

**AUTOMATIC TEST DATA GENERATION
FOR REGULAR EXPRESSION PREDICATES**

**By
Rana Ali Badawi Samhan**

**Supervisor
Dr. Mohammad Al-Shraideh**

**Co-Supervisor
Dr. Abedellatef Abu Dalhoom**

**This Thesis was submitted in Partial Fulfillment of the
Requirements
For the Master's Degree of Computer Science**

**Faculty of Graduate Studies
The University of Jordan**

August, 2009

COMMITT DECISION


COMMITTEE DECISION

This Thesis (Automatic test data generation for regular expression predicates) was successfully Defended and Approved on 4 / 8 /2009

Examination Committee

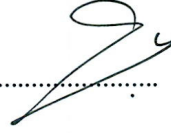
Dr. Mohammad Al-Shraideh (Supervisor)

Assist. Prof. of Software Testing using AI,
Image Recognition

Signature


Dr. Abdellatif AbuDalhoum (Co-Supervisor)

Assoc. Prof. of Genetic Algorithms and
Complex Systems



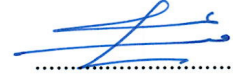
Dr. Saleh Al-Sharaeh (Member)

Assoc. Prof. of Parallel Processing and
Wireless Networks



Dr. Wesam A. AlMobaideen (Member)

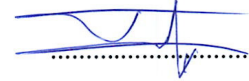
Assoc. Prof. of Complex System and Network



Dr. Hamed Al-bdour (Member)

Assoc. Prof. of Wireless Networks

(Mu'tah University)



تعتمد كلية الدراسات العليا
هذه النسخة من الرسالة
التوقيع.....التاريخ: ١٤٣٠/٨/٤

DEDICATION

*For every moment they prayed for me
For their endless support and patience
To my beloved Parents.*

ACKNOWLEDGEMENT

I am greatly expressing my sincere thanks to Allah, my Holy Lord, who was with me in every moment during my work on this thesis. Allah kindness and mercy gave me the ability and patience to proceed in bringing this piece of work to being.

I gratefully acknowledge all people who offered any help while working on this thesis. In particular, I would like to thank my supervisor, Dr. Mohammad Al-Shraideh, who helped, supported and leaded me during my work, and gave me desirable ideas and solutions for difficulties and problems.

I would also like to thank my dear friend, Noor Attari, for her efforts in reviewing my work.

In the last but not the least, I believe that this is the best opportunity to seize to provide special thanks to my family, who were always there, to support and guide me.

Rana Samhan

TABLE OF CONTENTS

COMMITT DECISION	ii
DEDICATION	iii
ACKNOWLEDGEMENT	iv
TABLE OF CONTENTS	v
LIST OF FIGURES	vii
LIST OF TABLES	viii
LIST OF EQUATIONS	ix
TERMINOLOGIES	x
ABSTRACT.....	xi
CHAPTER 1	1
Introduction.....	1
1.1 Overview	1
1.2 Problem Definition	4
1.3 Proposed Technique	5
1.4 Organization of the thesis	6
CHAPTER 2	7
Background and Literature Review	7
2.1 Software testing overview	7
2.1.1 Static Analysis	7
2.1.2 Dynamic Testing	8
2.1.3 Automatic Test Data Generation	14
2.2 Regular Expression and State Machine Overview	16
2.2.1 Regular Expression Overview	16
2.2.2 State Machine Overview.....	18
2.3 Genetic Algorithm overview	20
2.4 Related work in Genetic Algorithm with Software Testing	23
2.5 Related work in Numeric and String Predicates Testing	25
2.6 Related work in Regular Expression Matching	35
CHAPTER 3	36

Automatic Test Data Generation for Regular Expression Predicates.....	36
3.1 Introduction	36
3.2 Preprocessing Regular Expression Stage	39
3.2.1 Preprocessing RE that contains OR Operator	39
3.2.2 Preprocessing RE that contains Repetition Operator	40
3.3 Genetic Algorithm Stage	41
3.3.1 Regular Expression that contains OR Operator	41
3.3.2 Regular Expression that contains Repetition Operator	43
3.4 Case Study	44
CHAPTER 4	48
Experiments and Results	48
4.1 Introduction	48
4.2 Experiments Environment	48
4.3 Input Domain	49
4.4 Continuous and Binary Form	49
4.5 Experiments Design	50
4.6 GA Parameters Setup	52
4.7 Metric to record	56
4.8 Experiments	57
Program 1:.....	58
Program2:.....	64
Program3:.....	70
Program4:.....	76
Program5:.....	82
Program6:.....	88
Program7:.....	94
4.9 Analysis and Results	100
CHAPTER 5	104
Conclusion and Future Research	104
5.1 Accomplishments and Contributions to the Field	104
5.2 Future Researches	104
References.....	105
Appendix A.....	109
Summary in Arabic	123

LIST OF FIGURES

Figure 2-1: Simple GA Fundamental Mechanism.....	20
Figure 2-2: Single- point Crossover.....	22
Figure 2-3: Simple predicate example	25
Figure 2-4: example of string predicate.....	26
Figure 3-1: Flowchart of the proposed technique.....	37
Figure 3-2: Subprogram1.....	44
Figure 3-3: Subprogram 2.....	46
Figure 4-1: Regular Expression Contains OR operator at the start.	50
Figure 4-2: Regular Expression Contains OR operator at the middle.	50
Figure 4-3: Regular Expression Contains OR operator at the middle.	50
Figure4-4: Regular Expression contains one Closure Operator.	51
Figure 4-5: Regular Expression contains two Closure Operator.	51
Figure 4-6: Expression contains three Closure Operator.....	51
Figure 4-7: Expression contains Closure Operator and OR Operator.	51
Figure 4-8: The Flow of Experiments: E2 for CF, and E10 for BF, respectively.	60
Figure 4-9: Twenty Experiments for CF and BF in Program1	62
Figure 4-10: The Flow of Experiments: E3 for CF, and E10 for BF , respectively.	66
Figure 4-11: Twenty Experiments and Average Case for CF and BF in Program2.....	68
Figure 4-12: The Flow of Experiments: E4 for CF, and E6 for BF, respectively.	72
Figure 4-13: Twenty Experiments and Average Case for CF and BF in Program3.....	74
Figure 4-14: The Flow of Experiments: E8 for CF, and E10 for BF, respectively.	78
Figure 4-15: Twenty Experiments for CF and BF in Program 4.....	80
Figure 4-16: The Flow of Experiments: E2 for CF, and E5 for BF, respectively.	84
Figure 4-17: Twenty Experiments for CF and BF in Program 5.....	86
Figure 4-18: The Flow of Experiments: E5 for CF, and E6 for BF, respectively.	90
Figure 4-19: Experiments for CF and BF in Program 6	92
Figure 4-20: The Flow of Experiments: E9 for CF, and E2 for BF, respectively.	96
Figure 4-21: Experiments for CF and BF in Program 7.	98
Figure 4-22: Comparison between the Average of Generation Number in BF and Cf.	102
Figure 4-23: Comparison between the Average of Required Time in BF and Cf.....	103

LIST OF TABLES

Table 2-1: Summary of some special characters for RE	17
Table 2-2: Example of finding ED between R=aba and T=cab.....	31
Table 2-3: Example of finding OED between R=aba and T=cab.....	34
Table 4-1: Sample Experiments for population size=1000 in Program1.....	52
Table 4-2: Sample Experiments for population size=30 in Program1.....	53
Table 4-3: Sample Experiments for Program1 using two point Crossover.....	53
Table 4-4: Sample Experiments for Program1 using Gaussian mutation.....	54
Table 4-5: The Experiments' parameters and values.....	55
Table 4-6: Sample Experiments and Average Case for CF of Program 1.....	58
Table 4-7: Sample Experiments and Average Case for BF of Program 1.....	59
Table 4-8: Sample experiments and Average Case for CF of Program 2.....	64
Table 4-9: Sample Experiments and Average Case for BF of Program 2.....	65
Table 4-10: Sample Experiments and Average Case for CF of Program 3.....	70
Table 4-11: Sample Experiments and Average Case for BF of Program 3.....	71
Table 4-12: Sample Experiments and Average Case for CF of Program 4.....	76
Table 4-13: Sample Experiments and Average Case for BF of Program 4.....	77
Table 4-14: Sample Experiments and Average Case for CF of Program 5.....	82
Table 4-15: Sample Experiments and Average Case for BF of Program 5.....	83
Table 4-16: Sample Experiments and Average Case for CF of Program 6.....	88
Table 4-17: Sample Experiments and Average Case for BF of Program 6.....	89
Table 4-18: Sample Experiments and Average Case for CF of Program 7.....	94
Table 4-19: Sample Experiments and Average Case for BF of Program 7.....	95
Table 4-20: The Average Results for Seven Programs in CF and BF.....	100
Table A-1: Twenty Experiments for CF in Program 1.....	109
Table A-2: Twenty Experiments for BF in Program 1.....	110
Table A-3: Twenty Experiments for CF in Program 2.....	111
Table A-4: Twenty Experiments for BF in Program 2.....	112
Table A-5: Twenty Experiments for CF in Program 3.....	113
Table A-6: Twenty Experiments for BF in Program 3.....	114
Table A-7: Twenty Experiments for CF in Program 4.....	115
Table A-8: Twenty Experiments for BF in Program 4.....	116
Table A-9: Twenty Experiments for CF in Program 5.....	117
Table A-10: Twenty Experiments for BF in Program 5.....	118
Table A-11: Twenty Experiments for CF in Program 6.....	119
Table A-12: Twenty Experiments for BF in Program 6.....	120
Table A-13: Twenty Experiments for CF in Program 7.....	121
Table A-14: Twenty Experiments for BF in Program 7.....	122

LIST OF EQUATIONS

Equation 2-1: Hamming Distance Function	27
Equation 2-2: Character Distance Function.....	28
Equation 2-3: Edit Distance Function.....	29
Equation 2-4: Ordinal Edit Distance Function.	32

TERMINOLOGIES

ATG	Automated Test Data Generator
BF	Binary Form of GA
BHD	Binary Hamming Distance
CD	Character distance
CF	Continuous Form of GA
ED	Edit Distance
FSA	Finite State Automaton
FSM	Finite State Machine
GA	Genetic Algorithms
HD	Hamming Distance
OED	Ordinal Edit distance
RE	Regular Expression
SM	State Machine
SSF	Set String Form
TC	Test Cases

AUTOMATIC TEST DATA GENERATION FOR REGULAR EXPRESSION PREDICATES

**By
Rana Ali Badawi Samhan**

**Supervisor
Dr. Mohammad Alshraideh**

**Co-Supervisor
Dr. Abedellatef Abu Dalhoom**

ABSTRACT

Nowadays, Technology controls most of the areas in our day-to-day life. This technology, which depends mainly on software components, need to be reliable and user confident, therefore, it is necessary to concentrate on the testing stage of every technology, particularly, the software part.

In this thesis, we first present our new methodology for testing Regular Expression Predicate. To implement the proposed methodology, we use the state machine for regular expression recognition, in addition to using branch-testing technique to test each predicate in the program under test at least once. Finally, we use genetic algorithm as a search technique to find Test cases that will be used in the Regular Expression Predicate Testing.

We implement our proposed methodology using Matlab7.1. We execute seven programs using two forms: the Continuous and Binary Form. We apply deep analysis and study on the obtained results from the experiments. We found that the Continuous GA is faster (need less time) than the Binary GA in finding the Test case that used in Regular Expression Predicates Testing, while Binary GA need less Number of Generation to find the Test Case. In all experiments the Test Case was found and the Regular Expression Predicate was covered. The percentage between the BF Generation Number and the CF Generation Number is 0.58. On the other hand the percentage between the CF Required Time and the BF Required Time is 0.24.

CHAPTER 1

Introduction

1.1 Overview

Software Testing is an important stage of software development. It is the process of executing a program with the intent of finding errors and failure. The main goal of software testing process is to produce minimum number of test cases such that it reveals as many faults as possible. Software testing usually accounts for 50% to 80% of the software development cost (Pallavi et al. 2007), because producing input test cases is considered as an expensive component in software testing. The manual techniques are used in industry to generate test cases, but they are time consuming and labor-intensive techniques, usually resulting into poor coverage.

Software testing is applied on many levels of software development. These techniques are different in their nature and objectives (Lu Luo, 2002), and they include:

a) Unit Testing:

It is the testing that is implemented on the lowest level, which is used to test the basic and smallest unit of the program. The primary aim of the unit testing is to take the smallest unit in the program as isolated unit and determine if its behavior is as expected. As a result, each unit is tested separately before integrating it with other units.

b) Integration Testing:

It is the testing that is implemented after unit testing, and is used to integrate and combine the tested units as a group, and determine if its behavior is as expected. The integrated units are ready for system testing.

c) System Testing :

It is the testing that is applied to the complete integrated system as one unit system testing, which takes as its input all of the "integrated" software components that have successfully passed integration testing. Here, the testing attempts to discover defects that are properties of the entire system rather than of its individual components.

d) Regression Testing :

It is the testing that is implemented after any modification on the system. It is considered as a retesting process that is used to ensure the correctness of the modifications.

e) Acceptance Testing:

It is the testing that is implemented by the user or the customer after the developer hand over the system to them. The main aim of this testing is to gain the acceptance of the user rather than to ensure the correctness of the system.

f) Beta testing

When partial or full version of the software is available, the development organization can offer it free to one or more experienced users or beta testers. These users install the software and use it as they wish. The aim of this process is to report any errors revealed during using this version of the software.

Test Data Generation

Software Testing uses the Test data generation to identify a set of test data, which satisfies given testing criterion (Ruilian and Michael, 2003). Test data generation techniques are Manual (Static) and automatic (Dynamic).

Static analyzing tools analyze the software under test without executing the code; it is a limited analysis technique for programs including array references, pointer variables and other dynamic constructs. Experiments have shown that this kind of evaluation of code inspections (visual inspections) has found static analysis is very effective in finding 30% to 70% of the logic design and coding errors in a typical software symbolic execution. Evaluation is a typical static tool for generating test data.

In contrast to Static analysis, Dynamic testing tools involve the execution of the software under test and rely upon the feedback of the software in order to generate test data. Dynamic testing generator means reduction in time, effort, labor and cost for software testing (Boyapati, et al. 2002). There are many types of dynamic test data generators; pathwise, data specification and random test data generator. Dynamic test data generation is a popular approach to generate test cases that depends on executing specified programs in order to get needed information to generate suitable test cases.

1.2 Problem Definition

Predicate based testing is an approach in software testing which tests numerical predicate in the program that includes simple or compound predicate. This technique excludes non-arithmetic expressions such as character strings or regular expressions but it includes Boolean variables, relational expressions, and Boolean operators, therefore in the thesis we work to find similar approach for Regular Expression.

Regular Expressions (RE) is one of the components in some programs that need to be tested, often called a pattern, and Regular Expressions are expressions that describe a set of strings; they are set of characters that specify a pattern. The RE are usually used to give a concise description of a set, without having to list all elements, the functions of it is to check if a particular string matches a given RE.

In this thesis we aim to test Regular Expressions that are usually located in predicates with string characters, so we need to present a technique that is used to test non-numerical predicates e.g. predicates that contains regular expressions. For example, suppose that a program contains the following condition statement:

```
If (reg_exp == (alb)*)
{
    // execution required action
}
```

Here in this example, the predicate should be executed by finding value for **reg_exp** variable, which make the condition true. This input test case must belong to the language of **(alb)*** and accepted in the state machine of RE.

1.3 Proposed Technique

The aim of our research is to test the branches and predicates that contain regular expression. Our proposed technique combines three main concepts branch testing technique, Finite State Machine, and genetic algorithm. Branch testing technique is used to achieve the coverage branch for the program under test, while the Finite State Machine used to ensure that the test case belongs to the regular expressions language, and finally, genetic algorithm used as a search technique to find the test cases for the required branch.

The following steps summarize our proposed technique:

1. Use the branch testing technique to traverse all the branches in the program under test. If any branch or predicate contains simple or complicated regular expressions, proceed to the following steps.
2. Construct the state machine that relates to the RE that is in the predicate.
3. Preprocess and manipulate RE such that it will be able to enter GA stage.
4. Apply GA using preprocessed RE from the preceding step.
5. After applying the GA, the output is the test cases that execute the RE predicate.
6. Pass the TC (string) to the state machine in order to ensure that the output string belongs to the specified RE.
7. Use the TC in testing the RE predicate.
8. The specified RE branch is executed as a result, and our research aim is achieved.

1.4 Organization of the thesis

The rest of the thesis is organized as follows: Chapter 2 gives a brief background of software testing, regular expression, state machine and genetic algorithms. In addition we will review some of related work in string testing and regular expression matching. In Chapter 3 we presents our proposed technique that used to testing regular expression combined with some of examples that used to illustrate our techniques in details. Chapter 4 discusses the experiments setup, results, and analysis. Chapter 5 concludes the thesis work, thesis contributions, and future work.

CHAPTER 2

Background and Literature Review

2.1 Software testing overview

Software testing is the process of analyzing a software item to detect the differences between existing and required specifications, and to evaluate the features of the software item (Alshraideh and Bottaci, 2006). The Software testing should be done throughout the whole development process.

Techniques that related to software testing are usually classified into two categories: static analysis and dynamic testing. Static techniques are performed without actually executing programs. The program source code is reviewed statement by statement. It uses the program requirements and design documents. In contrast, dynamic testing techniques execute the program under test on test input data and notice its output. Usually, the term testing refers to just dynamic testing.

2.1.1 Static Analysis

Static testing is a type of software testing where the software isn't actually used. It is generally not a detailed testing, but checks mainly for the code syntax, algorithm, or document, so it is primarily a syntax checking of the code or and manually reading of the code or document to find errors. This type of testing can be used by the developer who wrote the code. Bugs discovered at this stage of development are less expensive to fix than later in the development cycle. Static technique is concerned with the analysis and checking of system representations throughout all stages of the software life cycle, it focuses on the range of methods that are used to determine or

estimate software quality without reference to actual executions. The advantage of static analysis is the ability to complete the process prior to actual coding; as a result, it prevents errors to occur before executing the system. Some of static analysis techniques in this area include code inspection, Code Walkthroughs Desk Checking, Code Reviews. Code inspections and walkthroughs are the two primary static analysis methods and they have a lot in common. Inspections and walkthroughs involve the reading or visual inspection of a program by a team of people (Lu Luo, 2002).

2.1.2 Dynamic Testing

It is a type used to describe the testing of the dynamic behavior of the code. In dynamic testing, the software must actually be compiled and run; Actually Dynamic Testing involves working with the software, giving input test values and checking if the output variables are as expected. Dynamic testing techniques execute the program under testing on test input data and observe its output.

Techniques in this area include synthesis of inputs, the use of structurally dictated testing procedures, and the automation of testing environment generation. Dynamic testing can apply only after compilation and linking. It may involve running several test cases each of which may take longer than compilation. It finds bugs only in parts of the code that are actually executed (Lu Luo, 2002).

There are Two Primary classifications of dynamic testing:

- Functional testing (Black box testing)
- Structural testing (White box testing)

Functional Testing

It is a black box testing that depends mainly on the requirements of the system and does not need to know the internal code of the system. Here, the tester does not know the internal structure of the item being tested. For example, in a black box test on software design the tester only knows the inputs and what the expected outcomes should be and not how the program arrives at those outputs. The tester does not need any further knowledge of the program other than its specifications. The selection of test cases for functional testing is based on the requirement or design specification of the software entity under test. Functional testing emphasizes on the external behavior of the software entity not the internal source code (Last, Eyal, and Kandel, 2005).

These are some of techniques that in functional testing:

- Equivalence partitioning :

Equivalence partitioning is a software testing technique that divides the input data of a software unit into partitions of data from which test cases can be derived. Equivalence classes form a partition of a set that is a collection of mutually disjoint subsets whose union is the entire set. This technique tries to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.

The use of this technique usually for two motivations: Sense of complete testing, and Avoid redundancy. The goal of equivalence class testing is to identify test cases by using one element from each equivalence class. In this technique there

are three types of techniques: Weak Equivalence Class Testing, Strong Equivalence Class Testing, and Traditional Equivalence Class Testing.

- Boundary value analysis:

Boundary value analysis technique focuses on the boundary of the input space to identify test cases. Usually the boundaries of input and output ranges of a software component are common locations for errors that result in software faults. Boundary value analysis assists with the design of test cases that will exercise these boundaries in an attempt to uncover faults in the software during the testing process. The expected input and output values should be extracted from the component specification. The input and output values to the software component are then grouped into sets with identifiable boundaries. It is important to consider both valid and invalid partitions when designing test cases. In this technique, there are three types of techniques Robustness Test Cases techniques, Worst Case Testing techniques, and Robust Worst Case Testing techniques.

- Decision table testing:

Decision tables are a precise yet a compact way to model complicated logic. Decision tables, like if-then-else and switch-case statements, it associates conditions with actions to perform. Decision tables can associate many independent conditions with several actions in an elegant way. Decision tables make it easy to observe that all possible conditions are accounted for. Each condition corresponds to a variable, relation or predicate.

Possible values for conditions are listed among the condition Alternatives:

- Boolean values (True / False) – Limited Entry Decision Tables.
- Several values – Extended Entry Decision Tables.
- Don't care value.

Structural Testing

The White box testing (Structural testing) generates the test cases depending on the knowledge of internal code of the system. It uses the internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all paths through the software. The tester chooses test case inputs to exercise paths through the code and determines the appropriate outputs. In this technique, where the software system is viewed as a “white box”, the selection of test cases is based on the implementation and the source code of the software system. The goal of selecting such test cases is to cause the execution of specific code segments in the system, such as specific statements, program branches or paths. The expected results are evaluated on a set of coverage criteria. Examples of coverage criteria include path coverage, branch coverage, and statement coverage. Structural testing emphasizes on the internal structure of the software system (Stamer, 1995).

There are several white box (structural) testing criteria:

a) Statement Testing:

Using this testing criterion, every statement will be executed at least once in the software under test during testing.

b) Branch Testing:

Branch coverage is a stronger criterion than statement coverage. It includes every possible outcome of all decisions or branch to be exercised at least once; this means that all control transfers are executed. It includes statement coverage since every statement is executed if every branch in a program is exercised once. In the proposed methodology we use the Branch coverage in order to ensure that each predicate is executed at least once.

c) Path Testing:

In path testing, every possible path in the software under test is executed; this increases the probability of error detection and is a stronger method than both statement and branch testing. A path through software can be described as the conjunction of predicates in relation to the software's input variables. In Path coverage it is not necessary to cover all predicates in the program.

These are some of the techniques that are used in structural testing, including Control flow testing, Data flow testing, and Program Slicing:

Control flow testing

This technique indicates the order in which the individual statements, instructions, or function calls of an imperative or functional program are executed or evaluated (Stamer, 1995).

Data flow testing

This technique uses the control flow graph to explore the unreasonable things that can happen to data during execution. It includes a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects.

Program Slicing

It is a decomposition technique that extracts statements relevant to a particular computation from a program. In general, the program slicing techniques are used to Program Debugging, Integration, Program Understanding, Software Maintenance, and Reverse Engineering.

2.1.3 Automatic Test Data Generation

Test Data Generation is a technique that is used to generate test data such that if the needed test requirements do not stratify, therefore the input test data will be used to determine how the input test data is close to the specific requirement (HUANG, 1975).

Some Automatic Test Data Generation techniques are:

- **Dynamic Test Data Generation:**

The Dynamic Test Data Generation technique depending on the feedback, the input test data are gradually modified, until they reach to the required requirements (B. Korel, 1990). The process of executing program repeatedly until reaching the required requirements can be reduced as function minimization, which can be performed using the gradient descent (K. C. Tai, 1996), genetic Search (Michael et al., 2001), and simulated annealing (Tracey et al.,1998).

Pargas, et al. (1999) classifies the automated test data generation techniques into random test data generator (Chu H.D, 1996), structural or path-oriented test data generator (Michael, et al., 1997), goal-oriented test data generator (B. Korel, 1990) ,and intelligent test data generator (Roper,1995).

