Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 03, March-2022

Automatic Test Case Generation Based on Monte Carlo Tree Search Algorithm

Angwech Kevin
Department of Information Science and Technology
Zhejiang Sci-Tech University,Hangzhou, China

Said Kabir Sulaiman
Department of Information Science and Technology
Zhejiang Sci-Tech University, Hangzhou, China

***Abstract*—Automatic test case generation is an important component of software debugging process. However, maximizing coverage is still a major problem in software testing. This paper proposed Monte Carlo Tree Search (MCTS) for automatic test case generation. The proposed MCTS Test Case Generation Method (M-TCG) maximizes branch coverage of derived test cases within the available test resources. M-TCG method formulates test case generation problem as a tree exploration problem and then uses MCTS to heuristically search optimal test cases. To evaluate the effectiveness and efficiency of our method, comparative experiments were conducted based on 17 projects with 66 non trivial Java classes from SF110 corpus and other benchmarks. Experiment results indicate that the proposed method improved average branch coverage by 2.1% compared with DynaMOSA, a state-of-the-art evolutionary technique. Our method also decreased memory consumption by 4% compared with the existing adaptive aDynaMOSA approach.**

Keywords— Software Testing; Automatic Test Case Generation; Monte Carlo Tree Search; Coverage

I. INTRODUCTION

In recent years, automatic testing has become widely used compared to manual testing. The aim of software testing is to provide the end users with good quality products [1] [2]. Generating test cases that covers many parts of the program ensures that the software application is not error prone. One critical task in software testing is to generate test data that satisfy a given adequacy criteria. However the automatic generation of test case with an effective method that maximizes coverage still poses a major problem in software testing.

Many researchers are focusing on Evolutionary algorithm strategies for test case generation, these strategies include Genetic algorithm, particle Swam optimization, simulated annealing and many others [3] [4]. The existing literature, Dynamic Many-Objective Sorting Algorithm (DynaMOSA) [5] and its extension aDynaMOSA [6] generate efficient test cases; however, branch coverage is not fully maximized. Branch coverage is one of the adequacy criterion highly considered in search based test case generation [7] [8] [9] [10]. Monte Carlo Tree Search (MCTS) techniques have successfully mitigated such complex problems.

MCTS [11] [12] is an optimization method for finding optimal decision in a given domain. It is based on random exploration search space. MCTS method can solve complex optimization problems [13] and has been very successful in games such as computer Go and planning problems [14].

In this paper, we proposed the MCTS-based approach for test case generation (M-TCG). We formulate test case generation problem as a tree-based optimization searching problem. To evaluate the performance of our methods, experiments were conducted on programs acquired from SF110 corpus [15] and other benchmarks that have been used in search-based methods [16] [17]. The proposed methods were compared to the state-of-art evolutionary method implemented in DynaMOSA and an adaptive approach implemented in aDynaMOSA, which extends DynaMOSA.

This article is summarized as follows: In section. 2 we present the related work, In section.3 we discuss the basic structure of Monte Carl tree search. Section .4 describes the the proposed M-TCG method. Section.5 discusses experimental configuration, design, experimental results and also threats and validity. Lastly we present our conclusion in section. 6.

II. RELATED RESEARCH

It is important to consider the performance of software being tested more effectively and at a lower cost through the use of automated methods. The purpose of test data generation automation system for programs evaluation is to generate test data that covers as many branches as possible from a program's code, with the least computational effort possible [18].

Several meta-heuristics have been suggested for path, statement and branch coverage. Genetic Algorithm (GA) is a global search heuristic that work on the principle of Darwin's theory of natural evolution. Techniques such as [5] [6] [19] have been proposed in literature. Dynamic Many Objective Algorithm (DynaMosa) which is a state-of-the-art evolutionary technique. An adaptive approach aDynaMOSA extends DynaMOSA. These studies employs branch coverage objective. Much as these methods are multi-objective, branch coverage is still not fully maximized. These technique might not scale well to very large numbers of objectives and limited search budgets. The findings of the study presented by Harman and Mcminn [20] indicates that sophisticated search techniques, such as evolutionary testing can often be outperformed by far simpler search techniques. However many methods have been put forward to learn the behavior of local search algorithms by applying reinforcement learning. Local search heuristics have been shown to be more effective than genetic algorithms for specific targets.

