by

Azmain Amin

2017

© 2017 Azmain Amin
All Rights Reserved

2.1 Web Applications 4

2.1.1 User Requests, Access Logs, and User Sessions 6

2.2 Web application testing 6

2.2.1 Importance and Challenges of testing 6
2.2.2 Types of Testing 7

2.2.2.1 User-session-based Testing 8
2.2.2.2 Regression testing 8

2.2.3 Evaluating Test Suites 8

2.3 Genetic Algorithms 9

2.3.1 Selection 9

2.3.1.1 Tournament Selection 9

2.3.2 Reproduction 10

2.3.2.1 Crossover 10

2.3.2.2 Mutation 10

2.4 Conclusion of Open Problems 10

3 APPLYING GENETIC ALGORITHM TO WEB APPLICATION TESTING 12

3.1 Problem Statement 12
3.2 Goals and Non-Goals 13
3.3 Model 13
3.4 Genetic Algorithm Process 13

3.4.1 Converting user sessions into genomes 14
3.4.2 Generating the initial population 14
3.4.3 Selection 15

3.4.3.1 Fitness Function 16

3.4.4 Reproduction 16

3.4.4.1 Crossover 17
3.4.4.2 Mutation 18

3.5 Related Work 18
3.6 Prototype Implementation 19

3.6.1 Converting Web App Usage into Genomes 20
3.6.2 Selection and Reproduction 20

3.6.2.1 Fitness Function 21

3.7 Conclusion 22

4 EXPERIMENTAL STUDY 23

4.1 Research Questions and Hypotheses 23

4.1.1 Research Questions 23
4.1.2 Hypotheses 23

4.2 Subject Application 23
4.3 Methodology 24

4.4 Results 25

4.4.1 Comparing the size of the initial population 25
4.4.2 Mutation Threshold 29
4.4.3 Number of Generations 33

4.5 Analysis 35

4.5.1 Comparing the size of the initial population 35
4.5.2 Mutation Threshold 36
4.5.3 Number of generation 36

4.6 Recommendations to Testers 37
4.7 Summary 37

5.1 Contributions 39
5.2 Future Work 40

4.1 Subject Application Characteristics 24

4.2 Characteristics of User Session Sets 24

LIST OF FIGURES

2.1 Web Application Architecture 5

2.2 A HTTP Request 6

2.3 Architecture of Genetic Algorithm 11

3.1 Gene, Chromosome, and Genome 14

3.2 Genetic Algorithms for Web Application Testing 15

3.3 Crossover for a genome 17

3.4 Mutation for a genome 18

4.1 Effect of changing initial number of chromosomes per genome on resource coverage and length of the output test suite 26

4.2 Effect of changing initial number of chromosomes per genome on resource coverage and length of the output test suite 27

4.3 Effect of changing initial number of chromosomes per genome on the average time taken to run GA for 30 generations. 29

4.4 Effect of changing mutation threshold on resource coverage and length of the output test suite. 30

4.5 Effect of changing mutation threshold on resource coverage and length of output test suite 31

4.6 Effect of changing mutation threshold on the average time taken to run GA for 30 generations. 32

4.7 Effect of number of generations on resource coverage and length of the output test suite 33

4.8 E ect of increasing number of generations on the average time taken for the framework 34

Web application testing is an integral part of the web application development process. Faults within a web application can damage a company's reputation and lead to nancial losses. Customers will lose con dence if they experience inconvenience. Rigorous testing is necessary to expose faults before production release. Test case generation is a time- and resource-consuming process. Testing requirements increase exponentially with code size, and it might be impossible to exhaustively test any suf- ciently complex software. This is specially true of web apps where you have multiple platforms integrating together.

In this thesis, I propose the use of genetic algorithm to generate usage-based test cases. Genetic-algorithm-based test case generation requires considerably less resources and is customizable and automated. I modeled usage-based test cases (i.e., user sessions) as components of genetic algorithm, namely genes, chromosomes and genomes, and created a customizable and automated genetic-algorithm-based testing framework. I carried out several sets of experiments, running the genetic algorithm and tuning various parameters to evaluate the e ect of each parameter on the resulting generated test suite. Our results show that genetic-algorithm-based test case generation is very cost e ective. The test suite is considerably smaller in size compared to the initial collection of user sessions and still maintained high resource coverage.

A web application, web app in short, is a web-based software application that users access over the Internet through a web browser. Web apps are dynamic in nature, involving interaction with users, whereas websites are static and primarily used to disseminate information. Web applications might include web pages, which are static, e.g., an about page, in addition to dynamic pages.

The use of web apps have dramatically increased over the last decade. Most of consumer-based software products are cloud-based since data has to be shared among millions of people. Examples include Facebook, Twitter, etc. Companies that provides software as a service (SaaS) heavily rely on cloud computing and most of these software are web-based, e.g., Salesforce. Web apps provide a general platform for users to share their data, be it consumers or enterprises. Web apps are not restricted to desktop computers, since they can be used on mobile devices as well.

Software testing is an integral part of the software development process. Testing improves performance and customer satisfaction, which translates to more pro t for companies. Software failures can cause millions of dollars worth of damage. In April 2015, Starbucks was forced to close down 60% of their chain stores in USA and Canada due to a software glitch that a ected the cash registers [3]. Software testing is expensive [6, 9] According to Srivasta et al., developers spend 50% of the software system development resources on testing [14]. A fundamental part of software testing is test-case generation. A test case is a set of conditions under which a tester will determine whether a web app under test satis es requirements or works correctly. Test cases are used to expose faults within the web app and they are building blocks of

software testing. Creating more test cases requires more resources. Moreover, manual test-case generation is very time consuming and does not guarantee the best test cases since it depends on the engineers writing them. Therefore, the goal is to reveal maximum number of faults using the least number of test cases.

I propose using genetic algorithms to automate the process of test-case generation for web applications. Using logged user sessions, we initialize our genetic algorithm. The goal is to generate an evolved set of test cases that will reveal faults in the web application.

My contributions in this thesis:

1. explored the use of genetic algorithms in test-case generation of web application testing,
2. designed and implemented a model for representing web application test cases as components of genetic algorithms, namely genes, chromosomes, and genomes,
3. de ned crossover and mutation functions in terms of user sessions relevant to web application testing,
4. created a framework for automatically generating test cases (evolved user sessions) using tournament selection,
5. analyzed the results and explored which parameters have the most impact on the results,
6. provided recommendations to web application testers for how to use the framework for their process.

This thesis aims to serve as a starting point for any future researcher looking into the use of genetic algorithms to automate test case generation for web applications. I believe my approach of appropriating a genetic algorithm to work with web applications will be invaluable.

The rest of the thesis is laid out as follows:

Chapter 2 describes web applications and web application testing, pointing out key di erences between desktop application testing and web application testing. It also discusses how genetic algorithms work, de ning key terms on the way. Chapter 3 lays out the approach I took to design and implement a genetic algorithm to process

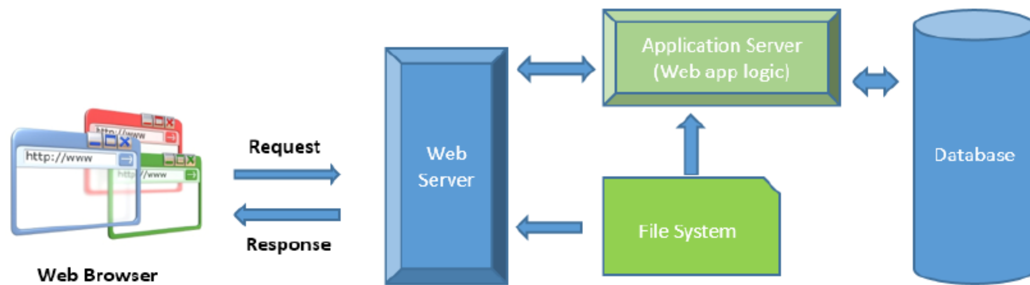
logged user sessions and generate evolved user sessions, which then become our test cases. Chapter 4 delves deeper into our hypotheses and how the experiment is designed to test out the hypotheses out. It will also present the ndings of the experiment and will analyze and evaluate them. Chapter 5 describes the limitations of the experiment and gives pointers to future work.

This chapter will give a general overview of web applications, web application testing why it is important and why it is hard, and genetic algorithms and its components. The chapter aims to arm the reader with relevant knowledge that they will require to understand the thesis.

Web applications are internet-based software applications, where the users interact with them by sending requests and receiving responses. As shown in Figure 2.1, a web application sits on a server, and the users makes requests to the server using a web browser. The server accepts the requests and generates the appropriate result and sends it back to the user. The browser renders the returned result and displays it through a user interface (UI), which is human friendly. The core component of a web app is its interactions with users. The back-end logic that handles these interactions are stored on the server that communicates with its database. Every time a user makes a request, the back-end logic determines what the returned results will be. The front-end logic handles how the results will be displayed to the user.

Consumers of web apps access data and features of the application via requests, usually a *HyperText Transfer Protocol* (HTTP) request. An HTTP request is made up of the following components: request method, resource and query string. Figure 2 provides a visual description of a HTTP request. When a web browser sends a HTTP request to the server, the server processes the method and resource and takes appropriate action. For example, if the request is to retrieve a particular web page (a GET method), then the server will look for that speci c web page by looking the

Figure 2.1: Web Application Architecture



resource and the query string. If such a web page exists, the server will then send back a response in the form of *HyperText Markup Language* (HTML) data, which the browser renders and displays in a human friendly format. If the server fails to nd the resource or there is a fault in the back-end logic of the web application, the server will respond with the appropriate error.

Web applications are usually dynamic, meaning the content of the web pages are not static. The content of the web page depends on the back-end code. If the content is derived from a database, the content might change as the database changes or updates. The content might also change due to user interactions. The database and other state is known as *persistent state* because state generated by one user can persist beyond the user's time interacting with the application.

A simple example is a web application that displays a calendar. The calendar value will change as days pass. On the other hand, the content of static pages do not change, unless the creator explicitly changes them manually. Faults in the dynamic parts of the web application are more common and more expensive to x than faults on a static page. Dynamic web applications are more complex since they involve intricate back-end logic and usually databases. A fault in a heavily used section/feature of the code can cause the whole web application to crash. Therefore, testing dynamic web applications is harder and more important.

Web browsers generally make requests to the server using the *HyperText Transfer Protocol*. A request has the following core components: request method that dictates what kind of request it is, request resource composed of resource path and resource name that acts as a *Uniform Resource Location (URL)* which allows the server to lter and serve the correct response and query string that passes in the data a server might need to return the correct response or save in the database. Figure 2.2 shows the overview of a HTTP request.

