NEAT Algorithm for Testsuite generation in Automated Software Testing

H.L Praveen Raj[1], K. Chandrasekaran[2]
[1,2] Department of Computer Science and Engineering,
[1, 2] National Institute of Technology Karnataka Surathkal
Email: [1] hlpr98@gmail.com, [2] kch@nitk.ac.in

***Abstract*— Software testing is one of the most essential and an indispensable part of Software production life cycle. Software testing helps in validating if the product meets with the requirements or not, and also testing helps to validate the performance of the product. Unfortunately, this process takes up about 50% of the production time and budget, due to its laboriosity. Hence, in order to reduce the time it takes, Automated Software Testing becomes essential. Here we propose a novel idea of using Machine Learning for automatically generating the test suites. In this paper we present an approach that uses NEAT (Neuroevolution of Augmenting Topologies) Algorithm to automatically generate new test suites or for improving the coverage of already produced test suite. Our approach automatically generates test suites for white box testing. White box testing refers to testing of the internal structure and the working of the Software Under Test.**

Keywords— Genetic Algorithm, NEAT Algorithm, Machine Learning, Software testing, statement coverage, branch coverage, loop coverage, Software Under Test.

I. Introduction

Software testing an essential part of Software production life cycle. Unfortunately, this process is very time consuming. Hence, in order to reduce the time it takes, Automated Software Testing becomes essential. Generation of Test Suites automatically is a very difficult task. Automated testing becomes difficult due to 1. The Diversity of the Softwares, 2. Different approaches and conventions followed in each software, 3. The presence of different types of constructive modules in the software, for instance, loops, branches, functions; etc. There have been several approaches suggested for overcoming these difficulties, namely symbolic execution, search-based testing, model-based testing, generation of Context Flow Graphs and many more[25]. The chief evaluating parameter for any Test suite generation approach is the *Coverage* achieved by the Test suite. Here "Coverage" refers to the statement, branch and loop coverages of the source code of the SUT[14].

There has been some amount of research in the field of automated software testing using Artificial Intelligence techniques. Some of the proposed techniques include Genetic Algorithms[7][8][9][10], Taboo search[18], Simulated annealing, Ant colony Optimisation[5][11] etc. These techniques have proved that use of the evolutionary algorithms is suitable for generation of test suites[1][2]. NEAT Algorithm is one such evolutionary algorithm[3].

The advantage of the use of NEAT Algorithm is that it combines the advantages of Artificial Neural networks and the evolutionary genetic algorithms. NEAT Algorithm helps us to evolve a neural network in both its topology and its weights. This algorithm evolves the neural network in a controlled manner, by following a specific mutation rate.

An evolutionary network is apt for the problem of Automated test suite generation because it helps to recognise and capture, the nuances and individuality of a software better than what a static algorithm could do. The neural network would help in easy generation of test cases by boiling down the process into a simple equation in terms of *weights of connection* between the layers of the network.

The problem of automated software testing can be represented as follows:

1. Create a test suite that can achieve the highest possible coverage of the Software Under Test (SUT).
2. Reduce the number of test cases in the test suite but keep the coverage same.
3. Generate the test suite in a short duration.

The scope of this problem in this research can be represented as follows:

1. To propose a novel approach for automatic generation of test suites.
2. To use Triangle Classifier problem as an example for analysing our approach.
3. To analyse our approach in terms of speed (time taken to generate test suites) and coverage achieved by the test suite.
4. To do comparative tests with other existing approaches.

In the sections that follow, we provide a literature review, a brief description of the problem, a description of the proposed algorithm and a discussion on the experimentation process and the results obtained.

II. Literature Review

In the beginning of the research, an extensive literature review was carried out regarding Software testing, automated software testing, genetic algorithms and their application in automated software testing. This section provides the gist of the literature review conducted.

A. Software testing

Software testing is an indispensable part of the Software production life cycle. It is the process of verification and validation of the Software Under Test (SUT). During this process, the testers verify if the SUT does meet the requirements specified by the stakeholders. Thus, Software Testing is a phase which ensures quality and performance of the product to the stakeholders. Testing also helps in identification of bugs in SUT.

Software Testing can be classified broadly into two methods, namely, White box testing and Black box testing[21]. White box testing, also called Structural testing, is the testing method where the internal structure and the working of the SUT are tested. On the other hand. Black box testing, also know as Behavioural testing, refers to the testing methodology where the functionality of the SUT is tested.

Software testing can be done both manually or automatically[15]. Manual testing is very difficult, laborious and time consuming. Hence Automated testing becomes very important[25]. Automated testing is performed using different approaches like search-based testing, testing using symbolic execution, combinatorial testing, adaptive random testing, model-based testing and many more[25].

