Application of Negative Selection Algorithm (NSA) for test data generation of path testing

Shayma Mustafa Mohi-Aldeen\textsuperscript{a,\textit{c},*}, Radziah Mohamad\textsuperscript{b}, Safaai Deris\textsuperscript{b}

\textsuperscript{a} Faculty of Computing, Universiti Teknologi Malaysia, 81310 UTM Skudai, Johor, Malaysia
\textsuperscript{b} Faculty Teknologi Kreatif Dan Warisan, Universiti Malaysia Kelantan, 16100 Kelantan, Malaysia
\textsuperscript{c} College of Computer Science and Mathematics, University of Mosul, 41002, Mosul, Iraq

\textbf{A R T I C L E   I N F O}

Article history:
Received 26 December 2015
Received in revised form 23 August 2016
Accepted 24 September 2016
Available online \textit{xxx}

Keywords:
Software testing
Path testing
Automatic test data generation
Negative Selection Algorithm

\textbf{A B S T R A C T}

Path testing is one of the areas covered in structural testing. In this process, it is a key challenge to search for a set of test data in the whole search space to satisfy path coverage. Thus, finding an efficient method for generating test data automatically is a key issue in software testing. This paper proposed a method based on Negative Selection Algorithm (NSA) for generating test data to satisfy the path coverage criterion. The results show that NSA could reduce the number of test data generated and improve the coverage percentage, as well as enhance the efficiency of the test data generation process. To evaluate the performance of the method, results from the proposed method were compared with random testing and a previous work that used Genetic Algorithm and Ant Colony Optimization. The results demonstrate that NSA outperforms other methods in reducing the number of test data that cover all program paths even the difficult ones.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

Software testing is an essential process which is used for improving quality and increasing the reliability of the software. Software testing is a difficult, labor-intensive, and time-consuming process, accounting for more than half of the total cost of software development \cite{1}. With the rapid growth of software being developed, generating a huge number of test data in software testing is needed. The main task of software testing is to generate a number of test data that satisfied the required adequacy criterion. However, path coverage is an extremely complicated, laborious and time-consuming process resulting to automatic generation of meaningful test data that can reduce testing time and improve the process of generation to satisfy a given test adequacy criterion \cite{2}.

The subject of automatically generated test data has gained the attention of researchers. However, comprehensive testing of the program’s input is unfeasible for any reasonable size of the program \cite{3}. The process of generating test data is an undecidable problem and could be non-deterministic (NP-hard) or existing solutions are infeasible. The extremely non-linear construction of the program represents a problem for search algorithms to find efficient and optimal test data from a nonlinear, complicated and intermittent input in the searching space \cite{3}.

The structural testing coverage criteria which have been proposed is used to determine which test data can be exercised or covered. The source code of the program could be divided into control flow coverage criteria, which is focused on testing the flow of the program and data flow coverage criteria, which focused on the flow of the data within the program \cite{1,2}.

There are several criteria of the control flow coverage such as: statement coverage (execute every statement in the program at least once), branch coverage (every logical branch in the program is executed with both consequences at least once), condition coverage (every condition in the decision of the program is executed), multiple condition coverage (exercising all True and False combinations of conditions in compound decisions at least once), and path coverage (every distinct path in the program is executed at least once), which has been arranged from the weakest to the strongest type of coverage \cite{1}.

Path coverage is the strongest coverage criterion of test adequacy criteria in white box testing, i.e. every path in a program must be exercised at least once \cite{4,5}. Searching for test data to traverse each path is a very difficult process, since there are too many paths in the program and it may be infinite in the case of loops \cite{5}, or infeasible in case of nested branches i.e. difficult for find-

* Corresponding author

E-mail addresses: shaima.mustafa76@yahoo.com (S.M. Mohi-Aldeen),
Radziahm@utm.my (R. Mohamad), Safaai@umk.edu.my (S. Deris).

http://dx.doi.org/10.1016/j.asoc.2016.09.044
1568-4946/© 2016 Elsevier B.V. All rights reserved.
ing the set of test data which can cover these paths. As a result, many researchers proposed different techniques for searching test data that could achieve high path coverage [4,6]. Information is needed to guide and decide where is the best test data lie in the search space. This process involves searching for test data from a set of input data that causes a certain path in a program to be executed. Such process can also solve optimization problems. Many studies have been made on how to generate test data automatically in the past few decades and many researchers have proposed several techniques from time to time [7,8].