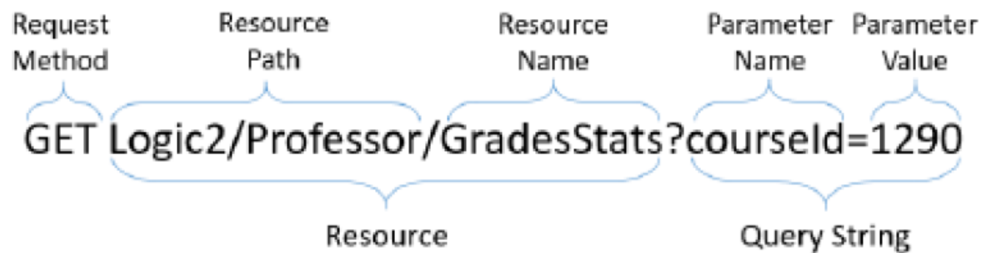


Figure 2.2: A HTTP Request

A user session is a session of activities of a user with a unique IP address on a web site for a speci ed time period, usually 30 minutes. If a user takes a break but comes back within the time limit i.e. 30 minutes, it is still considered one session. User sessions are a good way of tracking how the user interacts with the web app. Web applications store these user sessions in les called *access logs* that can parsed and processed later to analyze user behaviour. Access logs preserve the record of how the web app was used.

A software application should have less than 10 errors per 1000 lines of code to avoid functional failure [1]. Functional failures might reduce customer con dence and in uence them to switch over to competitor's product. This is specially true of web

applications since switching over to a competitors product is just a couple of clicks away. Losing a customer is very expensive for a company. Harvard Business Review reports that acquiring a new customer is anywhere from ve to 25 times more expensive than retaining an existing one [5]. Therefore, it is extremely pertinent for tech companies to keep their customer satis ed and their experience bug free. Moreover, faulty software can expose private data, which would not only decrease customer con dence but also extensively hurt lives too. Security is one of the biggest issues for todays developers and robust testing is required to minimize such risks.

Testing requirements increase exponentially with code size [1] and it might be impossible to exhaustively test any su ciently complex software [14]. Even the simplest web applications might have thousands of lines of code, and since testing requirements increase exponentially, it requires a lot of resources to extensively test such software. Software testing is laborious, tedious and time-consuming work. Moreover, since software are continuously evolving, their test cases also need to evolve. This makes testing labor and time-intensive. Voas and McGraw note that modern software systems are too large to be checked by white-box testing, and white-box testing cannot detect all program faults, such as missing code.[7].

Fisher et al. reports that testers su er from biases while developing test cases due to knowledge of the system internals, mental models, and expectations [2]. This might lead to under exploration of many testing regions. Moreover, test cases generated by testers might not robustly test heavily used sections/features of the web app since they might not focus on xed sections.

White box testing requires the tester to have knowledge of the underlying web application architecture and involves covering the code of the application [8]. For a web application, that means knowing before hand both the back-end and front-end architecture of the application. On the other side of spectrum is black box testing where the architecture of the program is not required to be known beforehand [8]. You

simply test if a input matches the expected output or not. In grey box testing, you need to have limited knowledge of the internal working of an application.

User-session-based testing involves creating test cases from selected parts of user sessions. Creating test cases from user sessions has been proven effective by Sprenkle et al, since user sessions reflect actual usage by the user that the tester might not have conceived during earlier development phase [12]. Moreover, test suites created from user sessions focus on areas that the users have interacted with higher frequency. These areas of the web app are critical and faults that disrupt the features provided by these areas are more costly than faults in areas not frequented by the users. User session based testing is also effective in regression testing.

Regression testing is used to make sure updated or changed software still performs the previous duties without any glitch. Test cases made from user sessions before the software update/patch can test whether the update has broken any previous functionality. New test cases can be made with the user sessions after the software update which can be used for regression testing after the next round of updates.

One of the challenge of testing is how we evaluate the efficacy of a test suite. One of the common methods of evaluating a test suite is to calculate its code coverage. Code coverage simply is a measure of what percentage of the total code did the test suite cover. The idea is that the more code coverage, the higher the probability of finding a bug.

In web applications testing, we can measure a different kind of coverage that is based on the HTTP requests. [11] Request-based coverage is correlated to code coverage since covering a request involves running specific code. For example:

and involve di erent lines of code and covering these two requests will run di erent parts of our code.

Genetic algorithms (GA) are adaptive search techniques used to solve optimization problems. GA mimics biological evolution process closely. In evolution, advantageous genes are given preference and has higher survival percentage. Similarly in GA, prede ned genes and chromosomes that have higher *tness* are allowed to breed to create genes/chromosomes of higher tness. A chromosome is made up of genes, where the genes are the basic unit of the GA. A suite of chromosomes is called a genome. Reproduction can be done in chromosome or genome level. Figure 2.3 shows the basic architecture of GA.

Genetic algorithm has been used in test case generation by [10],[2],[14], [1], [7] and [9].

A selection scheme is applied to determine how individuals are chosen for mating. Individuals chosen for mating are called parent population and the population generated by reproduction is called children population. Selection can be divided into two categories: initial population and reproducing population. Initial population is usually chosen at random but you can lter them based on the initial tness value. A cycle of parent population reproducing a children population is called a generation. Reproduction is done for multiple generations.

Tournament selection is a selection scheme used to choose parent chromosomes for reproduction. A speci ed number of individuals are chosen randomly from the parent population and each individual is tested for their tness. The one with the highest tness is chosen as a parent individual. The process is repeated to get the second parent. The two winners go through crossover to produce two child individuals. The

process is repeated as many times as required to get the child population, producing a new generation of individuals.

Reproduction is the process by which we generate new individuals from selected individuals. The process takes in the parent population and selects individuals for reproduction and then generates child population. The child population generate by reproduction is a new generation of individuals.

Crossover is one the two ways of reproduction. It involves swapping a part of a chromosome with another i.e. swapping the rst half of a chromosome with the bottom half of another. Crossover between two parents usually create two child chromosomes, but it depends on the programmer.

Mutation usually involves random insertion or deletion of genes in a chromosome. Sometime, a gene altered in minor ways. Mutation brings diversity within the population.

In the next section I will discuss the goals of the thesis, discuss research questions and my hypotheses. I will elaborate my approach of testing out the hypotheses.

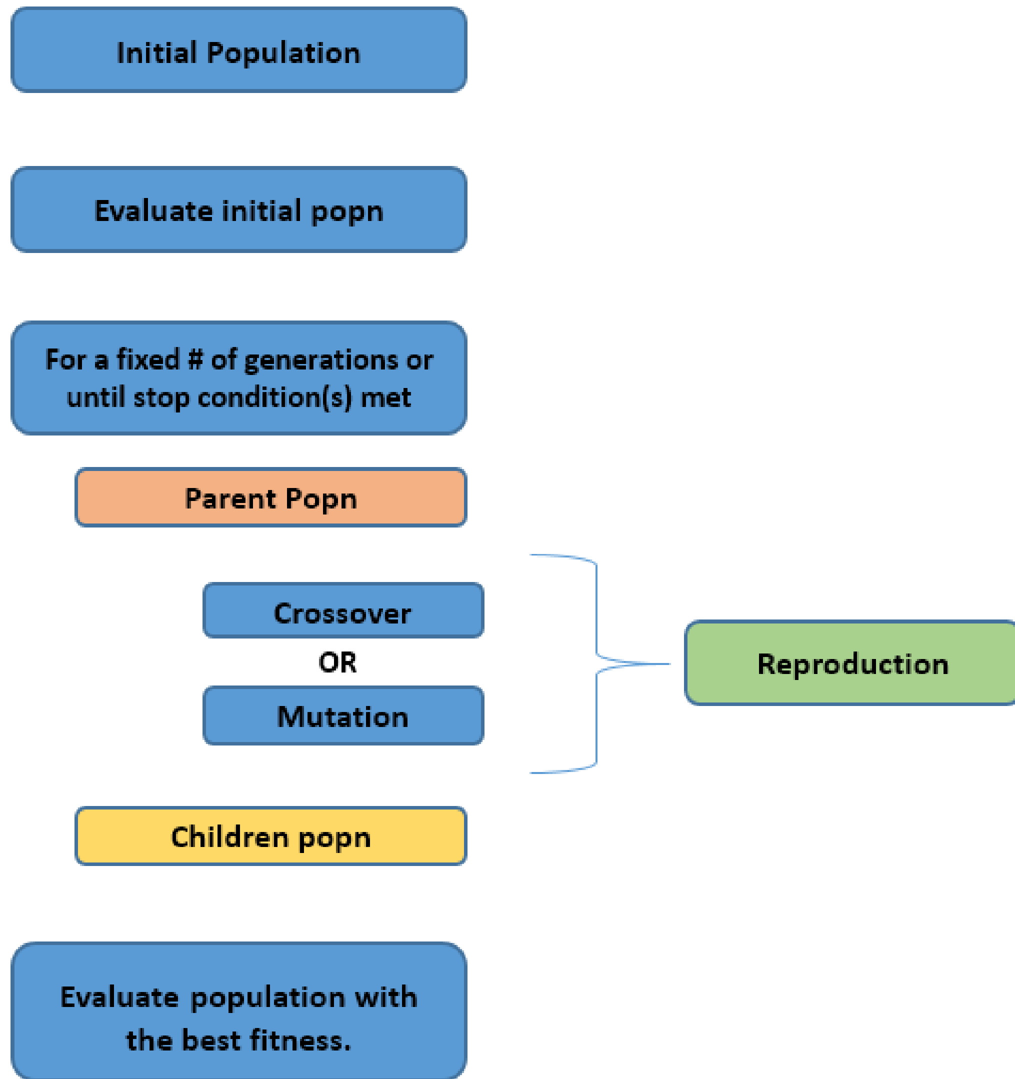


Figure 2.3: Architecture of Genetic Algorithm

This chapter presents the problem we are trying to solve and lays out our goals for the thesis. We describe how we model of web app usage into components of genetic algorithm, our design of applying genetic algorithm to web application testing, and the prototype implementation of the model and the genetic algorithm.