B. Genetic Algorithms

Genetic Algorithms belong to a larger class of algorithms called Evolutionary algorithms[22]. Genetic Algorithms are metaheuristics which are based on natural selection - a biological process where the fittest individual is selected, in nature over an unfit individual to thrive further. Genetic Algorithms are used to solve many real-world problems, such as optimisation problems, search problems etc[24].

Genetic Algorithms rely on three operators, namely 1. Mutation, 2. Crossing over and 3. Selection, for the generation of solution to the problem[23]. In genetic algorithm, initially a population starts with possible solutions to the problem under consideration and the fittest individuals among them are selected as parents and allowed to reproduce either sexually or asexually which gives rise to new generation. In each generation mutation is introduced to obtain new features (genotypes). In each generation selection is done such that the fitness follows a non-decreasing trend. Here each individual solution is called a Phenotype and the set of characteristics they have are called Genotype[22][23].

C. Automated testing using Genetic Algorithms

There has been a lot of research in the field of Automated Software Testing. One of the methods used for automated generation of Test suite is Genetic Algorithms[6][7][8][9][10]. The methodology used for generation of the Test-suite is as follows[6]

1. Generation either Context flow graph or Control dependence graph.
2. Creation of an initial population which may be a naive test suite.
3. Evolution of the population towards a better solution
4. Obtaining the fittest individual
5. Generation of the test suite from the fittest individual obtained.

Genetic algorithms improves the time taken for testing significantly when compared with other testing methodologies like manual testing, other automated testing methods etc[2].

D. NEAT Algorithm

NEAT Algorithm is an evolutionary algorithm more specifically it is a genetic algorithm. It belongs to a class of algorithms called TWEAN (Topology and Weight Evolving Artificial Neural network). This algorithm is used generation of evolving neural networks[3].

NEAT Algorithm takes advantage from both the evolving topology of the neural network and also the evolving weights. NEAT is based on three main techniques, namely 1. It tracks history of the genes, 2. It applies speciation in order to protect new traits 3. It develops topology incrementally from simple initial design[3]. NEAT uses Direct encoding scheme to encode the features of the neural network into the Genotypes of the Genetic algorithm and each network into a Phenotype of Genetic algorithm, this implies that each connection and neuron in the neural network are represented explicitly[3].

During the evolution process in NEAT implementation, the initial neural network generated would be simple perceptron like feed forward or recurrent neural network of input neurons and output neurons. As the evolution continues, the topology of the neural network becomes more and more complex, which is achieved by either added new neurons to the connection path or by addition of new layers of neurons i.e. addition of new neurons in between other neurons in the network.

III. Problem Descritpion

Software Testing is an essential and indispensable part of Software life cycle. Unfortunately software testing is a very difficult, laborious and time consuming process. Hence automated software testing becomes essential, so that the testing phase takes up lesser time and resources. Some of the chief reasons for why Automated Software testing is a very difficult task are, 1. Softwares are diverse. Each software has its own specifications and intricacies, which need to be captured during the testing phase and tested thoroughly 2. Different approaches and conventions followed in each software, 3. The presence of different types of constructive modules in the software, for instance, loops, branches, functions; etc.

The chief aims and goals of automated generation of Test suites are,

a) Obtain the highest possible (preferably 100%) statement, branch and loop coverages: In white box testing one of the measures for a good test suite is the Coverage it achieves. Coverages are of three types, namely, statement, branch and loop coverages. Detailed explainationation for each is provided in the following section.

b) Generate the test suite as fast as possible: Time is the most important factor during the Software testing phase, hence the generation of very good test suites by expending very short durations of time is very esseintal.

c) Reduce the number of test cases in the test suite but keep the coverage same: The execution of each test case in a test suite may take a lot of time depending on itss complexity and the software's complexity. Hence in order to have a faster testing phase, we need to be able to run the test cases as fast as possible. So, the test suite generator must be able to generate test suites which can provide high coverage but contain almost minimum number of test cases. The term minimum here is proportional to the number structural features or components to be tested.

There have been several approaches proposed to solve this problem, for instance, Combinatorial testing[25], Testing using Symbolic execution, Search-based testing[25], Adaptive random testing, Metaheuristics based testing[18], Model-based test case generation[19] and Genetic algorithm-based testing[7][8][9][10]. Each of these methods have their own advantages and disadvantages, some of them are extremely fast but generate test suites which achieve very poor coverage, some take longer time to generate test suites but can provide better coverages. But none of these methods answer the problem of achieving all the three goals of Automated software testing simultaneously.

For the above-mentioned reasons, we answer this problem by proposing a novel idea which uses Recurrent Artificial Neural Network along with the NEAT Algorithm for the automatically generating the Test Suite for Software Under Test (SUT).