Lakhotia, Tillmann, Harman and deHalleux [21] introduced a version of dynamic symbolic execution which uses search based software testing approach for handling floating point computation. The effectiveness of their approach increased although their study also showed that for the two solvers where no effective because it requires more execution time as and a large fitness budge. Poulding and Feldt [22] in their paper,

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 03, March-2022

applied Monte Carlo, which is nested to generate the programs used by GodelTest, and the results were efficient.

Harnam and Preet [23] proposed algorithm which combines Monte Carlo simulation and reliability computing by randomly sampling inputs from word file. Test that were done at different percentages on inputs was increased. The system was able to efficiently calculate the category of the document inputted. Cazenave and Mehat [24] proposed a method that combines Upper Confidence Bound (UCT) and Nested Monte Carlo Search for a single player general game playing. Their method show transposition table improves UCT. Much as there was improvement in their work, there is still a problem of whether or not the entire tree is search for huge data. This therefore calls for parallel MCTS to resolve this issue.

Various methods have been proposed in literature for test case generation, although not much is done by these techniques to measure and analyze coverage that maximizes the number of covered items [25]. A study by [26] discusses some of the limitations of genetic algorithm techniques in software testing. Generating test cases that satisfies a particular target criterion with higher coverage is still one of the challenges faced by structural testing. Thus MCTS and PMCTS for automatic test case generation are beneficial to software testing.

III. MONTE CARLO TREE SEARCH ALGORITHM

Monte Carlo search is a probabilistic, unique decision-making algorithm It is efficient with an enormous amount of possibilities suitable for problems with high branching factor as it does not waste computations on all possible branches [13]. The paradigm of MCTS combines information tree with Monte Carlo simulation. Monte Carlo search incrementally builds up the search tree and for each iteration, a tree policy is used to find the most promising child node to expand. The tree policy balances the exploration and exploitation. The four basic steps of MCTS operators are shown in Fig. 1.

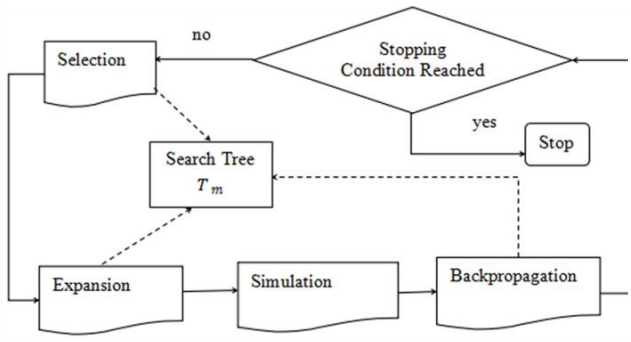

Fig. 1 The basic Monte Carlo Operation

In the selection step the T_m is traversed from the root until a leaf node is reached to be added to the tree. MCTS uses the Upper Confidence Bound (UCB) formula applied to trees as the strategy in the selection process to traverse T_m [27]. A well-known algorithm called UCB that can balance the trade-off between exploration and exploitation as seen in (1).

$$UCB_i = x_i + C * \sqrt{\frac{\ln(t)}{n_i}} \quad (1)$$

Where; UCB_i = value of a node, x_i = empirical mean of a node i(the average reward/value of all nodes beneath this node), C = a constant, t = total number of simulations (the number of times the parent node has been visited), n_i = is number of times child node i has been visited.

The second step is expansion which decides which node will be added to the T_m .

Monte Carlo simulation is run with random node selection. Heuristic knowledge can be used to give higher weight to promising node. This is done as a typical Monte Carlo simulation, either purely random or with some simple weighting heuristics if a light playout is desired, or by using some computationally expensive heuristics and evaluations for a heavy playout.

Propagation is a step where the results of simulation is propagated upwards through the search T_m and each node selected in step one is updated accordingly.

Termination: Once the time given as input expires or when all the nodes in the tree have been exhausted, the program stops expanding the tree and computes the final results.

IV. THE PROPOSED M-TCG METHOD

A program under test is taken as input and instrumented program is generated. The instrumented program is parsed and stored as a list. A control flow graph is generated from the stored data. The instrumented program executes the test data, corresponding branches are traced to check whether the test data meets the target requirement or not. Monte Carlo tree search operators play an important role in generation of test case.