- **Random Test Data Generation:**

It is based on developing test data randomly until the suitable test data is found (Duran and Ntafos, 1984). Although it is easy to implement, but randomly generated test data have difficulties in satisfying a specific requirement, such as domain testing for a predicate border associated with a

chosen path. In fact, random test data generation performs poorly, and in general, the Random test data generation is considered no effective on realistic programs (B. Korel, 1990).

- **Symbolic Execution**

The basic idea in a symbolic execution system is to allow numeric variables to take on symbolic values instead of numeric values. However, symbolic execution is very computational intensive, and a number of technical problems such as indefinite loops, subprogram calls, and array references and so on, are met in practice when symbolic execution is performed.

Moreover, if input variables are character string variables, symbolic expression becomes more difficult to apply Symbolic execution. Because Symbolic execution requires the systematic derivation of these expressions, which can take much computational effort, the values of all variables are maintained as algebraic expressions in terms of symbolic names. The value of each program variable is determined at every node of a flow graph as a symbolic formula (expression) for which the only unknown is the program input value. The symbolic expression for a variable carries enough information such that, if numerical values are as assigned to the inputs, a numerical value can be obtained for the variable, this is called symbolic evaluation.

2.2 Regular Expression and State Machine Overview

2.2.1 Regular Expression Overview

Regular Expression is a set of characters that specify a pattern. It's usually used when we want to search for specific lines of text containing a particular pattern. RE can contain both special and ordinary characters. They are used in many programming languages to search and manipulate text based on patterns. For example, Perl, Ruby and Tcl have a powerful regular expression engine built directly into their syntax.

The main functions of them are to check if a particular string matches a given RE, or if a given RE matches a particular string. One of the main problems of the RE is find all positions in a string where a RE matches (Kurtz, 2003).

The regular expression is recursively defined as follows (Muzatko, 1996) and (W.pratt, 4th).

1. Individual terminal symbols are regular expressions.
2. ϵ , \emptyset are regular expressions.
3. If, a and b are regular expressions, then so are:
 - a or b, (union)
 - ab, (concatenation)
 - (a), (parentheses)
 - a*, (closure)
4. Nothing else is a regular expression.

The value $h(r)$ of regular expression x is defined as follow (Muzatko, 1996).

1. $h(\epsilon) = \epsilon, h(\emptyset) = \emptyset$.
2. $h(x) = x$
3. $h(x+y) = h(x) + h(y)$ where y is regular expression.
4. $h(\mathbf{h.x}) = h(x).h(y)$ where y is regular expression.
5. $h(x^*) = (h(x))^*$

The following are examples of regular expression:

- alb^* : denotes $\{\epsilon, a, b, bb, bbb, \dots\}$
- $(alb)^*$: denotes the set of all strings with no symbols other than a and b , including the empty string: $\{\epsilon, a, b, aa, ab, ba, bb, \dots\}$

RE can contain both special and ordinary characters. The ordinary characters like "a", "b", and "0" are the simplest regular expressions, they simply match themselves.

The special characters like "|" and "*" (Table 2-1 summarizes some of special characters for RE).

Table 2-1: Summary of some special characters for RE

Special Character	Effect / Meaning
*	Causes the resulting RE to match 0 or more repetitions of the preceding RE, as many repetitions as are possible.
	$A B$, where A and B can be arbitrary REs, creates a regular expression that will match either A or B .
{ <i>m</i> }	Specifies that exactly m copies of the previous RE should be matched; fewer matches cause the entire RE not to match.
.	(Dot.) In the default mode, this matches any character except a newline.
^	(Caret.) Matches the start of the string and in MULTILINE mode also matches immediately after each newline.
?	Causes the resulting RE to match 0 or 1 repetitions of the preceding RE.

In this thesis, we focus on the two characters, (*) Repetition (Closure) Operator and (|) Or Operator because they are consider representative character for others, and they are the most used operator in the regular expression.

2.2.2 State Machine Overview

Finite State Machine (FSM), or finite state automaton (FSA) is an abstract model composed of a finite number of states, transitions between those states, and actions.

A finite state machine is a model of a machine with a primitive internal memory.

The FSA consists of starting node, one or more final states, and a set of transition (labeled arcs) from one state to another. State machine is used to recognize character of string such that, any string that takes the machine from the initial state to the final state through a series of transition is accepted by this FSA.

There are two type of FSM (Hanif, Ahmed, and Aqdas, 2006)

1. Deterministic FSM (DFSM)

A FSM where for each input event and state there is exactly one transition.

This means that the transition and output functions deterministic, and the output event and next state are uniquely determined by a single input event.

2. Non-deterministic FSM (NDFSM):

A FSM where for each input event and state there is not exactly one transition necessarily. In this FSM, the next state depends not only on the current input event, but also on a number of subsequent input events.

Formal definition

The FSM is defined as a 5-tuple, (Q, Σ, T, q_0, F) , consisting of:

- A finite set of states Q .
- A finite set of input symbols Σ .
- A transition function $T: Q \times \Sigma \rightarrow P(Q)$.
- A start state $q_0 \in Q$.
- A set of states F distinguished as accepting (final) states $F \subseteq Q$.

Construction of FSM for RE

There are two techniques to construct the FSM; the first one is Thompson [1968] construction, which, produces the FSM that has the size linear in the size of given expressions and does so in linear time. However, it also gives machines that have at most two transitions into and at most two out of each state.

A second construction method that is older than Thompson's construction is Glushkov construction it is a two-step Process. First, inductively compute three sets of symbols and then, second, compute the machine directly from these sets (Hanif, Ahmed, and Aqdas, 2006).

2.3 Genetic Algorithm overview

Genetic Algorithms (GAs) were invented by John Holland in the 1960s and were developed by Holland and his students and colleagues at the University of Michigan in the 1960s and the 1970s (Mitchell, 1999). It is an optimization and search technique based on the principles of genetics and natural selection, inspired by Darwin's theory about evolution. GA begins with population of randomly generated chromosomes, each chromosome is considered as a candidate solution to the problem being solved, and advances towards better chromosome by applying genetic operators based on the genetic processes occurring in nature. Each state of population is called generation. For each chromosome at every generation there is a fitness value, which indicates the goodness of the solution, represented by the chromosome values. Based on these fitness values (cost), the evaluation and the selection of the chromosomes are done, which is used to generate the new generation. The new chromosomes are created using genetic operators such as crossover and mutation (Abo-Hammour, 2002).

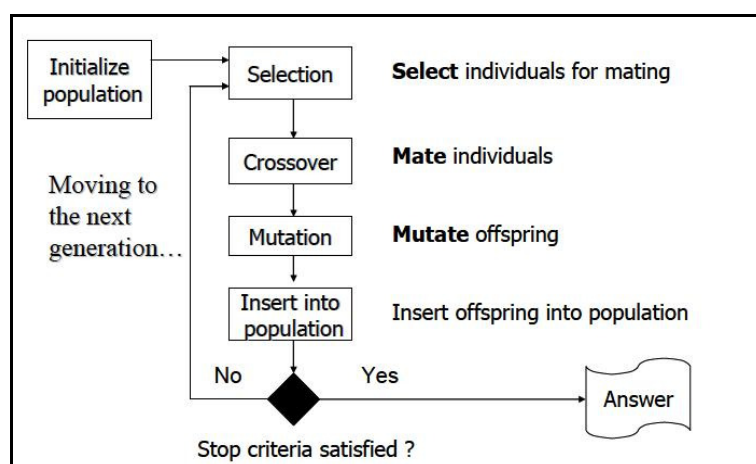


Figure 2-1: Simple GA Fundamental Mechanism

Figure 2-1 explained the main steps for the genetic algorithm, which is listed below:

1. Initialization

An initial population is randomly generated. The population consists from a set of individuals (chromosomes). Each chromosome contains a group of genes. Each gene represent variable in the potential problem.

2. Evaluation

It involves a function which is called objective function, used to rate the candidate solutions quality. This is the only single measure of how good a single chromosome is compared to the rest of the population. The fitness value is a nonnegative measure used for maximization or minimization purpose. In this thesis, we used it for minimization purpose. The cost function estimates the number of search operations that need to transform the candidate solution into the optimal solution. The main fitness function that we focus on it in thesis is the OED, which discuss later in this chapter.

3. Selection

In this step, the chromosomes are chosen from the current population in order to create new children for the next generation. The smaller fitness function value, the higher probability of the chromosome to contribute one or more children in the next generation. Usually the ranking operation happened for the chromosome then the selection operation occurred selection method according to specific techniques. In our implementation we use the stochastic uniform, lays out a line in which each parent corresponds to a section of the line of length proportional to its scaled value. The algorithm moves along the line in steps of equal size. At each step, the algorithm allocates a parent from the section it lands on. The first step is a uniform random number less than the step size (Haupt, 2004).

4. Crossover

In this step, each pair of chromosomes are taken and cut at some randomly chosen point to produce two segments in each chromosome then segments from different chromosome are swapped over to produce two new full-length chromosomes, which called children. Children inherit some genes from each parent and have new structures compared to those of their parents. We implement our proposed technique using Single-point crossover (Chen, 2002).

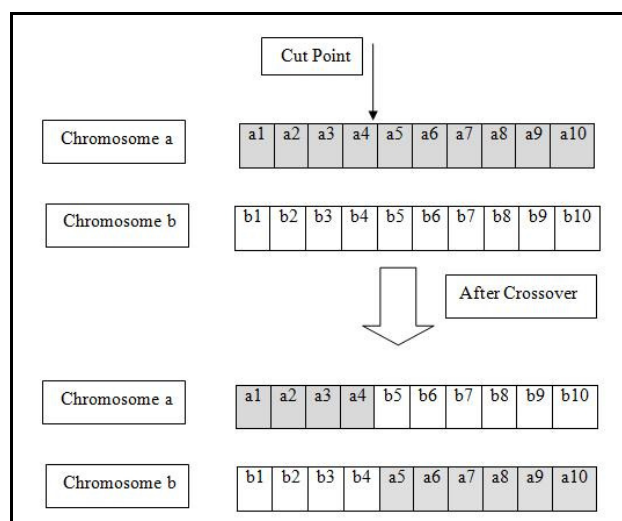


Figure 2-2: Single- point Crossover.

Figure 2-2 illustrates the single crossover operation which mainly depend on randomly choose cut of point and then exchanging the genes of the two chromosomes after the cut point.

5. Mutation

This step is applied after the selection and crossover operation. It is used to apply changes at chromosomes individually by making changes in some chromosome in order to ensure genetic diversity within the population. In our implementation we use

Uniform mutation method which in Continuous GA includes two steps. The first one is Selects a fraction of the vector entries of an individual for mutation, where each entry has a probability Rate of being mutated. In the other hand the second step, the algorithm replaces each selected entry by a random number selected uniformly from the range for that entry. In Binary GA the first step as in the Continuous GA while in the second step is different such that, the selected bits are inverted to other bits (Chen, 2002).

6. Termination

It is the step where the GA is terminating when some criterion is met such that maximum number of generation reached or the cost equal zero.

2.4 Related work in Genetic Algorithm with Software Testing

In literature, GA was used to solve many complicated problems in the computer science. One of these problems is software testing, such that GA helps in finding Test cases that is used in testing purpose.

Sthamer (1995) studied the use of GA as a Test Data Generator for structural testing.

Michael et al. (1997), performed experiments to compare between the random test data generation and genetic search for test data and demonstrated that genetic approaches outperform the random search in the more difficult setting.

Khor and Grogono (2004), introduced genet an Automated Test Data Generator (ATG) to generate test data for branch coverage.

Alzabidi, Kumar, and Shaligram (2009) proposed GA with different parameters combinations used to automate the test data generation for path coverage. The investigation involves crossover strategies and methods of selecting of parents for reproduction and mutation rates. The results of the study showed that double crossover was more successful in path coverage. The study results Also that, selecting parent for reproduction according to their fitness is more efficient than random selection.

2.5 Related work in Numeric and String Predicates Testing

As we mentioned previously, there are predicate testing techniques that are concerned with numeric predicates, which are used to compare numbers as illustrated in the example below in Figure 2-3:

```

If (y == 20)
{
//TARGET
}

```

Figure 2-3: Simple predicate example

In this example, we can use the Automatic methods, which aim is use the information that is gained by execution of the program under test. The most basic Automatic method is Random Test Data Generation technique and Dynamic Test Data Generation.

In Random Test Data Generation technique, test data is generated randomly. Each test case is then executed and either considered or discarded according to whether it executes the predicate goal. In this technique, the probability that a randomly generated input will set the variable to be equal to 0 may be very small. In general, random test data generation performs poorly and is generally considered ineffective at covering all branches in realistic programs (Zhao and Lyuv, 2003). In dynamic test data generation, if some desired test requirement is not reached, data generated in each test execution is used to identify how close the test input is to meeting the requirement. With the aid of feedback, test inputs are gradually modified until one of them satisfies the requirements using function minimization.

Also in numeric predicate testing, we can use Heuristic search techniques such as genetic algorithms and simulated annealing, which are high-level frameworks, which use heuristics to find solution without need to perform a full exhaustive enumeration of a search space. In fact, many test generation techniques are based around some notion of the coverage of the code. This coverage can be measured and incorporated into an objective or cost function. Better test values should be rewarded with lower cost values, whereas poorer test values should be related to higher cost values. With feedback from the cost function, the search looks for better tests based on a heuristic evaluation of existing tests. In the previous numeric predicate, the cost function is $(y - 20)$. Finding the value 20 of the cost function where $y=20$ is a required solution to achieve the predicate (Alshraideh, 2007).

Another predicate in the program that we usually need to test is the non-Numerical data types, including the string data type and regular expression.

String predicates take around 6% of expression predicates in programming language (Alshraideh, 2007). The following Figure 2-4 illustrated an example of string predicate.

```

If (S == "Master")
{
    // TARGET
}

```

Figure 2-4: example of string predicate.

The problem is to find an input string (S) so that the required branch is executed. If the branch is not executed, a cost is associated with S. This cost is used to guide the search as we mentioned in the numeric branch. Given the use of a particular search technique such as a genetic algorithm, a key problem is how to compute a useful cost for this predicate failure. For example, for two test cases $s1 = "Masor"$ and,

s2 = "Naster". The problem is to find which one, if any, should have the lower cost. Until the problem of a cost function for string equality is solved, it overly reduces software testing approaches for applications in practice, since string predicates are widely used in programming. Some of the string equality cost a function that is used in branch string testing is Binary Hamming distance (BHD), Character distance (CD), Edit distance (ED) and Ordinal Edit Distance (OED) (Alshraideh, 2007). We will concentrate on the OED fitness function, which we adapted in our proposed methodology.

- **Binary Hamming Distance (BHD)**

It is a fitness function that is used to find the number of bits that is different between Two-bit vectors (strings). This representation could be used for character strings by simply working with the underlying bit representation of each character. Once each string is converted to a bit string by concatenating the bit patterns of each character, the Hamming distance of two equal length strings may be easily computed.

The HD function was extended to deal with unequal length strings, so that any bits in one string that extend beyond the length of the shorter string are counted as mismatched. More formally, the distance between two strings A and B as shown in Equation 2-1:

$$HD(A, B) = \left(\sum_{j=0}^{minlen} \sum_{i=0}^{i<7} (A_i | B_i) \right) + 7(maxlen - minlen)$$

Equation 2-1: Hamming Distance Function

Where minlen, maxlen are the minimum length and maximum length of string A and B, A_i and B_i are bits number, i and j is XOR operator.

The BHD has limitations and problems. Some of them are that in some cases the solutions can be close to each other in decode solution space (character representation), but are far apart in the encoded binary representation.

Another problem is the maximum number of fitness values it can produce, which is $(7 * \text{maxlen})$, where *maxlen* is the maximum length of the two strings compared, taken in consideration that we can produce more than these test cases.

- **Character Distance (CD)**

It is a fitness function where characters may be mapped into an ordinal space according to each character's ordinal value ASCII Code. It represents the sum of the absolute differences between the ordinal character values of corresponding character pairs. For strings of unequal length, any character without a corresponding character increases the cost by 128 degree (represent insert new character).

Let string $s = s_0s_1 \dots s_{k-1}$ be of length k where S_i is the ordinal value of the i th character. Similarly, let string $t = t_0t_1 \dots t_{l-1}$ is a string of length $k \leq l$ then the character Distance function is shown in Equation 2-2:

$$CD(s, t) = \sum_{i=0}^{i=k-1} |s_i - t_i| + 128(l - k)$$

Equation 2-2: Character Distance Function.

- **Edit Distance (ED)**

ED is defined as the process of specifying the *minimum* number of *point mutations* required to transform a string (A) into another string (B).

The edit or mutation operations are:

1. Substitution a letter.
2. Insert a letter.
3. Delete a letter.

For example, the edit distance between "master" and "manar" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

1. master → masar (substitution of 'n' for 's')
2. masar → mastar (insert 't' after 's')
3. mastar → master (substitution of 'a' for 'e')

The edit distance function is defined by the recurrence relation below where $s: a$, $t: b$ are character strings, each consisting of a possibly empty string s , t , followed by the character a , b . the ED function shown in Equation 2-3.

$$ED(s: a; t: b) = \min(ED(s: a; t) + 1; ED(s; t: b) + 1; ED(s; t) + ED(a; b))$$

Equation 2-3: Edit Distance Function.

The edit distance of two characters is one unless they are equal, in which case it is zero. The edit distance of an empty string and a given string is the length of the given string.

The algorithm implementation for computing the Levenshtein distance involves the use of an $(n + 1) \times (m + 1)$ matrix, where n and m are the lengths of the regular expression R , and the string T .

Description of Edit Distance Algorithm

Step 1:

Set n to be the length of R .

Set m to be the length of T .

If $n = 0$, return m and exit.

If $m = 0$, return n and exit.

Construct a matrix containing $0...m$ rows and $0...n$ columns.

Step2:

Initialize the first row to $0...n$.

Initialize the first column to $0...m$.

Step3:

Insert each character of R (i from 1 to n).

Step4:

Insert each character of T (j from 1 to m).

Step5:

If $R[i]$ equals $T[j]$, the cost is 0.

If $R[i]$ does not equal $T[j]$, the cost is 1.

Step6:

Set cell $d[i,j]$ of the matrix equal to the minimum of:

- The cell immediately above plus 1: $d[i-1,j] + 1$. (Deletion)
- The cell immediately to the left plus 1: $d[i,j-1] + 1$. (Insertion)
- The cell diagonally above and to the left plus the cost: $d[i-1, j-1] + \text{cost}$ (Substitution).

Step7:

After the iteration steps (3, 4, 5, and 6) are complete, the distance is found in cell $d[n, m]$.

Example for ED:

Let $R=aba$, $T=cab$.

We apply the ED in order to find the differences between R and T .

Table 2-2: Example of finding ED between $R=aba$ and $T=cab$.

		a	b	a
	0	1	2	3
c	1	0	1	2
a	2	1	1	1
b	3	2	1	2

The ED = 2 which is the bottom right most corner.

- **Ordinal Edit Distance (OED):**

Ordinal Edit distance defined as the process of specify the minimum number of *point* mutations required to transform a string (A) into another string (B). Considering the ASCII Code of each characters such that the minimum number of character express using ASCII code instead of the number of characters.

The edit distance function can be modified to take account of the difference in character values whenever a character is substituted. The edit distance of two characters can be taken to be equal to the absolute difference in their ordinal values. The ordinal edit distance (OED) could thus be defined as the following Equation:

$$\text{OED}(s: a; t: b) = \min (\text{OED}(s: a, t) + k, \text{OED}(s, t: b) + k, \text{OED}(s, t) + |a - b|)$$

Equation 2-4: Ordinal Edit Distance Function.

Where k is the insertion or deletion cost and a, b in $|a - b|$ are interpreted as ordinal values.

Here, in OED the cost of insertion, k , was chosen to be 128. Given that any match that can be achieved by an insertion into one string can also be achieved by a deletion in the other, the cost of deletion was also chosen to be 128, instead 1 of insertion and deletion in ED.

Using 128 as the cost of insertion and deletion, however, gives $OED(XMaster, Master) = 128$ and yet $OED(Master, Thesis) = 6$ which is too low since the search Effort required to match (Master, Thesis), six substitutions, should be higher than the effort to match XMaster, Master, a single deletion. The problem is that substitution costs become unreasonably low as corresponding character values approach each other. The low, non-zero substitution costs were therefore offset away from zero

while retaining the maximum cost at 128. This was done by setting the substitution cost to be $128/4 + (3*|a - b|)/4$ when $|a - b| > 0$ and zero otherwise.

The OED, which depend on the ASCII code in calculation the cost of RE matching.

The algorithm implementation for computing the Levenshtein distance involves the use of an $(n + 1) \times (m + 1)$ matrix , where n and m are the lengths of the regular expression r , and the string s .

Description of Ordinal Edit Distance Algorithm:

Set r to be the regular expression.

Set s to be the string, which will compare with r .

Set n to be the length or r .

Set m to be the length s .

Construct a matrix containing $0 \dots m$ rows and $0 \dots n$ columns.

Step1:

Initialize the first rows to $0, 128 \dots n * 128$.

Initialize the first column to $0, 128 \dots m * 128$.

Step 2:

Insert each character of r (i from 1 to n).

Insert each character of s (j from 1 to m).

Step3:

If $s[i]$ equals $t[j]$, the cost is 0.

If $s[i]$ does not equal $t[j]$, the cost is $128/4 + (3*|a - b|)/4$.

Step4:

Set cell $d[i, j]$ of the matrix equal to the minimum of:

- The cell immediately above plus 128: $d[i-1, j] + 128$. (Deletion)

- The cell immediately to the left plus 128: $d[i, j-1] + 128$. (Insertion)
- The cell diagonally above and to the left plus the cost: $d[i-1, j-1] + \text{cost}$. (substitution)

Step5:

After the Steps 2, 3, 4 are complete the distance is found in cell $d[n, m]$ the right bottom cell.

Example for OED:

Let $R=aba$, $T=cab$. We apply the OED in order to find the differences taking in consideration the ASCII code between R and T.

Table 2-3: Example of finding OED between $R=aba$ and $T=cab$.

	S	a (97)	b (98)	a (97)
T	0	128	256	384
(99) c	128	33.50	160.75	288.75
(97) a	256	128.00	66.25	160.75
(98) b	384	256.00	128.00	99.00

The OED = **99.00** which is the bottom right most corner.

2.6 Related work in Regular Expression Matching

Muzatko (1996) proposed an algorithm, which is extended the hamming, and edit distance fitness function. The proposed algorithm constructs FSM that accepts strings with up to a defined number of mismatches. The algorithm constructs a non-deterministic machine containing l copies of the regular expression machine where l is the maximum number of mismatches to be detected but the algorithm complexity is exponential which is $O(2^k)$ in the worst case.

Alshraideh and Bottaci (2006) presented a proposed cost function that used the FSM to parse the given string to check its membership of the regular set. Instead of finite state machine producing a simple accept or reject output, however, the machine computes a cost by counting mismatched state transition. The complexity of the algorithm is $O(2^k)$.

CHAPTER 3

Automatic Test Data Generation for Regular Expression Predicates

3.1 Introduction

In this chapter, we present the proposed methodology, which relates to testing regular expression predicates in the program under test. Before proceeding in the methodology, it is important to mention that Regular expression Predicates occupy about 5% from 6% of string predicates in programming language (Alshraideh, 2007)

Which mean that they have an existence in the languages therefore, we try studying them, inorder discover techniques used to testing these predicates.

We use the branch testing, state machine and genetic algorithm inorder to achieve our research aim.

In chapter 1, we presented brief steps of our proposed technique, in this chapter; we discuss these steps in more details. Figure 3-1 below represents the flowchart of the proposed technique.