Testing web applications helps developers find and fix faults in the web application and ensures smooth functioning of the web app. Interrupted functioning of a web app will dissuade potential customers. Faults in web applications can cause millions of dollars of loss for businesses (Section 2.2.1), but testing is expensive and time consuming. Genetic algorithm (GA) can automate and expedite the process of test-case generation. GA specializes in optimizing solutions, since it takes a greedy approach when creating test cases: taking the best genome during reproduction. Therefore, given a set of user sessions, we will use GA to find a cost-effective test suite that reveals bugs/faults within a web application during regression testing. As discussed in Section 2.2.1, testers suffer from biases while developing test cases which leads to under exploration of many testing regions. Moreover, they might not robustly test heavily used sections of the web app. To overcome such biases, we automatically generate test cases from logged user sessions. Usage based test case generation reflects actual behaviours of authentic users and therefore focuses on areas that future users will potentially use.

Goals for my thesis are as follows:

1. Generate a test suite that is re ective of but not necessarily identical to actual web application usage
2. Reveal bugs/faults with a feasible number of test cases.
3. Automate the process of test case generation.
4. Make test-case-generation framework that is customizable for testers according to the web application's characteristics and the testers' demands.

My thesis does not claim to achieve the following:

1. Full code coverage of a web application
2. Minimum number of test cases that nd all faults.
3. Minimum amount of time to execute these test cases.

As discussed in Section 2.1.1, we represent web application usage as user sessions, which consist of requests made to the web app by users. To apply Genetic Algorithms to web application testing, we modeled web app usage as genes, chromosomes, and genomes. A gene consists of a request, a chromosome consists of a user session (sequence of requests for a given user), and a genome is a sequence of user sessions. Figure 3.1 illustrates a gene, a chromosome, and a genome.

The output of the GA is a sequence of user sessions that will form our test suite since the sequence is important, given the persistent states of the web application and our test suite will consist of the most cost-e ective sequence of test cases. Moreover, the output should re ect the usage of several user sessions to mimic the real world usage of web apps, given the web app is accessed by multiple users in reality.

Our approach can be broken down into three parts: converting user sessions into inputs for the genetic algorithm, creating an initial population from these inputs,

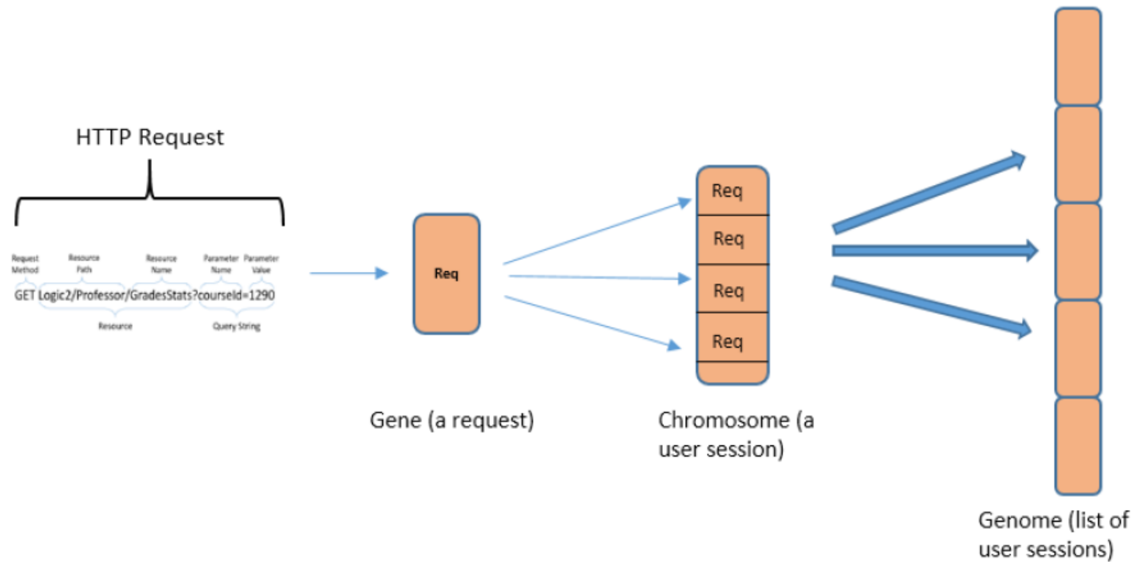


Figure 3.1: Gene, Chromosome, and Genome

and running the genetic algorithm. The input for the whole process are the access logs and the output is the most cost-e ective genome, which will be our test suite.

Figure 3.2 gives an overview of our approach.

Given a set of user sessions, our parser processes them and converts them into genomes, where a genome is a sequence of user sessions. The *length* of the genome is tunable and can be changed according to a testers' requirements. The length can be chosen based on the total number of logged user sessions and how good each user session is in terms of its number of requests. If you choose a longer length, the resulting genome may have increased coverage but will likely contain redundant requests. The initial population becomes the rst parent population for the genetic algorithm.

The initial population of user sessions is randomly selected from the set of user sessions from the previous step. With the generated genome, the user sessions are in their original, collected order. We used a initial tness threshold to lter out the

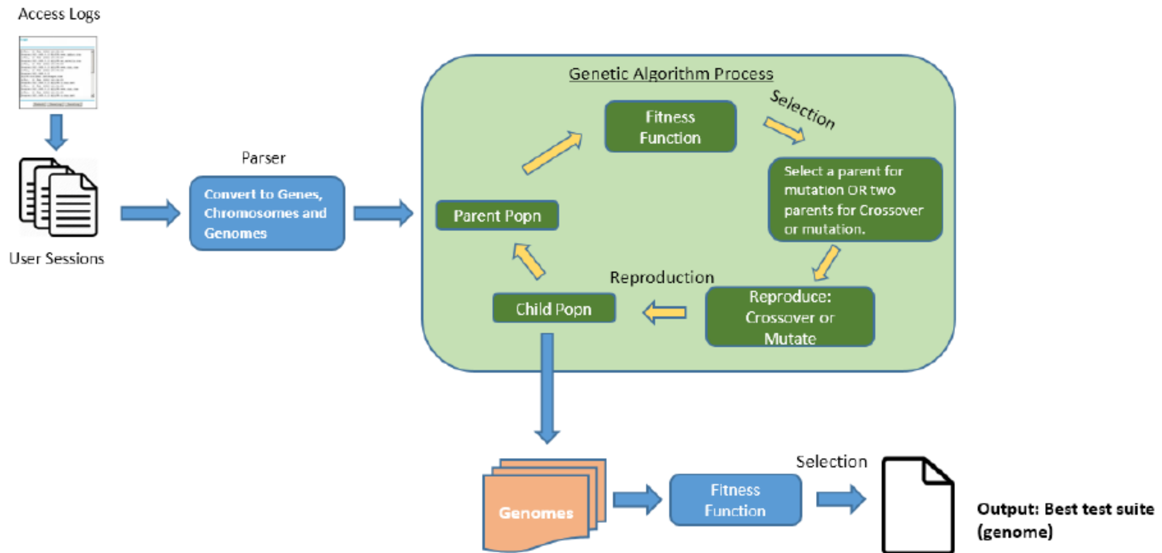


Figure 3.2: Genetic Algorithms for Web Application Testing

weakest user sessions. (We provide more information about how we de ne tness in Section 3.4.3.1.) The threshold should be around the mean tness of the genome pool. If you choose too high a threshold, then you may not be able to ll the initial population quota. If it is too low, then you might miss out on better genomes.

Selection is the process where we select an individual from the genome pool based on their tness. The standard algorithm for selection is *tournament selection* (2.3.1.1), in which we randomly select a xed number of genomes and evaluate their tness to nd the genome with the best tness, which then becomes a parent. For crossover, we repeat the process to get the second parent.

The random selection of a xed number of genomes during tournament selection ensures that we do not pick a select few genomes as the parents every time. The random selection distributes the probability of choosing a genome over the whole genome pool. If we only selected the best overall genomes, we would pick a select few every time, losing out on the unique resources of the other genomes.

We nd the best genome our test suite at the end of the GA process by evaluating the last generation and selecting the genome with the best tness.

A *tness function* evaluates the tness of a genome. Our tness function is comprised of two metrics: resource coverage and length of the genome (i.e., the total number of requests). In general, we want to increase coverage and maintain a feasible number of requests required to achieve the coverage. The tness function should reward coverage and punish length. The tness function can be altered according to the needs of the tester. The overall result of GA depends on the tness function, since the GA favors genomes with higher tness and the way you de ne tness will dictate the output.

Genetic algorithms have two methods of reproduction: crossover and mutation. Genomes go through either crossover or mutation in each cycle. These reproduction algorithms can be designed at the genome- or chromosome-level. For this thesis, I focused on designing crossover and mutation for *genomes* because web applications have persistent state and, therefore, realistic usage and code coverage is highly dependent on the sequence of requests.

The algorithm chooses whether to reproduce using mutation or crossover depending on the *mutation threshold*. We randomly generate a number and mutate the genome if the generated number is higher than the mutation threshold. A mutation threshold of 0.5 means we will mutate 50% of the time.

We stop reproducing when we have generated the desired number of child genomes. The generated child population then becomes the parent population for the next cycle of the GA.

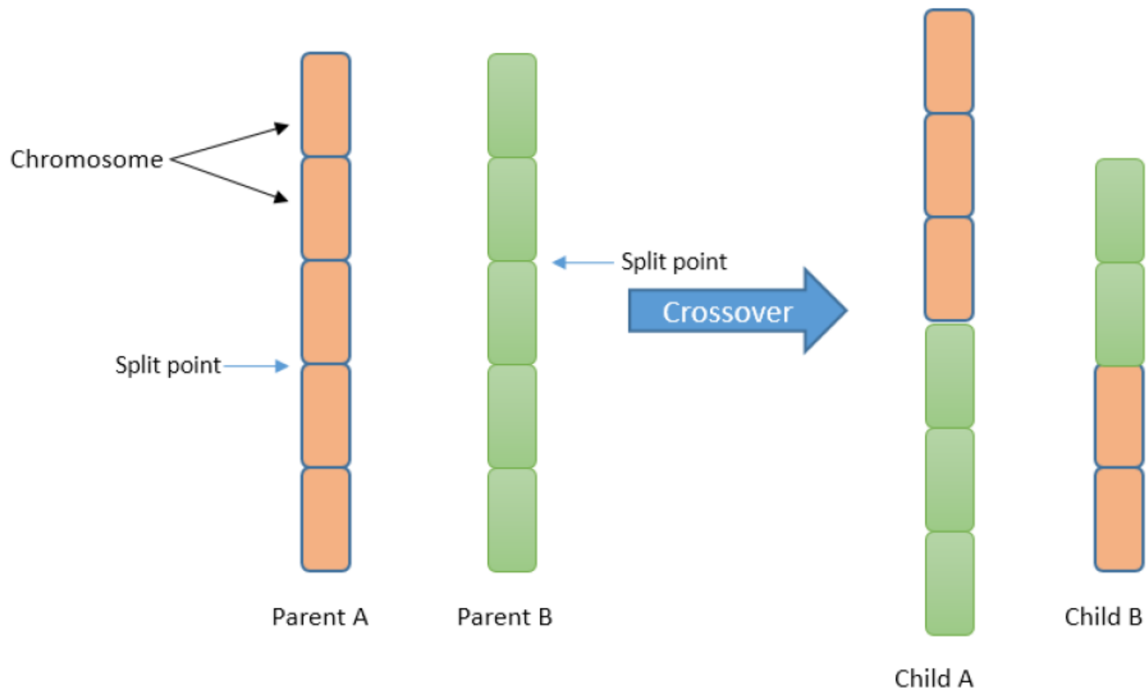


Figure 3.3: Crossover for a genome

The crossover function takes as input two genomes and produces two child genomes, where each child has traits from both parents. The child genomes consist of recombined fragments of their parents' genomes.