This method formulates test case generation as a search tree based optimization searching problem. The node of the search tree is traversed using the tree policy starting from the root node. Tree policy is used to construct simulated test case generation tree. Once a leaf node is reached, a new state of the node is added to the tree, Monte Simulation is perform with random policy and the results of the statistic is back propagated

Monte Carlo tester automatically and regularly generates new test case to monitor whether the target branches are covered or not and accordingly update Monte Carlo Tree Search parameters to lead new test to traverse uncovered branches. Test cases are chosen from their domain at random and the algorithm generates new test cases in order to achieve the target branch. Suitable test case that executes along the target branch is generated.

A. M-TCG Algorithm

This approach uses MCTS to generated test cases. The program branches are sampled and executed the algorithm. The output of the algorithm is a program branch that leads to highest coverage in the program. Algorithm 1 shows the general algorithm for MCTS Where s_0 is the initial state; d_{max} is the maximum tree depth; V_m is node set; ε_m is edge set; T_m is a search tree (a test case generation tree where each node represents a state of test case generation). Initially, T_m has only one node at the root.

The Instrumented program, the current rollout, simulation, initial state and the maximum tree depth are the inputs while the out is a test data arrays for the target node. The completed simulation is initialized and statistics are initialized to zero. The

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 03, March-2022

search tree containing nodes and edges are also initialized. While the stopping condition is not reached, a node is selected.

Selection: We select child the child node from root node S_0 that represents states S leading to highest value of $Q(s,a)$. The tree policy a_s to traverse a tree T_m is determined by (2).

Algorithm 1: The general algorithm for M-TCG method.

Input: instrumented program. Initial states$_0$; max tree depthd_{max}; simulation and rollout played.

Output: test data arrays for the target node

1.	Initialize	The completed simulation $n_{complete} \leftarrow 0$; A search tree T_m each with node set $V_m \leftarrow \{S_0\}$ and edge set $\varepsilon_m \leftarrow \emptyset$	Where the statistics are initialized to zero.
2.	while	time out is not expired and tree is not exhausted do	
3.		if (target is reached) Update the test data that reaches the target node. Continue with next branch	
4.		end	
5.		while $n_{complete} < N$ do	
6.		(Node selection) Select a node using UTC formula by traversing over T m starting from root node s0.	Node is selected according to equation (2). A successor of the node is selected based on if its depth exceeds d_{max}or not.
7.		(Expansion) Choose action a at state s to expand.	
8.		(Simulation) perform a simulation on the current node.	
9.		(Wait) wait until a simulation is complete and get return	
10.		(Backpropagation) update the current move sequence with the simulation result	
11.		if $n_{complete} \leftarrow n_{complete} + 1$ Terminate the current process	
13.		end	
14.		end while	
15.	end while		
16.	Return the best action for the initial state s_0.		

$$a_s = k \in \operatorname{argmax} a \in A(s)\left\{Q(s,a) + C * \sqrt{\frac{\log N(s)}{N(s,a)}}\right\} \quad (2)$$

where a - is an action; s - is the current state; A(s) - is a set of actions that are in state s; $(Q(s,a)$- is a value of an action a in state s; $N(s)$ - represents the total number of simulation; $N(s,a)$ - is a number of times an action is sampled in state s; c - is the uncertainty constant to be applied when calculating the bonuses of each action. $C > 0$.

Expansion: An action a for expansion is selected at state s and test case state is loaded. Choose the node that maximizes the UCT value. We keep doing this until we reach the leaf node. A new node added to the tree if it does not already exist or if the node already exists in the tree and check number of times a link was chosen for each action link.

Simulation: perform simulation until a result is achieved. Simulation is performed using a random default policy. The statistics are computed and the numbers of visits for each action are obtained per each play turn.

An integer representing the result is returned. Search tree T_m with node set $V_m \leftarrow \{s_0\}$, each corresponding to its state S that has been seen during simulations. Each node contains a total count for the state, $N(s)$ and an action value $Q(s,a)$ and count $N(s,a)$ for each action $a \in A$). A default policy is used to rollout simulations to completion. Every state and action in the search tree T_m is evaluated by its average value during simulations. After each simulation with value V, each node in the search tree T_m updates its count, and (s,a).