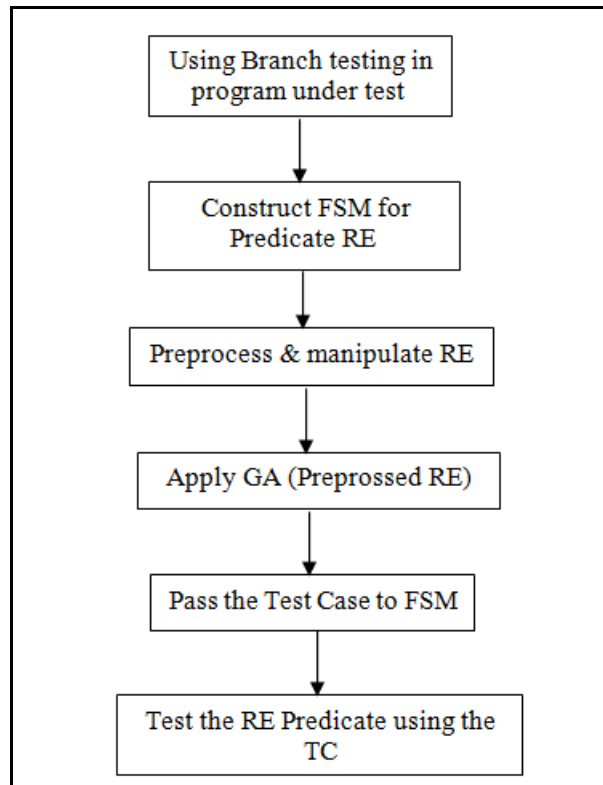


Figure 3-1: Flowchart of the proposed technique.

Figure 3-1 lists the main six steps in the proposed methodology. The first step is to use the branch testing technique, which, is used to execute each branch in the program under test at least once. This step is followed by constructing the FSA of the Regular Expression that is locates in predicate, then RE preprocessed such that is differentiated according to the type of RE that it contains. The Regular Expression is simplified to a set of string form (SSF), in order use these strings in GA, to be more specific, in the fitness function, as we proposed in our technique ((Illustrated in detail later)).

After simplifying the Regular Expression, the Genetic Algorithm is executed using the SSF and the proposed fitness functions. When the Test Case (TC) is discovered, it passed to the state machine, to ensure that the string belongs to the language of the RE. Finally, the TC is used to accomplish our aim in testing the Regular Expression

Predicate. Chapter 3 concentrates on two stages: the Preprocessing Regular Expression stage and Genetic Algorithm stage, because other stages were discussed previously in Chapter 2.

3.2 Preprocessing Regular Expression Stage

Input: Regular Expression (RE).

Output: Set of string forms (SSF) that equal to the Regular Expression.

Preprocessing Regular Expression stage is responsible for simplifying the RE, and transforming the RE into another form that is equal to the RE form.

In the proposed method, we focus on the two mainly special characters in regular expression syntax, which are:

- OR Operator (|) which described in Table 2.1.
- Repetitions of the preceding RE (*) also described in Table 2.1.

3.2.1 Preprocessing RE that contains OR Operator

OR Operator is one of the most used characters in the Regular Expression syntax. It is used to match either RE1 or RE2 with the rest of RE.

In this stage, we manipulate this RE as follows:

OR Operator (|) concatenates the rest of RE with one left or right character, not both of them, so as a result, the set of string form contains the rest of regular expressions concatenate with right side or with left side of the OR Operator, regardless of the length of the TC.

In other words, if we have the following regular expression, the set of string will be as follows:

RE: a(alb)a

SSF: {aaa, aba}.

As we see in this example, the proposed technique converts the RE to another equal form, which is a string form that facilitates testing of RE predicate, and let GA used it as an input set. Here in this point we should concentrate on that all the strings that belong to the SSF also belong to the language of regular expression and should be acceptable at RE finite state machine.

3.2.2 Preprocessing RE that contains Repetition Operator

The Repetitions Operator also is considered one of the most used characters in the Regular expression syntax. As we mentioned previously, it causes the resulting RE to match 0 or more Repetitions of the preceding RE, as many Repetitions as are possible.

In this stage, we manipulate this character as following:

Repetitions Operator (*) means 0 or more Repetitions of the preceding character. As a result, the set of string form contains zero Repetitions of a character and repeats a character more and more until we reach to the length of the TC (max length). In other words, if we have the following regular expression, the set of string will be as follows:

Consider that the length of Test Case = 6.

RE: ab*

SSF: {a, ab, abb, abbb, abbbb, abbbbbb}.

As we noticed in the previous example, the proposed technique converts the RE to another equal form, which is a SSF to facilitate testing RE predicate and using it in GA, as we will see later in this chapter. It worth mentioning that all the strings that belong to the SSF also belong to the language of regular expression, and it is also acceptable in the FSA of the RE.

3.3 Genetic Algorithm Stage

Input: SSF.

Output: The TC for testing RE predicate.

As illustrated in chapter 2, GA contains a set of steps. In this section, we mainly focus on the evaluation step of GA, in order to present our proposed fitness function that we used in the two cases of testing regular expression that we previously discussed. We implemented the GA in Continuous GA Form (CF) and in the Binary GA Form (BF).

Evaluation Step in GA:

This step is used to evaluate the chromosomes, using fitness function, which is a function used to evaluate the degree of goodness of the chromosomes. The fitness function that we used in our implementation for GA is OED, which was illustrated in Chapter 2.

In our proposed technique, we customize the OED such that the output from the Preprocessing Regular Expression stage that is SSF used as input for the OED fitness function (Genetic Algorithm Stage).

3.3.1 Regular Expression that contains OR Operator

In GA stage for OR Operator, we will use the proposed OED to find the costs value to the Chromosomes. The OED as we illustrated used to calculate the cost (differences) between two string that are TC and String. Therefore we apply the OED for each string in the set string form (SSF) that result from " Preprocessing Regular Expression Phase " with the TC in order to find the fitness value for each pair (each string in SSF and TC) , and then find the minimum value of the costs between all them.

Let $SSF = \{S1, S2\}$, $STR = S_n$, and $COST$

Then find: $OED(S1, S_n)$, $OED(S2, S_n)$

After that find: $COST = \text{MiN}(OED(S1, S_n), OED(S2, S_n))$

Where $SSF =$ set string form.

$STR =$ Test Case (string).

$COST =$ the cost of matching RE with string.

Example:

$RE = a(alb)a$ and $STR = aba$.

$SSF = \{aba, aaa\}$

$OED(aaa, aba) = 32.75$

$OED(aba, aba) = 0$

$COST = \text{MiN}(OED(S1, S_n), OED(S2, S_n))$

$COST = \text{MiN}(32.75, 0)$.

Then $COST = 0$.

3.3.2 Regular Expression that contains Repetition Operator

In RE which include Repetition Operator , we apply OED for each String SSF that result from "Preprocessing Regular Expression Phase "with the TC , in order find the cost for each pair, and then find the minimum value of the costs between all pairs.

Let $SSF = \{S1, S2, S3 \dots, Si\}$, $STR = Sn$, and COST

Calculate: $OED(S1, Sn), OED(S2, Sn), OED(S3, Sn) \dots OED(Si, Sn)$

Then find: $COST = \text{MiN}(OED(S1, Sn), OED(S2, Sn), OED(S3, Sn) \dots OED(Si, Sn))$.

Where $SSF =$ set string form.

$STR =$ test case (string).

$COST =$ the cost of matching RE with string.

Example:

$RE = ab^*$ and $STR = abbbb$.

$SSF = \{a, ab, abb, abbb, abbbb\}$

$OED(a, abbbb) = 512$

$OED(ab, abbbb) = 384$

$OED(abb, abbbb) = 256$

$OED(abbb, abbbb) = 128$

$OED(abbbb, abbbb) = 0$

$COST = \text{MiN}(OED(S1, Sn), OED(S2, Sn), OED(S3, Sn) \dots OED(Si, Sn))$.

$COST = \text{MiN}(512, 384, 256, 128, 0)$.

Then $COST = 0$.

3.4 Case Study

In this section, we present two cases studies, one for OR Operator and second for Repetitions Operator. There are two Case study Case 1 for OR Operator predicate as shown in Figure 3-2, and Case 2 for Repetitions Operator as we observe in Figure 3-3. We will apply our proposed method in these two programs.

Case1:

```
String STR;
Integer C=0;
IF (STR=aa|b)
{
  C=C+1
}
```

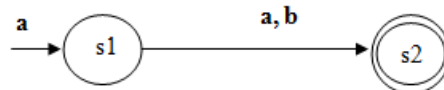
Figure 3-2: Subprogram1.

The proposed methodology applied to Program 1 as Follow:

- Traverse Program1 inorder finding the RE predicate P1.

```
If (STR=a (alb))
{
  C=C+1
}
```

- Using the branch testing technique to ensure the execution of RE predicate at least one.
- Construct the FSA that relates to the RE predicate.



- Preprocessing RE.
SSF = {ab, aa}
- Apply GA using preprocessed RE and proposed OED.

$$\text{SSF} = \{ab, aa\}$$

Let chosen individual in GA in specified generation = ac.

Then, calculate the fitness value using the proposed OED.

$$\text{OED}(ac, ab) = 32.7500$$

$$\text{OED}(ac, aa) = 33.5000$$

$$\text{COST} = \text{MIN}(\text{OED}(ac, ab), \text{OED}(ac, aa))$$

$$\text{COST} = \text{MIN}(32.7500, 33.5000).$$

However, COST Not equal 0, then continuo to followed generation.

In the following generation Let chosen individual in GA = ab

Then calculate the fitness value using the proposed OED.

$$\text{OED}(ab, ab) = 0$$

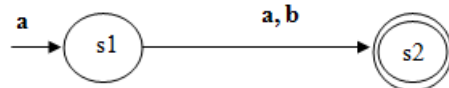
$$\text{OED}(ab, aa) = 32.7500$$

$$\text{COST} = \text{MIN}(\text{OED}(ab, ab), \text{OED}(ab, aa))$$

$$\text{COST} = \text{MIN}(0, 32.7500).$$

Then COST = zero, the TC that execute P1 is ab.

- Ensure that TC belongs to the RE using FSA



TC= ab

Traverse FSA using TC =ab, and the TC reached to the final state.

- Using TC, STR = TC and then execute P1 in Program1.

Case2:

```

String STR;
Integer c=0;
If (STR=aa*b*)
{
    c=c+1
}

```

Figure 3-3: Subprogram 2.

The proposed methodology applied to Program 2 as Follow:

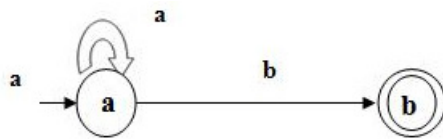
- Traverse Program2 inorder finding the RE predicate P1.

```

If (STR=aa*b)
{
    c=c+1
}

```

- Using the branch testing technique to ensure the execution of RE predicate at least one.
- Construct the FSA that relates to the RE predicate.



- Preprocessing RE regarded to the Max length of the TC.

$aa*b = \{ab, aab, aaab, aaaab, aaaaab\}$

- Apply GA using preprocessed RE and proposed OED.

SSF= {ab, aab, aaab, aaaab, aaaaab}

Let candidate individual in GA in specified generation to be = aaaab

Then, calculate the fitness value using the proposed OED.

$$\text{OED}(\text{aaaab}, \text{ab}) = 384$$

$$\text{OED}(\text{aaaab}, \text{aab}) = 256$$

$$\text{OED}(\text{aaaab}, \text{aaab}) = 128$$

$$\text{OED}(\text{aaaab}, \text{aaaab}) = 0$$

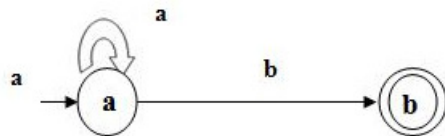
$$\text{OED}(\text{aaaab}, \text{aaaaab}) = 128$$

$$\text{COST} = \text{MIN}(384, 256, 128, 0, 128) = 0$$

COST = 0 the TC where the fitness value equal 0 is founded.

$$\text{TC} = \text{aaaab}$$

- Ensure that TC belongs to the RE using FSA



$$\text{TC} = \text{aaaab}$$

Traverse FSA using TC = aaaab, and the TC reached to the final state.

- Using TC, STR = TC and then execute P1 in Program 2.

CHAPTER 4

EXPERIMENTS AND RESULTS

4.1 Introduction

In chapter 4, we present our implementation and experiments that simulate our proposed methodology. The implementation includes Preprocessing Regular Expression Stage (Testing Regular Expression Techniques) and Genetic Algorithm Stage (The proposed fitness functions). As we mentioned previously, we implement the GA in two forms: Continuous GA and Binary GA. The chapter also presents a comparison between the Binary and Continuous implementation followed by the experimental result and analysis.

4.2 Experiments Environment

The experiments are conducted on Matlab 7.1. Using Genetic Algorithm, Direct Search Toolbox, and M-files concept. We run the experiments on a PC that run under windows operating system, with the following system specifications:

Manufacturer: DELL.

Processor: Intel(R) Core (TM) 2 Dou CPU T6400 @ 2.00 GZ.

Memory (RAM): 2:00 GZ.

System Type: 32 bit Operating system.

System Type: 32 bit Operating system.

4.3 Input Domain

The domain that we used in the implementation was [65,122]. The domain combined two ranges the ASCII codes for the small letters that is [97,122] and , the ASCII codes for the capital letters , which is [65,90]. The sub range from [66, 96] is represents some other printable characters e.g. (_).

4.4 Continuous and Binary Form

The proposed technique was implementing in two forms of GA: the Binary Form (BF) and Continuous Form (CF).

Binary Form that is a GA, which deals with population as, set bits. In this form the chromosomes contains just zero and one bits therefore; it should be exist an encoding and decoding methods to transform individuals from binary to decimal values and vice versa. This GA usually used when the variables are naturally quantized and not too large bits needed like variable used to full machine precision (floating- point) which need many numbers of bits to represent it (Haupt, 2004).

Continuous GA represents chromosomes as decimal value that means we used it when the number of variables is large and when the application care about accuracy of vales such that floating point. In this GA we need not to use encoding and decoding methods. The main advantage of CF that is requiring less storage than the Binary GA, also CF is inherently faster than the Binary GA, because the chromosomes do not have to be decoded prior to the evaluation of the cost function (Haupt, 2004).

4.5 Experiments Design

The Experiments include Continuous and Binary Form. At each form we have conducted 7 different experiments, which covers some possible scenario using both Repetition (*) and OR (|) operator.

The seven predicates are using the proposed fitness function to match a String with a Regular Expression that contains:

1. OR Operator (|) at the start of RE as shown in Figure 4-1.

```

If (STR = (a|b)babba)
{
  // Execute Target
}
```

Figure 4-1: Regular Expression Contains OR operator at the start.

2. OR Operator (|) at the middle of RE shown in Figure 4-2.

```

If (STR = bab(a|b)ba)
{
  // Execute Target
}
```

Figure 4-2: Regular Expression Contains OR operator at the middle.

3. OR Operator (|) at the end of RE as shown in Figure 4-3.

```

If (STR = babba(a|b))
{
  // Execute Target
}
```

Figure 4-3: Regular Expression Contains OR operator at the middle.

4. One Repetitions Operator (*) as shown in Figure 4-4.

```
If (STR = ba*b)
{
  // Execute Target
}
```

Figure4-4: Regular Expression contains one Closure Operator.

5. Two Repetitions Operator (*) as shown in Figure 4-5.

```
If (STR= ba*b*)
{
  // Execute Target
}
```

Figure 4-5: Regular Expression contains two Closure Operator.

6. Three Repetitions Operator (*) as shown in Figure 4-6.

```
If (STR = ba*b*a*)
{
  // Execute Target
}
```

Figure 4-6: Expression contains three Closure Operator.

7. A Closure Operator (*) and OR Operator (|) as shown in Figure 4-7.

```
If (STR = ba*(b|a) a)
{
  // Execute Target
}
```

Figure 4-7: Expression contains Closure Operator and OR Operator.

4.6 GA Parameters Setup

In using GA, the values of GA parameters must be set up before hand. Selection of these values was subject to trial-and-error practice. Initially, GA parameters are set to the values that are mostly used and considered promising in the previous related works. Gradually, based on the feedback from one experiment, parameters are refined in subsequent experiments.

The followings are the parameters that we use in the implementation of GA:

- **Population Size:**

It is used to determine the size of the population at each generation. Increasing the population size enables the GA to search more points and thereby obtain better results. However, the larger the population size, the longer the genetic algorithm takes to compute each generation. The population size in the implementation in two forms equal 100 individuals. The 100 individual was suggested depend on previous work in GA fields (Alshraideh, 2007). The 1000 individual was tested as shown in Table 4-1; in this case the TC found with less Required Time and Generation Numbers, however, it needs high computation and more resources to use. In addition, The 30 population size is tested as shown in Table 4-2, and found that, the TC is found with more Required Time and Number of Generations than in the 100 population size.

Table 4-1: Sample Experiments for population size=1000 in Program1.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	ababba	5	2.0592	131.3	minimum fitness limit reached
E2	bbabba	12	4.2900	38.42	minimum fitness limit reached

Table 4-2: Sample Experiments for population size=30 in Program1.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	ababba	1775	56.6908	33.2	minimum fitness limit reached
E2	bbabba	2621	98.5926	33.33	minimum fitness limit reached

- **Initial Population Range:**

The Genetic Algorithm usually produces a random initial population using creation function. The initial range that we used in our implementation is [65,122] for CF and (0, 1) for BF.

- **Selection Method:**

The selection function that we adopted in the implementation of BF and CF is stochastic uniform which described in chapter 2.

- **Crossover Method:**

The Single point crossover method is used in our implementation that discussed previously in chapter 2. We tested the two point crossover method as shown in Table and found that the Single and the Two point Crossover approximately need the same Required Time and Number of Generations.

Table 4-3: Sample Experiments for Program1 using two point Crossover.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	ababba	377	18.6889	32.98	minimum fitness limit reached
E2	bbabba	320	11.7001	33.77	minimum fitness limit reached

- **Crossover Probability:**

It is used to specify the fraction of the next generation, other than elite children, that are produced by crossover. In our implementation the Probability that chosen to use is 0.8 which is the default Probability of Crossover method in Matlab 7.1.

- **Mutation Method:**

In our implementation in each of BF and CF we use a Uniform mutation method that we describe it previously. We tested other Mutation Methods e.g. Gaussian mutation as shown in Table 4-4 , and found that it needed large Number of Generation compare to Uniform mutation and in some experiments the generations reach up to 10000 (the second termination reason) without find the solution (TC).

Table 4-4: Sample Experiments for Program1 using Gaussian mutation.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	Qbabba	10000	785.3558	195.8	reached 10000 Generations

- **Mutation Rate:**

It is a rate, which is used in Mutation Method as probability Rate for the mutation process. In the implementation we suggest to use the default value of Rate which is 0.1.

- **Number of Variables:**

A parameter represents the number of genes in the chromosome. In our method, it equals the length of string in SSF. In the first three experiments that related to OR Operator it, equal 6, while in the rest of the experiments, which used Repetitions Operator it,'s randomly generated from 1 to 6.

- **Chromosome:**

It consists from a set of genes. In our implementation, these genes are the ASCII code of characters in CF; in the other hand they are the bits 0 or 1, in the BF, that represent the underlying Binary representation for characters. The length of the chromosome depends on the number of variable parameters.

- **Stopping Criteria Parameters:**

The Stopping Criteria in our implementation are:

1. Finding the solution such that the fitness value equal zero.
2. Maximum number of generations equal to 10000.

In below Table 4-5 summarizes the parameters of our implementation and its values.

Table 4-5: The Experiments' parameters and values.

No	Parameter	Continuous	Binary
1	Population Size	100	100
2	Initial Population Range	[65,122]	(0,1)
3	Selection Method	Stochastic Uniform	Stochastic Uniform
4	Crossover Method	Single Point	Single Point
5	Crossover probability	0.8	0.8
6	Mutation Method	Uniform	Uniform
7	Mutation Probability	0.1	0.1
8	Number of Variables (genes)	<ul style="list-style-type: none"> • 6 characters in or operator. • Randomly generated in repetition operator. 	<ul style="list-style-type: none"> • 42 bits in or operator. • Randomly generated in repetition operator
9	Stopping Criteria	<ul style="list-style-type: none"> • Cost = 0 • Max Gen. No = 10000 	<ul style="list-style-type: none"> • Cost = 0 • Max Gen. No = 10000

These parameters were chosen to be suitable for all experiments in the Continuous and Binary GA.

4.7 Metric to record

The following pieces of information are very important to be noticed in every experiment for each generation, in order to assess the experiment outcomes.

- Generations Number: The numbers of generations needed to find the solution.
- Time: The time needed to find the solution.
- Best $f(x)$: Best fitness function value (the minimum one) where in the experiments all the fitness value equal to 0 in two forms.
- Mean $f(x)$: Mean fitness function value.
- GA Termination Reason: Usually the GA terminates when we find the solution, but also we can terminate it manually using stop button.

4.8 Experiments

We conducted the following programs in each form: Continuous and Binary for 20 times , and we noticed the five metrics that we discussed in the previous section,(The number of Generations, Required Time, Best $f(x)$, Mean $f(x)$, and GA Termination Reason for each time). (The obtained result for 20 experiments for each form is shown in detail in Appendix A).

In the seven programs, we will convert the RE to a set of strings form (SSF). Our aim in these experiments is to find the Test Case that belongs to SSF and execute the RE predicates e.g. the predicate is covered.

In each of the following programs we will present sample experiments about each form, and describe them in details, after that we show two diagrams for each form, that illustrate the flow of sample experiments from the starting point until it reaches the end point, where the cost equal 0 such that the Best $f(x)$ for all experiments in two forms where equal 0. In the upper part of these diagrams, the generation number is located on the x-axis and the fitness values on the y-axis. Also, in the lower part of these diagrams, there are two axes; the x-axis that represents the number of variables in best individual, and the y-axis that represents the current representation of best individual. Finally, we observed and noticed two charts that display the results of 20 experiments of the programs in the CF and BF. The x-axis in these two charts represents the 20 experiments from E1 to E20 and the Average case; while, the y- axis represents the generation number (black column) and the required time (gray column) for the experiments.

Program 1:

Testing a Regular Expression that includes OR Operator (|) at the start position of RE using the proposed OED fitness function (Predicate 1). In Program 1, the Regular Expression is RE = albbabba. The SSF is {ababba, bbabba}. The discovered TC should belong to SSF and execute Predicate1 which is shown in Figure 4-1 in section 4.5.

Table 4-6 describes the obtained results of sample experiments of CF Program 1, while Table 4-7 shows the obtained result of E14, E3, E8, and E20 form BF of Program 1.

Table 4-6: Sample Experiments and Average Case for CF of Program 1.

