STGFA-SMOG with a 30 generation search budget. In the **dib2calc** app, three exceptions are found: **IndexOutOfBoundsException, ArrayIndexOutOfBoundsException, and NullPointerException**. The first two exceptions are detected by all fitness function configurations, and the third exception is detected by the **CLM and CLN** fitness function configurations. **NullPointerException** is found in **palmcalc** app which is triggered only by the **CLC** fitness function configuration. Similarly, the **CLB** fitness function configuration of STGFA-SMOG revealed **ClassCastException** in the **sourceforgesolitaire** app which is not detected by the other fitness function configurations. STGFA-SMOG with a 10 generation search budget revealed three exceptions: **ActivityNotFoundException, IndexOutOfBoundsException, and ArrayIndexOutOfBoundsException** in two apps as shown in Table 6.13. Similar to the STGFA-SMOG with a 30 generation search budget, **ActivityNotFoundException** is triggered by all fitness function configurations of STGFA-SMOG with a 10 generation search budget in the **scoutantblokish** app. However, **IndexOut-**

OfBoundsException is triggered by **CLB, CLM, and CLN** fitness function configurations, and **ArrayIndexOutOfBoundsException** is detected only by the **CLC** fitness function configuration. The Random test generation also revealed two exceptions: **IndexOutOfBoundsException, and ArrayIndexOutOfBoundsException** in the **dib2calc** app as listed in Table 6.15.

When we compare the number of times crashes are triggered by the fitness function configurations keeping the search budget constant as shown in Table 6.12, we can see that **ActivityNotFoundException**, **IndexOutOfBoundsException**, and **ArrayIndexOutOfBoundsException** are more frequently triggered by the **CLM**, **CLB**, and **CLC** fitness function configurations sequentially. Similarly, as we can see in Table 6.14 **ActivityNotFoundException**, **IndexOutOfBoundsException**, and **ArrayIndexOutOfBoundsException** are activated more often by the **CLM**, **CLM**, and **CLC** fitness function configurations respectively.

STGFA-SMOG with a 30 generation revealed more unique crashes compared to STGFA-SMOG with a 10 generation search budget, and STGFA-SMOG with a 10 generation uncovered more crashes than Random test generation. The fitness function configurations under the 30 generation search budget activated crashes triggered under both budgets more often than the fitness function configurations under the 10 generation search budget.

CLM and CLN on the app **dib2calc**, **CLC** on the app **palmcalc**, and **CLB** on the app **sourceforgesolitaire** triggered unique crashes that are not revealed by the other fitness function configurations. Except the **IndexOutOfBoundsException** all of the crashes revealed under the 30 generation and 10 generation search budgets are triggered more often by the same fitness function configurations across all search budgets.

Table 6.11: Unique crashes detected by STGFA-SMOG with 30 generation across all trials

App	Exception	CLB	CLC	CLM	CLN
scoutantblokish	ActivityNotFoundException	✓	✓	✓	✓
dib2calc	IndexOutOfBoundsException	✓	✓	✓	✓
dib2calc	ArrayIndexOutOfBoundsException	✓	✓	✓	✓
dib2calc	NullPointerException			✓	✓
palmcalc	NullPointerException		✓		
sourceforgesolitaire	ClassCastException	✓			

Table 6.12: Average number of times the crashes are triggered by STGFA-SMOG with 30 generation per trial

App	**Exception**	**CLB**	**CLC**	**CLM**	**CLN**
scoutantblokish	ActivityNotFoundException	6.20	4.00	8.20	2.40
dib2calc	IndexOutOfBoundsException	6.80	0.60	1.80	3.60
dib2calc	ArrayIndexOutOfBoundsException	2.80	4.20	2.00	3.40
dib2calc	NullPointerException	0.00	0.00	0.20	0.20
palmcalc	NullPointerException	0.00	0.60	0.00	0.00
sourceforgesolitaire	ClassCastException	2.40	0.00	0.00	0.00

Table 6.13: Unique crashes detected by STGFA-SMOG with 10 generation across all trials

App	**Exception**	**CLB**	**CLC**	**CLM**	**CLN**
scoutantblokish	ActivityNotFoundException	✓	✓	✓	✓
dib2calc	IndexOutOfBoundsException	✓		✓	✓
dib2calc	ArrayIndexOutOfBoundsException		✓		✓
dib2calc	NullPointerException				
palmcalc	NullPointerException				
sourceforgesolitaire	ClassCastException				

Table 6.14: Average number of times the crashes are triggered by STGFA-SMOG with 10 generation per trial

App	**Exception**	**CLB**	**CLC**	**CLM**	**CLN**
scoutantblokish	ActivityNotFoundException	0.80	0.40	1.80	1.00
dib2calc	IndexOutOfBoundsException	0.60	0.00	1.60	0.40
dib2calc	ArrayIndexOutOfBoundsException	0.00	3.00	0.00	0.60
dib2calc	NullPointerException	0.00	0.00	0.00	0.00
palmcalc	NullPointerException	0.00	0.00	0.00	0.00
sourceforgesolitaire	ClassCastException	0.00	0.00	0.00	0.00

Table 6.15: Unique crashes detected by Random test generation across all trials

App	**Exception**	**Random**
scoutantblokish	ActivityNotFoundException	
dib2calc	IndexOutOfBoundsException	✓
dib2calc	ArrayIndexOutOfBoundsException	✓
dib2calc	NullPointerException	
Palmcalc	NullPointerException	
sourceforgesolitaire	ClassCastException	

Table 6.16: Average number of times the crashes are triggered by Random test generation per trial, (where Random-10 is Random test generation run for 10 generation, and Random-30 is Random test generation run for 30 generation)