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} I_i\,(s,a) V_i \quad (3)$$

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 03, March-2022

Where; $N(s,a)$ is the number of times action a has been selected from state s, $N(s)$ is the number of times a node has been played out through state s, V_i is the result of the ith simulation played out from s, and $I_i(s,a)$ is 1 if action a was selected from state s on the ith playout from state s or 0 otherwise.

Backpropagation: Update the current move sequence with the simulation result. The tree to which the node belongs is used to search for a node, also used to update tree statistics, including number of nodes and average depth. Update the data for the tree by iterating through all nodes. The weight of a node is its estimated value. A given reward is added to the total rewards for an action. An integer representing the action is selected and its reward added to proximate the test case-theoretic value of moves.

Termination: Once the time given as input expires or when all the nodes in the tree have been exhausted, the program stops expanding the tree and computes the final results.

B. TestData Model

The basis of the model is simple, the path selector identifies the paths. Once a set of test paths is determined the test generator derives input data for every path that results in the execution of the selected path. The test generator provides the selector with feedback concerning paths which are infeasible. For example, a program P with directed graph G = (V, E s,e) consisting of a set of vertices (nodes) and edges E={n,m|n,m∈V}, connecting the vertices.

Each node denotes a basic block which itself is a sequence of instructions. It is important to note that in every basic block the control enters through the entry node and leaves at the end without stopping or branching except at the end. Basically, a block is always executed as a whole. The entry and exit nodes are two special nodes denoted by s and e respectively.

An edge in a control flow graph represents possible transfer of control. All edges have associated with them a condition or a branch predicate. The branch predicate might be the empty predicate which is always true. In order to traverse the edge the condition of the edge must hold. If a node has more than one outgoing edge, the node is a condition and the edges are called branches.

The test data constructs the program flow graph, Select the Path and generate test data. The paths are identified by the path selector. After determining a set of test paths, the test generator derives input data for every path that results in the execution of the selected path. Essentially, our aim is to find an input data set that will traverse the path chosen by the path selector. The solution will ideally be a system of equations which will describe the nature of input data so as to traverse the path. In some cases the generator provides the selector with feedback concerning paths which are infeasible etc. Fig. 2 shows the process of test data generation.

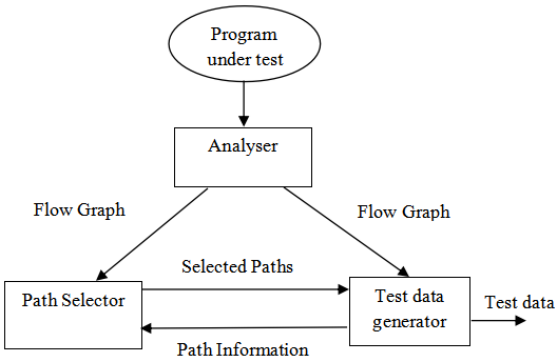

Fig. 2 Test Generation.

The flow graphs consist of nodes representing decisions and edges showing flow of control. The flow graph is constructed by replacing program control statements by equivalent diagrams. Each branch in a conditional statement is shown as a separate path. An arrow looping back to the condition node denotes a loop. Test cases are determined from the implementation of the software. A test case is a complete path from the entry vertex to the exit vertex of a flow graph. The function takes as inputs the generated values of variables as test cases, involved in the path constraint. It consists of decision-making statements.

V. EXPERIMENTAL CONFIGURATION AND DESIGN

A. Case Study

The projects were obtained from SF110 corpus and other search-based software testing benchmarks [5]. 10 Java projects whose average branch coverage is greater than 70% were randomly selected as seen in Table. 1 to study the effectiveness of our method and DynaMOSA technique in terms of branch coverage. The selected projects have a total of 56 non trivial classes.