IV. NEAT FOR TEST-SUITE GENERATION

In this section we describe our algorithm and the goals that we try to achieve with it. Here we discuss certain key aspects of the algorithm too.

A. The Algorithm

Here we propose a novel approach that uses a Recurrent Artificial Neural Network along with the NEAT Algorithm for the automatically generating the Test Suite for Software Under Test (SUT).

The Algorithm can be described as the following steps

a) Creating a very naive test suite (T): This test-suite may be created using any of the automated testing techniques mentioned in the literature review or may be created at random without any concern on the coverage or the result.

b) Creating an artificial neural network: The artificial neural network used can be either a Feed Forward Neural network or a Recurrent Neural network. The topology of the network would be fully connected network with any number of hidden layers. The number of hidden layers depends upon the complexity of the computation required to generate the test cases.

c) Setting the parameters: The ranges of weights and biases must be specified. These ranges depend upon the feasible range of input values of the SUT. The mutations rates also need to be specified. The number of generation of the algorithm must also be specified. Some of the other parameters include, maximum stagnation, i.e max number of continuous generations with no improvements in fitness, survival threshold, i.e the fraction of each species allowed to reproduce each generation, number of individuals in each generation, etc.

d) Defining the fitness function and fitness threshold: The fitness function specified here would be used to evaluate the fitness of each genome. The "fitness threshold" is the maximum achievable fitness, one a genome reches this fitness, the algorithm stops and returns this genome as the fittest genome.

e) Applying NEAT: The final step is to apply NEAT algorithm to this topology and parameters and allowing the algorithm to evolve to provide the fittest individual. While evolving, if an individual's fitness reaches the Fitness threshold, then the evolution stops and outputs the this individual. If this scenarios never occurs, then the evolution stops after completing the maximum number of generations specified, and outputs the genome with highest fitness in the current generation.

f) Evolution process: During evolution, we use the test-suite T as input to the neural network and get the newly generated test-suite N as the output and evolve to attain the highest fitness.

g) Generation of Test-Suite: Here we use the best genome obtained from the evolution process to generate the new test-suite. We feed the test-suite T as the input to the best genome, and the genome returns the best Test-suite N. The test-suite N would be the test-suite with highest statement, branch and loop coverage.

B. Aim and Goal of the Algorithm

The aim and goal of this algorithm are,

- Obtain the highest (preferably 100%) statement, branch and loop coverages using the least possible test cases.
- Generate the test suite as fast as possible.

This Algorithm does achieve both these goals since,

- The fitness function depends upon the statement, branch and loop coverages obtained.
- The NEAT algorithm introduces mutations in every generation, which helps in faster approach to the *Fitness threshold* unlike a traditional artificial neural network training process.
- The algorithm achieves the highest coverage with almost the least number of test cases

C. The Fitness function

The fitness function must capture all the desirable aspects of a Test suite, namely the coverages. The fitness function that we used during the experimentation process is,

$$\boldsymbol{Fitness = stmt * 2 + br * 3 - (1 - br) * 5 + tot * 5 + lo * 3 - (1 - lo) * 4} \qquad (1)$$

Here, stmt = % statement coverage
br = % branch coverage
lo = % loop coverage
tot = overall coverage (an average of statement, branch, loop coverages)

a) Statement coverage: Statement coverage refers to the coverage of all the lines in the source code the software. Here a line is said to be covered iff that line is executed atleast once. Percentage statement coverage provides the perrcentage of source code lines covered with respect to the number of lines in the source code.

b) Branch coverage: Branch coverage refers to the coverage of the branches in the source code.

$$Branch\ coverage = \begin{cases} true, & iff\ all\ the\ cases\ of\ the \\ & condition\ are\ exectued \\ false, & otherwise \end{cases} \qquad (2)$$

Percentage loop coverage provides the percentage of branches covered completeletly.

c) Loop Coverage: Loop coverage refers to the coverage of all the statements in a loop. A loop is said to be covered iff, all the statements in the loop are covered atleast once in each iteration of the loop, and the loop is iterated fot the number of iterations it is stipulated to. Hence the execution of "break" statement would result in unsuccessful coverage of the loop.

d) Ovearll coverage: Overall coverage is the cumulative of statement, loop and branch coverages. It also takes, partial loop and branch coverages, into account.

V. Experiments and Results

A. Experimentation methodology

We implemented our algorithm in Python 3.6. We used the "*neat-python*" library for implementing NEAT Algorithm in our experiment. For the analysis of statement, branch and loop coverages, we used an Open-source Python library named "*Coverage.py*" library. The graphs were obtained using the "*MATPLOT*" library.