App	Exception	Random-10	Random-30
scoutantblokish	ActivityNotFoundException	0.00	0.00
dib2calc	IndexOutOfBoundsException	0.05	0.05
dib2calc	ArrayIndexOutOfBoundsException	0.05	0.00
dib2calc	NullPointerException	0.00	0.00
Palmcalc	NullPointerException	0.00	0.00
sourceforgesolitaire	ClassCastException	0.00	0.00

Table 6.17: Time taken by STGFA-SMOG with 10 generation

Configurations	No. of Apps	No. of Trials	Average Time per Trial (hrs)
CLU	10	5	3
CLB	10	5	3
CLN	10	5	3
CLM	10	5	3
CLLC	1	5	6
CLMC	1	5	6

Table 6.18: Time taken by STGFA-SMOG with 30 generation

Configurations	No. of Apps	No. of Trials	Average Time per Trial (hrs)
CLU	10	5	10
CLB	10	5	10
CLN	10	5	10
CLM	10	5	10
CLLC	1	5	24
CLMC	1	5	24

6.2.2 Comparison of Techniques and Configurations

We performed a statistical analysis to assess our experimental results. Hence, in this subsection, we compared the techniques (i.e. STGFA-SMOG vs Random test generation) and each pair of fitness function configurations using statistical analysis. As the number of unique crashes was low, rather than comparing the number of unique crashes, we assess whether the discovered crashes are triggered more often by one configuration than Random test generation (or another configuration). This is indicated by the value of the "number of crashes" fitness function as measured on the final test suite produced in each trial. The test is run per each pair of fitness function configurations and techniques. Due to the stochastic nature of the search, a single run of the Random test generation may yield results of a favorable random solution or a badly selected random solution. Therefore, we perform one trial of Random test generation for each trial of any fitness function configuration of STGFA-SMOG. For each pair of techniques or configurations (combination of fitness functions and Random test generation), we perform the Mann-Whitney Wilcoxon test. Therefore, per each pair of configurations or techniques, we formulated null hypothesis H0 and its alternative hypothesis, H:

- H0: Given a fixed search budget, the observations of results for techniques or configurations A and B are drawn from the same distribution. 'A' and 'B' represent the fitness function configuration such as CLB, CLM, CLC, and CLN or a fitness function configuration and Random test generation.
- H: The observations of results for techniques or configurations A and B are drawn from different distributions.

In Table 6.19, 6.21, 6.23, 6.25, 6.27, and 6.28 the column **CLB** shows the p-value from the Mann-Whitney Wilcoxon test when comparing the distribution of fitness values for the crash fitness function between **CLB** and the other fitness function configurations or STGFA-SMOG optimizing the [(crash) + (test case length) + (Battery usage)] fitness function combination versus Random test generation. The column **CLC** in these tables compares fault-finding performance between **CLC** and the other fitness function configurations or STGFA-SMOG optimizing the [(crash) + (test case length) + (CPU usage)] fitness function combination versus Random test generation. Similarly, the column **CLM** shows the p-values from the Mann-Whitney Wilcoxon test when comparing the distribution of fitness values for the crash fitness function between **CLM** and the other configurations or STGFA-SMOG optimizing the [(crash) + (test case length) + (Memory usage)] fitness function combination versus Random test generation, while **CLN** shows p-value from the Mann-Whitney Wilcoxon test when comparing the distribution of fitness values of the crash fitness function between **CLN** and the other configurations or STGFA-SMOG optimizing the [(crash) + (test case length) + (Network usage)] fitness function combination versus Random test generation.

In Table 6.20, 6.22, 6.24, 6.26, and 6.29 the column **CLB** shows the effect size values from the Vargha-Delaney test when comparing the distribution of fitness values for the crash fitness function between **CLB** and the other configurations or STGFA-SMOG optimizing the [(crash) + (test case length) + (Battery usage)] and Random test generation. The column **CLC** shows the effect size values from the Vargha-Delaney test when comparing the distribution of fitness values for the crash fitness function between **CLC** and the other configurations or STGFA-SMOG optimizing the [(crash) + (test case length) + (CPU usage)] and Random test generation. Similarly, **CLN** shows the effect size values from the Vargha-Delaney test when comparing the distribution of fitness values for the crash fitness function between **CLN** and the other configurations or STGFA-SMOG optimizing the [(crash) + (test case length) + (Network usage)] versus Random test generation, while **CLM** shows the effect size values from the Vargha-Delaney test when comparing the distribution of fitness values for the crash fitness function between **CLN** and the other fitness function configurations or STGFA-SMOG versus Random test generation.

First, we examined the 10 generation search budget. The results in Table 6.19 and 6.21 indicate that a statistically significant difference is found in the distribution of the fitness values of the crash fitness function of STGFA-SMOG with 10 generation and Random test generation for **CLB, CLM, and CLN** configurations, and for **CLC and CLN** configurations respectively. Hence, we reject the null hypothesis for these configurations by claiming that results found by STGFA-SMOG with a 10 generation search budget and Random test generation are drawn from different distributions. Table 6.20 and 6.22 shows the corresponding effect size for the configurations where statistically significant difference is found.

At a 10 generation search budget, the **CLC** configuration outperforms random testing with small effect size on the **dib2calc** app. In all other significant cases, other configurations outperform random testing, but with only negligible effect size.

Table 6.19: P-Values for Mann-Whitney rank-sum test of the comparison of the different combination of the fitness functions of the STGFA-SMOG as well as Random test generation by the number of crashes detected in 10 generation in the **scoutantblokish** app.