Figure 3.3 illustrates crossover for a genome. A split point is randomly chosen in each genome. The part before the split point in genome A is joined with the part after the split point in genome B and vice versa. The result is two child genomes with chromosomes from both the parents. Figure 3.3 illustrates crossover for a genome.

Crossover increases the resource coverage of the test suite, since we cover more unique resource paths, making it more e ective in nding faults within a web app. The parent genomes are selected using tournament selection, which selects the best genome, in terms of tness, from a randomly picked number of genomes. Since, we always pick the genome with the highest tness, we prefer genomes that cover more unique resources. This results in the child genomes having a higher tness.

Mutation operates on one genome where a random chromosome (user sessions) is switched with another chromosome from the chromosome pool (the set of user sessions), also randomly picked. Figure 3.4 illustrates mutation for a genome.

Mutation increases diversity in our population. Since, we replace a chromosome in one of the genome with a chromosome from the chromosome pool, we add newer chromosomes that might not exist in the genome pool, since the genome pool is restricted to the initial 100 genomes with their randomly chosen chromosomes. More diversity means we have a wider variation of genomes, each with a di erent set of chromosomes, within our genome pool. Without mutation, our genome pool would have been restricted to the chromosomes chosen during initial population generation.

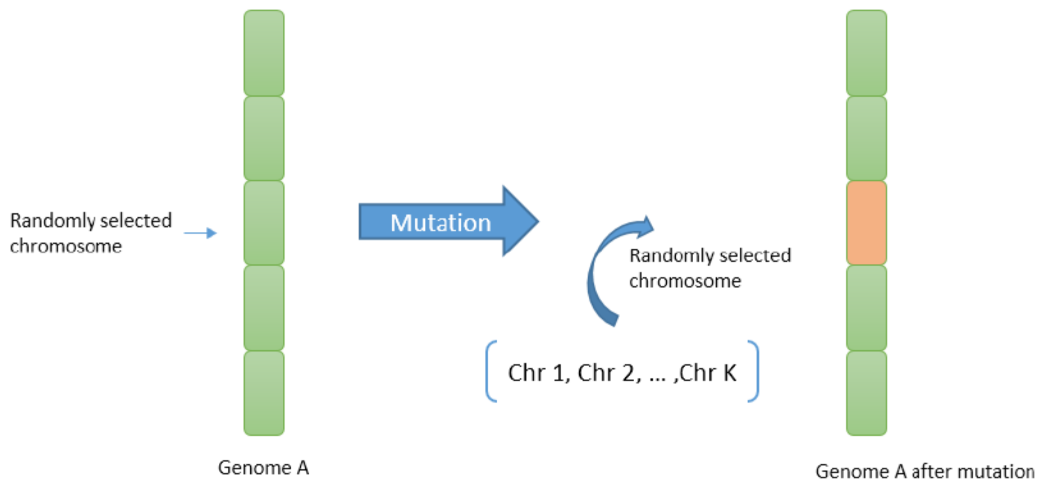


Figure 3.4: Mutation for a genome

Xuan Peng and Lu Lu proposed using genetic algorithm to generate test cases from logged user sessions [10]. They created a relation dependent graph of a web application and used a genetic algorithm to create test cases for the web app. They evaluated the test cases based on how many relational transitions the test cases covered.

There are fundamental ways their approach can be improved. Comparing their model with ours:

- Fitness function: Xuan Peng and Lu Lu’s tness function is evaluating the test suite (chromosome) based on its coverage of data dependence relation and link dependence relation. They are nding the percentage of total data dependence and link dependence relation of the test suite (chromosome) covers. They do not take length of the test suite into account and therefore their genetic algorithm do not try to minimize the length of the test suite. Genetic algorithm optimizes the solution i.e. test suite based on the tness function. Since their tness function focuses solely on increasing coverage, their GA will try to increase coverage by increasing length. Therefore, the generated test suite will be large and not cost e ective.

  Our tness function (3.2) rewards resource coverage and punishes length and therefore tries to make the generated test suite cost e ective.

- Xuan Peng and Lu Lu de ne transitional relation as page request page. This is confusing, since the request sent by the rst page is the same as the URL of the second request. It ultimately becomes: URL (rst page) → Request/URL of second page → URL (second page).

Comparing their experiment evaluation with our evaluation:

- There experiments and results are based on a small toy application. The web app they used is only used for research purposes. It is not a commercially used product. For data collection, they invited students to use the web app and logged their user sessions. Our subject application (4.2) is a fully functional commercial web app and data collected over a semester (3 months) from a number of students over multiple classes. Our data collection was with authentic users of the web app, not invited ones. Therefore, our data better represents the average usage of the web app.

- Their input size is very small with only 87 user sessions and 3219 requests, where as we used 842 user sessions and 32156 requests. Moreover, their subject application is also small with only 9 pages and 15 unique transitions, where as we have 167 unique resources.

We implemented our prototype primarily in Python. First, we create user sessions by parsing access logs. Then we convert the user sessions into the inputs for the genetic algorithm, namely: genes, chromosomes and genomes. We make the genomes

reproduce for a certain number of generations, evaluating their tness in each turn and selecting the ones with highest tness to reproduce (tournament selection). Our output is the genome with the highest tness value after the end of reproduction cycle.

In the remainder of this section, we describe these steps in more detail.

We parsed access logs of the web application to create user sessions based on the user's IP address and the time period of their interactions with the web app. Each user session was saved in a '.tc' (i.e., test case) le. The user sessions were converted into sub-components of genetic algorithm.

We model a chromosome as a user session, and a sequence of chromosomes make up a genome. Our initial genome pool have a length of 100 genomes that are randomly made by randomly choosing a number, that we can tune as required, of chromosomes from our chromosome pool. During this process, we evaluate the tness of each genome and put it in our initial population pool if its tness value is above the initial tness threshold, to lter out genomes with weak tness. Our threshold for this process was 0.1, since that was the median tness of the genome pool. We want to lter out the weakest genome but also needed to make sure we have at least 100 genomes with tness equal or above our threshold. Since our user sessions are from actual usage of the web app, some of them contain only a few requests i.e. just login and logout. Therefore, we lter out the user sessions that have weak tness.

The initial population is our rst parent population. We randomly generate a number between 0 and 1, and if the number exceeds the mutation threshold, the algorithm performs a crossover. Otherwise, the algorithm performs a mutation. Our base mutation threshold was 0.1, meaning we mutate 10% of the time. We prioritize increasing coverage over reducing the number of requests in our test suite, therefore

we want to do crossover more often than mutation since crossover tries to increase coverage.

We keep doing crossover or mutation until we have 100 genomes in our child population. The child population becomes the parent population for the next cycle of the GA.

We repeat the process for a xed number of generations. The number of generation we choose is tunable, and it a ects the generated test suite in terms of coverage and length. The tness stays same or increases with every iteration.

When we reach our generation limit, we evaluate the tness of each genome in our latest child generation. The genome with the highest tness becomes our test suite.

To measure how many requests a genome covers, we nd the number of unique requests in the genome and divide it by the total number of unique requests in the web app. A unique request has a unique request method plus resource signature, that no other request in our unique request pool has. They act as a *Unique Resource Locator (URL)*. We get the total number of unique requests in the web app by parsing all the logged user sessions and making a list of unique requests.

$$= \frac{\qquad}{\qquad} \quad 100 \tag{3.1}$$

Fitness is composed of coverage and the number of requests needed to get the coverage: the higher the coverage, the higher the tness, and the longer the length, the lower the tness. Since, the range of length is in the thousands of requests, we took the log of length to better t the curve. We have two weights: w and w . These weights determine our prioritization of coverage and length. If we want to prioritize coverage more, then w will be bigger than w and vice versa. Since the primary goal of the thesis is to increase coverage, our w is higher than w . But if w is too low then the output genome will be too long and won't be cost-e ective.

$$= \qquad\qquad + \frac{1}{(\qquad)} \tag{3.2}$$

We have designed our approach to automatically convert web app usage into parts of genetic algorithm, so that we can run the genetic algorithm to generate a cost-e ective test suite. The framework is customizable, so that we can tune parameters and analyze their e ect on the output. We evaluate the e ect of the parameters in the next chapter.

This chapter will address the research questions and our hypotheses, how we are testing out our hypotheses, and evaluate and analyze the experimental results. The chapter will also give a set of recommendations for future testers.

The research questions this thesis is aiming to address are:

1. Is genetic-algorithm-based test case generation cost-e ective?
2. What are the characteristics of the genetic-algorithm-derived test cases under di erent parameters?

Our hypotheses are as follows:

1. Genetic-algorithm-based test-case generation is cost e ective.
2. Increasing the number of generations will yield more cost-e ective test suites.
3. Increasing the mutation rate will decrease the resulting test suite's resource coverage.
4. Increasing the initial number of chromosome per genome will increase the resulting test suite's resource coverage.

Since our user-session-based testing techniques are language-independent, requiring user sessions but not source code for testing, our techniques can be easily extended to other web technologies.

Table 4.1: Subject Application Characteristics

Table 4.2: Characteristics of User Session Sets

For this thesis, we evaluate our approach on an online symbolic logic tutorial, which we call Logic. Logic is written in Java using servlets and JSPs. The application consists of a back-end data store, a Web server, and a client browser. Table 4.1 summarizes the applications' code characteristics. The application has two primary users: students and professors. Students have access to quizzes which they can take and submit. They also have access to practice quizzes. Professors can access submitted quizzes and grade them. They can also make new quizzes and make them accessible to students.