The paper is prepared as follows: Section 2 presents the works related to this study; Section 3 explains the basic concept of the Negative Selection Algorithm; Section 4 describes the proposed algorithm; Section 5 presents the experimental evaluations; while conclusions are shown in Section 6.

2. Related works

The generation of test data which satisfied the coverage criteria is an essential problem in software testing. Many researches have been made to solve this difficult problem. Several methods have been used from time to time to generate test data automatically based on structural testing for different coverage criteria in order to increase the coverage percentage of program testing, decrease the number of test data, as well as reduce the time execution of the testing process [9].

Commonly, automatic test data generation techniques could be classified into: random test data generation, symbolic test data generation, dynamic test data generation, and search-based test data generation techniques [1].

In random technique, the test data have been selected arbitrarily from an input domain. Although it is simple and could generate a lot of test data quickly, it generates a great number of redundant data [10].

In the early days of automation of software testing, majority of the test data generators used symbolic or dynamic technique. Symbolic test data generators are static, and assign symbolic values to variables instead of actual values in the program. Dynamic test data generators, on the other hand, require actual execution of the program with some selected inputs. If there are some desirable test requirements that have not been accomplished, the data which have been collected during the execution process could still be used to decide which data is nearest to satisfy the test requirements. This feedback helps in modifying the test inputs incrementally until all requirements are satisfied. However, these techniques were incompetent, time consuming, and got stuck in local optima of the searching space of the possible input data domain. Problems also happen when loops, array or pointers occur in Software Under Test (SUT) [5,10].

Search-based test data generation is part of a very wide area of research in search-based software engineering. It includes exploring for data in the input scope of the testing program to fulfill certain test data coverage criterion [6,7]. Search-based test data generation techniques like Genetic Algorithm (GA), Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), Simulated Annealing (SA), Artificial Bee Colony (ABC), and Memetic Algorithm (MA) are the most widely used methods in generating test data automatically, and meta-heuristic optimizing technique is used to find the best solution for automating the generation of test data by directing the search toward the most likely areas of the domain [8].

Automating the test data generation is still a popular topic of research. Although several methods have been recently proposed for automating the process of generating test data to fulfill the adequacy criteria, the application of these methods to find the optimal solution, minimum number of test data, maximum coverage of the required criterion, reasonable completion time, and avoiding redundancies is still limited. Despite the advantages of these methods, there is a need to improve the efficiency and effectiveness of the test generation process, since these techniques may yield a great amount of test data or redundant test data that wastes both time and cost. In addition, there may also be cases of low probability execution path which are not covered. There may also be a problem with infeasible paths in the program.

2.1. Existing literature on search-based test data generation techniques

This section will present the existing literature on search-based test data generation techniques. Many search-based techniques have been used in generating test data automatically with the main goal to maximize the selected coverage with minimum test suite size and minimum time consumed.

Local search-based methods such as Simulated Annealing (SA) proposed by Mansour and Salame [11] are simpler and offer better runtime than random testing and are more efficient for simple problems; however, they can become trapped in local optima and are very slow.

Global search methods like GA, PSO, ACO, and ABC solved the problems of local search methods by finding more global optimal solutions, but these methods are more complex and difficult to implement. GA has been applied in Srivastava and Kim [12], Nirpal and Kale [13], Suresh and Rath [14], Rao et al. [15], Xibo and Na [16], Sun and Jiang [17], Singh [18], Gupta and Gupta [19], Ahmed and Hermadi [20], Shimin and Zhangang [10], Al-Zabidi et al. [21], Athar and Gupta [22], Ghiduk [5], and Hermadi et al. [4]. The results show that GA is better than random testing in terms of coverage and time consuming as well as in reducing the number of test data, but the efficiency and effectiveness of this method depend on its fitness function.

PSO was applied in Li and Zhang [23], Peng [24], and Latiu et al. [25]. It has a fast convergence time and needs short computation time, which can effectively improve the slow convergence problems of the GA. However, it suffers from prematurity, affecting its speed and direction. For these reasons, Zhang et al. [26] and Li et al. [27] proposed a hybrid algorithm of GA with PSO to tackle these problems. Although the hybrid algorithm improved the number of generations, it increases the time needed for testing. In addition, the value of fitness is affected by the efficiency of the algorithm and in many problems, it gives the same coverage percentage as in GA and PSO.