TABLE I. 10 JAVA PROGRAMS USED IN THE STUDY FOR BRANCH COVERAGE

Project Name	Classes
biblestudy	1
Commons-lang	14
firebired	2
guava	2
jdom	5
Jfree-chart	12
saxpath	2
scribe	6
tullibee	5
twitter4j	7
total	56

The second set experiment was conducted on Java projects randomly selected from subject projects used in the existing research paper [6]. The projects were obtained from different benchmarks such as SF110 corpus [15] and other benchmarks used in the existing research paper.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 03, March-2022

TABLE II. 7 JAVA PROGRAMS USED IN THE STUDY FOR RUNTIME AND MEMORY CONSUMPTION

Project Name	Classes
a4j	1
freemind	1
javaviewcontrol	1
jdbcl	1
jmca	2
Jsecurity	1
shop	3
total	10

7 Java project with non-trivial classes (with cyclomatic complexity below five) were randomly selected as shown in Table. 2 to study performance of our methods compared with existing DynaMOSA and aDynaMOSA. Each run was repeated 50 times.

B. Experimental Design

The proposed M-TCG method is a test data generation method for automatic generation of test cases guided by MCTS algorithm to derive test cases. This method automatically generates test cases for Java classes, targeting branch coverage criterion. Program's branches are executed at least once. Both the true and false branches of all conditions must be executed. For example, in a decision like if- statements the conditions should be at least once true and once false during testing and thus all branches should be taken. A batch of test cases are generated, executed and the branch coverage is determined. We proposed MCTS method for test case generation with the aim of maximizing branch coverage. Each of these research questions were investigated in the light of branch coverage criteria alongside other metrics, such as the, runtime and heap memory consumption. The study is designed around the following research questions:

RQ1: Is M-TCG effective for generating test cases in terms of branch coverage criterion?

RQ2: How does the M-TCG method perform compared to DynaMOSA on branch coverage?

RQ3: What is the performance of M-TCG in reducing runtime compared to existing DynaMOSA and aDynaMOSA?

RQ4: what is the memory consumption of M-TCG compared to DynaMOSA and aDynaMOSA?

In order to answer the research questions, two sets of experiments are performed to compare the effectiveness and efficiency of the proposed M-TCG to the existing methods. For each subject program, we measured average branch coverage achieved on the programs, alongside other metrics, such as the number of test case generated and the elapsed time. Branch coverage criterion is use measures the extent to which a set of test cases covers a program is use for calculating branch coverage as shown in (4).

$$\text{branch coverage} = \frac{\text{covered branches}}{\text{total branches to be covered}} * 100\% \quad (4)$$

C. Experimental Results

The statistical difference is measured with Wilcoxon test and we used Cohen's d to quantify improvement. Our cutoff for significance is 5% (P-value = 0.05). A result below 5% means significant does exist. A result above 5% means that we can't conclude that a significant difference exists. Effects size quantifies the magnitude of difference found between the variables of our technique and the existing techniques used for comparison. Cohen's d = 0.2 is a small effect size, d = 0.5 represents a medium effect size and 0.8 indicates a large effect size. Lager effect size means greater height difference between our approach and the existing approach for comparison. Effect size less than 0.2 means the difference is negligible although significant.

To answer our research question RQ1, experiments were conducted and observation from the experiment results show that the proposed method effectively generate test cases and achieve branch coverage with fairly good performance. Our method's performance of was compared with the performance of the existing methods.

To answer research RQ2, experiments were performed on 10 randomly selected projects and each run repeated 10 times. Table. 3 shows the results of mean branch coverage performed by M-TCG and DynaMOSA methods.

TABLE III. MEAN BRANCH COVERAGE ACHIEVED BY M-TCG AND DYNAMOSA

Project Name	Branch Coverage (%)		Min Branch coverage (%)	Max Branch coverage (%)
	DynaMOSA	***M-TCG***		
biblestudy	84.85	90.79	75.03	94.58
Commonslang	95.40	93.48	90.52	95.20
firebired	75.62	87.05	77.62	90.85
guava	72.24	78.53	68.31	84.57
jdom	84.16	87.64	79.47	90.66
Jfree-chart	85.16	85.34	77.98	93.29
saxpath	95.04	92.44	86.73	95.64

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 03, March-2022

scribe	100	96.02	93.43	97.95
tubllibee	100	99.56	97.90	100
twitter4j	94.59	97.00	94.24	98.87
Average	88.71	90.77	84.12	94.16
Total	887.06	907.85	841.23	941.61