Logic's user sessions were previously collected during two academic semesters. We converted the user accesses into user sessions using Sprenkle et al.'s framework [13]. Table 4.2 shows the characteristics of the collected user sessions, in terms of the number of user sessions and the number of user requests.

We run our prototype implementation (3.6), choosing and tuning the parameters as required for the evaluation of our hypotheses. For all the runs of the test-case-generation algorithm, the initial tness threshold during initial population generation is xed. We nd the average tness of all of our user sessions and set our threshold around that, so that we lter out the weakest genomes but at the same time have at least 100 genomes for our initial population. Moreover, we gave more priority to increasing coverage than reducing length by making w in our tness function (3.2) twice as big as w (w =2, w = 1).

We measure resource+type coverage and the number of requests (length) in each genome for every generation as well as the time taken for each run of our genetic algorithm. These metrics are also the components of fitness: the higher the coverage, the higher the fitness, while the higher the length, the lower the fitness.

We evaluate the effect of tuning the following parameters on our resulting test suite:

1. Initial number of chromosomes per genome. We performed three experiments with the following values of initial number of chromosomes per genome: 25, 50, 100. The mutation threshold was fixed at 0.1, and the number of generations was fixed at 30.
2. Mutation threshold. We performed three experiments with the following values of mutation threshold: 0.1, 0.25 , 0.5. In these experiments, the initial number of chromosomes per genome was fixed at 100, and the number of generations was fixed at 30.
3. Number of generations. We performed one experiment with 100 generations to see how the number of generations affect fitness of the genomes. During this experiment, the initial number of chromosomes per genome was fixed at 100 and mutation threshold was fixed at 0.25.

Each experiment was repeated 30 times, taking the mean value of resource coverage and length for each generation. We also calculated the standard deviation for resource coverage and genome length for each generation.

4.4 Results

This section presents the results of the experiments described in the previous section.

4.4.1 Comparing the size of the initial population

Figure 4.1 provides an overview of the effect of increasing the initial number of chromosomes per genome. The left y-axis represents resource coverage of the best genome, our final test suite, from the last generation. The right y-axis represents the length, in terms of number of requests, of our test suite from the last generation.

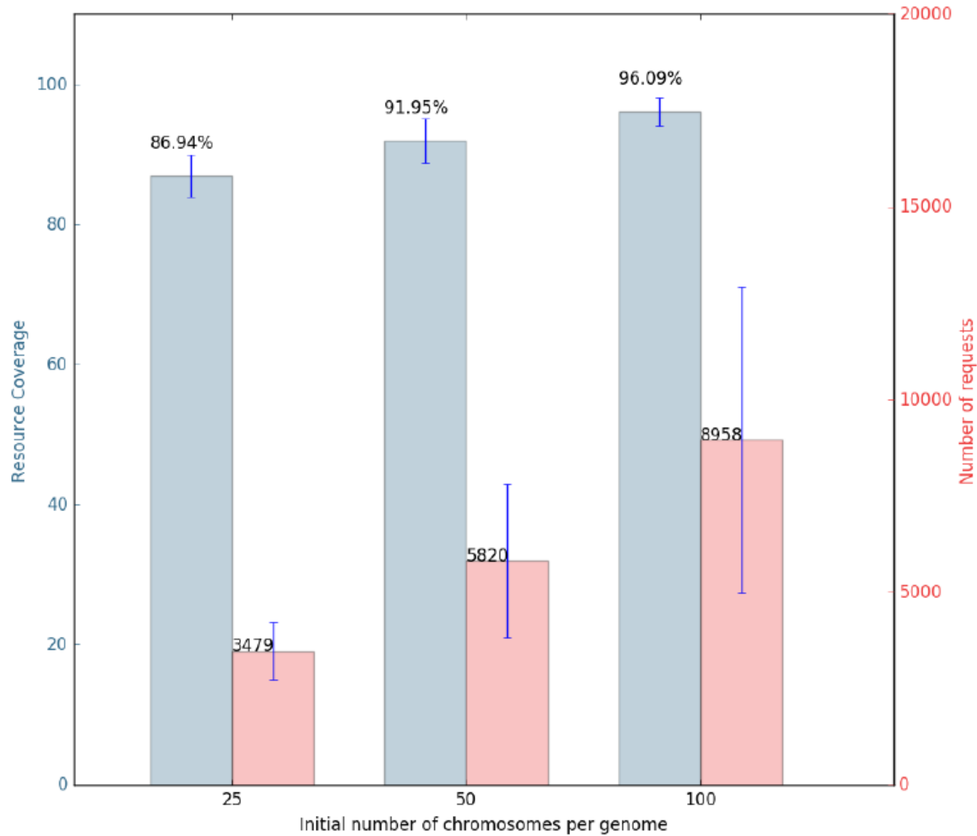


Figure 4.1: E ect of changing initial number of chromosomes per genome on resource coverage and length of the output test suite

The x-axis represents the initial number of chromosomes per genome. Since tness is composed of resource coverage and length, representing them both on two di erent y-axes helps us to visualize the e ect on each one better. The vertical lines on the bars represent the standard deviation respectively.

As the initial number of chromosomes per genomes increases, the resource coverage and length of the test suite also increase. The rate of increase of resource coverage slows down as we increase the initial number of chromosome per genome it is bounded by 100% coverage. Length, on the other hand, keeps increasing steadily.

Figure 4.2 gives detailed results for the experiments. The left y-axis represents resource coverage of the best genome in each generation, the right y-axis represents

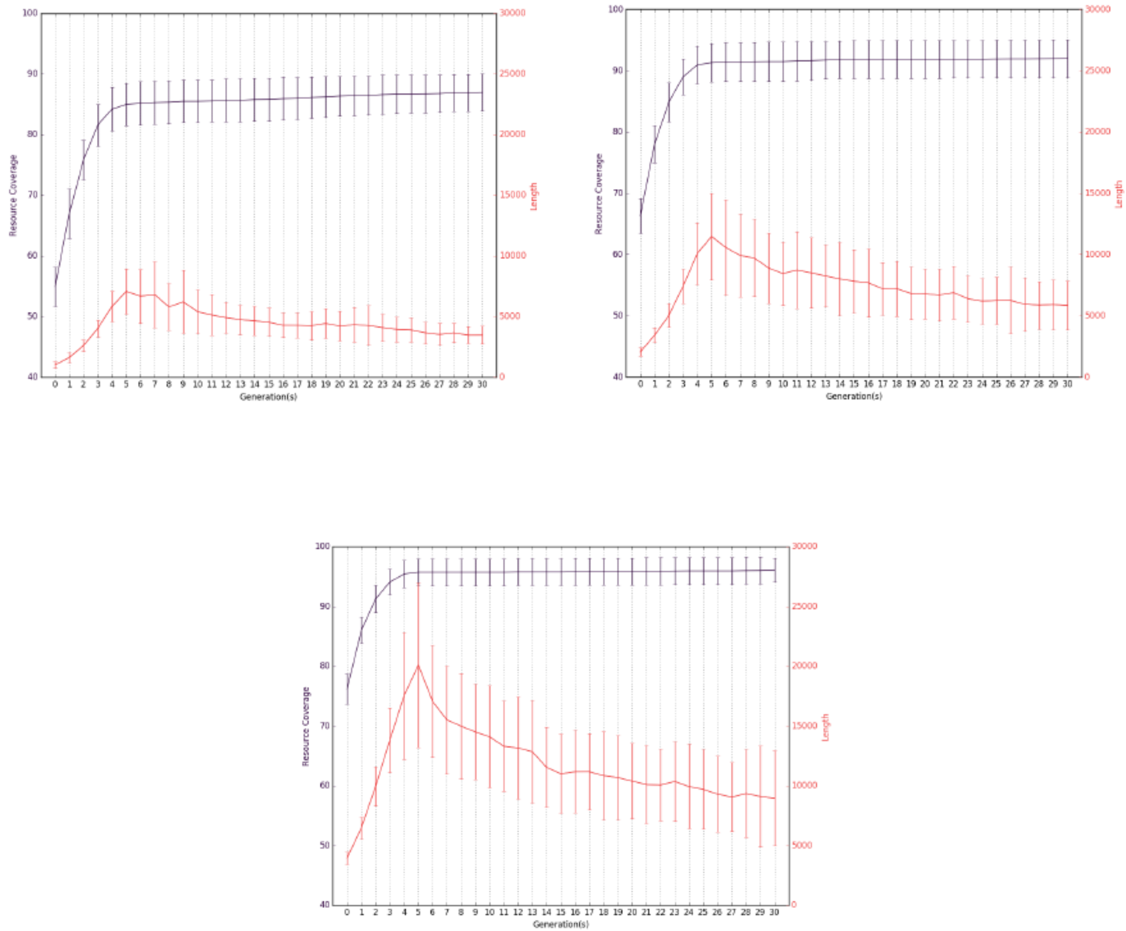


Figure 4.2: E ect of changing initial number of chromosomes per genome on resource coverage and length of the output test suite

the length, in terms of number of requests, of the best genome in each generation, and the x-axis represents the generation number. The 0 generation represents the initial population of the genetic algorithm. The vertical lines on the curves represents the standard deviation respectively.

In Figure 4.2a, the rate of increase of resource coverage is high up to the 5 generation. The rate decreases and there is a slight and steady increase in the resource coverage for the rest of the time. The length of the genome increases initially, and starts to decrease after reaching a peak at 5 generation. The peak coincides with the start of the plateau for the resource coverage curve. The length curve steadily decreases, with a few bumps in between.

Figure 4.2b shows similar results. The curve for resource coverage plateaus around the 4 generation, a bit earlier than in Figure 4.2a. The start of the plateau coincides with the peak of the length curve. Resource coverage of the initial population is higher than the resource coverage of the initial population of the previous experiment.

Figure 4.2c shows similar results as Figure 4.2a and Figure 4.2b. The overall resource coverage and length is higher compared to the previous two experiments. The curve for resource coverage levels out around 4 generation, but the curve for length peaks around 5 generation. The rate of decrease of length, after its peak, is higher than the rates in the previous two experiments. Resource coverage for the initial population is the highest of the three experiments. Standard deviation of length for this experiment is also higher than the standard deviation of the previous two experiments.

Figure 4.3 gives the overview of the e ect of increasing initial number of chromosomes per genome on the average time taken to run the framework for 30 generations. The x-axis represents the initial number of chromosomes per genome, the y-axis represents the average time taken and the vertical lines are the standard deviation of the average time taken. As we increase the initial number of chromosomes per genome, the average time taken to run the framework and the standard deviation of the average time increases.