B. The Experiment

For the experimentation purpose we used the "*Triangle Classification"* program which is shown the figure.1.

The implementation and experimentation can be described as the following steps:

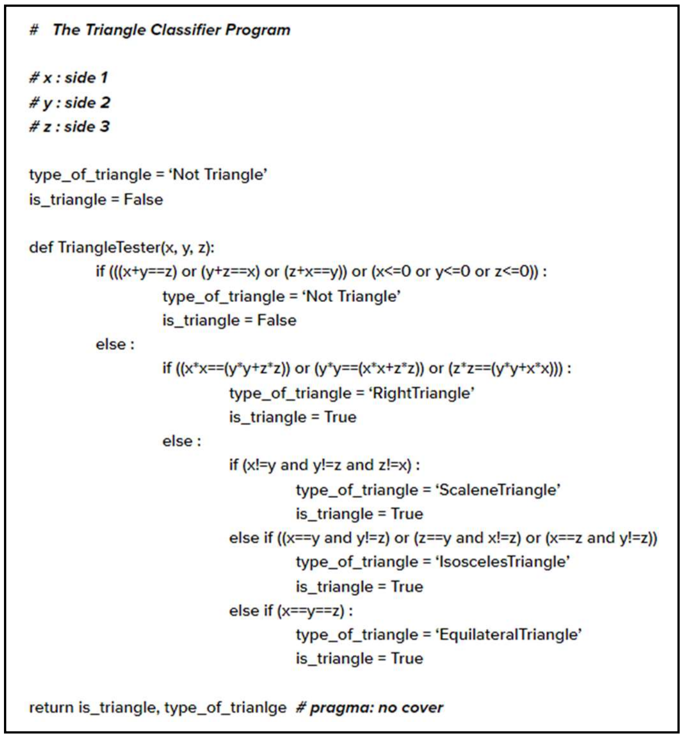
a) The naive test suite used: We used a randomly generated naive test suite, represented in figure.2

b) The Artificial neural network: We used a Recurrent neural network with 4 hidden layers, 9 inputs and 3 outputs. It had full undirected connection with all the nodes in the beginning.

c) Setting parameters: The parameters of the network were as follows,

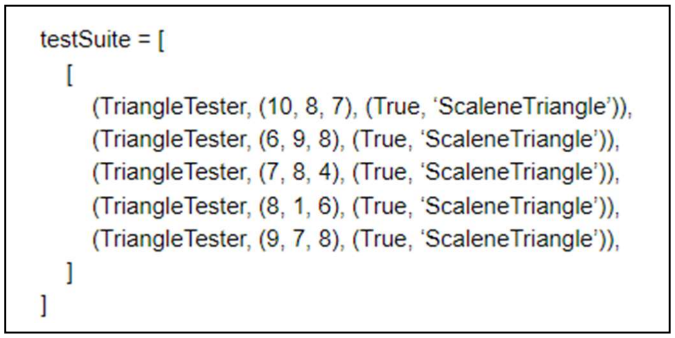
Range of weights = -10 to 100
Range of biases = 1 to 100
Activation function = Random choice between exponential, square, cube, softplus and tanh.
Max stagnation = 2
Survival threshold = 0.2
Individuals per generation = 50
Max number of generations = 30; etc.

d) Fitness function and Fitness threshold: The Fitness threshold was 9.675 and the Fitness function was the same as mentioned before. The fitness function used, depends upon the coverage achieved by the Test suite. Here we evaluated the coverage scores using an Open-source Python library named "Coverage.py".



```
#  The Triangle Classifier Program

# x : side 1
# y : side 2
# z : side 3

type_of_triangle = 'Not Triangle'
is_triangle = False

def TriangleTester(x, y, z):
        if (((x+y==z) or (y+z==x) or (z+x==y)) or (x<=0 or y<=0 or z<=0)) :
                type_of_triangle = 'Not Triangle'
                is_triangle = False
        else :
                if ((x*x==(y*y+z*z)) or (y*y==(x*x+z*z)) or (z*z==(y*y+x*x))) :
                        type_of_triangle = 'RightTriangle'
                        is_triangle = True
                else :
                        if (x!=y and y!=z and z!=x) :
                                type_of_triangle = 'ScaleneTriangle'
                                is_triangle = True
                        else if ((x==y and y!=z) or (z==y and x!=z) or (x==z and y!=z))
                                type_of_triangle = 'IsoscelesTriangle'
                                is_triangle = True
                        else if (x==y==z) :
                                type_of_triangle = 'EquilateralTriangle'
                                is_triangle = True

return is_triangle, type_of_trianlge  # pragma: no cover
```