	CLB	**CLC**	**CLM**	**CLN**	**Random**
CLB	-	0.41	0.15	0.74	**0.04**
CLC	0.41	-	**0.03**	0.25	0.16
CLM	0.15	**0.03**	-	0.27	**0.00**
CLN	0.74	0.25	0.27	-	**0.02**
Random	**0.04**	0.16	**0.00**	**0.02**	-

Table 6.20: Results of Vargha-Delaney A Measure of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 10 generation in the **scoutantblokish** app. Note that large effect sizes are bolded and effect size is only measured when statistical significant difference is found using the Mann-Whitney Wilcoxon test.

	CLB	**CLC**	**CLM**	**CLN**	**Random**
CLB	-	-	-	-	0.52
CLC	-	-	0.46	-	-
CLM	-	0.54	-	-	0.55
CLN	-	-	-	-	0.53
Random	0.48	-	0.45	0.47	-

Table 6.21: P-Values for Mann-Whitney rank-sum test of the comparison of the different combination of the fitness functions of the STGFA-SMOG as well as Random test generation by the number of crashes detected in 10 generation in the **dib2calc** app

	CLB	**CLC**	**CLM**	**CLN**	**Random**
CLB	-	**0.00**	0.12	0.47	0.08
CLC	**0.00**	-	0.08	**0.01**	**3.22e-05**
CLM	0.12	0.08	-	0.39	0.05
CLN	0.47	**0.01**	0.39	-	**0.02**
Random	0.08	**3.22e-05**	0.05	**0.02**	

Table 6.22: Results of Vargha-Delaney A Measure of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 10 generation in the **dib2calc** app. Note that large effect sizes are bolded and effect size is only measured when statistical significant difference is found using the Mann-Whitney Wilcoxon test.

	CLB	**CLC**	**CLM**	**CLN**	**Random**
CLB	-	0.43	-	-	-
CLC	0.57	-	-	0.56	0.58
CLM	-	-	-	-	-
CLN	-	0.44	-	-	0.53
Random	-	0.42	-	0.47	-

Next, we compared at the 30 generation search budget. As we can see in Tables 6.23 and 6.25 there exists a statistically significant difference for all the comparisons between the fitness function configurations and Random test generation. Therefore, we reject the null hypothesis for these comparisons and we claim that the results are from a different distribution. In addition, a statistically significant difference exists between the comparison of **CLB** configuration and Random test generation as shown in Table 6.28. Hence, we failed to reject the null hypothesis for all comparisons between the configurations and Random test generation except between **CLB** and Random test generation. In Table 6.24 and 6.26 we can see that STGFA-SMOG with 30 generation has a medium effect size over Random test generation for **CLB** configuration and for **CLN** configuration sequentially. Moreover, STGFA-SMOG has a large effect size over Random test generation for **CLM** configuration and for **CLB** configuration respectively. In contrast, the other effect size values appear to be a small effect size which can be interpreted as STGFA-SMOG with 30 generation with a small effect size over Random test generation in the number of crashes revealed.

At a 30 generation search budget, STGFA-SMOG outperforms random testing in all cases where results are found to be drawn from different distributions (2 large effect size, 2 medium, 5 small). In addition, we see improvements between 10 and 30 generations.

We now compared the configurations of STGFA-SMOG to each other. First, we examined a 10 generation search budget. As shown in Table 6.19 we can see a statistically significant difference only in the comparison of the configurations **CLC vs CLM**. Similarly, in Table 6.21 only the comparison of the configurations **CLB vs CLC** and **CLC vs CLN** have a statistically significant difference. Based on these two tables, it seems that statistical difference does not exist for the majority of the entries. Hence, we reject the null hypothesis only for the comparison of the configurations **CLC vs CLM** in Table 6.19 and **CLB vs CLC**, and **CLC vs CLN** in Table 6.21. The corresponding effect size measure for the comparison of configurations where their null hypothesis is rejected is shown in Tables 6.20 and 6.22. Table 6.22 shows **CLC** configuration outperformed the **CLB and CLN** configurations and their effect size values belong to a small effect size. Similarly, as we can see in Table 6.20 **CLM** has a higher effect size value than **CLC**. However, the effect size value is below the small effect size range.

At a 10 generation budget, **CLC** outperforms **CLB** and **CLN** with small effect size for the **dib2calc** app. For **scoutantblockish**, **CLM** outperforms **CLC** with negligible effect size. We do not see a discernible pattern where one configuration outperforms the others.

Next, we compared the configurations at 30 generations. As we can see in Tables 6.25 and 6.28 configuration **CLB** compared to **CLC**, and configuration **CLB** compared to **CLM** have a statistically significant difference. Similarly, configuration **CLB** compared to **CLN** has a statistically significant difference as shown in Tables 6.23 and 6.28. Furthermore, configuration **CLC** compared to **CLM** has a statistically significant difference in the **scoutantblokish** app as in Table 6.23 and configuration **CLM** compared to **CLN** does on **dib2calc** and **scoutantblokish** apps as shown in Table 6.25 and 6.23 sequentially. We reject the null hypothesis for the **CLB vs CLC and CLB vs CLM** in Table 6.25 and 6.28, and **CLB vs CLN** in Table 6.23 and 6.28. For the remaining configurations compared in Tables 6.23, 6.25, 6.27, and 6.28, we failed to reject the null hypothesis. Configuration **CLB** performed better than **CLC and CLM** in the **dib2calc and sourceforgesolitaire** apps as shown in Tables 6.26 and 6.29, and configuration **CLB** performed better than **CLN** in the **scoutantblokish and sourceforgesolitaire** apps as listed in Tables 6.24 and 6.29 whereas configuration **CLM** outperformed **CLC and CLN** configurations in the **scoutantblokish** app (i.e. Table 6.24) and configuration **CLN** outperformed **CLM** in the **dib2calc** app (i.e. Table 6.26).