Although the ACO has a better success rate compared to GA, it has several weaknesses namely increasing the size of the test suite and repeating the test data within the same test suite without any improvement in test coverage criteria. It also needs to update its pheromone value that negatively impacts its efficiency, and it requires a lot of time as shown in Srivastava et al. [28] and Dahiya et al. [29].

ABC has been applied by Mala and Mohan [30], Mala et al. [31], Jeya Mala et al. [32], Kulkarni et al. [33], and Lam et al. [3]. In generating test data, the results show that ABC is superior to GA and ACO in reducing the number of generations and time, but it is lacking in terms of coverage percentage. In addition, the colony size has an effect on the method’s performance.

MA which combines local search and global search has been applied by Zhang and Wang [34], MA is more efficient than pure local or global search, but it fails to cover new branch and requires more computational cost.

Although the evolutionary algorithm could direct the search and generate number of test data to guarantee the maximum per-
3. Overview of Negative Selection Algorithm (NSA)

Negative Selection Algorithm (NSA) is one of the most important methods in an Artificial Immune System (AIS), which is a branch of computational intelligence models. AIS was inspired by the biological behaviour of Natural Immune System (NIS), which is a compound biological network using fast and active techniques to defend the body versus a specified foreign body called antigens [36]. AIS is one of the different kinds of algorithms inspired by biologic systems, like evolutionary algorithms, swarm intelligence and neural networks, which have attracted a lot of researchers’ attention. Its objective is to develop immune-based techniques for solving complicated computation [36].

NSA was introduced by Forrest (1994) and it has been applied in fields such computer security, pattern recognition, anomaly detection and fault detection. The main goal of NSA is to distinguish between self and non-self samples while only the self samples are available. In particular, the goal is covering the non-self space with a specific number of detectors [37].

The basic idea of NSA is generating a number of detectors in the search space and then applying these detectors for classifying new data as self or non-self [38]. NSA consists of two stages: generation stage (also called training stage) and detection stage (also called testing stage). Firstly, in the generation stage, a random process is used to generate the detectors and the process is monitored. The matched candidates are discarded and the residues are stored as detectors. The generation stage is terminated when a sufficient number of detectors (detector sets) are generated that could be determined by certain stopping criteria [39]. While in the detection stage, the detector sets which have been generated in the first stage are used to check whether the input samples correspond to self or non-self samples. If the new sample matched any detector in the detectors set, then it is classified as a non-self which indicates anomaly in most applications [40,41]. Fig. 1 describes the main concept of NSA.

4. Applying Negative Selection Algorithm (NSA) in test data generation

Path testing is the main strategy in structural testing. A basic way to solve path testing is finding the specified input data in the search space that is likely to cover a path in the testing program. Many works have been done to generate test data automatically

centage of path coverage, it can still be improved in terms of its limitation in generating the optimal test data set [9,35].

![Fig. 1. Main concepts of Negative Selection Algorithm (NSA)](image)

![Fig. 2. Overview of the proposed method.](image)
to achieve path coverage. This research proposed a new test data generator by using the NSA to generate a set of test data which satisfied the path coverage criterion and reduce the size of the test suite of the program under test by removing redundant test data. NSA is one of the most successful methods of AIL and it has been used in many applications, but its utilization in the field of test data generation is still limited. The algorithm consists of two stages, generation stage and detection stage, as explained in the previous section. In this research, we focus on the generation stage, which is responsible for generating the detector sets. In this study, it is used to generate the set of test data or test suite. At the beginning of this stage, a candidate test data is generated randomly using a process to find duplicate data within the set of test data, and if the new test data match any test data in the set, it is eliminated. This process prevents the test data from being redundant. Otherwise, the matching rule is determined by calculating the Hamming distance between the two test data and the minimum distance obtained will be compared with a threshold value. If the distance is less than the threshold value, then this test data is removed, otherwise it is added to the set of test data. This process helps to cover as much of the search area as possible, and here it could cover more paths with limited number of test data for the SUT. This algorithm not only generates a specific number of test data for each path to guarantee test adequacy, but it could also reduce the redundancy of test data. By comparing the results with random test data generation technique, it is noted that this method is a much better technique. Fig. 2 shows the overview of the method.