Fig. 1. Triangle Classifier Program



```
testSuite = [
    [
        (TriangleTester, (10, 8, 7), (True, 'ScaleneTriangle')),
        (TriangleTester, (6, 9, 8), (True, 'ScaleneTriangle')),
        (TriangleTester, (7, 8, 4), (True, 'ScaleneTriangle')),
        (TriangleTester, (8, 1, 6), (True, 'ScaleneTriangle')),
        (TriangleTester, (9, 7, 8), (True, 'ScaleneTriangle')),
    ]
]
```

Fig. 2. Naive Test-suite T

e) Generation of test-suite: We applied the NEAT algorithm to this neural network and trained the network for several generation. The initial neural network was a fully connected undirected Recurrent neural network with 4 hidden layers. The training process took 67 sec for completing 30 generations, 108 sec for completing 500 generations, 134 sec for completing 800 generations and 480 sec for completing 2000 generations. The final evolved neural network can be seen in figure.3.

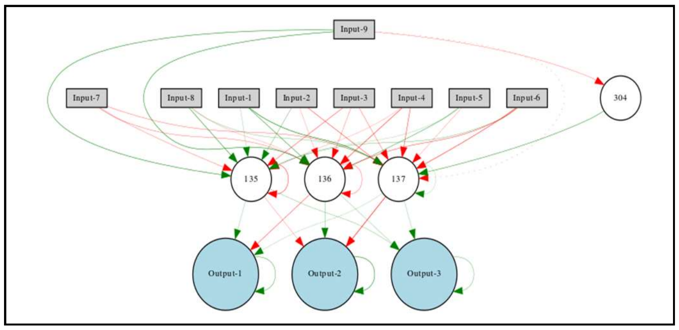
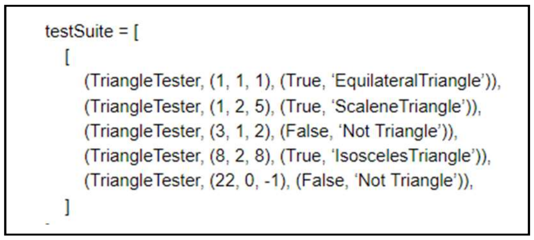


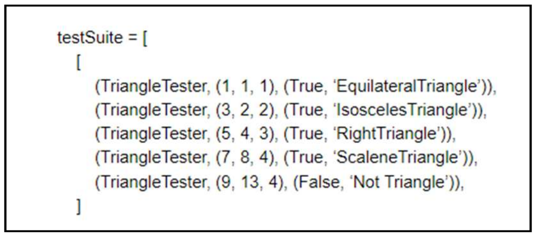
Fig. 3. The final evolved Recurrent neural network after 2000 generations, which started as a fully connected network.

f) The final test suite: Each generation produces multiple indiviiduals, but the test suite taken into consideration is the one generated by the fittest individual in that generation. After training the generator for 30 generations, which took 67 sec, we could obtain the test suite shown in figure.4. And the test suite obtained after training the generator for 800 generations, which took 134 sec is shown in figure.5. The coverages of theses "Best Test-suites N" were analysed using the "Coverage.py" open-source python library. We analysed the statement coverage, branch coverage, loop coverage and the overall coverage provided by each test suite.



```
testSuite = [
  [
    (TriangleTester, (1, 1, 1), (True, 'EquilateralTriangle')),
    (TriangleTester, (1, 2, 5), (True, 'ScaleneTriangle')),
    (TriangleTester, (3, 1, 2), (False, 'Not Triangle')),
    (TriangleTester, (8, 2, 8), (True, 'IsoscelesTriangle')),
    (TriangleTester, (22, 0, -1), (False, 'Not Triangle')),
  ]
```

Fig. 4. Best Test-suite N obtained after 30 generations.



```
testSuite = [
  [
    (TriangleTester, (1, 1, 1), (True, 'EquilateralTriangle')),
    (TriangleTester, (3, 2, 2), (True, 'IsoscelesTriangle')),
    (TriangleTester, (5, 4, 3), (True, 'RightTriangle')),
    (TriangleTester, (7, 8, 4), (True, 'ScaleneTriangle')),
    (TriangleTester, (9, 13, 4), (False, 'Not Triangle')),
  ]
```

Fig. 5. Best Test-suite N obtained after 800 generations.

C. *Results of the Experiment*

The results of our experiment are represented in Table1 and figure.6. From Table1 it is evident that, greater the number of generations higher would be the fitness.