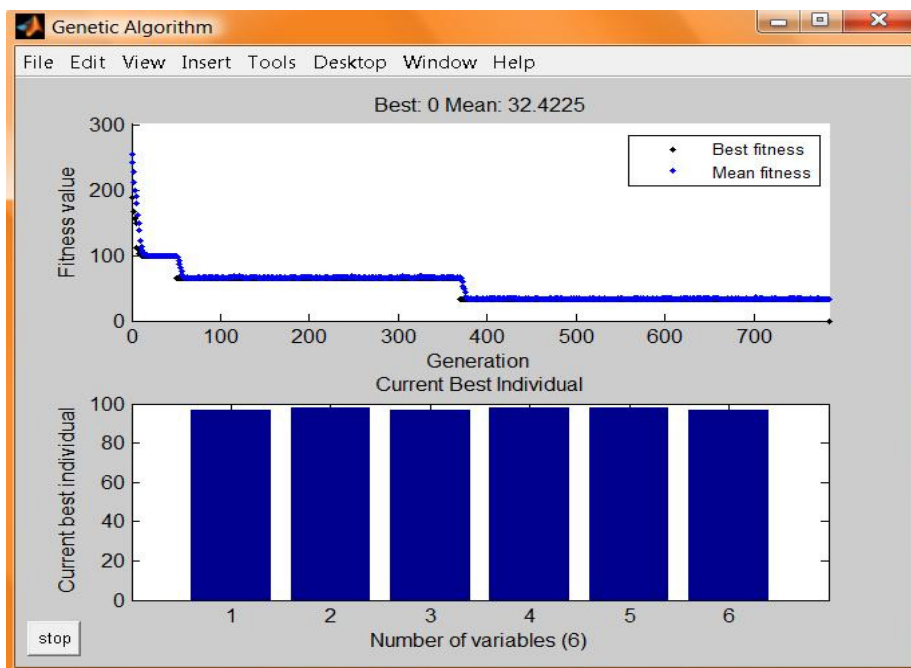
Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	bbabba	649	19.3129	33.68	minimum fitness limit reached
E3	bbabba	114	3.7284	33.41	minimum fitness limit reached
E8	ababba	682	27.4562	33.66	minimum fitness limit reached
E18	ababba	116	4.3524	34.31	minimum fitness limit reached
Average	-----	524	19.3371	33.5235	-----

Table 4-6 gives a brief description for the sample of experiments for CF of Program1. For example, in experiment 1 (E1), the founded test case "bbabba", which is used to excute predicate1, is discoverd at Generation number 649 during 19.3129 seconds, at fitness value equal 0, and with avarege of fitness value equal 33.68 . Finally, the termrrnation reaseon is that the minimum fitness limit is reached. Avarege for the 20 expriments was calculated and the results were the generation number equal 524.05 rounded to 524 , the needed time is 19.3371, and the mean of fitness value is 33.5235.

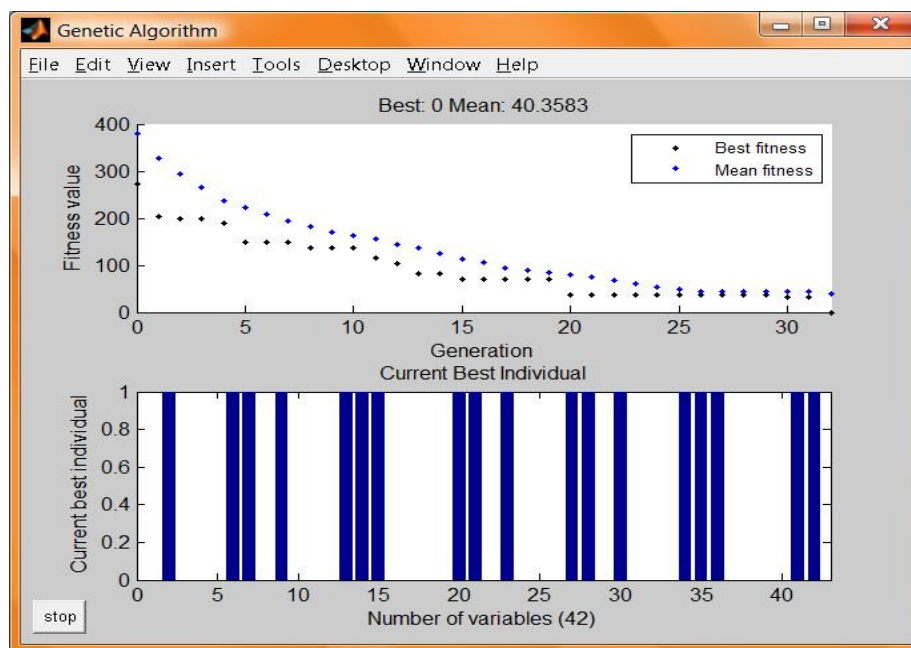
Table 4-7: Sample Experiments and Average Case for BF of Program 1.

Time No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E3	bbabba	85	12.4177	37.15	minimum fitness limit reached
E7	ababba	1219	176.7491	37.2	minimum fitness limit reached
E14	ababba	1271	181.9128	40.59	minimum fitness limit reached
E20	bbabba	34	5.0856	39.09	minimum fitness limit reached
Average	-----	355	51.410125	41.7365	-----

Table 4-7 describes sample experiments of BF for Program 1. For example, in experiment 14 (E14), the founded test case was "ababba", which is used to excute predicate1, is discoverd at Generation number 1271 during 181.9128 seconds at fitness value equal 0, and with avarege of fitness value equal 40.59, and finally the termrrnation reaseon is that the minimum fitness limit is reached. Avarege for the 20 expriments was calculated and the results were the generation number equal 355.45 rounded to 355, the needed time is 51.410125, and the mean of fitness value is 41.7365.



(a) Experiment 2 flow



(b) Experiment 10 flow

Figure 4-8: The Flow of Experiments: E2 for CF, and E10 for BF, respectively.

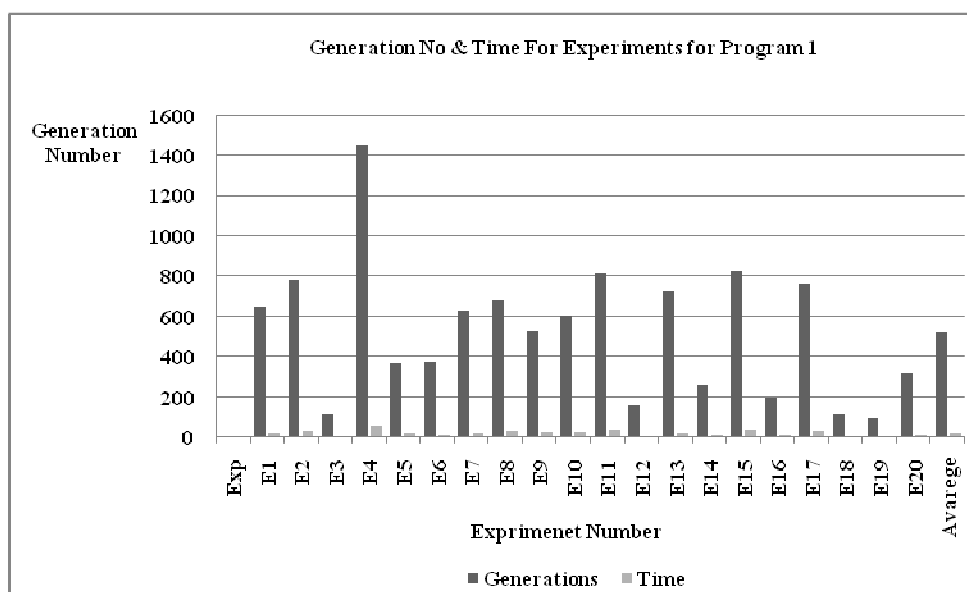
Figure 4-8 shows two Diagrams that explain the flows of the experiments.

Figure 4-8 (a) represents the flow for experiment 2 in Continuous Form. It starts with 188.5 fitness values and is then followed with 166.3, 155.8, 148.3, 111, 109.5, 108, 103.5, 100.5, 98.25, 65.5, and 32.75, until it reaches to 0. The generation is where the zero fitness value found was 784. The lower part of diagrams contains the ASCII codes for the individual, which is selected as the current best individual which is:

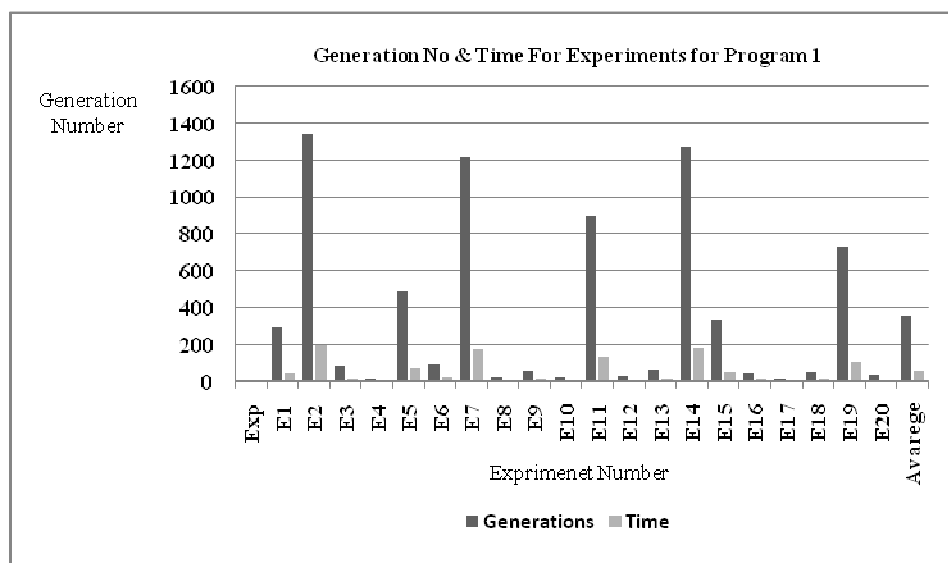
{98, 98, 97, 98, 98, and 97}.

Figure 4-8 (b) explains experiment 10's flow in BF. It began with cost equals 170.8 then decreases to 141.8, 139.5, 129, 93.25, 91, 79, 76.75, 76, 70.75, 70, 68.5, and 67, 65.5, 32.75 and ended with cost equals 0. The generation where the test case found was 22. In the lower part of the diagram there are the ASCII codes for the best individual characters which are:

{ (1 0 0 0 0 1 1), (0 1 0 0 0 1 1),
 (1 0 0 0 0 1 1), (0 1 0 0 0 1 1),
 (0 1 0 0 0 1 1) , (1 0 0 0 0 1 1) }.



(a) Continuous Form



(b) Binary Form

Figure 4-9: Twenty Experiments for CF and BF in Program1

Figure 4-9 shows two charts that display the results of 20 experiments of program1 in the CF and BF. In Figure 4-9 chart (a) we noticed that the generation numbers for experiments are approximately close to each other, on the other hand the generation number in chart (b) have diversified values; therefore they are not close to each other. In

addition, we note from Average cases that the needed generation number to find the test case in BF is less than the generation number in the CF. The required time to find the test case in the BF is greater than the needed time in the CF.

Program 2:

Testing Regular Expression that includes OR Operator (|) at the middle position of RE using Proposed OED fitness function (Predicate 2). In Program 2 the Regular expression under test is RE = babalbba. The SSF is {bababa, babbba}. Predicate as shown in Figure 4-2. In Table 4-8, we show the obtained results of sample CF experiments, which are E4, E11, E17 and E19 from Program 2. Table 4-9 presents the obtained results of sample experiments that are in BF of Program 2.

Table 4-8: Sample experiments and Average Case for CF of Program 2.

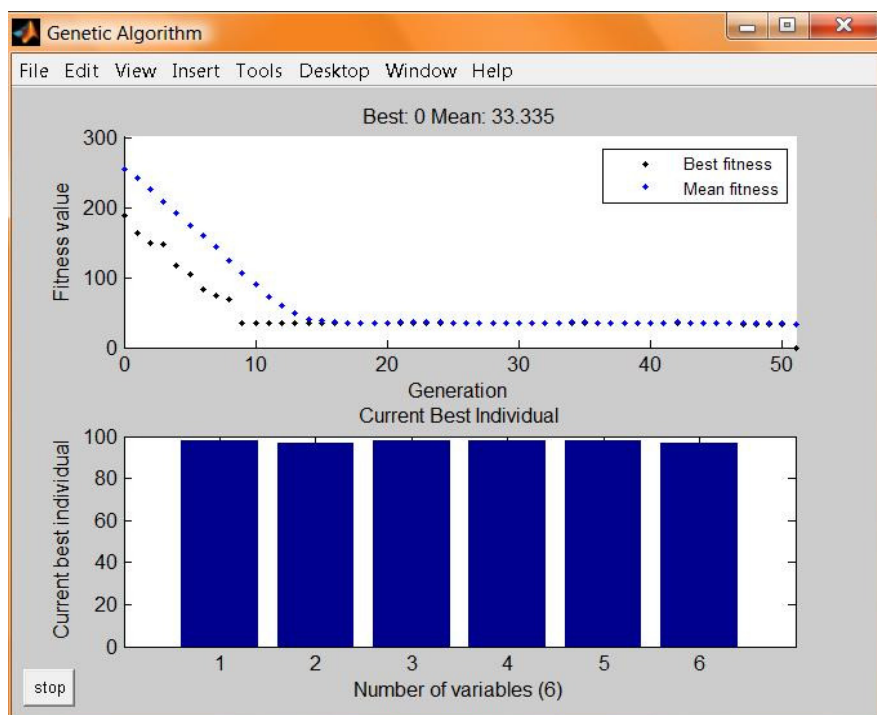
Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E4	babbba	226	8.8453	34.78	minimum fitness limit reached
E17	babbba	1691	73.1021	32.91	minimum fitness limit reached
E11	bababa	676	27.1286	33.25	minimum fitness limit reached
E19	bababa	72	3.0888	35.6	minimum fitness limit reached
Average	-----	449	17.16635	33.6155	

Table 4-8 gives a brief description for the 20 time experiments for CF of Program 2. In experiment 4 (E4) the founded test case "babbba". Discovered at Generation number 226 during 8.8453 seconds at fitness value equal 0 and with average of fitness value equal 34.78. Finally, the termination reason is that the minimum fitness limit reached. The Average case for the 20 experiments was founded and the results were the generation number equal 448.65 rounded to 449, the needed time is 17.16635, and the mean of fitness value is 33.6155 as shown in Table 4-8.

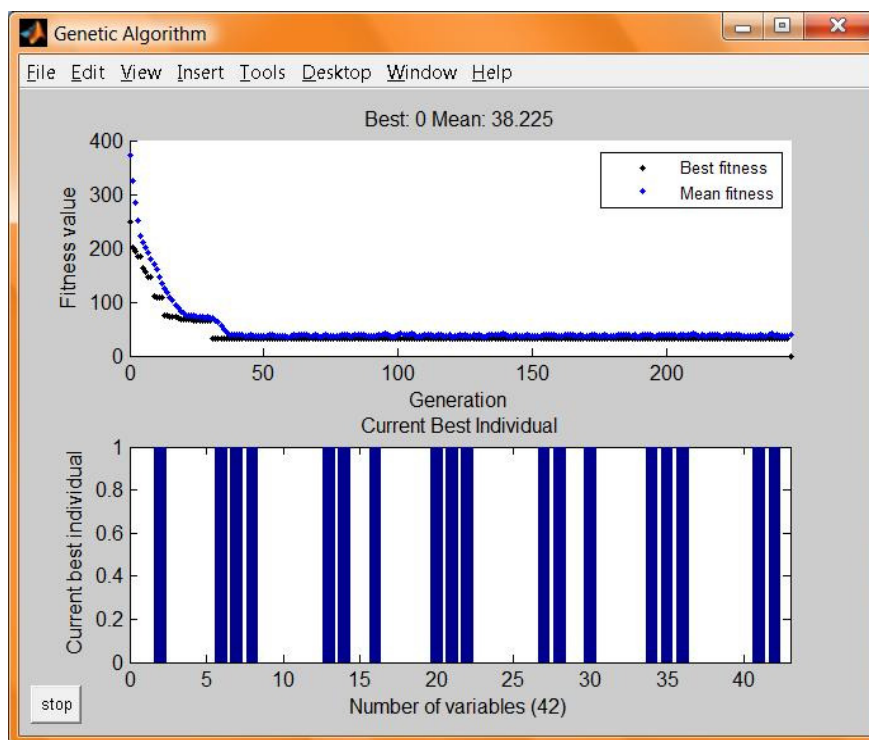
Table 4-9: Sample Experiments and Average Case for BF of Program 2.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E4	babbba	303	43.2903	37.47	minimum fitness limit reached
E7	babbba	439	62.4628	38.47	minimum fitness limit reached
E11	bababa	18	2.808	43.23	minimum fitness limit reached
E15	bababa	71	9.9841	37.32	minimum fitness limit reached
Average	-----	195	27.77832	38.3015	-----

Table 4-9 gives a the results for some sample experiments for BF of Program 2 . one of the sample experiment is experiment 4 (E4) where the founded test case "babbba", discovered at Generation number 303 during 43.2903 seconds at fitness value equal 0 and with average of fitness value equal 37.47. Finally, the termination reason is that the minimum fitness limit reached . The Average for the 20 experiments was calculated and the results were the generation number equal 195.05 rounded to 195 , the needed time is 27.77832, and the mean of fitness value is 38.3015.



(a) Experiment 3 flow



(b) Experiment 10 flow

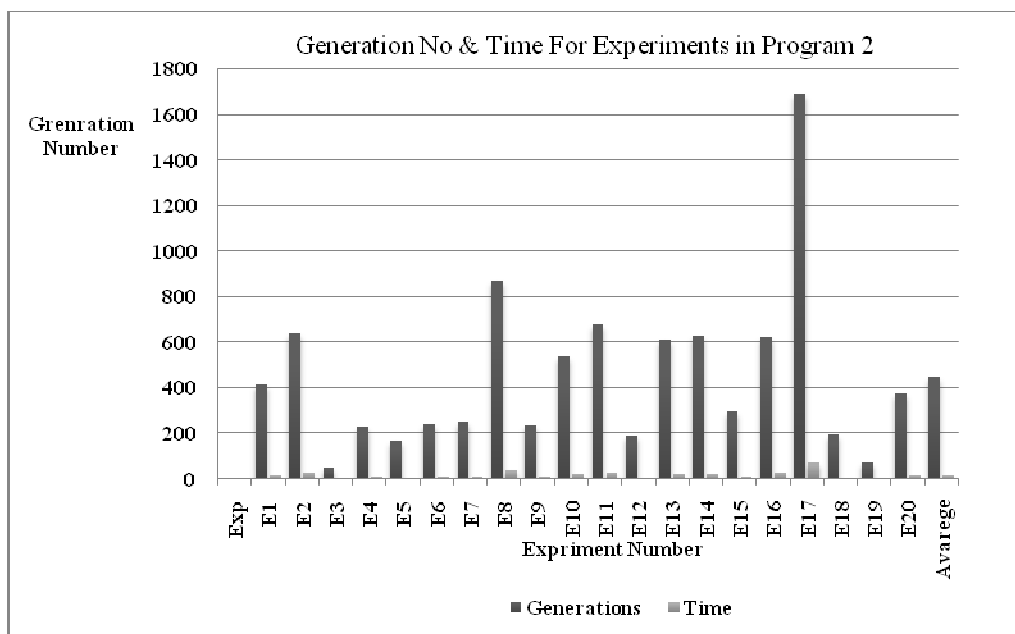
Figure 4-10: The Flow of Experiments: E3 for CF, and E10 for BF , respectively.

Figure 4-10 shows two diagrams that explain the flows of the experiments in Program 2. Figure 4-10 (a) represents the flow for experiment 3 in Continuous Form. It starts with 163. Then 149.3, 147, 117, 105, 82.75, 74.5, 68.5, 34.25, 32.75, until it reaches to 0. The generation where the zero fitness value founded was 51, with test case equals to "babbbba". The lower part of diagrams contains the ASCII codes for the individual that was chosen as the current best individual which is:

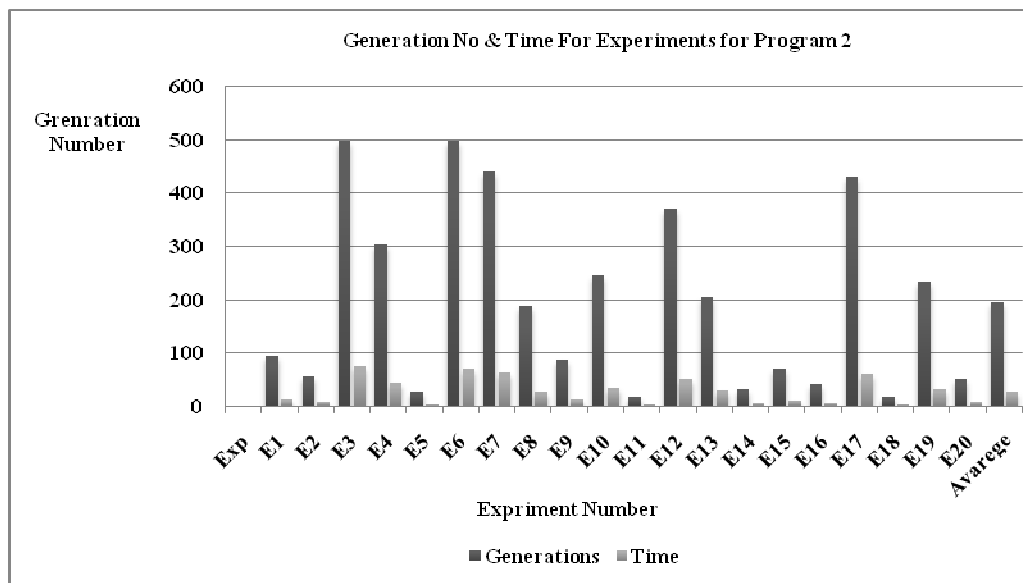
{98, 97, 98, 98, 98, and 97}.

Figure 4-10 (b) explains experiment 10's flow in Binary Form. It began with cost equals 200.5 then decreased to 193, 185.5, 184, 162.5, 156.5, 146.8, 145.3, 111.8, 108.8, 76, 71.5, 70, 67, 65.5, and 32.75 and ended with cost equal 0. The generation where the test case (bababa) found was 246. In the lower part of the diagram, there are the ASCII codes for the best individual characters which are:

{ (0 1 0 0 0 1 1), (1 0 0 0 0 1 1)
 (0 1 0 0 0 1 1), (1 0 0 0 0 1 1)
 (0 1 0 0 0 1 1), (1 0 0 0 0 1 1) }.



(a) Continuous Form



(b) Binary Form

Figure 4-11: Twenty Experiments and Average Case for CF and BF in Program2.

Figure 4-11 shows two charts that display the results of 20 experiments of Program 2 in CF and BF. In Figure 4-11 chart (a), we observed that the generation number for experiments are approximately close to each other except for some extreme values like

E17. On the other hand, the generation number in Figure 4-11 chart (b) has diversified values; therefore, they are not close to each other. In addition, we note from the Average case for the experiments that the needed generation number to find the test case in BF of Program 2 is less than the generation number in the CF, and the needed time to find the test case in the BF is greater than the needed time in the CF.

Program 3:

Testing Regular Expression that includes OR Operator (|) at the end position of RE using the proposed OED fitness function (Predicate 3). In Program 3 the Regular expression is RE = babbaalb. The SSF for this program is {babbaa, babbab}. Figure 4-3 shows Predicate 3. In Table 4-10, we describe the results of sample CF experiments which Program 3. Table 4-11 presents the obtained results related to BF of Program 3.

Table 4-10: Sample Experiments and Average Case for CF of Program 3.

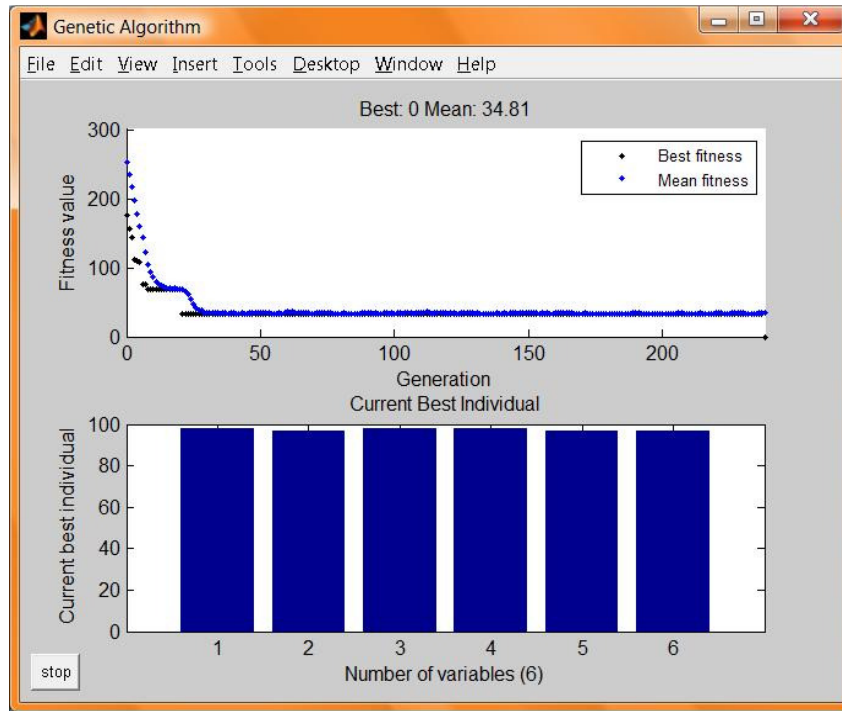
Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E2	babbab	24	2.4804	34.36	minimum fitness limit reached
E8	babbab	7	0.4836	114.6	minimum fitness limit reached
E14	babbaa	382	11.3881	33.06	minimum fitness limit reached
E17	babbab	140	5.2884	32.92	minimum fitness limit reached
Average	-----	206	7.41629	38.391	-----

Table 4-10 summarizes sample of CF experiments for Program 3. As we can see in experiment8 (E8) the founded test case "babbab" that used to execute predicate 3 which aroused at Generation number 7 during 0.4836 seconds with fitness value equals zero and with average of fitness value equal 114.6. The experiment terminated due to the minimum fitness limit reached. In Table 4-10 the Average for the 20 experiments were calculated and the generation number equal 206.4 rounded to 206, the needed time is 7.41629, and the mean of fitness value is 38.391.