The path test data generation can be explained as follows: \( P \) represents the program under test, \( U \) represents the program paths, \( S \) is the input search space of \( P \), the generator determines \( x \in S \), where \( x \) is an input test data which caused the path \( u \in U \) to be traversed through the execution, and \( U \) is one of program paths \( U \).

Application of NSA in generating the set of test data represented by \( D \) begins with randomly generating a candidate population of test data that will then mature during the iterative process such that \( \forall d_i \in D \) and \( \forall x \in S \), where \( d_i \) is a test data within the set \( D \) which covered the path \( u \). The matching rule is then to utilize the distance between the test data input with all the test data in the set, the minimum distance, \( \tau \), is checked, which represents the threshold value of affinity in order to eliminate and remove redundant detectors in test data.

While the matching threshold (affinity matching distance) represents the allowable variation between the self samples and detectors, it is important and affects the NSA performance. With large threshold, more detectors are necessary to cover a larger region of the non self space which are not neighbours to the self space. On the other hand, if the threshold is too small, it may not be possible to generate a reasonable sized detector set from the available self set. The selection of a suitable value is desirable for optimal performance of the algorithm. The determination of the threshold value is done by trial and error and the value which gives the best performance is selected. Thus, experimentally, the best threshold value used in the proposed method was set to 0.5.

The proposed algorithm selects the test data with maximum distance which allows the algorithm to cover the whole search space, improving its ability to find test data and cover as many paths as possible. The iteration is continued until the stopping criterion has been met. This criterion is fulfilled when the maximum number of detectors has been met or all paths of the program have been covered. The steps of implementing NSA in test data generation algorithm and its detailed flow chart are shown in Figs. 3 and 4 respectively.

### 5. Empirical evaluation

#### 5.1. Empirical setup

To evaluate the performance of the proposed method, a comparison has been made between programs that are popular from the
Table 1
Description of the benchmark programs.

<table>
<thead>
<tr>
<th>Benchmark Abbreviation</th>
<th>Description</th>
<th>Number of Arguments</th>
<th>Lines of Code</th>
<th>No. of Loops</th>
<th>Number of Decision Nodes</th>
<th>Number of Paths</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trityp</td>
<td>Triangle Type: aims to receive three numbers as input and deciding the type of triangle they represent: equilateral, isosceles, scalene, or no triangle.</td>
<td>3</td>
<td>18</td>
<td>0</td>
<td>6</td>
<td>7</td>
<td>[4,5,21,44]</td>
</tr>
<tr>
<td>Mid</td>
<td>Find the middle value between three values.</td>
<td>3</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>[5,21]</td>
</tr>
<tr>
<td>QuadEq</td>
<td>Find the root of a quadratic equation and specify whether they are real or imaginary.</td>
<td>3</td>
<td>28</td>
<td>0</td>
<td>6</td>
<td>7</td>
<td>[5,32]</td>
</tr>
<tr>
<td>STD-Scale</td>
<td>Output the scale of a student based on the marks in three subjects.</td>
<td>3</td>
<td>46</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>[21]</td>
</tr>
<tr>
<td>P-lie</td>
<td>Find if x and y point lie on x-axis, y-axis or origin.</td>
<td>2</td>
<td>21</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>[21]</td>
</tr>
<tr>
<td>MinMax</td>
<td>Find the max and min value in array of integers and floats numbers.</td>
<td>1</td>
<td>16</td>
<td>1</td>
<td>3</td>
<td>13</td>
<td>[4,5,21]</td>
</tr>
<tr>
<td>L-Search</td>
<td>Searching a key in array of integer, floats, characters and string.</td>
<td>2</td>
<td>27</td>
<td>3</td>
<td>5</td>
<td>32</td>
<td>[4,21]</td>
</tr>
<tr>
<td>BubSort</td>
<td>Sort an array of elements in an increasing order.</td>
<td>1</td>
<td>20</td>
<td>2</td>
<td>1</td>
<td>15</td>
<td>[4,42]</td>
</tr>
<tr>
<td>NumDays</td>
<td>Calculating number of days between two dates.</td>
<td>6</td>
<td>233</td>
<td>3</td>
<td>43</td>
<td>86</td>
<td>[5,42,44]</td>
</tr>
<tr>
<td>CalDay</td>
<td>Calculates what day of the week for a specific date.</td>
<td>3</td>
<td>119</td>
<td>0</td>
<td>24</td>
<td>25</td>
<td>[1,44]</td>
</tr>
<tr>
<td>AllTrue32</td>
<td>Checks the Boolean values of an array if are all true.</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>32</td>
<td>2^{32}</td>
<td>[45]</td>
</tr>
</tbody>
</table>