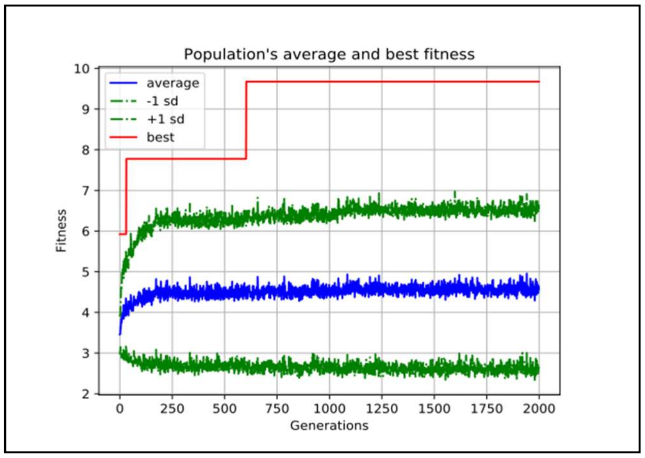


Fig. 6. Progress of best and average fitness values from generation to generation.

During our experiments we also considered the trade off between the time taken for training and generation of Test-suite, and the Coverage obtained, which can be seen in figure.7. It can be inferred from the graph that, the coverage obtained increase with the increase in the number of generations it is trained for and finally obtains the maximum coverage and becomes stagnant. The graph is a non-linear graph because the mutations do not always give an individual which has the ability to provide a higher fitness (which here is directly dependent on the coverages obtained).

A comparison of the performance of our approach was done with that of 1. A Naive automated test suite generation algorithm and 2. EvoSuite-a genetic algorithm based automated Test suite generator. EvoSuite is currently rated as one of the best Automated Test Suite Generators. This comparison shows us that EvoSuite obtains almost the same coverage as our algorithm (trained for 30 generations which took 67 sec),but takes greater time (113 sec) and utilises higher number of test cases (18 test cases) than our approach (our approach could achieve a slightly better coverage with just 5 test cases whose generation took 67 sec).

Our Algorithm outperforms other algorithms in the coverage obtained, by achieving the highest possible coverage of 96%, when run for 800 generations which took 134 sec. The comparison results are represented in figure.8.

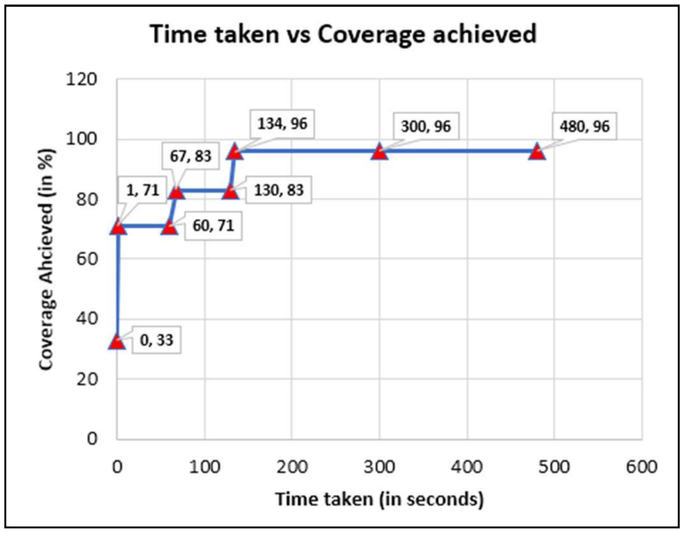


Fig. 7. Comparison between the time taken for training and generation of Test suite and the coverage obtained for the respective Test suite.

The experiments show that this algorithm is able to produce a test suite with nearly the least number of test cases which provides at least as good coverage as that provided by the test suites of other methods, and also takes lesser time to generate the test suite than other methods. This method is a better choice for creation of a generic test suite generator than other methods due to its ability to better recognise and learn the nuances of the software and create test suites which can test them better.

TABLE I. PROGRESS OF THE EVOLUTION PROCESS.

This tables shows the evolution of each fitness parameter as the generations increase.

Generation	**Statement Coverage**	**Branch Coverage**	**Overall Coverage**	**Fitness Value**	**Fitness**	**Time taken (in sec)**
Naive Initial	0.3125	0.625	0.33	2.275	0.2351421189	0
1	0.6875	0.75	0.71	5.925	0.6124031008	0.02
2	0.6875	0.75	0.71	5.925	0.6124031008	0.06
...	**...**	**...**	**...**	**...**	**...**	**...**
29	0.8125	0.875	0.83	7.775	0.8036175711	66.58
30	0.8125	0.875	0.83	7.775	0.8036175711	67.13
31	0.8125	0.875	0.83	7.775	0.8036175711	68.01
...	**...**	**...**	**...**	**...**	**...**	**...**
499	0.8125	0.875	0.83	7.775	0.8036175711	107.10
500	0.8125	0.875	0.83	7.775	0.8036175711	108
501	0.8125	0.875	0.83	7.775	0.8036175711	108.57
...	**...**	**...**	**...**	**...**	**...**	**...**
798	0.9375	1	0.96	9.675	1	133.55
799	0.9375	1	0.96	9.675	1	133.98
800	0.9375	1	0.96	9.675	1	134.17
801	0.9375	1	0.96	9.675	1	134.71
...	**...**	**...**	**...**	**...**	**...**	**...**
1999	0.9375	1	0.96	9.675	1	479.88
2000	0.9375	1	0.96	9.675	1	480.30