Table 6.23: P-Values for Mann-Whitney rank-sum test of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 30 generation in the **scoutantblokish** app.

	CLB	**CLC**	**CLM**	**CLN**	**Random**
CLB	-	0.08	0.14	**0.00**	**1.54e-09**
CLC	0.08	-	**0.00**	0.12	**2.60e-06**
CLM	0.14	**0.00**	-	**3.60e-06**	**7.97e-13**
CLN	**0.00**	0.12	**3.60e-06**	-	**0.00**
Random	**1.54e-09**	**2.60e-06**	**7.97e-13**	**0.00**	-

At a 30 generation search budget, the **CLB** configuration shows limited evidence for being the best configuration (for **dib2calc**, it outperforms **CLM** with medium effect size and **CLC** with small; for **sourceforgesolitaire**, it outperforms **CLC**, **CLM**, and **CLN** with small effect size). However, the best configuration is likely dependent on the app. For example, for **scoutantblockish**, **CLM** outperformed **CLN** with medium effect and **CLC** with small). More research is needed to find clear patterns when to use certain configurations.

Table 6.24: Results of Vargha-Delaney A Measure of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 30 generation in the **scoutantblokish** app. Note that large effect sizes are bolded and effect size is only measured when statistical significant difference is found using the Mann-Whitney Wilcoxon test.

	CLB	**CLC**	**CLM**	**CLN**	**Random**
CLB	-	-	-	0.60	0.66
CLC	-	-	0.39	-	0.60
CLM	-	0.61	-	0.64	**0.71**
CLN	0.40	-	0.36	-	0.56
Random	0.34	0.40	0.29	0.44	

Table 6.25: P-Values for Mann-Whitney rank-sum test of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 30 generation in the **dib2calc** app.

	CLB	**CLC**	**CLM**	**CLN**	**Random**
CLB	-	**0.00**	**4.95e-05**	0.10	**5.67e-15**
CLC	**0.00**	-	0.50	0.06	**9.43e-07**
CLM	**4.95e-05**	0.50	-	**0.01**	**2.60e-06**
CLN	0.10	0.06	**0.01**	-	**3.91e-11**
Random	**5.67e-15**	**9.43e-07**	**2.60e-06**	**3.91e-11**	

Table 6.26: Results of Vargha-Delaney A Measure of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 30 generation in the **dib2calc** app. Note that large effect sizes are bolded and effect size is only measured when statistical significant difference is found using the Mann-Whitney Wilcoxon test.

	CLB	**CLC**	**CLM**	**CLN**	**Random**
CLB	-	0.62	0.64	-	**0.74**
CLC	0.38	-	-	-	0.62
CLM	0.36	-	-	0.42	0.60
CLN	-	-	0.58	-	0.68
Random	0.26	0.38	0.40	0.32	-

Table 6.27: P-Values for Mann-Whitney rank-sum test of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 30 generation in the **palmcalc** app.

	CLB	**CLC**	**CLM**	**CLN**	**Random**
CLB	-	0.08	NA	NA	NA
CLC	0.08	-	0.08	0.08	0.08
CLM	NA	0.08	-	NA	NA
CLN	NA	0.08	NA	-	NA
Random	NA	0.08	NA	NA	-

Table 6.28: P-Values for Mann-Whitney rank-sum test of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 30 generation in the **sourceforgesolitaire** app.

	CLB	**CLC**	**CLM**	**CLN**	**Random**
CLB	-	**0.00**	**0.00**	**0.00**	**0.00**
CLC	**0.00**	-	NA	NA	NA
CLM	**0.00**	NA	-	NA	NA
CLN	**0.00**	NA	NA	-	NA
Random	**0.00**	NA	NA	NA	-

Table 6.29: Results of Vargha-Delaney A Measure of the comparison of the different combination of the fitness functions of the STGFA-SMOG and Random test generation by the number of crashes detected in 30 generation in the **sourceforgesolitaire** app. Note that large effect sizes are bolded and effect size is only measured when statistical significant difference is found using the Mann-Whitney Wilcoxon test.

	CLB	**CLC**	**CLM**	**CLN**	**Random**
CLB	-	0.56	0.56	0.56	0.56
CLC	0.44	-	-	-	-
CLM	0.44	-	-	-	-
CLN	0.44	-	-	-	-
Random	0.44	-	-	-	

6.2.3 Effect of Search Budget

In this subsection, we compared the effect of the search budget on the number of crashes detected by the different fitness function configurations. We carry out the test per each pair of the fitness function configurations of each pair of search budgets. Hence, per the pair of the search budgets, we formulated null hypothesis H0 and its alternative hypothesis, H:

- H0: The observations of results for crashes revealed by STGFA-SMOG with both 30 and 10 generation search budgets are drawn from the same distribution.
- H: The observations of results for crashes revealed by STGFA-SMOG with 30 generation search budget have a different distribution than the observations of results for crashes revealed by STGFA-SMOG with 10 generation search budget.

In Table 6.30 we can see that a statistically significant difference between the distribution of the fitness values of STGFA-SMOG with 30 generation and 10 generation appears in the **scoutantblokish** app for three configurations: **CLB**, **CLC**, and **CLM**, in the **dib2calc** app for three configurations: **CLB**, **CLM**, and **CLN**, and in the **sourceforgesolitaire** app for one configuration: **CLB**. As a result, we reject the null hypotheses for these configurations claiming that the fitness values of these fitness function configurations are from different distributions. The effect size measure for these configurations where a statistically significant difference exists is presented in Table 6.31. STGFA-SMOG with 30 generation for **CLB** configuration on the **dib2calc** app is shown to have a large effect size over STGFA-SMOG with 10 generation. In addition, it can be seen that a medium effect size on the **scoutantblokish** app for two configurations: **CLB** and **CLM**, and on the **dib2calc** app for one configuration: **CLN**, and small effect size for the rest of the configurations.