These experiments support our hypothesis that increasing the initial number of

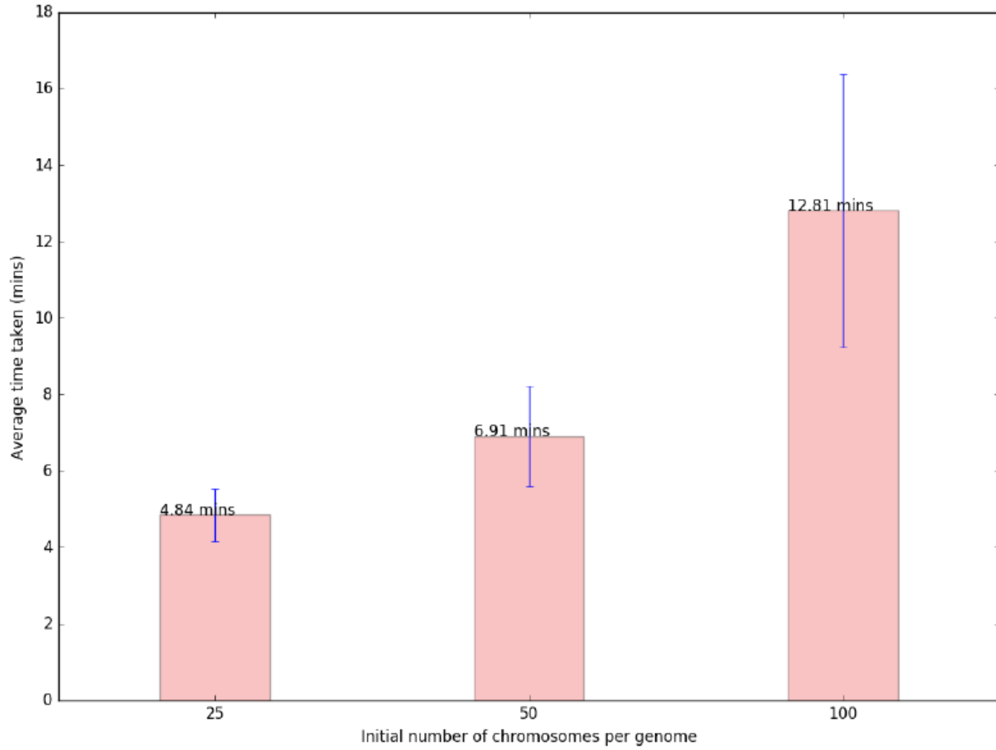


Figure 4.3: E ect of changing initial number of chromosomes per genome on the average time taken to run GA for 30 generations.

chromosome per genome will increase resource coverage.

Figure 4.4 exhibits the e ect of changing mutation threshold on the overall result i.e. our overall test suite. The left y-axis represents resource coverage of the best genome, our nal test suite, from the last generation, the right y-axis represents the length, in terms of number of requests, of our test suite from the last generation and the x-axis represents the mutation threshold. The vertical lines on the bars represent the standard deviation respectively.

We are measuring the tness of genomes. Since, tness is composed of resource coverage and length, representing them both on two di erent y-axes helps us to visualize the e ect on each one better. Both the resource coverage and length of our test suite

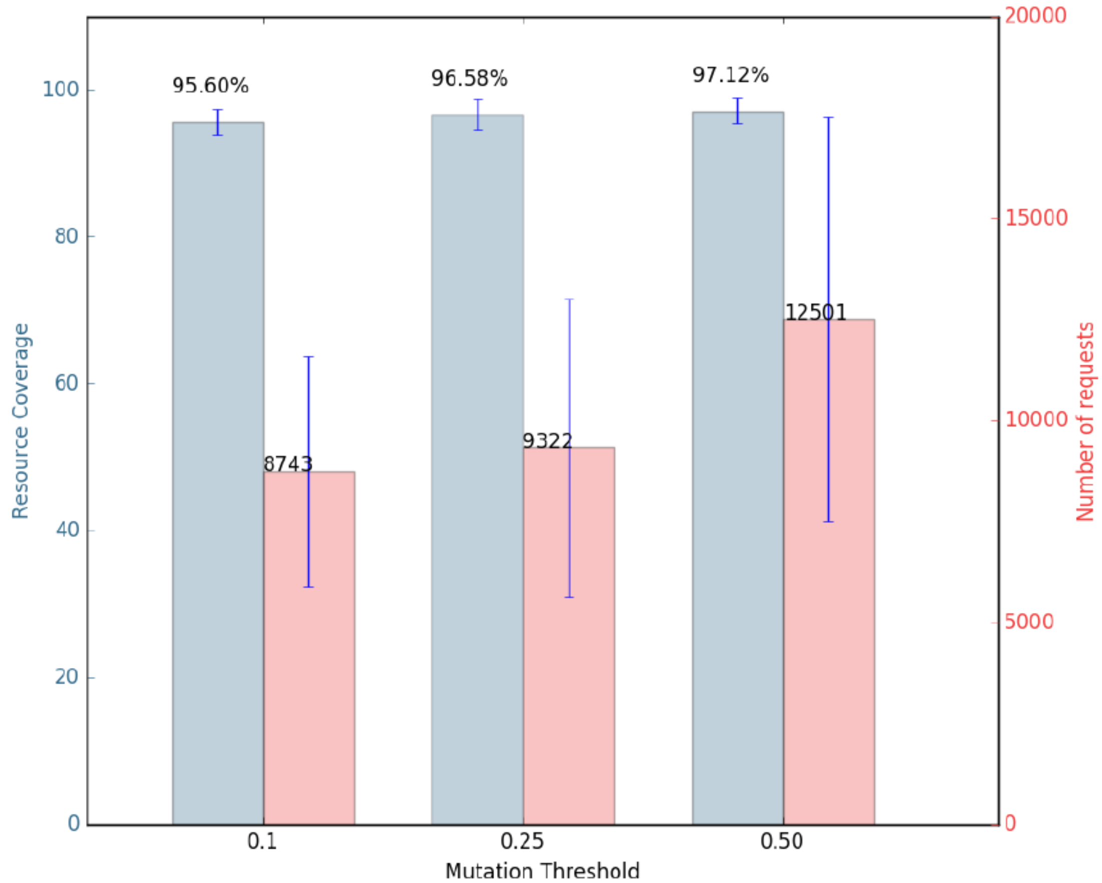


Figure 4.4: E ect of changing mutation threshold on resource coverage and length of the output test suite.

increases as we increase the mutation threshold. Increase in resource coverage is small, where as increase in length is signi cant.

Figure 4.5 gives a detailed overview of the results for the three experiments. The left y-axis represents resource coverage of the best genome, our nal test suite, from the last generation, the right y-axis represents the length, in terms of number of requests, of our test suite from the last generation and the x-axis represents the number of generation. The vertical lines on the curves represent the standard deviation respectively.

In all of the graphs, the resource coverage of the initial population is close to each other, around 75%. The curve for resource coverage, for all the graphs, increases

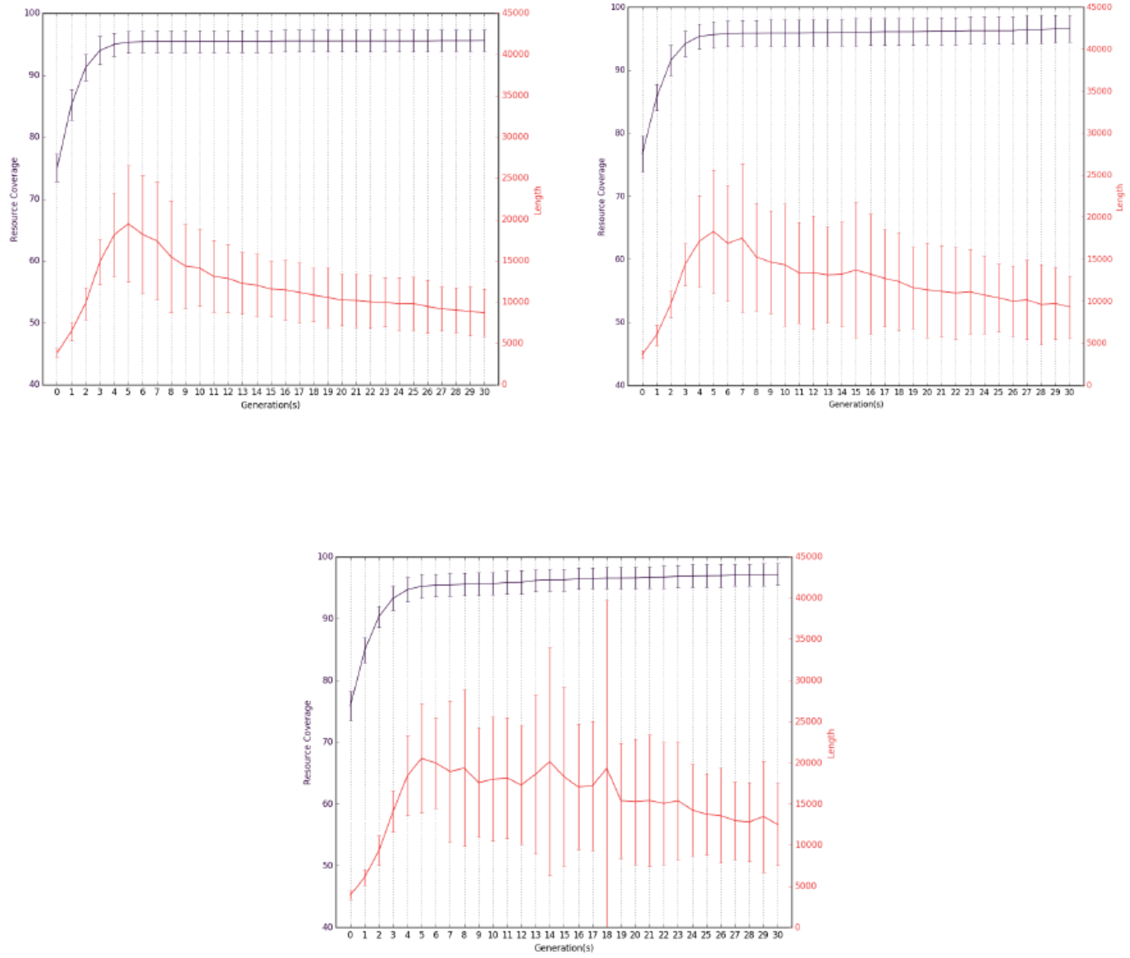


Figure 4.5: E ect of changing mutation threshold on resource coverage and length of output test suite

rapidly and starts to plateau around 4 and 5 generation.In Figure 4.5c, resource coverage increases slightly after the start of the plateau. In all the three graphs, the curve for length increases rapidly and peaks around 5 generation. Then it starts decreasing. Interestingly, the rate of reduction after the peak decreases, as we increase the mutation threshold, as evidenced by Figure 4.5a and Figure 4.5b. Figure 4.5b also has more bumps after the peak than the other two graphs. Standard deviation of length increases as we increase the mutation threshold. Standard deviation of length is highest for Figure 4.5c.