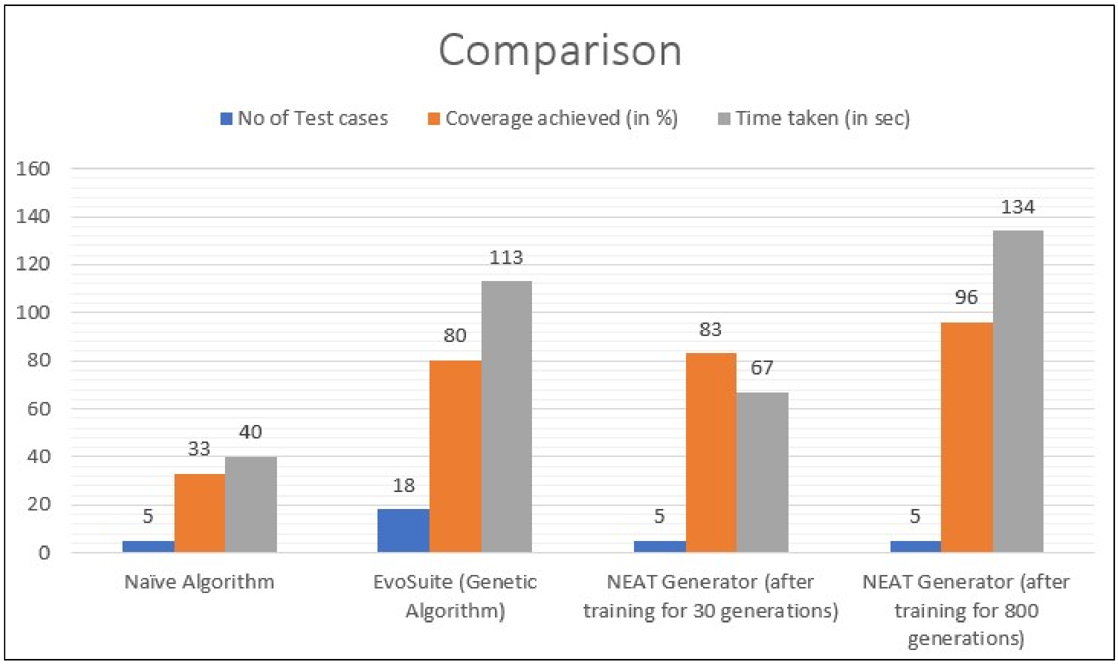


Fig. 8. Comparison between a. Naive Algorithm based Generator, b. EvoSuite-a Genetic Algorithm based Generator, c. NEAT Generator (which is the proposed approach) after training for 30 generations, d. NEAT Generator after training for 800 generations. Here the comparison is done by taking the following parameters into account a. Time taken to generate Test Suite, b. Coverage obtained by the Test Suite, c. No of test cases required for obtaining the coverage that was obtained.

VI. Conclusion and Future Works

During this research we have tried to answer the problem of Automated software testing. Here we have proposed a novel approach for the generation of the test suites for a Software Under Test using the NEAT Algorithm.

This research has shown that our approach for automatic generation of test suite for Software Under Test (SUT), obtains the better coverage than other approaches using lesser time and fewer number of test cases.

From this research we can conclude that, the NEAT Generator can be effectively used to automatically generate test suites for the SUT. The NEAT Generator generates test suites which obtain the best coverage scores. It also generates the test suites in a very short time. One other feature of the NEAT Generator is that, it can obtain the best coverage scores by generating the test suites containing almost minimum number of test cases.

We feel that future research should be focused on improving the time taken for the generation of test suites. We also think that further research is required for better evaluation of the branch and loop coverages in a Software.

References

[1] J. Wegener, A. Baresel, and H. Sthamer, "*Suitability of evolutionary algorithms for evolutionary testing*", in Proceedings 26th Annual International Computer Software and Applications, 2002, pages. 287–289. J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[2] Rizal Broer Bahaweres, Khoirunnisya Zawaw and Dewi Khairani, "*Analysis of statement branch and loop coverage in software testing with genetic algorithm*", 2017 4th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI), pages. 1-6, 2017.