Table 6.30 shows that a statistically significant difference was found in three configurations (i.e. **CLB, CLC, and CLM**) on the scoutantblokish app, three configurations (i.e. **CLB, CLM, and CLN**) on the dib2calc app, and one configuration (i.e. **CLB**) on the sourceforgesolitaire app. From the statistical analysis of the effect of search budget, we observed that STGFA-SMOG with 30 generations outperformed STGFA-SMOG with 10 generations in all the configurations where a statistically significant difference is found.

Table 6.30: P-Values for Mann-Whitney rank-sum test of the crashes detected by STGFA-SMOG with 30 generation versus 10 generation

	CLB	**CLC**	**CLM**	**CLN**
scoutantblokish	**5.44e-07**	**5.00e-05**	**1.88e-07**	0.08
dib2calc	**7.80e-13**	0.16	**0.01**	**6.15e-08**
palmcalc	NA	0.08274	NA	NA
sourceforgesolitaire	**0.00**	NA	NA	NA

Table 6.31: Results of Vargha-Delaney A Measure of the crashes detected by STGFA-SMOG with 30 generation versus 10 generation

	CLB	**CLC**	**CLM**	**CLN**
scoutantblokish	0.64	0.59	0.66	-
dib2calc	**0.72**	-	0.56	0.66
palmcalc	-	-	-	-
sourceforgesolitaire	0.56	-	-	-

7 Discussion

In this research, an empirical study was conducted on a newly developed search-based test generation framework for Android GUI-based testing to investigate whether a certain combination of fitness functions reveal more crashes than other configurations and whether a large search budget uncovers more crashes than a small search budget, which is a number of generations of solution evolution in this study. Hence, this chapter discusses the findings of the research with respect to the four research questions based on the experimental results depicted in Section 6.2.

7.1 RQ1: Framework Design

How can a search-based test generation framework for Android GUI-based testing best be designed to support multi-objective generation with support for the inclusion of additional fitness functions over time?

To design multi-objective generation, we applied a non-dominated sorting genetic algorithm (NSGA-II). The use of NSGA-II enabled STGFA-SMOG to prevent the loss of elitist test cases during the generation of test cases. Some elitist test cases found in one generation may be lost at some point in the next generation. By adopting NSGA-II we are able to generate multi-objective test cases and overcome the challenge of losing elitist test cases. Using NSGA-II our framework is also able to generate diversified Pareto-optimal test cases that fulfill different goals at the same time. The adoption of a factory method design pattern to implement the fitness functions enabled our framework to be open for the inclusion of new additional fitness functions over time. We demonstrated this capability by implementing the fitness functions in three cycles. In the first cycle, we implemented CPU usage and Memory usage fitness functions by following the factory method design pattern. In the second cycle, we added the Battery usage and Network usage fitness functions without modifying the existing implementation of the existing fitness functions. Finally, we added the class, method, and line coverage fitness functions by simply adding the implementation of these fitness functions and including them in the factory pattern without modifying the existing implementation.

In the intermediate evaluation, we compared the fitness values of STGFA-SMOG fitness functions configurations and Random test generation. We also compared the fitness values of the STGFA-SMOG fitness function configurations at two search budgets (i.e. 10 and 15 generation search budgets) The intermediate evaluation showed that our framework was able to generate tests with higher fitness values in each of these function configurations than Random testing. Similarly, the test cases generated by the large search budget (i.e. 15 generation) gave higher fitness values than the test cases generated by the small search budget (i.e. 10 generation). This means that our framework is better able to show certain qualities of interest (CPU usage, memory usage, battery usage, network usage, code coverage) than random test generation. Regardless of the crash results, our framework is effectively able to generate tests for assessment of non-functional qualities. If testers are interested in a quality of interest, they can generate tests and see if any pass a set threshold indicating an issue. For example, a tester concerned with CPU usage could set a threshold (e.g., 80% usage). They could generate tests, and investigate the results of any where CPU usage passed 80%. This framework is better able to generate such tests than the baseline. This can benefit testers interested in non-functional qualities or performance tests. The intermediate results show that RQ1 has been answered successfully.

7.2 RQ2: Effectiveness of Test Cases

How effective are the generated test cases at causing crashes in the assessed apps, in comparison to random test generation?

Our intermediate experimental results confirmed that there is a significant difference between STGFA-SMOG and random test generation that the effect size turned in favor of STGFA-SMOG. Some irregularities are observed on the significant difference when it comes to the final experimental results mainly on the comparison between STGFA-SMOG with 10 generation and Random test generation where we can see a small effect size for many of the entries. The size of the search budget might be the reason for the small significance. As of the final experiment from STGFA-SMOG with 30 generations, the exceptions ActivityNotFoundException, IndexOutOfBoundsException, and ArrayIndexOutOfBoundsException have been triggered four times each since NullPointerException has been triggered three times (one fault is triggered twice in app dib2calc while another fault of this type trigger once in app palmcalc) and ClassCastException was only triggered once. Exception ActivityNotFoundException was triggered four times, IndexOutOfBoundsException triggered three times while ArrayIndexOutOfBoundsException was triggered twice. Only two types of exceptions (IndexOutOfBoundsException and ArrayIndexOutOfBoundsException) are found by the Random test data generation which both were triggered only once. These results confirm the claim that STGFA-SMOG outperforms Random test generation. The triggering of these faults throughout trials is also a good indication of the credibility of our results. When it comes to the frequency of these crashes, the average number of times the crashes are triggered by STGFA-SMOG is found to be much higher than the random test generation for both search budgets.