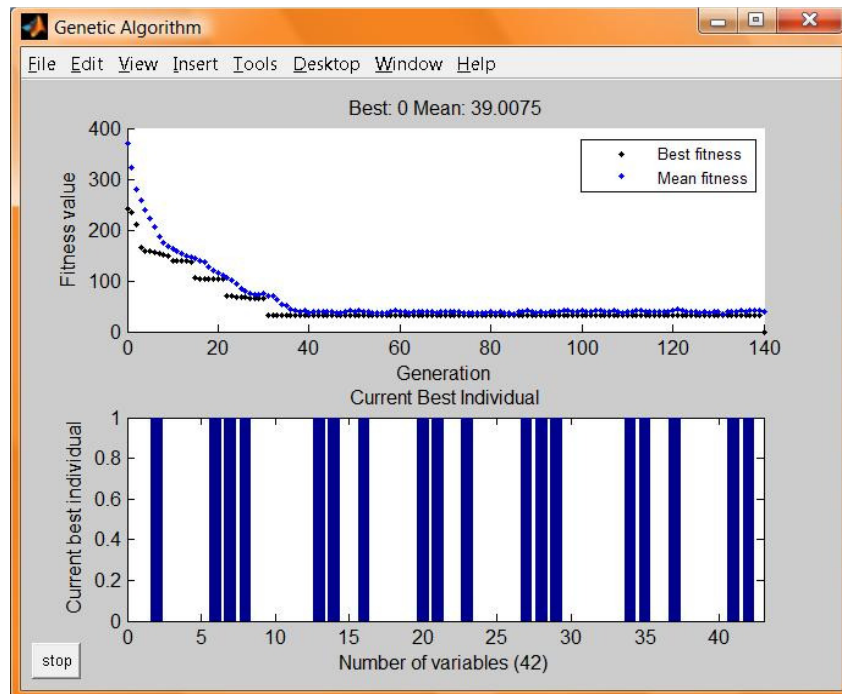
Table 4-11: Sample Experiments and Average Case for BF of Program 3.

Time No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	babbaa	419	52.9623	35.21	minimum fitness limit reached
E5	babbab	390	55.5208	36.3	minimum fitness limit reached
E16	babbab	344	45.8643	38.4	minimum fitness limit reached
E20	babbab	108	14.4145	38.8	minimum fitness limit reached
Average	-----	154	21.256695	40.633	-----

Table 4-11 gives a brief description for sample experiments for Program 3 but in BF. In experiment 5 (E5), the founded test case was "babbab", discovered at generation number 390 with needed time equal to 55.5208 seconds at fitness value equal 0 and with average of fitness value equal 36.3, E5 terminated due to the minimum fitness limit reached. The Average for the twenty experiments the generation number was equal to 154.1 rounded to 154, the needed time is 21.256695, and the mean of fitness value is 40.633. The charts below display some of sample experiments and their flows.



(a) Experiment 4 flow



(b) Experiment 6 flow

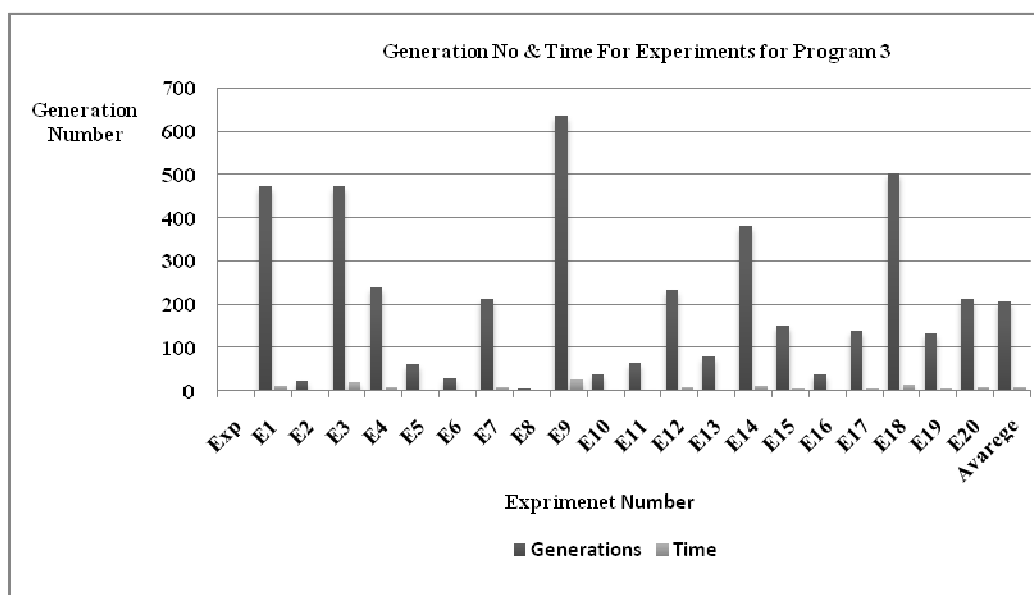
Figure 4-12: The Flow of Experiments: E4 for CF, and E6 for BF, respectively.

As we can observe in Figure 4-12, two diagrams illustrate the experiments and their flows. Figure 4-12 (a) presents the flow for experiment 4 in Continuous Form. It starts with 156.5 fitness value and decreases until it reaches zero. The generation where the test case (babbaa) was discovered equal to 238. In the lower part of diagrams, the ASCII codes for current best individual appeared, which are:

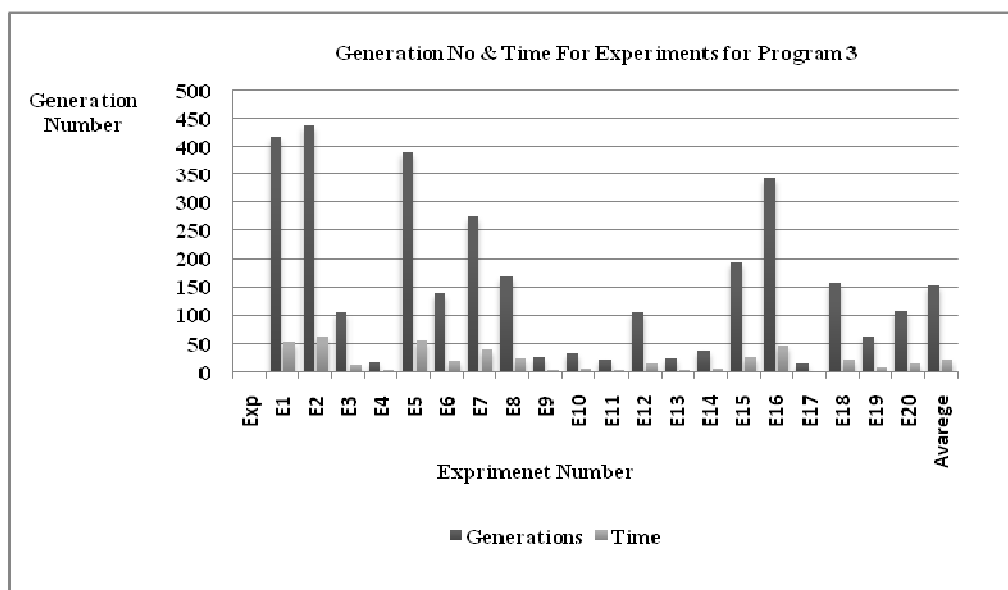
{98, 97, 98, 98, 97, and 97}.

Figure 4-12 (b) explains the flow of experiment 6 in Binary Form. It began with cost that equals 233.5 then decreased until it reached to fitness value equal 0. The generation where the test case (babbab) found was 140. In the lower part of the diagram there is the binary representation for the best individual characters which is:

{ (0 1 0 0 0 1 1), (1 0 0 0 0 1 1)
 (0 1 0 0 0 1 1), (0 1 0 0 0 1 1)
 (1 0 0 0 0 1 1), (0 1 0 0 0 1 1) }.



(a) Continuous Form



(b) Binary Form

Figure 4-13: Twenty Experiments and Average Case for CF and BF in Program3.

In Figure 4-13, there are two charts that display the results of 20 experiments of Program 3 the CF in Figure 4-13 (a), and the BF in Figure 4-13 (b). In chart (a), the generation number for experiments are approximately close to each other except for some extreme values such as E9. On the other hand, the generation number in chart (b)

has diversified values; therefore, they are not close to each other. In addition, we note from the Average of the 20 experiments that the needed generation number to find the test case in BF of Program 3 is less than the generation number in the CF, and the needed time to find the test case in the BF is greater than the needed time in the CF.

Program 4:

Testing Regular Expression that includes One Repetitions Operator (*) using the proposed OED fitness function (Predicate 4). In Program 4 the Regular expression is $RE = ba^*b$. The SSF is {bb, bab, baab, baaab, baaaab}. The founded Test Case should belong to SSF, therefore executes predicate 4 that is shown in Figure 4-4. Table 4-12 summarizes the sample of obtained results of 20 CF experiments of Program 4, which have been run 20 times and had different results., while Table 4-13 shows the obtained results related to BF of Program 4.

Table 4-12: Sample Experiments and Average Case for CF of Program 4.

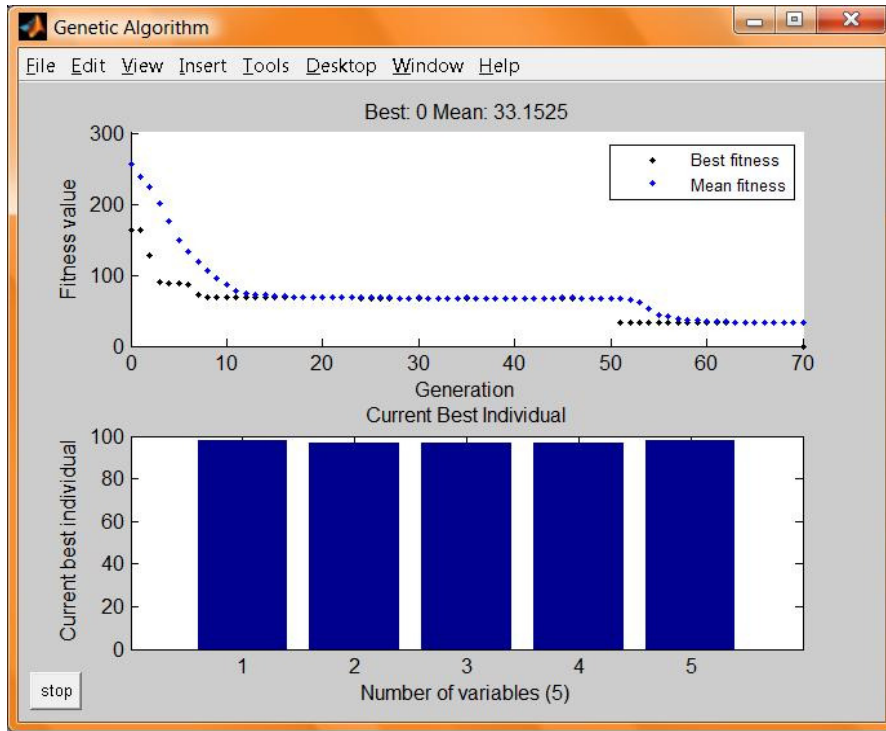
Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E2	baaab	204	4.3680	33.7	minimum fitness limit reached
E6	baaab	723	31.2002	33.83	minimum fitness limit reached
E13	bab	48	1.5600	33.16	minimum fitness limit reached
E17	bab	1056	35.7866	32.51	minimum fitness limit reached
Average	-----	478	17.18818	33.3735	-----

In Table 4-12 we summarize the result of sample experiments for CF of Program 4. In experiment 2 (E2) the found TC was "baaab" that used to execute predicate 4 discovered at Generation number 204 during 4.3680 seconds with cost equal zero and with average of fitness value equal 33.7. The last row in Table 4-12 represent The Average for the 20 experiments were the generation number equal 478.25 rounded to 478, the needed time is 17.18818, and the mean of fitness value is 33.3735.

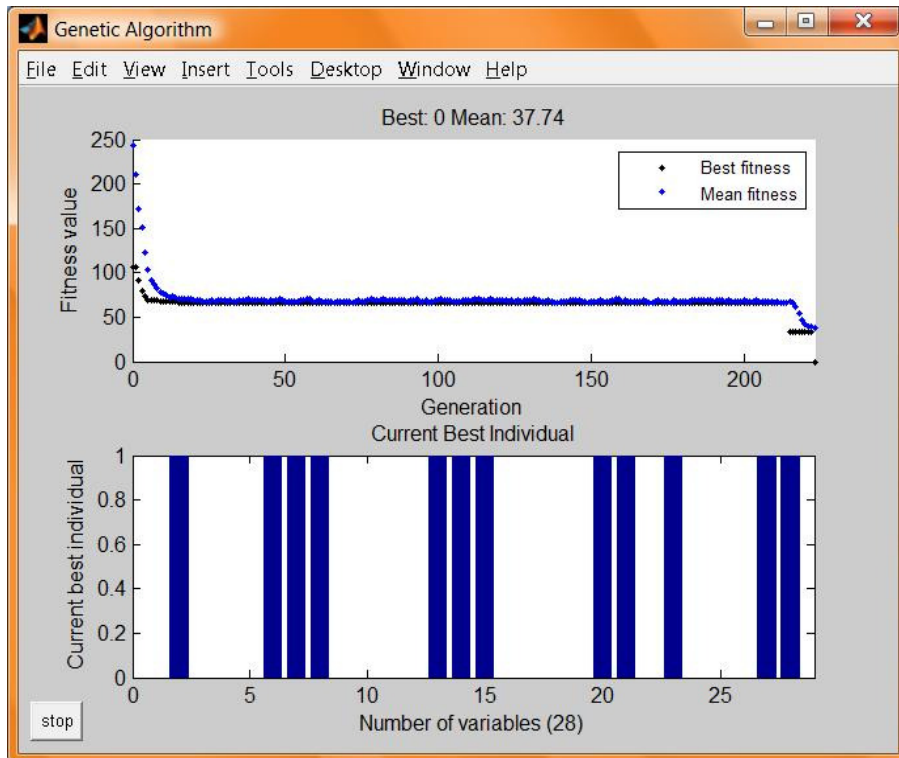
Table 4-13: Sample Experiments and Average Case for BF of Program 4.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E2	bb	56	5.8188	34.23	minimum fitness limit reached
E5	baaab	25	5.8968	45.55	minimum fitness limit reached
E13	baab	126	25.1318	34.68	minimum fitness limit reached
E20	bab	162	26.0678	36.38	minimum fitness limit reached
Average	-----	240	50.12156	42.6645	-----

Table 4-13 gives a brief description for sample experiments for Program 4 in BF. In experiment2 (E2) the founded test case was "bb", discovered at Generation number 56 with needed time equal to 5.8188 seconds at fitness value equal 0 and with average of fitness value equal 34.23, and termination reason is that the minimum fitness limit reached. Average for the 20 experiments the generation number was equal to 240.4 rounded to 240, the needed time is 50.12156, and the mean of fitness value is 42.6645. The following charts display some of chosen experiments and their flows.



(a) Experiment 8 flow.



(b) Experiment 10 flow.

Figure 4-14: The Flow of Experiments: E8 for CF, and E10 for BF, respectively.

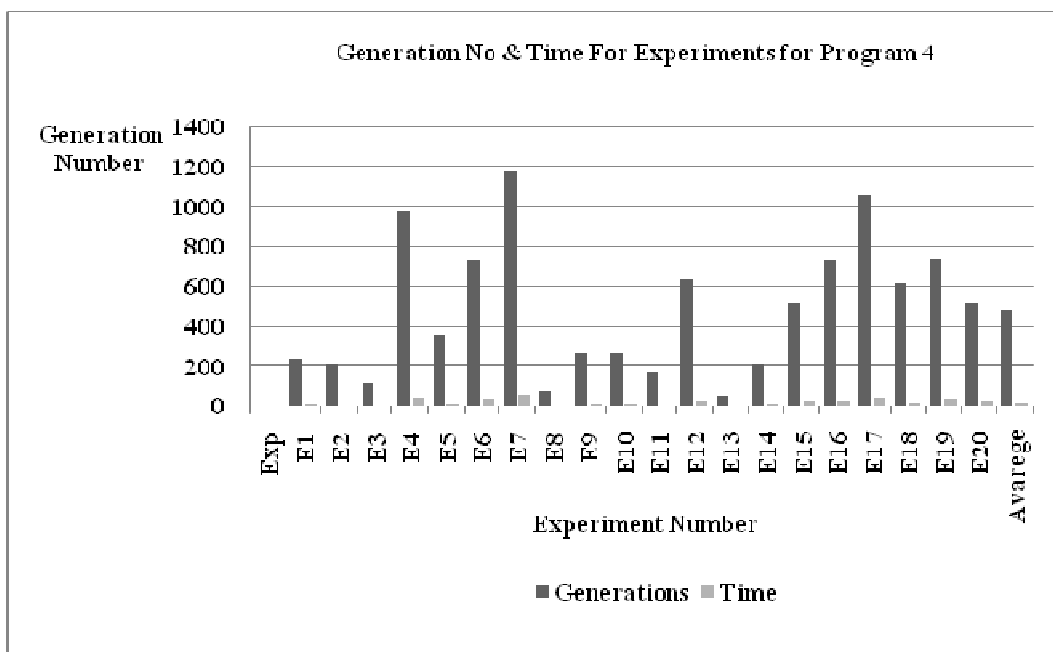
In Figure 4-14, we can see two diagrams that illustrate the experiments and their flows.

Figure 4-14 (a) presents the flow for experiment 8 in Continuous Form. It starts with 163.3 fitness value and then 127.5, 90.25, 88.75, 87.25, 72.25, 68.5, 67, and 32.75, until reach to 0. The generation where the test case (baaab) discovered was 70. In The lower part of diagrams the ASCII codes for current best individual is appeared that are:

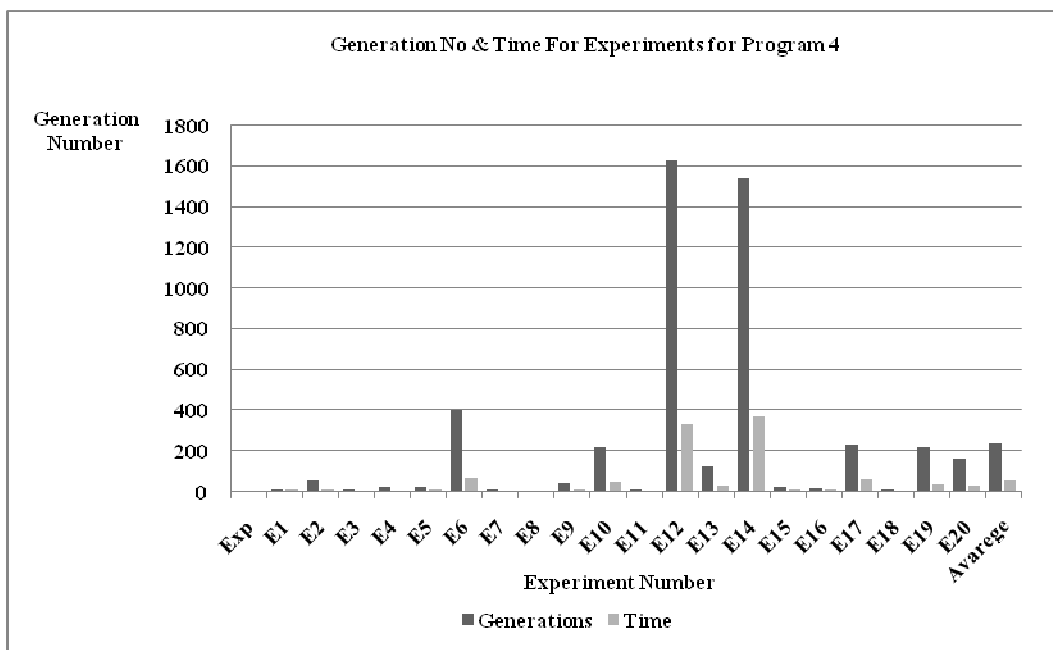
{98, 97, 97, 97, and 98}.

Figure 4-14 (b) explains flow of experiment 10 in Binary Form. It began with cost equals 106.8 then decreases to 91.75, 79, 73.75, 68.5, 67, and 66.25, 65.5, 32.75 and ended with cost equal 0. The generation where the test case (baab) found was 223. In the lower part of the diagram there are the ASCII codes for the best individual characters which are:

{ (0 1 0 0 0 1 1), (1 0 0 0 0 1 1)
 (1 0 0 0 0 1 1), (0 1 0 0 0 1 1) }.



(a) Continuous Form



(b) Binary Form

Figure 4-15: Twenty Experiments for CF and BF in Program 4.

Figure 4-15 shows two charts that display the results of 20 experiments of Program 4.

In Figure 4-15 chart (a), we observed that the generation number for experiments are approximately close to each other except some extreme value such as E13 and E7. On

the other hand, the generation numbers in chart (b) have diversified values; therefore, they are not close to each other. In addition, we note from the Average of the 20 experiments that the needed generation number to find the test case in BF of Program 4 is less than the generation number in the CF. The needed time to find the test case in the BF is greater than the needed time in the CF.

Program 5:

Testing Regular Expression that includes Two Repetitions Operator (*) using the proposed OED fitness function (Predicate 5). In Program 5 the Regular expression is $RE = ba*b^*$. The SSF is {b, ba, baa, baaa, baaaa, bb, bbb, bbbb, bbbbb, bbbbbb, bab, baabb, baaabb, baabbb, babbbb, baaaab}. Predicate five which shown in Figure 4-5.

In Table 4-14, we preview the obtained results of sample CF experiments, which are E4, E8, E13 and E17 from Program 5. Table 4-15 presents the obtained results of sample experiments that are in BF of Program 5.

Table 4-14: Sample Experiments and Average Case for CF of Program 5.