literature. These programs have been widely used as test studies by researchers in the search-based software testing (SBST) field. The programs are written in JAVA and C++, and they contain a selection of IF, IF-ELSE statements with sequence, and nested complex structures like nested IF statements or deeply nested predicates. These programs also contain conditions with equality conditions or relational operators, i.e., $\{=, \neq, <, >, \leq, \geq\}$ and compound conditions of AND and OR logical operators, as well as existing arithmetic operators, i.e., $\{+,-,/,^\}$ which made these programs appropriate in testing several techniques of test data generation. A complex data structure with different data types, i.e., integers, floats, characters and string have also been included in these programs [4,21]. Table 1 illustrates these programs. For each program, the table lists a short description of the program, number of arguments in each program, lines of code (LOC), number of decision nodes, number of paths, and source. The common parameter of the method is the maximum number of detectors which was set to 5000. The experiment was run on 64-bits, 2.10 GHz Intel® Core™ i7 computer running Microsoft Windows 7 with 8 GB memory. The method was executed in the Delphi 5 platform.

5.2. Results and discussion

In this work, NSA is used for generating the test data automatically. The results show NSA’s ability in generating test data which can minimize the number of generations needed to cover the programs’ paths, reducing testing time. To investigate the effect of the proposed algorithm, we compared the results of this method with those obtained from random test data generation method.

Generally, in NSA, the number of generated detectors in each generation or each run time (population size) has a large effect on the efficiency. The efficiency is represented by the number of generations needed to cover all paths of the program while the effectiveness is represented by the number of generated test data which needs to cover all paths of the program.

The increase in population size will increase the effectiveness of the algorithm, but an increase in the number of the generations will affect the required testing time.

If the population size is increased, the coverage percentage of the paths will be increased. However, at the same time, the number of test data will increase and this is not desirable. When the population size is decreased, the number of generations will be increased, which means more time needed, so we need to choose a sufficient number of detectors.

The algorithm has been performed and ran several times with different population sizes (different number of detectors in each generation) for each stopping time from 50 to 5000 generated detectors, and the results have been recorded (number of test data that cover the difficult path). In each run, the number of test data generated, number of generations, and execution time have been recorded, and the results showed that the best solution for the algorithm is when the population size is 500 and the number of generation is 5 we got more than 600 times to cover the equalateral path. The results showed that when the population number exceeds 500 or the generation exceeds 5, the relations move in linear fashion, and will decrease with the effectiveness which is represented by the number of test data needed to cover all paths and linearly increase the number of generations which affects the efficiency of the method.

Fig. 5 displays the relationship between the number of test data needed to cover the Equilateral path according to population size and number of generations. This is the most difficult path in the famous triangle classification test program (TriTyp).

The experiments have been done on well-known benchmark programs, and in this section we will illustrate the results for each testing program. Triangle type classifier (TriTyp), which is the most popular program in software testing, received three input num-
Fig. 5. Relationship between the number of test data that covers the equilateral path with (a) number of generations; (b) population size.

Fig. 6. The source code and control flow graph of Trityp.