[3] Kenneth O. Stanley, Risto Miikkulainen, "*Evolving Neural Networks through Augmenting Topologies*", Evolution Computation, Summer 2002, vol. 10 iss. 2, pages. 99-127

[4] IEEE-SA Standards Board, "*IEEE Standard for software unit testing*", IEEE Standards, pages. 1-28, 1987.

[5] Sumon Biswas, M. S. Kaiser, S. A. Mamun, "*Applying Ant Colony Optimization in software testing to generate prioritized optimal path and test data*", Electrical Engineering and Information Communication Technology (ICEEICT), 2015 International Conference, 21-23 May 2015.

[6] Roy P. Pargas, Mary Jean Harrold, Robert R. Peck, "*Test-Data Generation using Genetic Algorithms*", Journal of Software Testing, Wiley, 1999.

[7] B.F. Jones, H. H. Sthamer, D.E. Eyres, "*Automatic structural testing using genetic algorithms*", IET Software engineering journal, September 1996, pages. 299 – 306.

[8] Praveen Ranjan Srivastava, Tai-hoon Kim, "*Application of Genetic Algorithm in Software Testing*", International Journal of Software Engineering and Its Applications, vol. 3, iss. 4, October 2009.

[9] C.C. Michael, G.E. McGraw, M.A. Schatz, C.C. Walton, "*Genetic algorithms for dynamic test data generation*", Automated Software Engineering, 1997. Proceedings., 12th IEEE International Conference, 1-5 Nov. 1997.

[10] D. Berndt, J. Fisher, L. Johnson, J. Pinglikar, A. Watkins, "*Breeding software test cases with genetic algorithms*", System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference, 6-9 Jan. 2003.

[11] Huaizhong Li and C. Peng Lam, "*Software Test Data Generation using Ant Colony Optimization*", World Academy of Science, Engineering and Technology International Journal of Computer, Electrical, Automation, Control and Information Engineering, vol. 1, iss. 1, 2007.

[12] Donghwan Shin, Shin Yoo, Mike Papadakis and Doo-Hwan Bae, "*Empirical Evaluation of Mutation-based Test Case Prioritization Techniques*", Wiley InterScience, 2017.

[13] A. Podgurski and L. A. Clarke, "*A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance*", IEEE Transactions on Software Engineering, vol. 16 iss. 9, 1990.

[14] Pavneet Singh Kochhar, Ferdian Thung, and David Lo, "*Code Coverage and Test Suite Effectiveness: Empirical Study with Real Bugs in Large Systems*", IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, 2015.

[15] Jussi Kasurinen, Ossi Taipale, Kari Smolander, "*Software Test Automation in Practice: Empirical Observations*", Advances in Software Engineering, 2010.

[16] A. Jefferson Offutt, "*Investigations of the Software Testing Coupling Effect*", ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 1 iss. 1, 1992.

[17] Mario Linares-Vasquez, Carlos Bernal-Cardenas, Kevin Moran and Denys Poshyvanyk, "*How do developers test Android Applications*", 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages. 613-622, 2017.

[18] Eugenia Díaz, Javier Tuya, Raquel Blanco, "*Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search*", 18th IEEE International Conference on Automated Software Engineering, pages. 310-313, 2003.

[19] John Callahan, Francis Schneider, Steve Easterbrook, "*Automated Software Testing Using Model-Checking*", Proceedings 1996 SPIN Workshop,Rutgers,NJ, August 1996. Also, WVU Technical Report #NASA-IVU-96-022.

[20] E. Diaz, J. Tuya, R. Blanco, "*A modular tool for automated coverage in software testing*", Software Technology and Engineering Practice, 2003. Eleventh Annual International Workshop, 19-21 Sept. 2003.

[21] Srinivas Nidhra and Jagruthi Dondeti, "*Black box and White box testing techniques – A Literature review.*", International Journal of Embedded Systems and Applications (IJESA), vol. 2, iss. 2, June 2012

[22] Darrell Whitely, "*A genetic algorithm tutorial*", Statistics and Computing, Springer, June 1994, vol. 4 iss. 2, pages. 65–85.

[23] John A. Miller, Walter D. Potter, Ravi V. Gandham, and Chito N. Lapena, "*An Evaluation of Local Improvement Operators for Genetic Algorithms*", IEEE Transactions on Systems, Man, and Cybernetics, vol. 23 iss. 5, pages. 1340-1351, Sep/Oct 1993.

[24] D. E. Goldberg, "*Genetic Algorithms in Search, Optimization, and Machine Learning*", Reading M.A:Addison-Wesley, 1989.

[25] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil Mcminn, "*An orchestrated survey of methodologies for automated software test case generation*", Journal of Systems and Software, vol. 86 iss. 8, pages. 1978-2001, August, 2013.