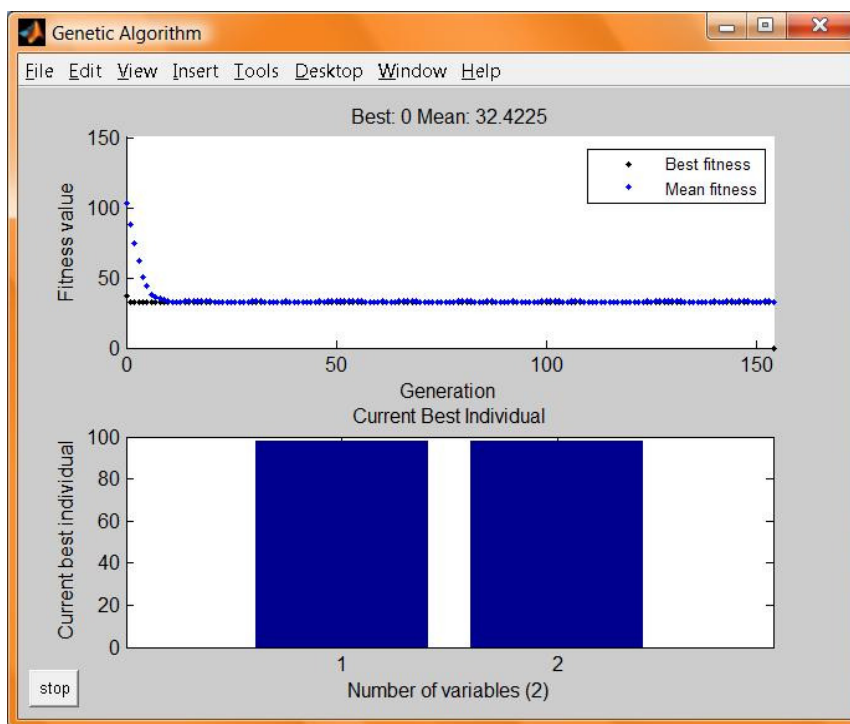
Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E4	baaa	107	4.5552	33.23	minimum fitness limit reached
E8	bbbb	318	16.3177	33.55	minimum fitness limit reached
E13	bbbbbb	15	0.9984	34.4	minimum fitness limit reached
E17	ba	264	12.9949	32.62	minimum fitness limit reached
Average	-----	199	9.606535	34.4535	-----

In Table 4-14 we summarize the results sample Experiment that taken from 20 time experiments conducted for CF of Program 5 . For example in experiment 8 (E8) the founded test case " bbbb " that used to excute predicate 5 found at Generation number 318 during 16.3177 seconds with cost equal zero and with avarege of fitness value equal 33.55. The last row in Table 4-14 shown The Avarege for the 20 experiments were the generation number equal 198.9 rounded to 199 , the needed time is 9.606535, and the mean of fitness value is 34.4535.

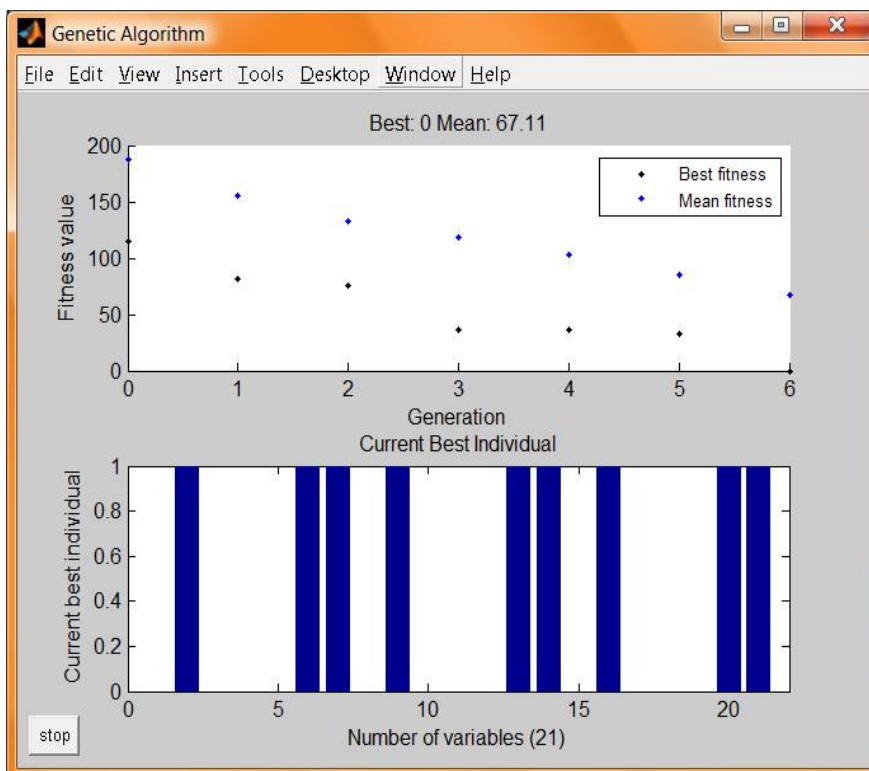
Table 4-15: Sample Experiments and Average Case for BF of Program 5.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E2	baabb	1004	622.7872	37.84	minimum fitness limit reached
E9	baa	234	100.8702	34.03	minimum fitness limit reached
E17	bbbbbb	21	14.2585	41.3	minimum fitness limit reached
E19	ba	2	1.1076	87.04	Minimum fitness limit reached
Average	-----	139.2	83.660995	47.3075	-----

Table 4-15 present a brief description of sample experiments for Program 5 but in BF. In experiment2 (E2) the revealed test case was " baabb" , discovered at Generation number 1004 with needed time equal to 622.7872 seconds at fitness value equal 0 and with avarege of fitness value equal 37.84 , and termrrnation reaseon is that the minimum fitness limit reached . Avarege for the 20 experiments the generation number was equal to 139 , the needed time is 83.660995 , and the mean of fitness value is 42.6645. The belowed chartes display some of Sample experiments and their flows .



(a) Experiment 2 flow



(b) Experiment 5 flow

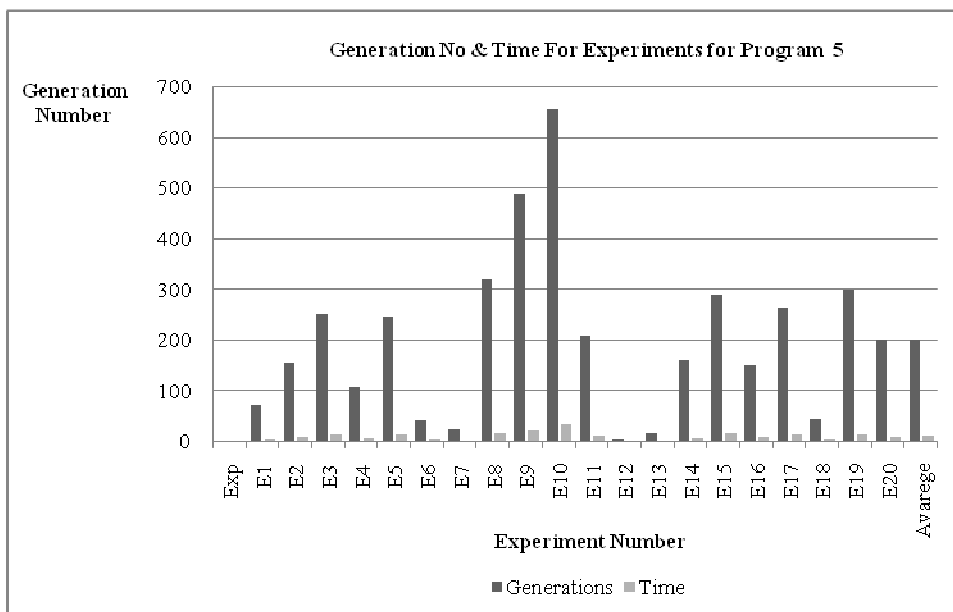
Figure 4-16: The Flow of Experiments: E2 for CF, and E5 for BF, respectively.

As we can observe in Figure 4-16, two diagrams that illustrate the experiments and their flows. Figure 4-16 (a) presents the flow for experiment 2 in CF. It starts with 32.75 fitness value and decreases until reach to zero. The generation where the test case (bb) discovered was 154. In The lower part of diagrams the ASCII codes for current best individual is appeared that are:

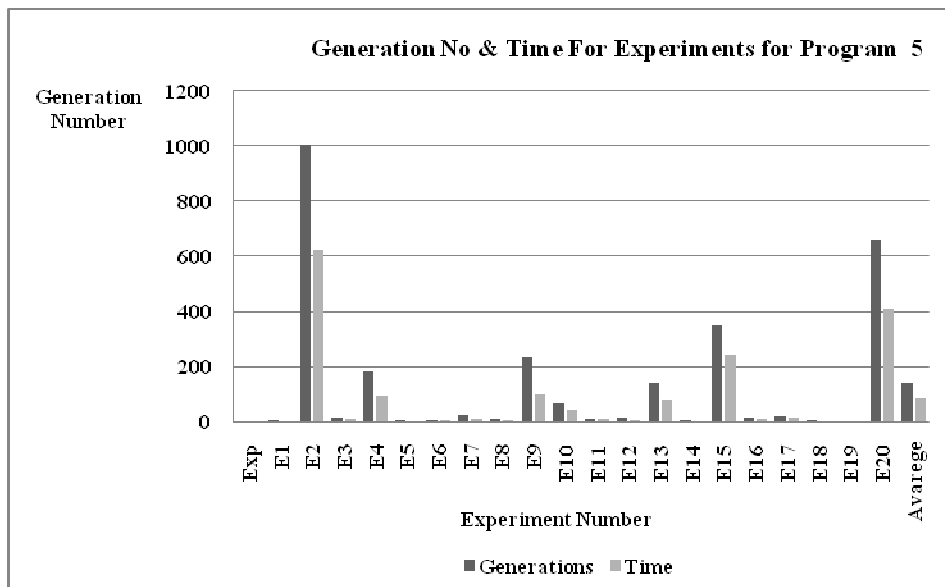
{98 and 98}.

Figure 4-16 (b) explains flow of experiment 5 in BF. It began with cost equals 81.25 then decreases to 75.25, 36.5, and 32.75 and ended with fitness value equal 0. The generation where the test case (bbb) found was 6. In the lower part of the diagram there are the ASCII codes for the best individual characters which are:

{ (0 1 0 0 0 1 1), (0 1 0 0 0 1 1),
(0 1 0 0 0 1 1)}.



(a) Continuous Form



(b) Binary Form

Figure 4-17: Twenty Experiments for CF and BF in Program 5.

In Figure 4-17, we can observe two charts that display the results of 20 experiments of Program 5. In Figure 4-17 chart (a), we observed that the generation number for experiments are approximately close to each other except some extreme value such as

E19 and E10. On the other hand, the generation number in chart (b) has diversified values; therefore, they are not close to each other. In addition, we note from the Average of the 20 experiments that the needed generation number to find the test case in binary form of Program 5 is less than the generation number in the CF. The needed time to find the test case in the BF is greater than the needed time in the Continuous Form.

Program 6:

Testing Regular Expression that includes Three Repetitions Operator (*) using the proposed OED fitness function (Predicate 6). In Program 6 the Regular expression is $RE = ba^*b^*a^*$. The SSF is { b, ba, baa, baaa, baaaa, bb, bbb, bbbb, bbbbbb, bbbbbbb, bab, baabb, baaabb, baabbb, babbbb, baaaab, baabba, baabaa, babaaa, babbaa}. Predicate 6 shown in Figure 4-6. Table 4-16 summarizes the sample of obtained results of 20 CF experiments of Program 6, which have been run 20 times and had different results, While Table 4-17 shows the obtained results related to BF of Program 6.

Table 4-16: Sample Experiments and Average Case for CF of Program 6.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E2	baaaaa	1120	49.7175	32.85	minimum fitness limit reached
E10	bbbbbb	237	12.4333	34.27	minimum fitness limit reached
E13	bab	76	3.9624	32.42	minimum fitness limit reached
E20	baabb	73	3.9000	34.14	minimum fitness limit reached
Average	-----	291	14.67735	33.2945	-----

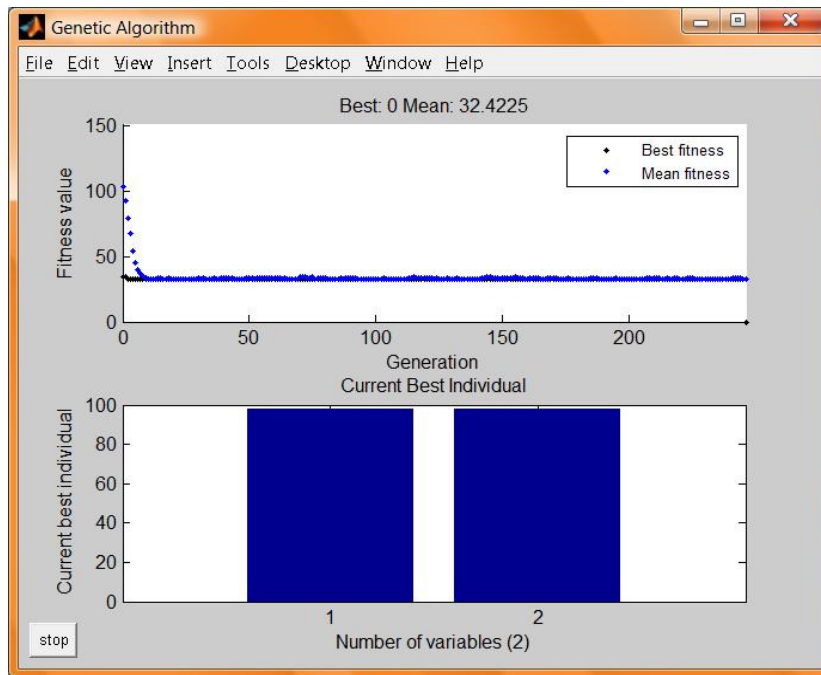
In Table 4-16 we summarize the sample result from 20 experiments for CF of Program 6. In experiment2 (E2) the found test case was "baaaaa" that used to execute predicate 6 discovered at Generation number 1120 during 49.7175 seconds with cost equal zero and with average of fitness value equal 32.85. The experiment terminated due to the minimum fitness limit reached. In last row in Table 4-16 we can Notice The

Average for the twenty experiments were the generation number equal 290.95 rounded to 291, the needed time is 14.67735, and the mean of fitness value is 33.2945.

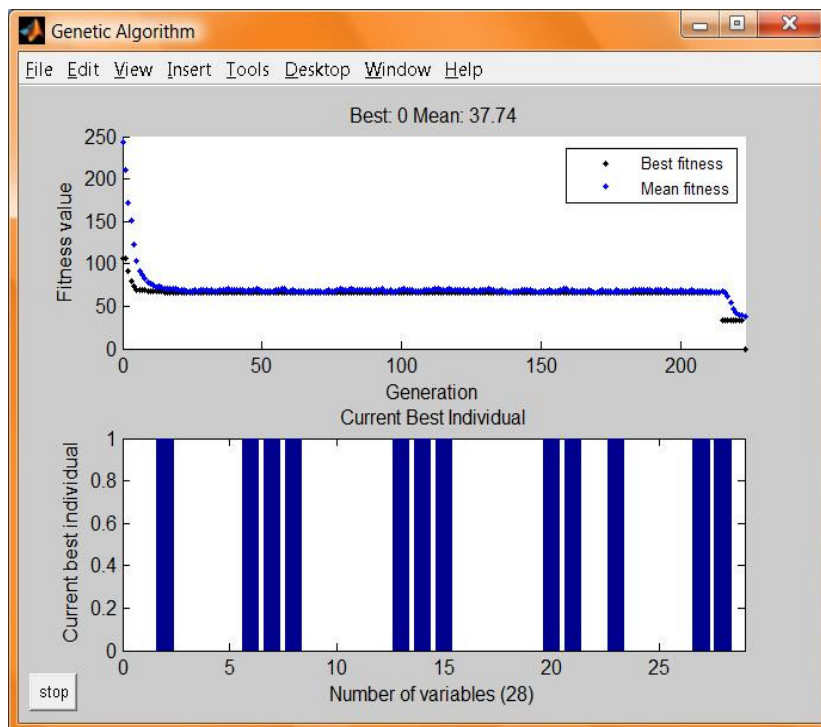
Table 4-17: Sample Experiments and Average Case for BF of Program 6.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E3	baabb	263	207.2317	36.09	minimum fitness limit reached
E8	b	1	0.5772	46.56	minimum fitness limit reached
E15	bbbbbb	743	570.5737	35.6	minimum fitness limit reached
E20	baaaa	16	10.6549	36.32	minimum fitness limit reached
Average	-----	139	99.49432	45.367	-----

Table 4-17, gives a brief description for sample experiments for Program 6 but in BF. In experiment 3 (E3) the discovered test case was "baabb", found at Generation number 263, and needed time equal to 207.2317 seconds at fitness value equal 0 and with average of fitness value equal 36.09, and termination reason is that the minimum fitness limit reached. Average for the 20 experiments was calculated and the results were as follow: the generation number was 138.9 rounded to 139, the needed time is 99.49432, and the mean of fitness value is 45.367. The belowed chartes display some of chosen experiments and their flows.



(a) Experiment 5 flow



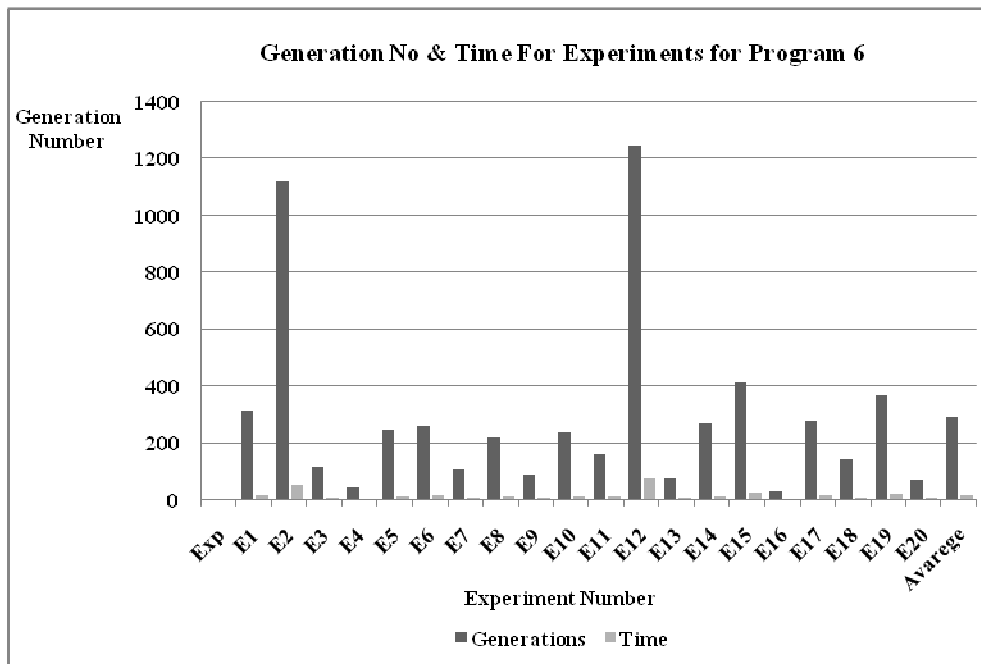
(c) Experiment 6 flow

Figure 4-18: The Flow of Experiments: E5 for CF, and E6 for BF, respectively.

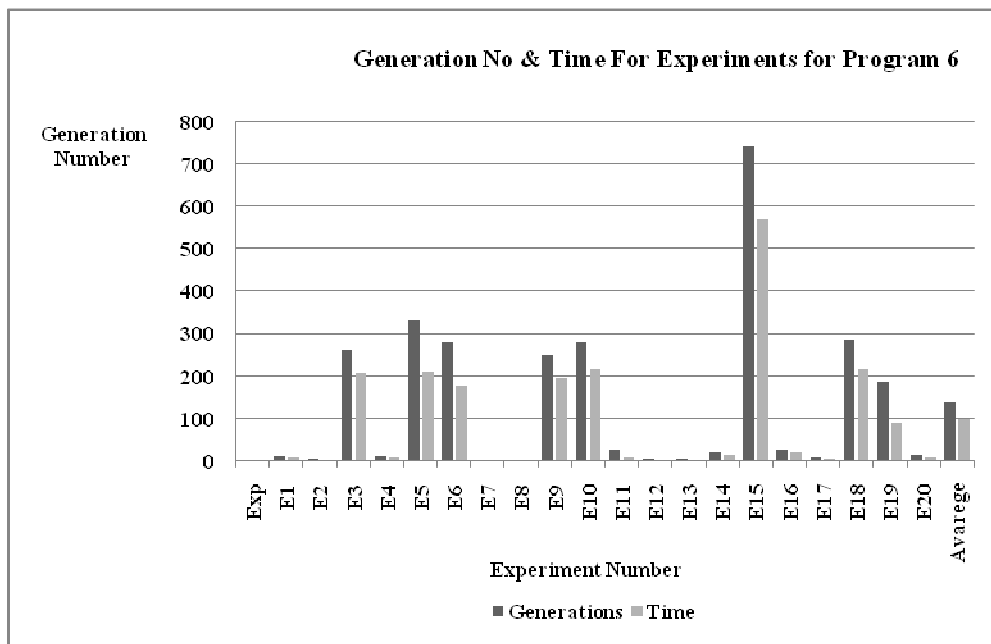
In Figure 4-18, we can observe two diagrams that illustrate the experiments. In Figure 4-18 (a) presents the flow for experiment 5 in CF. It starts with 34.25 fitness value and then 32.75, until reach to 0. The generation where the test case (bb) discovered was 246. In The lower part of diagrams the ASCII codes for current best individual is appeared that are {98, and 98}.

Figure 4-18 (b) explains flow of experiment 6 in Binary Form. It began with cost equals 142.5 then decreases to 120, 71.5, 68.5, 34.25, 33.5, and 32.75 and ended with cost equal 0. The generation where the test case (bbbb) found was 280. In the lower part of the diagram there are the ASCII codes for the best individual characters which are:

{ (0 1 0 0 0 1 1), (0 1 0 0 0 1 1),
 (0 1 0 0 0 1 1), (0 1 0 0 0 1 1) }.



(a) Continuous Form



(b) Binary Form

Figure 4-19: Experiments for CF and BF in Program 6

In Figure 4-19, we notice two charts that show the results of 20 experiments of Program 6. Figure 4-19 chart (a) we found that the generation number for experiments are approximately close to each other except some extreme value such as E2 and E12. On the other hand, the generation number in chart (b) has diversified values; therefore, they are not close to each other. Also we note from the Average of the 20 experiments that the needed generation number to find the test case in BF of Program 6 is less than the generation number in the CF, also the required time to find the test case in the BF is greater than the needed time in the CF. We will explain these notices later in this chapter.

Program 7:

Testing Regular Expression that includes Repetitions Operator (*) and OR Operator (|) using Proposed OED fitness function (Predicate 7). In Program 7 the Regular expression is $RE = ba^*ab$. SSF is {ba, bb, baa, bab, baaa, baab, baaaa, baaab, baaaaa, baaaab}. Predicate 7 shown in Figure 4-7. Table 4-18 summarizes the sample of obtained results of 20 CF experiments of Program 7, which have been run 20 times and had different results, While Table 4-19 shows the obtained results related to BF of Program 7.

Table 4-18: Sample Experiments and Average Case for CF of Program 7.