<table>
<thead>
<tr>
<th>Path no.</th>
<th>Path Generated</th>
<th>Triangle Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0→1→13→14</td>
<td>Not triangle</td>
</tr>
<tr>
<td>2</td>
<td>0→1→2→12→14</td>
<td>Not triangle</td>
</tr>
<tr>
<td>3</td>
<td>0→1→2→3→4→5→14</td>
<td>Equilateral</td>
</tr>
<tr>
<td>4</td>
<td>0→1→2→3→4→6→14</td>
<td>Isosceles</td>
</tr>
<tr>
<td>5</td>
<td>0→1→2→3→7→8→14</td>
<td>Isosceles</td>
</tr>
<tr>
<td>6</td>
<td>0→1→2→3→7→9→10→14</td>
<td>Isosceles</td>
</tr>
<tr>
<td>7</td>
<td>0→1→2→3→7→9→11→14</td>
<td>Scalene</td>
</tr>
</tbody>
</table>

Please cite this article in press as: S.M. Mohi-Aldeen, et al., Application of Negative Selection Algorithm (NSA) for test data generation of path testing, Appl. Soft Comput. J. (2016), http://dx.doi.org/10.1016/j.asoc.2016.09.044
eral types of triangle. Thus the probability of selecting three inputs randomly that traverse the equilateral path is $2^{17}/2^{48}$ which is approximately equal to $1/2^{31}$. This makes it a typical program to evaluate several methods in software testing [43]. The source code and control flow graph (CFG) of the program are shown in Fig. 6. As shown, the most difficult path to be covered in the program is the equilateral triangle.

An example of how to represent the test data of the triangle classification program and how to calculate Hamming distance is presented in Fig. 7, where a, b and c represents the value of three input numbers which are used to decide the type of triangle: whether it is not a triangle, isosceles, equilateral or scalene. Two test data have been presented and converted into binary form, and their Hamming distance is calculated.

The effectiveness of the program that is represented by the number of test data needed to cover each path of the program is shown in Fig. 8. The figure involves 100 detectors and a maximum of 10 generations.

From the figure, we can see that the proposed method overcame random testing in generating data for equilateral path, achieving 100% of path coverage while random testing covered only 75%. The number of test data and execution time for all paths tested also decreased at least 90% as shown in Fig. 9.

Fig. 10 presents the number of test data and paths covered using NSA and random testing for each benchmarking programs.

As shown in the figures, NSA has more chance in generating test data for all paths even for difficult program paths.

The proposed method decreases the number of test data by more than 75% and also reduces the time required to cover all paths of the programs by more than 85% as shown in Fig. 11.

To evaluate the efficiency and the effectiveness of the proposed method, we compared the proposed method’s results with Alzabidy’s [21] study which have the same type of coverage and used the same five well-known benchmark programs. In addition, in random testing, the comparison was based on the average values of two performance metrics which are average number of test data and average number of generations needed for total coverage. The results have been presented in Table 2.

The table shows that the generated test data using the proposed method covers all programs paths with minimum number of iteration and reduced the average number of test data produced. In the P-lie program, GA could not achieve full path coverage (denoted as NA) since it failed to generate the test data which satisfy the path of the points that lie in the origin because the input (0,0) was not generated within the maximum number of generations. The table also shows the path coverage percentage of the programs compared with random testing and from the results, NSA has highest coverage percentage.

Figs. 12 and 13 described the number of test data needed to cover all paths of the programs and the percentage of path coverage in each program by using random testing and NSA respectively. It is clear from Fig. 12 that the number of generated test data in NSA is lesser than random testing technique, but the proposed method could not cover all paths covered by the NumDays program because of the complex structure of this program, while, Alltrue program needs more than 90000 detector to cover all paths. Fig. 13 illustrates the percentage of path coverage for all programs used in the experiment.

For further evaluation, we compared the triangle classification program results with previous works. Mao et al. [44] used ACO based test data generation and the results were compared with SA, GA and PSO. The findings indicate that ACO outperformed SA and GA, while it is comparable to PSO. Other comparisons were made with Singh [18] and Alzabidy [21] which used GA to generate test data for path coverage. Table 3 shows the comparison between our method with previous works.

From the table of comparison, the results show that NSA needs less than the average number of test data since it has filtered and checked for duplicate test data, and lesser number of generations. However, it is also shown that ACO outperforms NSA in execution time.

In order to increase the validity of the results, we implemented a statistical analysis, T-test, of the results to compare the two meth-
ods. Table 4 shows the results for all programs with 10 runs for each program.

The values in Table 4 were obtained by computing the number of test data which have been generated by the programs from 10 runs of each testing program. The table shows the mean, median, standard deviation (Std-Dev), minimum and maximum number of test data that were generated by using both algorithms. As shown in the table, the proposed method achieved better results and better efficiency compared with random testing. The small Std-Dev values in NSA indicates that the algorithm could achieve better coverage of the program paths with less number of test data and less number of generations. This decreases the testing time because some paths are simpler to cover using NSA.