7.3 RQ3: Effectiveness of the Combinations of Fitness Functions

Which combinations of fitness functions is the most effective at causing the assessed apps to crash?

In the final experiment, this study assessed the effect of having a certain type of combination of fitness function on finding crashes. Combinations of fitness functions compared with other combinations to see if they affect the effectiveness of the test cases in terms of the frequency of crashes triggered. This was done by setting the search budget as a control group. For the 10 generation budget, though statistical difference exists between combination **CLB** vs **CLC**, **CLC** vs **CLM** and **CLC** vs **CLN** for apps **dib2calc**, **scoutantblokish** and **dib2calc** respectively all the effect sizes are negligible. For the 30 generation search budget though there are some more combinations with a statistical difference the effect sizes are limited to medium and small. Configuration **CLB** can be seen outperforming the other configuration on **dib2calc**. But it is difficult to claim significant effect with the data we have since we were unable to see any large effect size and we believe further study is needed to see if this configuration outperforms the configurations in other apps. Here in our experiment, there is no permanent configuration that is consistently best at revealing faults for different apps. In future work, the scope of experiments will be extended to look at a larger number of apps, a wider variety of fitness function combinations, and additional search budgets.

7.4 RQ4: Effect of Search Budget

Does an increased search budget (the number of generations) improve the effectiveness of the resulting test cases?

As it has been mentioned a couple of times in the above sections increasing the search budget significantly improves the fitness scores and the ability of test cases to find faults. This is true for the intermediate and final evaluations. The statistical difference with quite many large effect sizes for the comparing STGFA-SMOG with 15 versus 10 generations is shown in intermediate evaluation while we can also confirm based on the number of crashes found in the final evaluation. Based on the final experiment from STGFA-SMOG with 30 generations, the exceptions ActivityNotFoundException, IndexOutOfBoundsException, and ArrayIndexOutOfBoundsException have been triggered four times each since NullPointerException has been triggered three times and ClassCastException was only triggered once. Exception ActivityNotFoundException was triggered four times, IndexOutOfBoundsException triggered three times while ArrayIndexOutOfBoundsException was triggered twice. In this case, we can see how the frequency where the occurrence of the faults improved on the 30 generation search budget. The unique exceptions found by STGFA-SMOG with 30 generations which in this case are five is a little higher than the 10 generation search budget(three in this case). Similarly, the average number of times the crashes are triggered by STGFA-SMOG with a 30 generation search budget per trial is higher than this of STGFA-SMOG with 10 generations for all configurations included in our experiment. This shows the test suite's capability to find error increases as the search budget increases.

7.5 Threats to Validity

7.5.1 Internal Validity

The test generation takes a long time (see Tables 6.17 and 6.18) which makes it difficult to have sufficient data for analysis. Having limitations in the quantity of input data can affect the conclusion of the experiment. It would have been helpful if we got data for more search budgets as some of the results show inconsistencies though we can take the pattern seen with the given search budgets. Studying the effect of selecting a certain type of combination of fitness function on a quality of test cases which is RQ2 is a good example of this. Ultimately, we were not able to use line coverage and method coverage in the final experiment and we could not make it due to time limitation. However, we believe that the quantity of data we gathered in this experiment is sufficient to demonstrate the potential effectiveness of this framework (i.e., to demonstrate that it can attain higher fitness values and trigger more crashes than a baseline), even if we were not able to attain clear answers regarding which fitness configurations were more effective.

7.5.2 External Validity

We limited the number of android apps in our experiment due to time factors that may not optimally represent all the other apps out of the study. We chose to randomly select the apps to minimize bias on input selection that might lead to a wrong conclusion. In addition, our framework is only compared with random test case generation since comparison with random search is a standard procedure. Hence, we do not know how it performs in comparison to other existing techniques such as Swift-Hand, Sapienz, and EvoDroid. As a result, we do not have any claims, suggestions, or results related to such comparisons. We preferred to study which combination of fitness functions better reveals more fault as there is no standard to guide testers which configuration to use and when. In case a specific configuration reveals more fault than others it will minimize testing cost by saving time that would have been wasted in generating test cases by many configurations for repetitive work and instead use the better configuration only.

Moreover, we took volume control events, simple atomic events like swipe and tap, but not complex user gestures such as long-press and sophisticated system events such as fingerprint recognizer. Our framework is effective on Android UI that does not have login pages. This is because we did not validate the credentials and that could be one trade-off when applying our framework beyond the context mentioned here. In the comparison of performance among different configurations (RQ3) the result seems inconsistent and what we observed is an unstable pattern. When conclusions are made here in our experiment we carefully saw the results not to conclude prematurely that we left some issues open for further investigation mainly for those we were unable to see a clear pattern.

7.5.3 Conclusion Validity

Conclusion validity concerns the empirical design to ensure that a statistical relationship exists between treatment and outcome. So, when selecting statistical analyses to analyze our experimental results, we have attempted to ensure the base assumptions of the selected statistical analyses are met. We have favored non-parametric methods over parametric methods to analyze our experimental results. This is because the distribution characteristics of the results of search-based test generation are not generally known a priori, and normality cannot be assumed.

8 Conclusion

8.1 Conclusion

In multi-objective search-based test generation, the effectiveness of the test cases depends on the selection of effective fitness functions. However, it is not yet known which combination of fitness functions is more effective at triggering more crashes and achieving other test goals, such as identifying excessive CPU usage, memory usage, and maximizing code coverage for Android GUI-based testing. It is also not well known the effect of search budgets on the effectiveness of the test cases for Android GUI-based testing. Moreover, existing multi-objective search-based test generation tools for Android GUI-based testing do not allow testing to be adapted to different specific goals such as maximizing code coverage, detecting faults, and identifying excessive CPU, memory, and battery usage. Hence, we are motivated to conduct this study so that we can add some scientific knowledge to the multi-objective search-based test generation for the Android GUI-based testing domain.