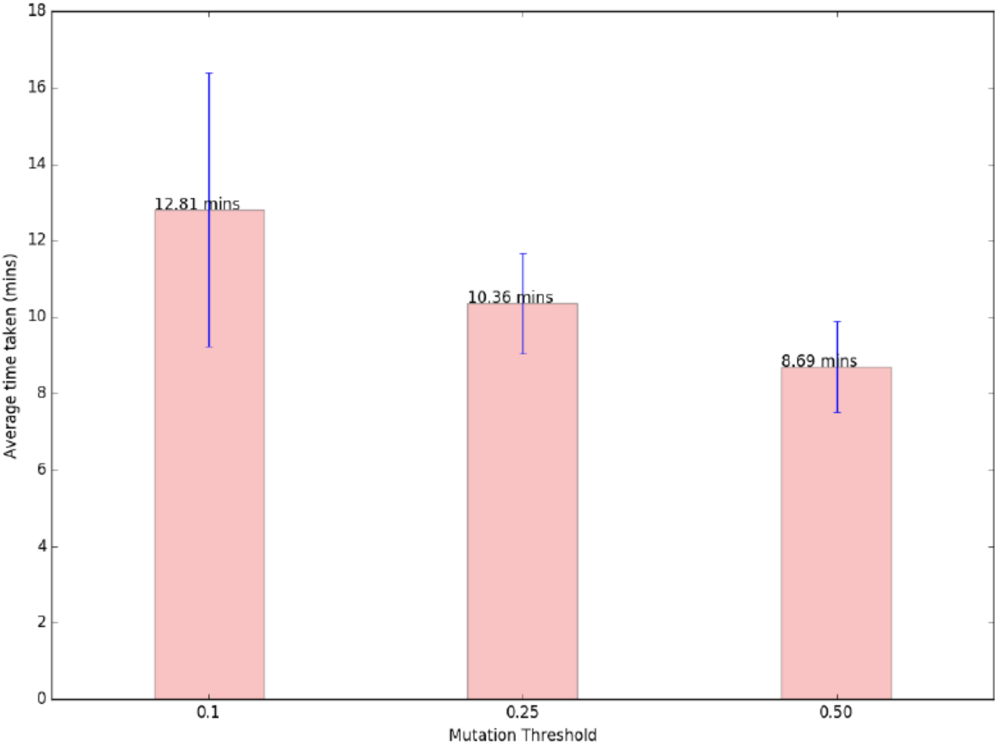


Figure 4.6: E ect of changing mutation threshold on the average time taken to run GA for 30 generations.

Figure 4.6 gives the overview of the e ect of changing mutation threshold on the average time taken to run the framework for 30 generations. The x-axis represents the mutation threshold, the y-axis represents the average time taken and the vertical lines

are the standard deviation. As we increase the mutation threshold, the average time taken to run the framework and the standard deviation of the average time decreases.

The results from this set of experiments do not support our hypothesis that increasing the mutation threshold decreases coverage, since we see a slight increase in coverage.

We ran the experiment for 100 generations to see the e ect of number of generations on the result test suite. Figure 4.7 demonstrates the results of the experiment. The left y-axis represents resource coverage of the best genome in each generation, the right y-axis represents the length, in terms of number of requests, of the best genome in each generation and the x-axis represents the number of generation.

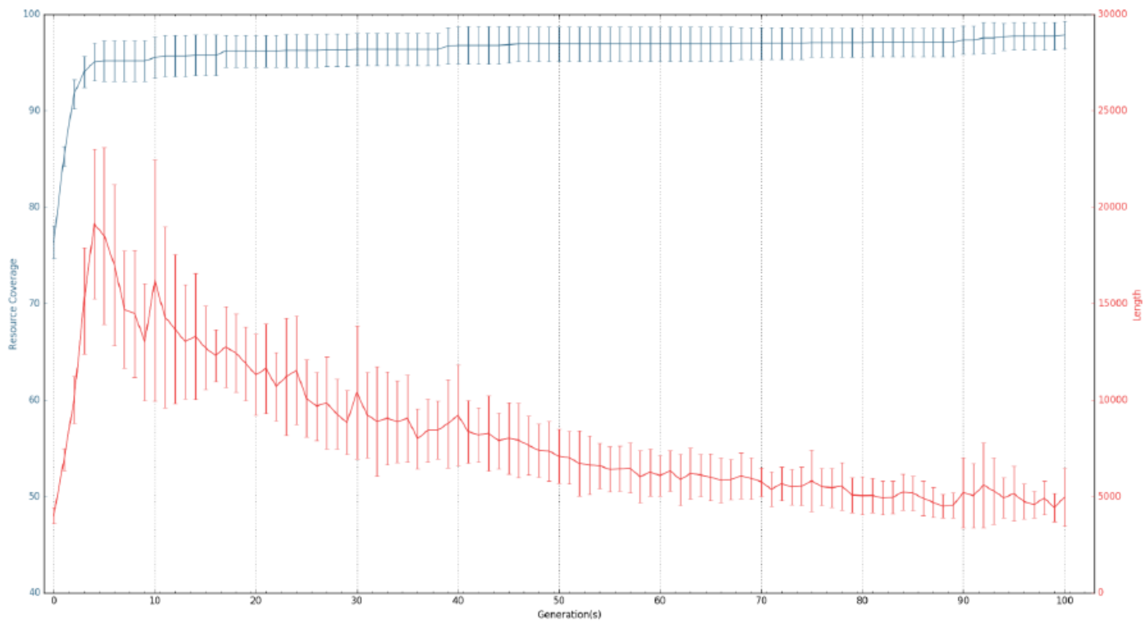


Figure 4.7: E ect of number of generations on resource coverage and length of the output test suite

The curve for resource coverage increases rapidly initially and begins to plateau around 5 generation. Resource coverage increases slowly, in bumps, till the 100 generation. There are slight upward bumps at around 9 , 16 , 38 and 90 generation.

The length curve increases rapidly and peaks at around 5 generation. It decreases rapidly till 100 generation, with few spikes in between.

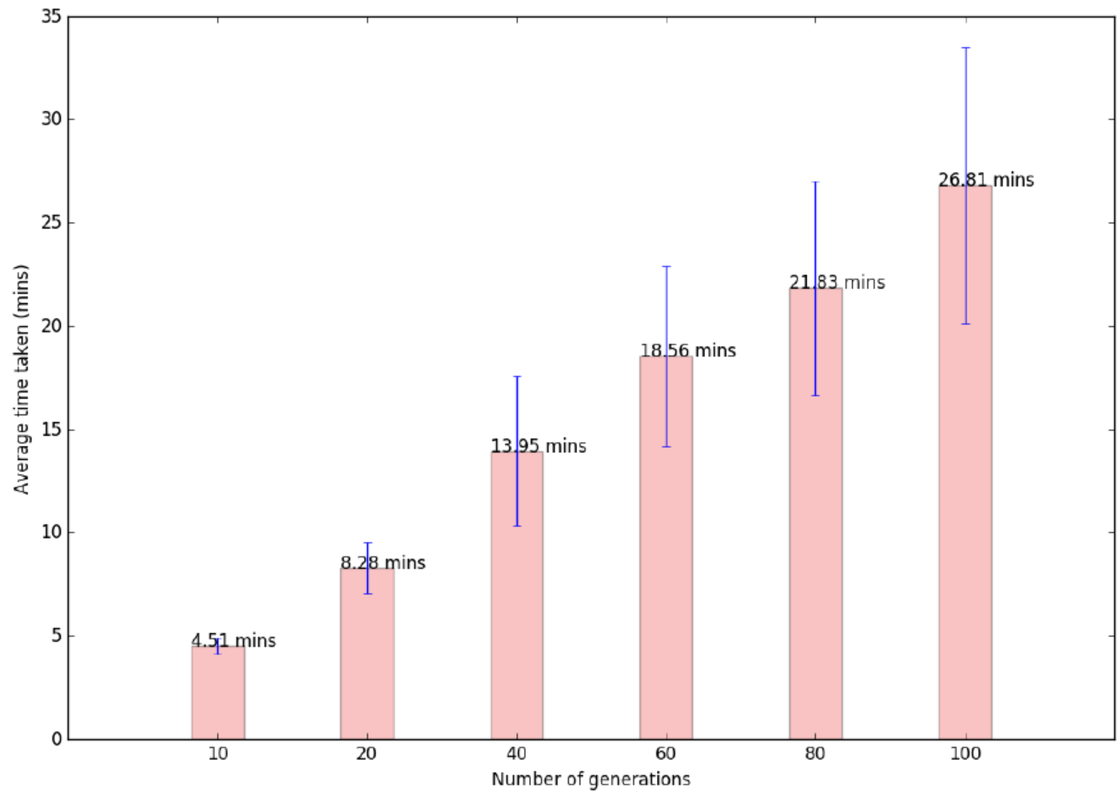


Figure 4.8: E ect of increasing number of generations on the average time taken for the framework

Figure 4.8 shows the e ect of increasing the number of generations the framework is run for on the average time taken for the framework to run. The x-axis represents number of generations, the y-axis represents the average time taken in minutes and the vertical lines represent standard deviation of average time. As we increase the number of generations, average time taken and standard deviation of average time increases.

The results from this experiments support our hypothesis that increasing the number of generations will yield better results.

We will analyse the results from the previous section here.

From Figure 4.1, we see that increasing the initial number of chromosome per genome increases the resource coverage and length and from Figure 4.2, we see that increasing the initial number of chromosome per genome increases the resource coverage of the initial population. Increasing the initial number of chromosome per genome increases the total number of chromosomes in the genome pool. Since there are more number of unique resources from the beginning, the resource coverage of the initial population is higher. The genetic algorithm also has more choice of chromosomes and therefore can lter out the weak ones and replace them with better ones.