The proposed NSA method could perform better in covering programs paths even when dealing with difficult paths that include Nested-if statements, loops, different data types, and even string compared to other methods. This method also produce good results in reducing the number of test data generated, the only issue shortcoming of this method is it could not cover the whole paths of the NumDays program (it only covers 84% of the program’s paths).

6. Conclusions

This paper used NSA in generating the test data automatically to achieve path coverage criterion. List of benchmark programs have been used to validate the efficiency and effectiveness of the
Table 2

Experimental results of all programs compared with Alzabidi [21].

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Average Number of Generations needed for total coverage</th>
<th>Average Number of Test data needed for total coverage</th>
<th>Average Path Coverage%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trityp</td>
<td>50</td>
<td>3</td>
<td>10.3</td>
</tr>
<tr>
<td>Mid</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P-Lie</td>
<td>49</td>
<td>7</td>
<td>NA</td>
</tr>
<tr>
<td>QuadEq</td>
<td>39</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>STD-Scale</td>
<td>26</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>MinMax</td>
<td>5</td>
<td>2</td>
<td>–</td>
</tr>
<tr>
<td>L-Search</td>
<td>7</td>
<td>3</td>
<td>–</td>
</tr>
<tr>
<td>L-SearchChar</td>
<td>2</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>L-SearchString</td>
<td>7</td>
<td>3</td>
<td>–</td>
</tr>
<tr>
<td>BubSort</td>
<td>4</td>
<td>2</td>
<td>–</td>
</tr>
<tr>
<td>CalDay</td>
<td>87</td>
<td>54</td>
<td>–</td>
</tr>
<tr>
<td>NumDays</td>
<td>NA</td>
<td>NA</td>
<td>–</td>
</tr>
<tr>
<td>AllTrue32</td>
<td>NA</td>
<td>940</td>
<td>–</td>
</tr>
</tbody>
</table>

Fig. 12. Average number of test data to cover all paths of the programs.

Fig. 13. Path coverage percentage of the programs.

Table 3

Comparison with previous works.

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of Generations needed for total coverage</th>
<th>Average Time (ms)</th>
<th>Average Number of Test data</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSA</td>
<td>3</td>
<td>31</td>
<td>229</td>
</tr>
<tr>
<td>ACO [44]</td>
<td>6</td>
<td>6.22</td>
<td>–</td>
</tr>
<tr>
<td>GA [18]</td>
<td>27</td>
<td>4000</td>
<td>24421</td>
</tr>
<tr>
<td>GA [21]</td>
<td>10</td>
<td>–</td>
<td>11091</td>
</tr>
</tbody>
</table>

Method in generating and reducing the test data. The experimental results showed that NSA is more efficient and more effective than random generation because of the method’s ability to move the search space to the optimal search range with fewer number of generations. The results also showed that NSA has the capability to reduce the number of test data when compared with ACO.
Table 4
Statistical comparison of the results.