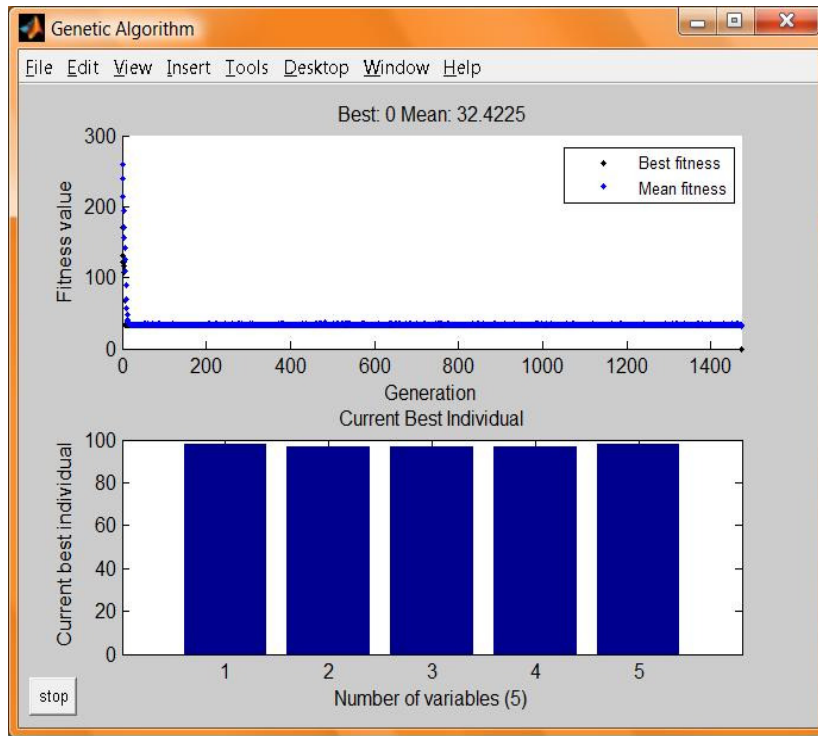
Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E2	baaaab	548	24.4922	32.47	minimum fitness limit reached
E7	baab	809	36.6446	32.87	minimum fitness limit reached
E13	baaaa	536	18.8917	32.98	minimum fitness limit reached
E20	baaab	668	30.5138	32.42	minimum fitness limit reached
Average	-----	457	18.98767	33.509	-----

In Table 4-18 we summarize the result of sample experiments for CF of Program 7. In experiment2 (E2) the founded test case "baaab" that used to excute predicate 7 arise at Generation number 548 during 24.4922 seconds with avarege of fitness value equal 32.47, E2 terminated due to the minimum fitness limit reached. Avarege for the 20 expriments were the generation number equal 456.65 rounded to 457, the needed time is 18.98767, and the mean of fitness value is 33.509.

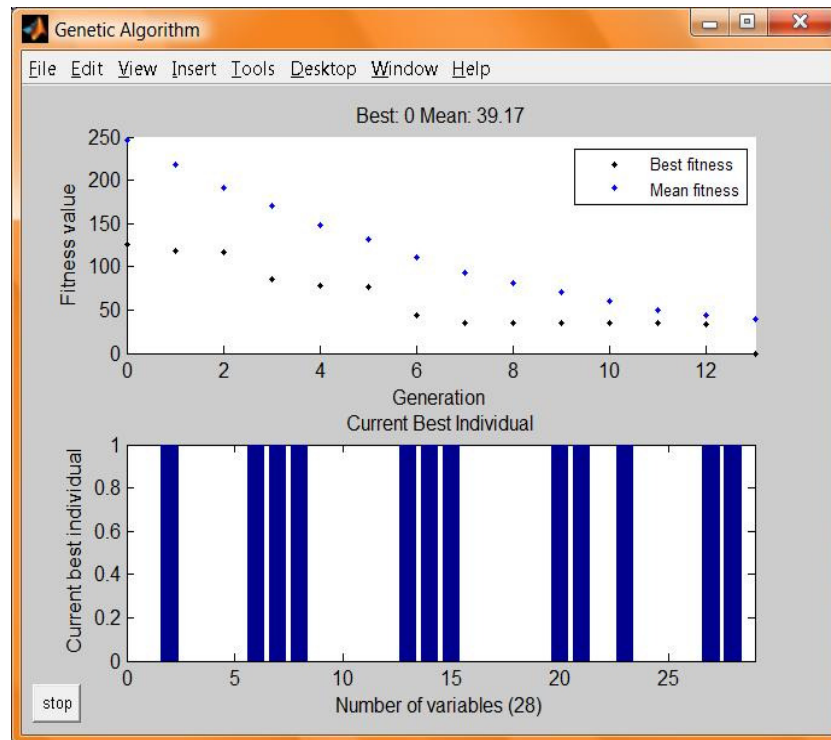
Table 4-19: Sample Experiments and Average Case for BF of Program 7.

Exp No.	Test Case	Generations No.	Time	Best f(x)	Mean f(x)	Termination Reason
E4	baaa	693	213.7994	0	36.7	minimum fitness limit reached
E11	baaab	46	18.6577	0	35.47	minimum fitness limit reached
E17	bb	8	1.9032	0	41.08	minimum fitness limit reached
E20	baaa	10	3.8688	0	67.06	minimum fitness limit reached
Average	-----	275.4	102.76938	0	42.9155	-----

Table 4-19 gives a briaif description for sample experiments for Program 7 but in BF. In experiment 4 (E4) the found test case was "baaa", discovered at Generation number 693, and needed time equal to 213.7994 seconds at fitness value equal 0 and with avarege of fitness value equal 36.7, and termrrnation reaseon is that the minimum fitness limit reached. The Avarege case for the 20 expriments was calculated and the results were as follow: the generation number was 275.4 rounded to 275, the needed time is 102.76938, and the mean of fitness value is 42.9155. The following chartes display some of choosen expriments and their flows.



(a) Experiment 9 flow



(b) Experiment 2 flow

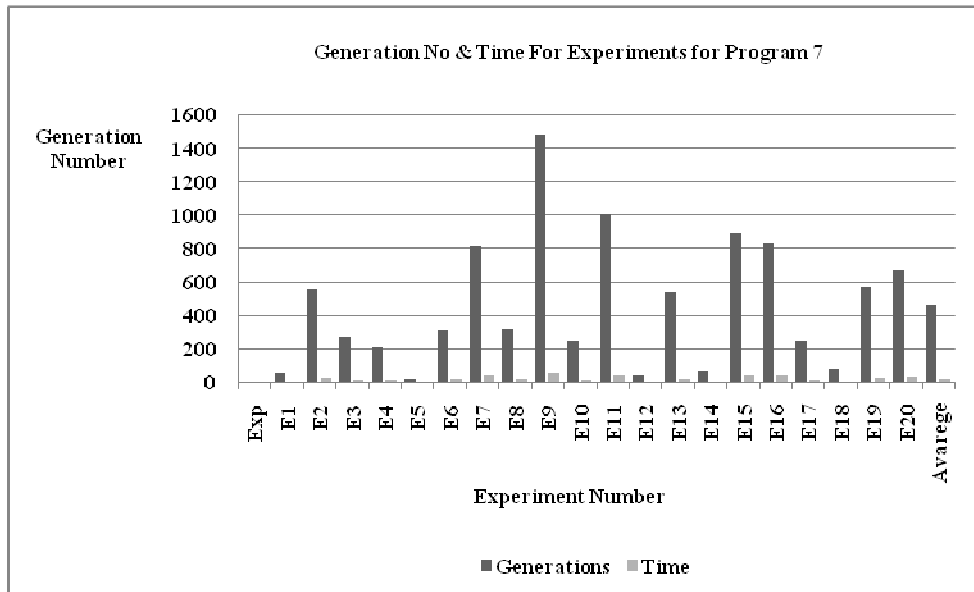
Figure 4-20: The Flow of Experiments: E9 for CF, and E2 for BF, respectively.

In Figure 4-20, we can observe two diagrams that illustrate the experiments and their flows. Figure 4-20 (a) presents the flow for experiment 9 (E9) , which is in CF. It starts with 131.3 fitness value and then 122.3, 116.3, 108, 67, and 32.75, until reach to 0. The generation where the test case (baaab) discovered was 1474. In The lower part of diagrams the ASCII codes for current best individual is appeared that are:

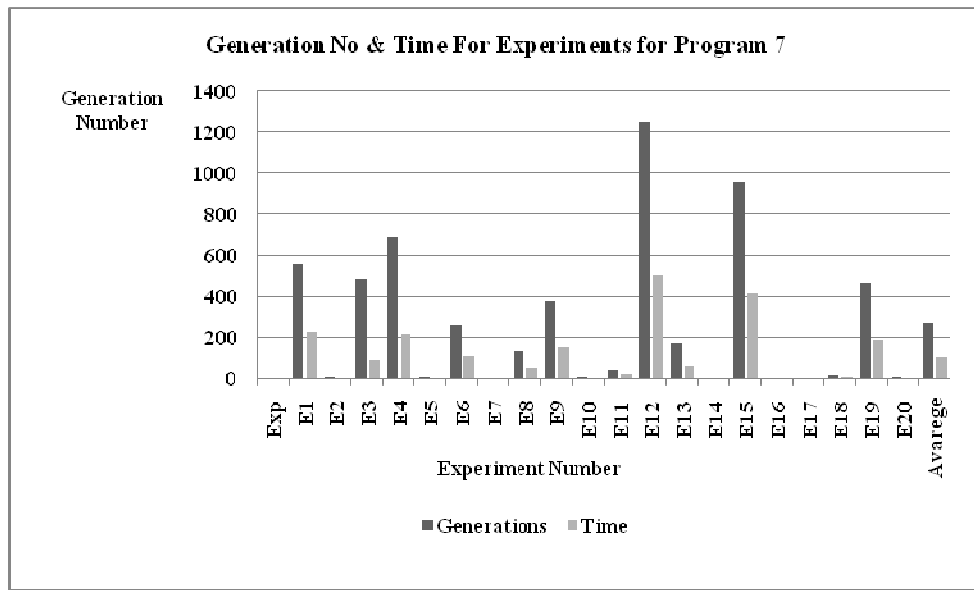
{98, 97, 97 ,97, and 98}.

Figure 4-20 (b) explains flow of experiment 2 (E2) that is in Binary Form. It began with cost equals 117.8 then decreases to 116.3, 85, 77.5, 76.75, 44, 35, 34.25 and 32.75 then the fitness value ended with cost equal **0**. The generation where the test case (baab) found was 13. In the lower part of the diagram, there are the ASCII codes for the best individual characters, which are:

{ (0 1 0 0 0 1 1), (1 0 0 0 0 1 1),
(1 0 0 0 0 1 1), (0 1 0 0 0 1 1) }.



(a) Continuous Form



(b) Binary Form

Figure 4-21: Experiments for CF and BF in Program 7.

In Figure 4-21, we observe two charts that display the results of 20 experiments of Program 7. In Figure 4-21 chart (a), we observed that the generation number for experiments are approximately close to each other except some extreme value such as

E12 and E15. On the other hand, the generation number in chart (b) has diversity values; therefore, they are not close to each other. In addition, we noticed from the Average of the 20 experiments that the needed generation number to find the test case in BF of Program 7 is less than the generation number in the CF. Also the required time to find the test case in the BF is greater than the needed time in the CF. We will explain these notices in the next section.

4.9 Analysis and Results

In the previous sections seven programs, each program represents a regular expression predicate, and each one is conducted 20 times for Binary and Continuous Form as a separate experiment. To study and analyze each experiment, we noticed five parameters: Numbers of generations, Required Time to find the TC, Best Fitness Function, Mean Fitness Function, and GA Termination Reason. After finishing the execution of the seven programs for 20 times, the average for each experiment was calculated, then, we studied, analyzed and compared each experiment, to others. Table 4-20 presents the average results for the seven programs in CF and BF.

Table 4-20: The Average Results for Seven Programs in CF and BF.

Form	Continuous (CF)			Binary (BF)		
	Gen. No.	Time (Seconds)	Mean of fitness value	Gen. No.	Time (Seconds)	Mean of fitness value
Program1	524	19.34	33.52	356	51.41	41.74
Program2	449	17.18	33.2	195	27.78	38.30
Program3	206	7.47	38.39	154	21.26	40.63
Program4	478	17.19	33.37	240	50.12	42.67
Program5	199	9.61	34.45	139	83.66	47.31
Program6	291	14.68	33.29	139	99.49	45.37
Program7	457	18.99	33.51	275	102.77	42.92

Table 4-20 summarizes the average results for each program in CF and BF. We have two main factors we used to compare between CF and BF:

- Generation Number.
- Required Time.

From Table 4-20, we conclude the following Conclusions and Notices:

1. Generation Number:

CF Generation Number is greater than BF's in all the Programs and Experiments. For example, in Program 4 the average of the generation number in CF is to 478, while the average of the generation number in BF is 240. The percentage between the BF Generation Number and the CF Generation Number is 0.58

2. Required Time :

The Required Time in CF is less than the Required Time in BF. For example in Program 1 the Required Time to find TC in CF is 19.34, on the other hand the Required Time to find TC in the BF is 51.41. The percentage between the CF Required Time and the BF Required Time is 0.24.

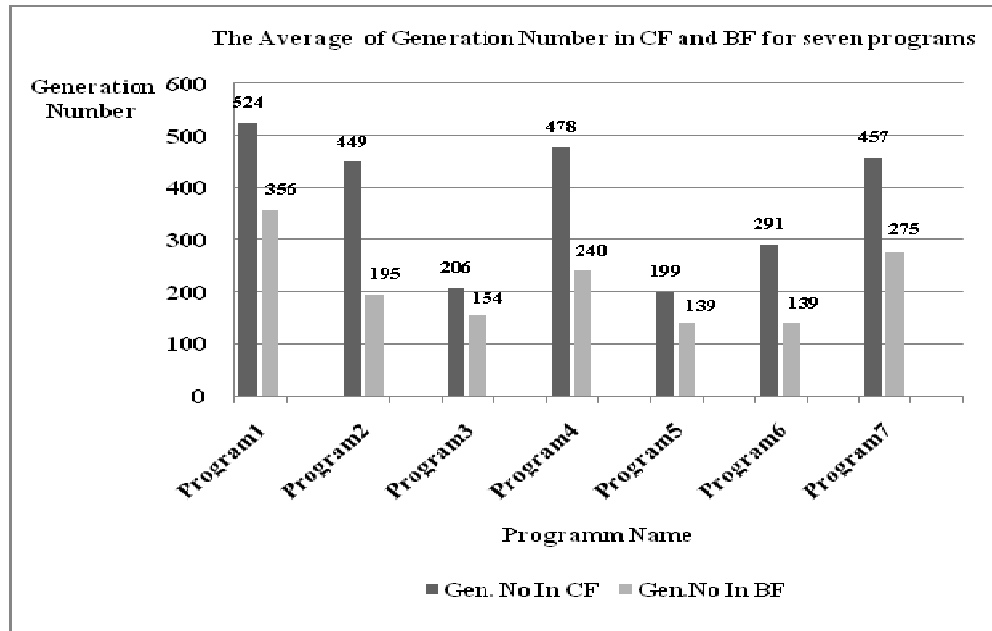


Figure 4-22: Comparison between the Average of Generation Number in BF and Cf.

Figure 4-22 shows the number of generations needed to find the solution. The black columns represent the Average of Generation Number in CF, while the Gray columns represent the average of Generation Number in BF. From Figure 4-22, we can conclude that the number of needed generation in BF is less than the one in the CF since BF GA deal with the character as a set of bits e.g. (0, 1) therefore, during the mutation operation, any changes in one bit (underlying binary representation) may lead to a huge changes in the characters representation, which, means that the GA need less generation numbers to reach the solution (TC). On the other hand, in CF, we deal with characters as ASCII codes, which are decimal numbers, where during the mutation operation, the character is considered as one unit, which is the ASCII code value; therefore, the changes in CF are not strong as the changes in BF, which deal with the character as a set of units (bits).

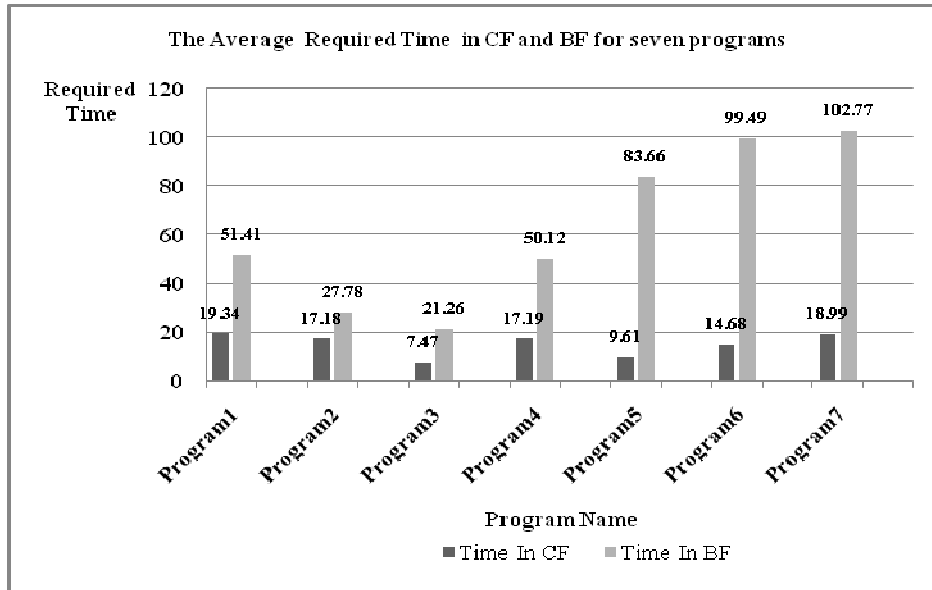


Figure 4-23: Comparison between the Average of Required Time in BF and Cf.

In Figure 4-23, the black columns point to the Average Required Time in CF for every program and the Gray columns refer to the Average Required Time in BF. We can conclude from Figure 4-23 that the needed time to find the TC in BF is greater than the time in CF, and this due to dealing with one character in BF indicates that GA deals with a set of bits (units), which means more time to manipulate these bits in the whole operation of GA. On the other hand, CF GA manipulates each character as one unit, which reduces the needed time. Another reason is that GA in CF needs not to use encoding and decoding methods as in BF, which leads to less Required Time.

CHAPTER 5

Conclusion and Future Research

5.1 Accomplishments and Contributions to the Field

In this research, we have accomplished several goals. We propose new methodology that used to Test RE predicates. In addition, we used heuristic search (GA) in RE Testing. We adapt OED Cost function which used in RE Testing.

We have implemented our proposed methodology using Matlab7.1. Our simulation included seven experiments for each Form; Binary and Continuous Form. The obtained results studies and analyzed. The obtained results proof our proposed technique therefore the RE Predicate covered.

The RE predicate experiments are implemented in tow forms BF and CF. the BF needed more time to find the TC with less number of generations, while CF needed less time with greater number of generations. The percentage between the BF Generation Number and the CF Generation Number is 0.58. On the other hand the percentage between the CF Required Time and the BF Required Time is 0.24.

5.2 Future Researches

Ongoing researches have been established to find new proposed methodology and techniques that include all the special characters for regular expression. Therefore, we test regular expression predicates that contains all type of regular expression. In addition use other cost function other than OED to implement our proposed technique.

References

- Abo-Hammour Z.S, (2002), Advance Continuous Genetic Algorithm and their Application in the Motion Planning of Robot Manipulators and In the Numerical Solution of Boundary value Problems, **PhD .D.Dissertation.**
- Alshraideh Mohammad, (2007), USE OF PROGRAM AND DATA-SPECIFIC HEURISTICS FOR AUTOMATIC SOFTWARE TEST DATA GENERATION, **PhD Thesis in the University of Hull.**
- Alshraideh Mohammad and Bottaci Leonardo, (2006) , Automatic Software Test Data Generation for String Data Using Heuristic Search with Domain Specific Search Operators, **Department of Computer Science, the University Of Hull, HULL, HU6 7RX, UK.**
- Alzabidi Maha, Kumar Ajay, and Shaligram A.D. ,(2009), Automatic Software Structural Testing by Using Evolutionary Algorithms for Test Data Generations, **IJCSNS International Journal of Computer Science and Network Security, VOL.9 No.4.**
- Boyapati Chandrasekhar, Khurshid Sarfraz, and Marinov Darko,(2002), Korat: Automated Testing Based on Java Predicates , **MIT Laboratory for Computer Science 200 Technology Square Cambridge, MA 02139 USA.**
- Chen S.M, (2002), Automatically Constructing Membership Functions and Generating Fuzzy Rules Using Genetic Algorithm. **An International Journal, Vol, 33, pp.841-862.**
- Chu, H.D, (1996), **an Evaluation Scheme of Software Testing Techniques.**
<http://citeseer.ist.psu.edu/68763.html>

- Hanif Ayesha, Ahmed Zaheer, and Hanif Muhammad, and Aqdas Ali (2006), Regular Expression to Finite State Machine , **Journal of Applied Sciences Research, 2(12): 1359-1362, INSInet Publication.**
- Haupt Randy L. and Haupt Sue Ellen (2004), PRACTICAL GENETIC ALGORITHMS, SECOND EDITION, Copyright © 2004 by John Wiley & Sons, Inc. All rights reserved.
- HUANG J.C,(1975), An Approach to Program Testing , **Department of Computer Science, University of Houston, Houston, Texas 77004,** Computing Surveys Vol. 7, No. 3 September.
- Khor S. and Grogono P. (2004). Using a Genetic Algorithm and Formal Concept Analysis to Generate Branch Coverage Test Data Automatically, **In the Proceedings of the 19th International Conference on Automated Software Engineering.**
- Korel .B, (1990), Automated Software Test Data Generation, IEEE Transactions on Software Engineering, **IEEE Transactions on Software Engineering, Volume 16, No. 8, pp. 870-879.**
- Kurtz Stefan, 2003, Foundations of Sequence Analysis “**Lecture notes for a course In the Summer Semester 2003**”.
- J. Duran and S. Ntafos, (1984), an Evaluation of Random Testing, **IEEE Transactions on Software Engineering, 10 (4).**
- Last Mark ,Eyal Shay, and Kandel Abraham,(2005), Effective Black-Box Testing with Genetic Algorithms, **Department of Information Systems Engineering, Ben-Gurion University of the Negev, and, 2Department of Computer Science and Engineering, University of South Florida.**

- Lu Luo, (2002), Software Testing Techniques Technology Maturation and Research Strategy, **Institute for Software Research International Carnegie Mellon University Pittsburgh, PA15232 USA.**
- Michael, C.C., McGraw, G.E., Schatz, M.A., and Walton, C.C, (1997), Genetic algorithms for dynamic test data generation, **In Proceedings of the 12th IEEE International Automated Software Engineering Conference (ASE 97), pp. 307-308, Tahoe, NV.**
- Mitchell, M (1999), an Introduction to Genetic Algorithms. **Reading, MA: MIT.**
- Muzatko P, (1996), approximates regular expression matching, **faculty electrical engineering Czech Technical University.**
- Pallavi Joshi, Koushik Sen and Mark Shlimovich, (2007), Predictive Testing: Amplifying the Effectiveness of Software Testing, **EECS Department, university of Californi, Berkeley, USA, ACM (978-1-59593-812-1/07/0009).**
- Pargas, R.P., Harrold, M.J., and Peck, (1999), R.R. Test-Data Generation Using Genetic Algorithms. **Journal of Software Testing, Verification and Reliability.**
- Roper, M., Maclean, I., Brooks, A., Miller, J., and Wood, M, (1995) Genetic Algorithms and the Automatic Generation of Test Data.
- Ruilian Zhao and Michael R. Lyu, (2003), Character String Predicate Based Automatic Software Test Data Generation, **Beijing University of Chemical Technology, Chinese University of Hong Kong, IEEE.**
- Sthamer Harmen, (1995), the Automatic Generation of Software Test Data Using Genetic Algorithms, **University of Glamorgan / Prifvsgol Morgannwg for the degree of a Doctor of Philosophy.**

- Tai K. C, (1996), Theory of Fault-based Predicate Testing for Computer Programs, **IEEE Transactions on Software Engineering**.
- THOMPSON Ken (1968) , Regular Expression Search Algorithm, Bell Telephone Laboratories, Inc., Murray Hill, New Jersey, Volume 11 / Number 6 .
- Tracey N, J. Clark, and K. Mander, (1998), Automated Program Flaw Finding Using Simulated Annealing, **Proceedings of International Symposium on Software Testing and Analysis, Software Engineering, Notes, March**.
- W.pratt Terrence, and zelkowitz Marvin, programming languages design and implementation ", **fourth edition**.
- Zhao Ruilian and Lyuv Michael R, (2003), Character String Predicate Based Automatic Software Test Data Generation, **Proceedings of the Third International Conference on Quality Software (QSIC'03) 0-7695-2015-4/03**.

Appendix A

Results and Experiments

Program 1

Continuous Form Experiments:

Table A-1: Twenty Experiments for CF in Program 1.