We also see from Figure 4.2 that the peak of length conincides with the start of the plateau of the resource coverage curve. Since we gave more priority to increasing coverage than reducing length by making w twice as big as w , the GA focuses on increasing resource coverage during crossover at the cost of increasing length. Around 5 generation, increase in resource coverage is minimal, therefore the GA starts to increase tness by reducing the length. The rate of decrease of length after the peak is highest in Figure 4.2c since the GA has a bigger genome pool, in terms of number of chromosomes, and has more choices of chromosomes and therefore can lter out the weak ones and replace them with better ones.

The standard deviation is lower overall in Figure 4.2a compared to Figure 4.2b and Figure 4.2c, since the GA with lower initial number of chromosome has a smaller total number of chromosomes in the genome pool.

As we increase the initial number of chromosomes per genome, the average time taken to run the framework for 30 generations increases (4.3) since we have a higher total number of chromosomes in the genome pool and our input size is bigger. Computationally, the system has more processing to do, specially during evaluating tness and rejoining genomes during crossover, as they have O(n) run time.

The slight increase in resource coverage we see in Figure 4.4 can be explained by the high initial number of chromosomes per genome. Increasing the mutation threshold reduces the tness by increasing the length considerably, as seen in Figure 4.5. Mutation is done randomly and it does not lter out chromosomes using the tness function. As we increase the mutation threshold, we do fewer crossovers, meaning fewer number of ltering out the weaker genomes. Therefore, the reduction of length is lower. This also explains the lower rate of decrease of length in Figure 4.5c compared to Figure 4.5a and Figure 4.5b. Moreover, since mutation is a random swap of chromosome for chromosome, the algorithm might be swapping chromosomes of di erent length. Therefore we get the high standard deviation in Figure 4.5c.

As we increase the mutation threshold, the average time take to run the framework for 30 generations decreases (4.6). With a higher mutation threshold, we are doing fewer crossovers. Mutations are faster than crossover, since mutation is just a straight swap, while crossover requires rejoining genomes, O(1) vs O(n). Moreover, fewer crossovers mean that we have fewer possibilities of having genomes with a large length.

From Figure 4.7, we see that increasing the number of generation increases the tness of the output test suite, both in terms of increasing resource coverage and reducing length. Since we run the experiment for more generations, we do more number of crossovers, which always try to maximize the tness through tournament selection (2.3.1.1). The parents chosen by tournament selection for crossover, have the highest tness among the randomly chosen genomes. Therefore, during a cycle of the genetic algorithm, the child generation has a higher mean tness than the parent generation. As we run the GA for more generations, the mean tness of the child generation, which becomes parent generation for the next cycle (3.2), increases.

Once we reach a high resource coverage, around 98%, crossover increases tness by reducing length. The decrease in length continues until the 88 generation, where it starts to level o . The GA has maximized the reduction and any further reduction might lead to reduction in resource coverage. We also see slight bumps in the resource coverage curve. As we do more crossovers and mutations,

From Figure 4.8, we see that increasing the number of generations increases the average time taken for the framework to run. As we run the framework for more number of generations, we have genomes of longer length and we also do more computation. Therefore, the average time taken increases.

From the above experiments, we can gure out the values of the parameters that will give us the best set of results. Here are the recommendations for future testers using this framework:

- If you are short in time, you can run the framework for lower number of generations. Running the framework for at least 10 generations gives good resource coverage (4.7), though the length of the test suite is not the most cost e ective.
- If you want to prioritize increasing resource coverage over reducing length, choose a higher value for w than w (3.2).
- Choose a reasonable initial number of chromosomes per genome. You can experiment with a few high values and see which one works better, since higher initial number of chromosomes per genome yields better results (4.1).
- Choose a mutation threshold around 0.25 since that seems to work best (4.4). Too low of a threshold will reduce the diversity of the population and the overall tness might not be satisfactory. Too high of a threshold will increase the length unnecessarily.
- Run the experiment for at least 100 generations since increasing the number of generations yields better results (4.7).

Increasing the initial number of chromosomes per genome increases the tness of the output test suite (4.1), since there is a higher total number of chromosomes in

the genome pool. Therefore, we have a high fitness of the initial population and the GA has more options to choose from and is more effective at replacing weak genomes with stronger ones. Increasing the initial number of chromosomes per genome increases the average time taken for the framework to run.

Increasing the mutation threshold reduces the fitness by increasing the length considerably. As we increase the mutation threshold, we do fewer crossovers, meaning fewer number of filtering out the weaker genomes.Increasing the mutation threshold decreases the average time taken for the framework to run.

Increasing the number of generation increases the fitness of the output test suite, both in terms of increasing resource coverage and reducing length. Since we run the experiment for more generations, we do more number of crossovers, which always try to maximize the fitness. Increasing the number of generations the framework is run for increases the average time taken for the framework to run.

The next chapter will discuss the contributions of this thesis and the future work that needs to be done to further improve the method in detecting faults in web applications.

Chapter 5

CONTRIBUTIONS AND FUTURE WORK

This chapter will serve to summarize my thesis' contributions as well as to outline future work that needs to be done to improve the method in detecting faults in web applications.

5.1 Contributions

The primary goal of the thesis was to create a customizable automated genetic algorithm framework that generated cost-effective test cases for web applications, with the aim of finding faults within the web application. The contributions of the thesis are as follows:

1. explored the use of genetic algorithms in test-case generation for web applications: Chapter 3 gives a detailed overview of the use of genetic algorithm in test case generation and details how to implement genetic algorithm to generate test cases. Chapter 4 gives the results of the experiments carried out to explore the effectiveness of genetic-algorithm-based test case generation.

2. designed and implemented a model for representing web application test cases as components of GA, namely, genes, chromosomes, and genomes: Section 3.6.1 describes how to convert to web app usage into components of GA. This allows testers to run the GA and generate test cases.

3. defined crossover and mutation functions in terms of user sessions relevant to web application testing: Section 3.4.4 describes how we defined crossover and mutation in the genome level. The crossover and mutation functions composes reproduction for the GA. Reproduction allows GA to optimize the solution of a problem. The problem in our case is to generate cost effective test cases given a set of user sessions.

4. created a framework for automatically generating test cases (evolved user sessions) using tournament selection: Section 3.4 gives an overview of our framework that generates a cost effective test suite from a given set of user sessions. The framework is automated and customizable to cater the different needs of testers.

5. analyzed the results and explored which parameters have the most impact on the results: Chapter 4 details the experiments ran to test the efficacy of genetic algorithm based test generation. It also presents the findings of the experiments and provides detailed analysis of the results.

6. provided recommendations to web application testers for how to use the framework for their process: Section 4.6 gives a set of recommendations that will enhance test case generation for future testers.

5.2 Future Work

Future work in improving the model and evaluating the algorithm include:

1. Different coverage metrics: In this thesis, we are calculating coverage of unique resources (request type + resource). To find faults, it is not enough to calculate resource coverage. Future work should be done with different coverage metrics, e.g., coverage of request type + resource + parameter name-value pairs [11].

2. Plug-able fitness functions: Coverage of critical and frequently used section of the web app should also be taken into account. We can create a new fitness function, which calculates the fitness based on how many frequently used resources the test suite covers. This fitness function will increase the coverage of such resources and can be used as a high priority test suite which quickly tests the most commonly used features of the web app.

3. Replacing tournament selection with NSGA-II algorithm: Multi-objective evolutionary algorithms (MOEA) solves problems that consists of multiple objectives [4]. NSGA-II algorithm is an example of a MOEA. Our GA tries to maximize the fitness, composed of resource coverage and length, of the output test suit. The problem has two objectives: increase resource coverage and reduce length. Therefore, with NSGA-II algorithm, we will be better able to prefer one over the other as required.

4. Take persistent state into account during crossover and mutation: If we take persistent state into consideration, then we may have requests that are not compatible with other requests, i.e., a request trying to access data before it was saved by another request; the request that saves data has to be played before the request that accesses it. Many faults occur due to corrupted data or trying to access data that does not exist. Future work should be done by taking the persistent states of the web app into consideration while making test cases.

5. Implement crossover and mutation for chromosome level: Our framework defines crossover and mutation for genome level–we mutate a genome or do crossover with two genomes. Implementing crossover and mutation for chromosomes may fine tune the algorithm further and may allow the GA to optimize the solution

more by merging two compatible user sessions together. This will allow us to create new user sessions from existing ones and may increase the effectiveness of test cases in finding faults, since the newly created user session might cover a different part of the web app that the original user sessions does not.

6. Evaluation: Do experiments on different applications with more and different user sessions. Evaluate test suites based on their ability to reveal faults.

BIBLIOGRAPHY

[1] Jarmo T. Alander, Timo Mantere, and Pekka Turunen. Genetic algorithm based software testing.

[2] Donald J. Berndt, John W. Fisher, L. Johnson, J. Pinglikar, and Alison Watkins. Breeding software test cases with genetic algorithms. In *36th Hawaii International Conference on System Sciences (HICSS-36 2003), CD-ROM / Abstracts Proceedings, January 6-9, 2003, Big Island, HI, USA*, page 338, 2003.

[3] Candice Choi. Starbucks cash registers reveal a scary truth about modern business, Apr 2015.

[4] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[5] Amy Gallo. The value of keeping the right customers, Nov 2014.

[6] Security Innovation. 2012 application security gap study:a survey of it security & developers. *Independently Conducted by Ponemon Institute LLC*, 2012.

[7] Shanshan Lv and Ruilian Zhao. Neural-network based test cases generation using genetic algorithm. *Pacific Rim International Symposium on Dependable Computing, IEEE*, 00:97–100, 2007.

[8] Farmeena Khan Mohd Ehmer Khan. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *International Journal of Advanced Computer Science and Applications(IJACSA)*, 3(6), 2012.

[9] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification And Reliability*, 9:263–282, 1999.

[10] Xuan Peng and Lu Lu. User-session-based automatic test case generation using ga. *International Journal of Physical Sciences*, 6(13):3232–3245, 2011.

[11] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, and Lori Pollock. Web application testing with customized test requirements–an experimental comparison study. In *International Symposium on Software Reliability Engineering*. IEEE Computer Society, Nov 2006.

[12] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, Lori Pollock, and Amie Souter. Analyzing clusters of web application user sessions. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005.

[13] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated replay and fault detection for web applications. In *International Conference on Automated Software Engineering (ASE)*, November 2005.

[14] Praveen Ranjan Srivastava and Tai hoon Kim. Application of genetic algorithm in software testing, 2009.