Out of 10 projects, M-TCG performed better than DynaMOSA in five projects. M-TCG achieved an average coverage of all the 10 projects by 90.77% while DynaMOSA achieved an average coverage of all the 10 projects by 88.71%. We can observe that for all the projects, DynaMOSA covers 887.06 total percentage branches while M-TCG covers 907.85 total percentage branches of all the projects. The "min branch coverage" and "max branch coverage" column data are for our method and can be use to get the range between the minimum and maximum coverage of the ten runs. Fig. 3 shows branch coverage for both methods with the effect size of 0.25 and p-value of 0.28, the result for M-TCG shows relative increases in mean branch coverage with small effect size. Hence M-TCG method achieved an improvement with weak mean branch coverage. Experiment results show that M-TCG method improved mean branch coverage by 2.1% compared with DynaMOSA.

To answer our research question RQ3, an experiment was conducted on 7 projects with each run repeated 50 times. Table. 4 and Fig. 4 show that M-TCG outperformed DynaMOSA and aDynaMOSA in only two projects out of seven projects. The average runtime of project freemind and jsecurity is lowest in M-TCG than in DynaMOSA and aDynaMOSA. The mean runtime of all projects in M-TCG is greater than that of DynaMOSA and aDynaMOSA.

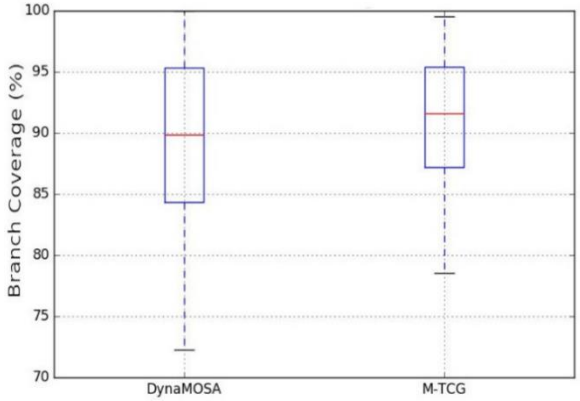

Fig. 3 Branch Coverage Achieved by M-TCG and DynaMOSA

TABLE IV. MEAN RUNTIME AND MEMORY CONSUMPTION ACHIEVED BY M-TCG AND DYNAMOSA, ADYNAMOSA

Project Name	**Runtime (ms)**			**Memory Consumption (MB)**		
	DynaMOSA	***aDynaMOSA***	***M-TCG***	***DynaMOSA***	***aDynaMOSA***	***M-TCG***
aj4	49.26	48.75	309.62	22.78	22.83	20.00
freemind	781.37	1,188.46	441.12	305.05	384.81	19.13
javaviewcontrol	169.81	112.79	403.94	224.47	384.81	19.45
jdbcl	14.34	2.28	315.40	2.84	5.68	17.94
jmca	130.87	155.12	505.12	590.46	607.21	18.04
jsecurity	552.76	490.33	372.42	390.03	349.12	13.81
shop	57.18	75.42	439.12	285.34	281.42	18.77
average	250.80	296.16	398.11	260.14	260.08	18.16
total	1,755.59	2,073.15	2,834.15	1,820.97	2,035.88	127.14

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 03, March-2022

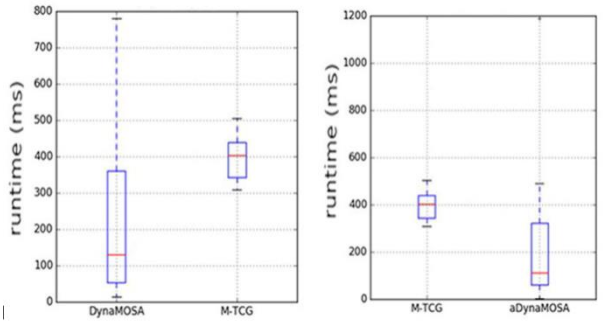

Fig. 4 Runtime of M-TCG and DynaMOSA, aDynaMOSA

The effect size is 0.68 and the p-value of 0.18 when comparing M-TCG and DynaMOSA. Since the effect size is greater than 0.5, this therefore the mean runtime of M-TCG has a strong effect size. The proposed technique achieved the highest mean runtime when compared the existing DynaMOSA and aDynaMOSA. So to generate tests and achieve coverage, the proposed method requires more runtime.