Exp No.	Test Case	Generations No.	Time	Best f(x)	Mean f(x)	Termination Reason
E1	bbabba	649	19.3129	0	33.68	minimum fitness limit reached
E2	ababba	784	28.5014	0	32.75	minimum fitness limit reached
E3	bbabba	114	3.7284	0	33.41	minimum fitness limit reached
E4	ababba	1452	57.2836	0	33.75	minimum fitness limit reached
E5	ababba	369	14.5393	0	33.58	minimum fitness limit reached
E6	ababba	375	10.9825	0	33.3	minimum fitness limit reached
E7	ababba	631	19.5001	0	32.55	minimum fitness limit reached
E8	ababba	682	27.4562	0	33.66	minimum fitness limit reached
E9	ababba	531	21.3253	0	32.85	minimum fitness limit reached
E10	bbabba	606	24.5078	0	34.74	minimum fitness limit reached
E11	bbabba	815	33.4778	0	32.85	minimum fitness limit reached
E12	bbabba	160	4.9608	0	32.86	minimum fitness limit reached
E13	bbabba	727	18.1585	0	33.02	minimum fitness limit reached
E14	bbabba	262	10.0153	0	33.45	minimum fitness limit reached
E15	ababba	831	33.7118	0	33.4	minimum fitness limit reached
E16	bbabba	196	7.6596	0	33.54	minimum fitness limit reached
E17	bbabba	766	30.8414	0	34.5	minimum fitness limit reached
E18	ababba	116	4.3524	0	34.31	minimum fitness limit reached
E19	bbabba	97	3.7908	0	35	minimum fitness limit reached
E20	ababba	318	12.6361	0	33.27	minimum fitness limit reached
Average	-----	524	19.3371	0	33.5235	-----

Binary Form Experiments:

Table A-2: Twenty Experiments for BF in Program 1.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	bbabba	291	40.5915	38.73	minimum fitness limit reached
E2	ababba	1343	199.0573	41.05	minimum fitness limit reached
E3	bbabba	85	12.4177	37.15	minimum fitness limit reached
E4	ababba	17	2.8392	7.48	minimum fitness limit reached
E5	ababba	486	69.8260	37.27	minimum fitness limit reached
E6	bbabba	95	13.5097	38.64	minimum fitness limit reached
E7	ababba	1219	176.7491	37.2	minimum fitness limit reached
E8	ababba	20	3.1512	64.48	minimum fitness limit reached
E9	ababba	56	8.1589	37.54	minimum fitness limit reached
E10	ababba	22	3.3852	67.26	minimum fitness limit reached
E11	bbabba	895	129.2000	38.07	minimum fitness limit reached
E12	ababba	31	4.6332	49.27	minimum fitness limit reached
E13	ababba	66	9.6253	38.26	minimum fitness limit reached
E14	ababba	1271	181.9128	40.59	minimum fitness limit reached
E15	bbabba	338	48.0015	39.07	minimum fitness limit reached
E16	ababba	42	6.1620	37.66	minimum fitness limit reached
E17	ababba	14	2.2308	69.58	minimum fitness limit reached
E18	bbabba	51	7.4256	37.98	minimum fitness limit reached
E19	ababba	733	104.2399	38.36	minimum fitness limit reached
E20	bbabba	34	5.0856	39.09	minimum fitness limit reached
Average	-----	355	51.410125	41.7365	-----

Program 2

Continuous Form Experiments:

Table A-3: Twenty Experiments for CF in Program 2.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	babbba	413	15.5845	32.85	minimum fitness limit reached
E2	babbba	639	26.4266	33.28	minimum fitness limit reached
E3	babbba	51	2.1996	33.34	minimum fitness limit reached
E4	babbba	226	8.8453	34.78	minimum fitness limit reached
E5	bababa	164	5.3664	32.96	minimum fitness limit reached
E6	babbba	240	9.2509	34.84	minimum fitness limit reached
E7	bababa	249	9.5473	34.91	minimum fitness limit reached
E8	bababa	872	36.0518	33.31	minimum fitness limit reached
E9	bababa	233	6.9732	32.42	minimum fitness limit reached
E10	bababa	536	19.2661	32.89	minimum fitness limit reached
E11	bababa	676	27.1286	33.25	minimum fitness limit reached
E12	babbba	190	4.5240	32.42	minimum fitness limit reached
E13	bababa	609	19.7965	33.69	minimum fitness limit reached
E14	bababa	625	20.6077	32.42	minimum fitness limit reached
E15	bababa	296	11.4037	34.12	minimum fitness limit reached
E16	babbba	621	25.1162	34.08	minimum fitness limit reached
E17	babbba	1691	73.1021	32.91	minimum fitness limit reached
E18	babbba	195	4.5864	33.91	minimum fitness limit reached
E19	bababa	72	3.0888	35.6	minimum fitness limit reached
E20	bababa	375	14.4613	34.33	minimum fitness limit reached
Average	-----	449	17.16635	33.6155	-----

Binary Form Experiments:

Table A-4: Twenty Experiments for BF in Program 2.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	bababa	94	11.8093	37.17	minimum fitness limit reached
E2	babbba	55	8.0653	36.95	minimum fitness limit reached
E3	bababa	496	74.4125	39.54	minimum fitness limit reached
E4	babbba	303	43.2903	37.47	minimum fitness limit reached
E5	babbba	27	4.0248	36.01	minimum fitness limit reached
E6	bababa	497	70.7777	38.38	minimum fitness limit reached
E7	babbba	439	62.4628	38.47	minimum fitness limit reached
E8	bababa	188	26.9726	40.44	minimum fitness limit reached
E9	babbba	88	12.4333	37.97	minimum fitness limit reached
E10	bababa	246	34.6322	38.23	minimum fitness limit reached
E11	bababa	18	2.808	43.23	minimum fitness limit reached
E12	bababa	370	51.7143	35.52	minimum fitness limit reached
E13	babbba	205	28.7822	37.2	minimum fitness limit reached
E14	babbba	31	4.5552	38.45	minimum fitness limit reached
E15	bababa	71	9.9841	37.32	minimum fitness limit reached
E16	babbba	41	5.7252	41.13	minimum fitness limit reached
E17	bababa	431	60.4816	39.78	minimum fitness limit reached
E18	bababa	18	2.7768	39.17	minimum fitness limit reached
E19	bababa	233	32.6198	38.03	minimum fitness limit reached
E20	bababa	50	7.2384	35.57	minimum fitness limit reached
Average	-----	195	27.77832	38.3015	-----

Program 3

Continuous Form Experiments:

Table A-5: Twenty Experiments for CF in Program 3.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	babbab	472	10.1089	34.17	minimum fitness limit reached
E2	babbab	24	2.4804	34.36	minimum fitness limit reached
E3	babbab	472	19.7185	34.17	minimum fitness limit reached
E4	babbaa	238	9.0169	34.81	minimum fitness limit reached
E5	babbab	63	2.4960	33.14	minimum fitness limit reached
E6	babbab	28	1.2324	45.02	minimum fitness limit reached
E7	babbab	212	7.8469	33.85	minimum fitness limit reached
E8	babbab	7	0.4836	114.6	minimum fitness limit reached
E9	babbab	636	26.7542	34.17	minimum fitness limit reached
E10	babbaa	38	1.6380	33.51	minimum fitness limit reached
E11	babbaa	66	2.6520	32.93	minimum fitness limit reached
E12	babbab	233	9.0481	34.02	minimum fitness limit reached
E13	babbaa	80	3.2760	32.53	minimum fitness limit reached
E14	babbaa	382	11.3881	33.06	minimum fitness limit reached
E15	babbaa	150	4.5708	33.16	minimum fitness limit reached
E16	babbab	38	1.2948	36.49	minimum fitness limit reached
E17	babbab	140	5.2884	32.92	minimum fitness limit reached
E18	babbab	504	15.7249	34.06	Minimum fitness limit reached
E19	babbab	133	5.0700	33.69	Minimum fitness limit reached
E20	babbab	212	8.2369	33.16	minimum fitness limit reached
Average	-----	206	7.41629	38.391	-----

Binary Form Experiments:

Table A-6: Twenty Experiments for BF in Program 3.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	babbaa	419	52.9623	35.21	minimum fitness limit reached
E2	babbab	438	62.6656	39.33	minimum fitness limit reached
E3	babbaa	105	13.2601	36.07	minimum fitness limit reached
E4	babbaa	18	2.8860	44.28	minimum fitness limit reached
E5	babbab	390	55.5208	36.3	minimum fitness limit reached
E6	babbab	140	18.6577	39.01	minimum fitness limit reached
E7	babbab	276	39.2343	39.15	minimum fitness limit reached
E8	babbaa	168	23.9774	37.66	minimum fitness limit reached
E9	babbaa	26	3.9000	54.42	minimum fitness limit reached
E10	babbab	34	5.0076	37.19	minimum fitness limit reached
E11	babbab	21	3.1668	39.38	minimum fitness limit reached
E12	babbaa	105	15.0541	41.04	minimum fitness limit reached
E13	babbaa	24	3.4944	40.84	minimum fitness limit reached
E14	babbaa	37	5.6004	39.16	minimum fitness limit reached
E15	babbaa	194	27.4250	38.54	minimum fitness limit reached
E16	babbab	344	45.8643	38.4	minimum fitness limit reached
E17	babbab	16	2.2776	59.56	minimum fitness limit reached
E18	babbaa	157	20.8885	38.68	Minimum fitness limit reached
E19	babbaa	62	8.8765	39.64	Minimum fitness limit reached
E20	babbab	108	14.4145	38.8	minimum fitness limit reached
Average	-----	154	21.256695	40.633	-----

Program 4

Continuous Form Experiments:

Table A-7: Twenty Experiments for CF in Program 4.

Time No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	baab	235	5.4288	33.12	minimum fitness limit reached
E2	baaab	204	4.3680	33.7	minimum fitness limit reached
E3	bb	111	2.3244	32.49	minimum fitness limit reached
E4	baaaab	971	39.9675	33.19	minimum fitness limit reached
E5	baaaab	353	6.6456	36.63	minimum fitness limit reached
E6	baaaab	723	31.2002	33.83	minimum fitness limit reached
E7	baaaab	1178	51.8547	32.46	minimum fitness limit reached
E8	baaab	70	1.8876	33.15	minimum fitness limit reached
E9	baab	263	8.3929	33.78	minimum fitness limit reached
E10	bab	265	8.4397	32.93	minimum fitness limit reached
E11	bab	168	3.7128	33.63	minimum fitness limit reached
E12	bab	626	26.2706	33.26	minimum fitness limit reached
E13	bab	48	1.5600	33.16	minimum fitness limit reached
E14	baab	199	5.7252	32.88	minimum fitness limit reached
E15	baaaab	510	21.6997	33.89	minimum fitness limit reached
E16	baaab	725	20.7949	33.06	minimum fitness limit reached
E17	bab	1056	35.7866	32.51	minimum fitness limit reached
E18	baaaab	615	12.9013	33.45	minimum fitness limit reached
E19	baaab	733	32.7134	33.32	minimum fitness limit reached
E20	baab	512	22.0897	33.03	minimum fitness limit reached
Average	-----	478	17.18818	33.3735	-----

Binary Form Experiments:

Table A-8: Twenty Experiments for BF in Program 4.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	baaab	11	4.6800	74.09	minimum fitness limit reached
E2	bb	56	5.8188	34.23	minimum fitness limit reached
E3	baab	10	2.1528	53.91	minimum fitness limit reached
E4	bb	26	3.1044	34.54	minimum fitness limit reached
E5	baaab	25	5.8968	45.55	minimum fitness limit reached
E6	bab	404	61.2616	34.6	minimum fitness limit reached
E7	bab	14	2.3712	37.24	minimum fitness limit reached
E8	bab	6	1.2792	69.69	minimum fitness limit reached
E9	baaab	42	10.2961	38.15	minimum fitness limit reached
E10	baab	223	44.4447	37.74	minimum fitness limit reached
E11	baaab	12	3.2916	60.28	minimum fitness limit reached
E12	baab	1632	330.8937	38.26	minimum fitness limit reached
E13	baab	126	25.1318	34.68	minimum fitness limit reached
E14	baaab	1545	369.1296	38.99	minimum fitness limit reached
E15	baaaab	27	7.8469	38.67	minimum fitness limit reached
E16	baaab	18	4.7736	38.78	minimum fitness limit reached
E17	baaab	235	55.9576	36.42	minimum fitness limit reached
E18	bab	13	2.4180	36.95	minimum fitness limit reached
E19	bab	221	35.6150	34.14	minimum fitness limit reached
E20	bab	162	26.0678	36.38	minimum fitness limit reached
Average	-----	240	50.12156	42.6645	-----

Program 5

Continuous Form Experiments:

Table A-9: Twenty Experiments for CF in Program 5.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	ba	71	2.6052	32.42	minimum fitness limit reached
E2	bb	154	7.6752	32.42	minimum fitness limit reached
E3	bbbbbb	251	12.8857	32.88	minimum fitness limit reached
E4	baaa	107	4.5552	33.23	minimum fitness limit reached
E5	bbbbbb	245	12.4177	33.8	minimum fitness limit reached
E6	bb	41	2.2308	33.16	minimum fitness limit reached
E7	baaaaa	24	1.4820	33.3	minimum fitness limit reached
E8	bbbb	318	16.3177	33.55	minimum fitness limit reached
E9	baaa	487	20.8417	33.16	minimum fitness limit reached
E10	bbbbbb	654	34.1798	32.5	minimum fitness limit reached
E11	ba	207	10.0777	32.48	minimum fitness limit reached
E12	ba	3	0.3432	61.26	minimum fitness limit reached
E13	bbbbbb	15	0.9984	34.4	minimum fitness limit reached
E14	bb	159	6.5052	32.65	minimum fitness limit reached
E15	bbbbbb	287	15.3817	33.48	minimum fitness limit reached
E16	baaa	149	7.6128	32.9	minimum fitness limit reached
E17	ba	264	12.9949	32.62	minimum fitness limit reached
E18	bbb	44	2.1996	33.16	Minimum fitness limit reached
E19	bb	299	12.3709	32.43	Minimum fitness limit reached
E20	bbbb	199	8.4553	33.27	minimum fitness limit reached
Average	-----	199	9.606535	34.4535	-----

Binary Form Experiments:

Table A-10: Twenty Experiments for BF in Program 5

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	b	5	1.2012	36.85	minimum fitness limit reached
E2	baabb	1004	622.7872	37.84	minimum fitness limit reached
E3	bbbbbb	14	9.8593	51.6	minimum fitness limit reached
E4	bbbb	183	93.2262	36.52	minimum fitness limit reached
E5	bbb	6	3.0420	67.11	minimum fitness limit reached
E6	bab	8	3.6816	59.32	minimum fitness limit reached
E7	baaa	23	12.6205	37.16	minimum fitness limit reached
E8	bbb	9	4.0872	52.67	minimum fitness limit reached
E9	baa	234	100.8702	34.03	minimum fitness limit reached
E10	bbbbbb	67	43.5711	35.77	minimum fitness limit reached
E11	baabb	13	8.8921	59.82	minimum fitness limit reached
E12	baa	16	7.4724	38.31	minimum fitness limit reached
E13	baaa	143	77.9069	34.18	minimum fitness limit reached
E14	bb	5	2.0748	48.96	minimum fitness limit reached
E15	baabb	350	242.4412	38.12	minimum fitness limit reached
E16	baabb	15	10.5145	42.41	minimum fitness limit reached
E17	bbbbbb	21	14.2585	41.3	minimum fitness limit reached
E18	bab	6	2.9640	71.67	Minimum fitness limit reached
E19	ba	2	1.1076	87.04	Minimum fitness limit reached
E20	baabb	660	410.6414	35.47	minimum fitness limit reached
Average	-----	139	83.660995	47.3075	-----

Program 6

Continuous Form Experiments:

Table A-11: Twenty Experiments for CF in Program 6.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	baaa	313	14.7265	33.15	minimum fitness limit reached
E2	baaaaa	1120	49.7175	32.85	minimum fitness limit reached
E3	baaa	117	6.3648	33.03	minimum fitness limit reached
E4	baaabb	43	2.6832	33.2	minimum fitness limit reached
E5	bb	246	8.6425	32.42	minimum fitness limit reached
E6	bbbb	261	14.2117	34.32	minimum fitness limit reached
E7	baabb	108	4.8672	34.88	minimum fitness limit reached
E8	bab	219	11.1853	32.42	minimum fitness limit reached
E9	bbbb	90	4.8984	34.26	minimum fitness limit reached
E10	bbbbbb	237	12.4333	34.27	minimum fitness limit reached
E11	baa	163	8.3461	32.42	minimum fitness limit reached
E12	baabba	1248	71.5109	32.42	minimum fitness limit reached
E13	bab	76	3.9624	32.42	minimum fitness limit reached
E14	bbbb	271	11.8873	33.76	minimum fitness limit reached
E15	baabb	414	21.9181	32.92	minimum fitness limit reached
E16	baaaab	30	1.3572	33.43	minimum fitness limit reached
E17	baaa	278	14.4145	33.33	minimum fitness limit reached
E18	bb	144	7.1760	32.74	minimum fitness limit reached
E19	bbbb	368	19.3441	33.51	minimum fitness limit reached
E20	baabb	73	3.9000	34.14	minimum fitness limit reached
Average	-----	291	14.67735	33.2945	-----

Binary Form Experiments:

Table A-12: Twenty Experiments for BF in Program 6.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	baabb	12	10.1401	47.26	minimum fitness limit reached
E2	bbb	6	3.5724	66.65	minimum fitness limit reached
E3	baabb	263	207.2317	36.09	minimum fitness limit reached
E4	bbbbbb	12	10.0777	70.45	minimum fitness limit reached
E5	bbbb	334	210.8354	34.73	minimum fitness limit reached
E6	bbbb	280	179.1671	35.78	minimum fitness limit reached
E7	bb	4	3.3072	65.1	minimum fitness limit reached
E8	b	1	0.5772	46.56	minimum fitness limit reached
E9	baabb	252	195.1573	35.75	minimum fitness limit reached
E10	bbbbbb	281	216.9194	37.7	minimum fitness limit reached
E11	bb	28	10.2493	35.5	minimum fitness limit reached
E12	bb	6	2.5896	49.14	minimum fitness limit reached
E13	bab	7	4.0716	64.73	minimum fitness limit reached
E14	bbbbbb	21	16.8013	36.81	minimum fitness limit reached
E15	bbbbbb	743	570.5737	35.6	minimum fitness limit reached
E16	baabb	27	21.3097	38.93	minimum fitness limit reached
E17	bbbb	10	7.0200	64.16	minimum fitness limit reached
E18	baabb	286	217.2782	36.6	Minimum fitness limit reached
E19	bbb	189	92.3526	33.48	Minimum fitness limit reached
E20	baaa	16	10.6549	36.32	minimum fitness limit reached
Average	-----	139	99.49432	45.367	-----

Program 7

Continuous Form Experiments:

Table A-13: Twenty Experiments for CF in Program 7.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	baaaaa	53	2.4024	33.66	minimum fitness limit reached
E2	baaaab	548	24.4922	32.47	minimum fitness limit reached
E3	baa	265	8.8453	33.3	minimum fitness limit reached
E4	baaaab	208	8.9389	34.3	minimum fitness limit reached
E5	baa	14	0.6552	36.28	minimum fitness limit reached
E6	baaab	308	13.1353	32.96	minimum fitness limit reached
E7	baab	809	36.6446	32.87	minimum fitness limit reached
E8	baaaaa	311	13.3537	33.3	minimum fitness limit reached
E9	baaab	1474	56.2696	32.42	minimum fitness limit reached
E10	baaab	242	8.6737	32.85	minimum fitness limit reached
E11	baaaaa	1002	39.4371	35.69	minimum fitness limit reached
E12	baa	38	1.3884	32.75	minimum fitness limit reached
E13	baaaa	536	18.8917	32.98	minimum fitness limit reached
E14	baaa	58	2.5740	33.03	minimum fitness limit reached
E15	baaaaa	889	35.2406	32.42	minimum fitness limit reached
E16	baaaa	823	37.9862	33.85	minimum fitness limit reached
E17	bab	244	10.8421	32.91	minimum fitness limit reached
E18	baaab	80	3.4944	35.62	minimum fitness limit reached
E19	baaaab	563	25.9742	34.1	minimum fitness limit reached
E20	baaab	668	30.5138	32.42	minimum fitness limit reached
Average	-----	457	18.98767	33.509	-----

Binary Form Experiments:

Table A-14: Twenty Experiments for BF in Program 7.

Exp No.	Test Case	Generations No.	Time	Mean f(x)	Termination Reason
E1	baaaa	562	227.0127	36.58	minimum fitness limit reached
E2	baab	13	4.8516	39.17	minimum fitness limit reached
E3	baaab	485	89.9936	34.53	minimum fitness limit reached
E4	baaa	693	213.7994	36.7	minimum fitness limit reached
E5	bb	14	2.7924	33.63	minimum fitness limit reached
E6	baaaa	262	106.2211	36	minimum fitness limit reached
E7	baa	7	2.1060	70.19	minimum fitness limit reached
E8	baaaa	136	54.0231	35.43	minimum fitness limit reached
E9	baaaa	379	150.6502	37.53	minimum fitness limit reached
E10	baaab	10	4.4928	63.99	minimum fitness limit reached
E11	baaab	46	18.6577	35.47	minimum fitness limit reached
E12	baaab	1250	503.5088	37.59	minimum fitness limit reached
E13	baab	176	57.7516	36.7	minimum fitness limit reached
E14	baaa	8	3.4632	58.99	minimum fitness limit reached
E15	baaaa	956	412.6850	37.88	minimum fitness limit reached
E16	Baab	9	3.4632	47.47	minimum fitness limit reached
E17	bb	8	1.9032	41.08	minimum fitness limit reached
E18	baab	18	6.3336	35.23	minimum fitness limit reached
E19	baaaa	466	187.8096	37.09	minimum fitness limit reached
E20	baaa	10	3.8688	67.06	minimum fitness limit reached
Average	-----	275	102.76938	42.9155	-----

توليد الفحص الالي للتعابير المنتظمة المتوقعة

إعداد

رنا علي بدوي سمحان

المشرف

الدكتور محمد الشريدة

المشرف المشارك

الدكتور عبد اللطيف ابو دلهوم

ملخص

في هذه الايام تحث التكنولوجيا مكانة متميزة في جميع مجالات حياتنا اليومية . لذا يجب ان نتصف هذه التكنولوجيا بالدقة و الموثوقية العالية للمستخدم , اخذا بعين الاعتبار انها تتكون من البرمجيات بشكل رئيسي ؛ نتيجة لذلك نجد من الضروري التركيز على مرحلة الفحص في التكنولوجيا وخاصة فحص البرمجيات .

هذه الرسالة اولا تقترح طريقة جديدة لايجاد مجموعة من حالات الاختبار المصممة لفحص الجمل الشرطية التي تحتوي على التعابير المنتظمة . واحتجنا لتصميم هذه الطريقة المقترحة الى استخدام مجموعة من التقنيات و الاساليب منها : نموذج الحالة و ذلك لايجاد حل للتعابير المنتظمة ، و استخدمنا ايضا طريقة فحص الجمل الشرطية و ذلك لعمل فحص لكل جملة شرطية في البرنامج الذي يتم فحصه مرة واحد على الاقل ، و كذلك استخدمنا اسلوب الخوارزميات الجينية كوسيلة بحث عن حالات الاختبار اللازمة لعمل الفحص الالي على الجمل الشرطية التي تحتوي على التعابير المنتظمة .

لتنفيذ الطريقة المقترحة استخدمنا برمجية Matlab7.1 . حيث اعتمدنا سبع تجارب باستخدام طريقتين : التمثيل الثنائي و التمثيل العشري . و من ثم اخضعنا نتائج التجارب الى الدراسة والبحث . و اخيرا بناء على ما سبق تم تحقيق الهدف من هذا البحث و هو فحص الجمل الشرطية التي تحتوي على التعابير المنتظمة. ووجدنا ان التمثيل العشري اسرع في ايجاد الحل ولكن التمثيل الثنائي يساعد على ايجاد الحل بعدد اجيال اقل .