In this study, to investigate the problems mentioned in the previous paragraph, we developed a multi-objective search-based test generation for Android GUI-based testing called STGFA-SMOG. STGFA-SMOG allows users to select three fitness functions depending on the goal the user wants to achieve and set the search budget (i.e. number of generations). It also allows the addition of more fitness functions over time. The tool can be used by developers or testers to generate test cases that can be used to achieve different goals. It can be used to generate test cases to trigger faults in the AUT. Other developers can also use it to generate test cases for performance tests such as for identifying test cases that achieve excessive memory, CPU, battery, and network usage. Using the framework developed, we also conducted empirical research intending to guide how to choose fitness functions and which of them to combine for effective test case generation. It also aims to guide which search budgets performed best with these combinations of fitness functions that gave effective test cases with respect to the goal set. To achieve these goals from the empirical research, we formulated a number of research questions related to how the framework should be designed to ensure effective test cases, the effectiveness of the generated test cases at triggering crashes in comparison to Random test generation, which combinations of fitness functions best trigger crashes, and effect of search budgets on the effectiveness of the test cases at triggering crashes.

The intermediate evaluation shows that STGFA-SMOG generates test cases with higher fitness values than Random test generation. Hence, we concluded that STGFA-SMOG outperformed Random test generation at generating effective test cases for non-functional tests such as CPU usage, memory usage, battery usage, network usage, and code coverage. Moreover, the final evaluation also shows that STGFA-SMOG outperformed Random test generation at generating test cases that trigger more crashes than test cases generated by Random test generation. For the comparison of the fitness function configuration of STGFA-SMOG to each other, we did not see a consistent pattern where one configuration outperforms the other configurations. When the search budget is increased, we found that some configurations outperformed the other configurations. However, the best configuration is dependent on the AUT. Our study also revealed that STGFA-SMOG with a large search budget generated test cases with higher fitness values and test cases that trigger crashes more often than STGFA-SMOG with a small search budget. That is an increase in search budget results in increasing the performance of the STGFA-SMOG.

Our study constitutes a step towards understanding the design of multi-objective search-based test generation for Android GUI-based testing with support for the addition of more fitness functions over time, the effect of search budget, and effective combination of fitness functions to chose for Android GUI-based testing. Our study results are promising and more research is needed to better understand them. In the future, we recommend expanding the scope of the experiment to look at a larger number of apps, a wider variety of fitness function combinations, and additional search budgets. We also recommend implementing an additional feature that enables the framework to generate test cases based on the widgets in the current context of the AUT in addition to randomly generating test cases.

Bibliography

[1] P. McMinn, "Search-based software testing: Past, present and future," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 153–163.

[2] "Smartphone users 2020." [Online]. Available: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

[3] "Number of Android applications on the Google Play store." [Online]. Available: https://www.appbrain.com/stats/number-of-android-apps

[4] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, and A. Bertolino, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013, publisher: Elsevier.

[5] P. McMinn, "Search-based software test data generation: a survey," *Software testing, Verification and reliability*, vol. 14, no. 2, pp. 105–156, 2004, publisher: Wiley Online Library.

[6] A. Salahirad, H. Almulla, and G. Gay, "Choosing the fitness function for the job: Automated generation of test suites that detect real faults," *Software Testing, Verification and Reliability*, vol. 29, no. 4-5, p. e1701, 2019, publisher: Wiley Online Library.

[7] S. Shamshiri, J. M. Rojas, G. Fraser, and P. McMinn, "Random or genetic algorithm search for object-oriented test suite generation?" in *Proceedings of the 2015 annual conference on genetic and evolutionary computation*, 2015, pp. 1367–1374.

[8] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 2007, pp. 1098–1105.

[9] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 94–105.

[10] S. Holla and M. M. Katti, "Android based mobile application development and its security," *International Journal of Computer Trends and Technology*, vol. 3, no. 3, pp. 486–490, 2012, publisher: Citeseer.

[11] P. Kong, L. Li, J. Gao, K. Liu, T. F. Bissyandé, and J. Klein, "Automated testing of android apps: A systematic literature review," *IEEE Transactions on Reliability*, vol. 68, no. 1, pp. 45–66, 2018, publisher: IEEE.

[12] "3.1 The testing pyramid · GitBook." [Online]. Available: https://sttp.site/chapters/pragmatic-testing/testing-pyramid.html

[13] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 280–291.

[14] T. Manning, R. D. Sleator, and P. Walsh, "Naturally selecting solutions: the use of genetic algorithms in bioinformatics," *Bioengineered*, vol. 4, no. 5, pp. 266–278, 2013, publisher: Taylor & Francis.

[15] P. Patel, G. Srinivasan, S. Rahaman, and I. Neamtiu, "On the effectiveness of random testing for Android: or how i learned to stop worrying and love the monkey," in *Proceedings of the 13th International Workshop on Automation of Software Test*, 2018, pp. 34–37.

[16] "UI/Application Exerciser Monkey." [Online]. Available: https://developer.android.com/studio/test/monkey

[17] T. Wetzlmaier, R. Ramler, and W. Putschögl, "A framework for monkey GUI testing," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 416–423.

[18] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.

[19] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 599–609.

[20] A. S. Sayyad and H. Ammar, "Pareto-optimal search-based software engineering (POSBSE): A literature survey," in *2013 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*. IEEE, 2013, pp. 21–27.

[21] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 429–440.