<table>
<thead>
<tr>
<th>Program</th>
<th>Method</th>
<th>Runs</th>
<th>Mean</th>
<th>Median</th>
<th>Std-Dev</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trityp</td>
<td>Random</td>
<td>10</td>
<td>4945</td>
<td>3187.5</td>
<td>3246.63</td>
<td>2520</td>
<td>10987</td>
</tr>
<tr>
<td></td>
<td>NSA</td>
<td>10</td>
<td>229</td>
<td>230.5</td>
<td>39.07</td>
<td>153</td>
<td>287</td>
</tr>
<tr>
<td>Mid</td>
<td>Random</td>
<td>10</td>
<td>6</td>
<td>23.5</td>
<td>7.69</td>
<td>13</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>NSA</td>
<td>10</td>
<td>5</td>
<td>3.8</td>
<td>3</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>P-λie</td>
<td>Random</td>
<td>10</td>
<td>4848</td>
<td>4712.5</td>
<td>2037.12</td>
<td>1800</td>
<td>9048</td>
</tr>
<tr>
<td></td>
<td>NSA</td>
<td>10</td>
<td>691</td>
<td>675.5</td>
<td>86.9</td>
<td>590</td>
<td>845</td>
</tr>
<tr>
<td>QuadEq</td>
<td>Random</td>
<td>10</td>
<td>3816</td>
<td>3524.5</td>
<td>2509.91</td>
<td>1028</td>
<td>8023</td>
</tr>
<tr>
<td></td>
<td>NSA</td>
<td>10</td>
<td>362</td>
<td>383.5</td>
<td>91.06</td>
<td>225</td>
<td>460</td>
</tr>
<tr>
<td>STD-Scale</td>
<td>Random</td>
<td>10</td>
<td>2534</td>
<td>2085.5</td>
<td>2406.89</td>
<td>1019</td>
<td>9122</td>
</tr>
<tr>
<td></td>
<td>NSA</td>
<td>10</td>
<td>215</td>
<td>213</td>
<td>74.24</td>
<td>118</td>
<td>293</td>
</tr>
<tr>
<td>MinMax</td>
<td>Random</td>
<td>10</td>
<td>497.2</td>
<td>512</td>
<td>82.2905</td>
<td>321</td>
<td>618</td>
</tr>
<tr>
<td></td>
<td>NSA</td>
<td>10</td>
<td>113.6</td>
<td>95</td>
<td>78</td>
<td>16</td>
<td>244</td>
</tr>
<tr>
<td>L-Search</td>
<td>Random</td>
<td>10</td>
<td>698.6</td>
<td>319</td>
<td>247.2</td>
<td>205</td>
<td>985</td>
</tr>
<tr>
<td></td>
<td>NSA</td>
<td>10</td>
<td>290.5</td>
<td>260</td>
<td>116.9</td>
<td>209</td>
<td>585</td>
</tr>
<tr>
<td>L-SearchChar</td>
<td>Random</td>
<td>10</td>
<td>157.7</td>
<td>137.5</td>
<td>48.2</td>
<td>101</td>
<td>235</td>
</tr>
<tr>
<td>L-SearchString</td>
<td>Random</td>
<td>10</td>
<td>77.5</td>
<td>70</td>
<td>32.9</td>
<td>43</td>
<td>147</td>
</tr>
<tr>
<td>BubSort</td>
<td>Random</td>
<td>10</td>
<td>242.8</td>
<td>500</td>
<td>386.6</td>
<td>236</td>
<td>1290</td>
</tr>
<tr>
<td></td>
<td>NSA</td>
<td>10</td>
<td>379.9</td>
<td>177.5</td>
<td>2076.3</td>
<td>132</td>
<td>215</td>
</tr>
<tr>
<td>CalDay</td>
<td>Random</td>
<td>10</td>
<td>5238</td>
<td>7787.5</td>
<td>4720.3</td>
<td>3106</td>
<td>18168</td>
</tr>
<tr>
<td></td>
<td>NSA</td>
<td>10</td>
<td>6038</td>
<td>4436.5</td>
<td>5490.1</td>
<td>1930</td>
<td>19511</td>
</tr>
<tr>
<td>AllTrue32</td>
<td>Random</td>
<td>10</td>
<td>93966</td>
<td>95600</td>
<td>5356.2</td>
<td>85630</td>
<td>99172</td>
</tr>
</tbody>
</table>

and GA used in previous works. For future works, the proposed test data generation using NSA algorithm could be enhanced by using adequate fitness function or hybridization with other optimization algorithms to overcome its lack in terms of full coverage.

Acknowledgments

We would like to thank Ministry of Higher Education Malaysia for sponsoring the research through the Fundamental grant Search-Based Mobile Applications Test Data Generation Model in Universiti Teknologi Malaysia for providing the facilities and support for the research. The author also would like to thank Ministry of Higher Education & Scientific Research of Iraq and University of Mosul for sponsoring the research.

References


Please cite this article in press as: S.M. Mohi-aldeen, et al., Application of Negative Selection Algorithm (NSA) for test data generation of path testing, Appl. Soft Comput. J. (2016), http://dx.doi.org/10.1016/j.asoc.2016.09.044


Please cite this article in press as: S.M. Mohi-Aldeen, et al., Application of Negative Selection Algorithm (NSA) for test data generation of path testing, Appl. Soft Comput. J. (2016), http://dx.doi.org/10.1016/j.asoc.2016.09.044