[22] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE software*, vol. 32, no. 5, pp. 53–59, 2014, publisher: IEEE.

[23] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," *Acm Sigplan Notices*, vol. 48, no. 10, pp. 623–640, 2013, publisher: ACM New York, NY, USA.

[24] W. Wang, D. Li, W. Yang, Y. Cao, Z. Zhang, Y. Deng, and T. Xie, "An empirical study of android test generation tools in industrial cases," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 738–748.

[25] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of Android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[26] L. Sell, M. Auer, C. Frädrich, M. Gruber, P. Werli, and G. Fraser, "An empirical evaluation of search algorithms for app testing," in *IFIP International Conference on Testing Software and Systems.* Springer, 2019, pp. 123–139.

[27] I. A. Moreno, J. P. Galeotti, and D. Garbervetsky, "Algorithm or Representation? An empirical study on how SAPIENZ achieves coverage," in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, 2020, pp. 61–70.

[28] S. Dashevskyi, O. Gadyatskaya, A. Pilgun, and Y. Zhauniarovich, "The influence of code coverage metrics on automated testing efficiency in android," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2216–2218.

[29] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for android: Are we really there yet in an industrial case?" in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 987–992.

[30] J. Campos, Y. Ge, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for test suite generation," in *International Symposium on Search Based Software Engineering.* Springer, 2017, pp. 33–48.

[31] T. Vogel, C. Tran, and L. Grunske, "Does Diversity Improve the Test Suite Generation for Mobile Applications?" in *International Symposium on Search Based Software Engineering.* Springer, 2019, pp. 58–74.

[32] N. Albunian, G. Fraser, and D. Sudholt, "Causes and effects of fitness landscapes in unit test generation," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, 2020, pp. 1204–1212.

[33] H. N. Yasin, S. H. A. Hamid, R. J. R. Yusof, and M. Hamzah, "An empirical analysis of test input generation tools for android apps through a sequence of events," *Symmetry*, vol. 12, no. 11, p. 1894, 2020, publisher: Multidisciplinary Digital Publishing Institute.

[34] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012, publisher: IEEE.

[35] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002, publisher: IEEE.

[36] V. Khare, X. Yao, and K. Deb, "Performance scaling of multi-objective evolutionary algorithms," in *International conference on evolutionary multi-criterion optimization.* Springer, 2003, pp. 376–390.

[37] M. Köppen and K. Yoshida, "Substitute distance assignments in NSGA-II for handling many-objective optimization problems," in *International Conference on Evolutionary Multi-Criterion Optimization.* Springer, 2007, pp. 727–741.

[38] M. Elarbi, S. Bechikh, A. Gupta, L. B. Said, and Y.-S. Ong, "A new decomposition-based NSGA-II for many-objective optimization," *IEEE transactions on systems, man, and cybernetics: systems*, vol. 48, no. 7, pp. 1191–1210, 2017, publisher: IEEE.

[39] "Factory Method." [Online]. Available: https://refactoring.guru/design-patterns/factory-method

[40] A. Pilgun, O. Gadyatskaya, Y. Zhauniarovich, S. Dashevskyi, A. Kushniarou, and S. Mauw, "Fine-grained code coverage measurement in automated black-box Android testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–35, 2020, publisher: ACM New York, NY, USA.

[41] N. Li, X. Meng, J. Offutt, and L. Deng, "Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (experience report)," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 380–389.

[42] D. Tengeri, F. Horvath, A. Beszedes, T. Gergely, and T. Gyimothy, "Negative effects of bytecode instrumentation on Java source code coverage," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 225–235.

[43] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of management information systems*, vol. 24, no. 3, pp. 45–77, 2007, publisher: Taylor & Francis.

[44] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS quarterly*, pp. 75–105, 2004, publisher: JSTOR.

[45] F.-M. De Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, and C. Gagné, "Deap: A python framework for evolutionary algorithms," in *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, 2012, pp. 85–92.

[46] I. A. Moreno, "Search-based test generation for Android apps," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2020, pp. 230–233.

[47] S. Paydar, "An Empirical Study on the Effectiveness of Monkey Testing for Android Applications," *Iranian Journal of Science and Technology, Transactions of Electrical Engineering*, vol. 44, no. 2, pp. 1013–1029, 2020, publisher: Springer.

[48] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1–10.

[49] H. Almulla and G. Gay, "Learning how to search: generating exception-triggering tests through adaptive fitness function selection," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 63–73.

[50] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *International Symposium on Search Based Software Engineering*. Springer, 2011, pp. 33–47.

[51] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[52] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000, publisher: Sage Publications Sage CA: Los Angeles, CA.

A
Appendix 1

settings.py

```
# === GA parameters ===
SEQUENCE_LENGTH_MIN = 20
SEQUENCE_LENGTH_MAX = 50
POPULATION_SIZE = 10
OFFSPRING_SIZE = 10
NGENERATION = 10
# Crossover probability
CXPB = 0.3
# Mutation probability
MUTPB = 0.3
# Reproduction probability
REPROPB = 0.15
# Configure combination of fitness function to use
# e.g ["crash", "length", "cpu"/"memory"/"network"/"battery"]
FITNESS_FUNCS = ["crash", "length", "cpu"]
# Configure fitness weights
# (crash, length, cpu/memory/network/battery)
# (1.0, -1.0, 1.0) order does matter
FITNESS_WEIGHTS = (1.0, -1.0, 1.0,)

# === Query CPU, Memory, Network, and Battery usage ===
CPU_INTERVAL = 3
MEM_INTERVAL = 3
NET_INTERVAL = 3
BATT_INTERVAL = 3

# === Top best individuals to return ===
BEST_INDIV = 10
```