Comparing M-TCG and aDynaMOSA, the effect size of is 0.33 and a p-value of 0.31. So there is a relative increase in mean runtime with a weak effect size. This therefore mean that aDynaMOSA require less runtime to achieve branch coverage and generate test case compared to our M-TCG method.

To answer the research question RQ4, experiment was conducted on 7 projects and each run was repeated 50 times. Table. 5 and Fig. 5 show that the average memory consumption of all projects in M-TCG is about fourteen times less than that of DynaMOSA and aDynaMOSA.

Comparing M-TCG with DynaMOSA, the effect size is -1.67 and a of p-value of 0.04 indicating that the relative decrease in the mean of heap memory usage and the p-value shows there is a statistical significant difference between M-TCG and DynaMOSA.

Considering M-TCG and aDynaMOSA, the effect size and p-value is -1.65 and 0.04 respectively. Here the effect also decreased the mean of memory consumption. M-TCG method consumed less memory than DynaMOSA and aDynaMOSA when generating tests. The average memory consumption of all the projects in our method is about fourteen times less than that of DynaMOSA and aDynaMOSA. Therefore the proposed method outperformed DynaMOSA and aDynaMOSA by achieving 4% decrease in statistical memory consumption.

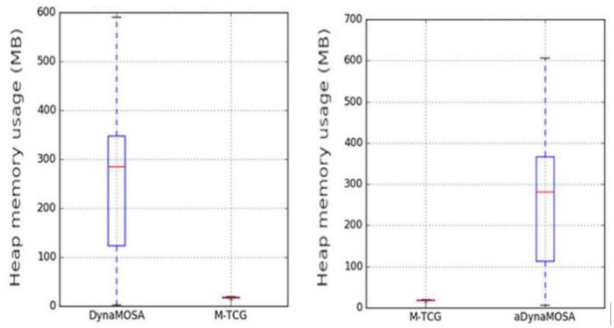

Fig. 5 Heap Memory Consumption of M-TCG and DynaMOSA, aDynaMOSA

D. Summary of the Results

Experimental results showed that M-TCG method achieved the highest mean branch coverage when compared to that of both DynaMOSA and aDynaMOSA The effect size tells us there is a strong increase in coverage when using M-TCG over DynaMOSA. M-TCG method outperformed DynaMOSA and aDynaMOSA by achieving the lowest memory consumption. However M-TCG achieved a lower runtime than that of DynaMOSA and aDynaMOSA in only two out of seven projects. The proposed technique achieved the highest mean runtime when compared to that of both DynaMOSA and aDynaMOSA.

E. Threats to validity

Although MCTS is powerful heuristic search method, its computational cost is expensive especially in rollouts to construct the search tree. In our future work, we will introduce Parallel Monte Carlo Tree Search.

After implementing Parallel Monte Carlo Tree Search, we will compare the performance of our two methods using DynaMOSA and aDynaMOSA techniques on the same data set.

The choice of existing techniques and tool for comparison is also a possible threat. There are limited recent publications for test case generation dealing with branch coverage and having available dataset for java projects. For future work, further experimental studies are required to solve the identified limitations of our tool.

The external threat to validity arises from the quality of java projects selected and the scale used to select the projects.

VI. CONCLUSION

We proposed MCTS method for automatic generation of test cases. The proposed method solves test case generation problem as a tree exploration problem and heuristically searches optimal test cases.

A prototype system was designed and implemented to evaluate the effectiveness of our method on branch coverage and study the performance of runtime and heap memory consumption.

Experiments were conducted on the popular test data sets from SF110 corpus and other benchmarks. Two sets of comparative experiments were performed on our method and promising results were obtained by our method. Experimental results showed that the M-TCG approach generates test cases with improved average branch coverage of 2.1% of all the subject projects over DynaMOSA. In addition, our method significantly decreased heap memory consumption by 4% over the existing DynaMOSA and aDynaMOSA. However for our method, more runtime was required to generate test cases and for this reason, one of our future work will be to find solution to this matter.

REFERENCES

[1] M. Ahmed, M. Nazir, S. A. Awan, “Optimization of Test Case Generation using Genetic Algorithm (GA),” ArXiv, abs/1612.08813, 2016.

[2] E. Nikravan, F. Feyzi, S. Parsa, “Enhancing path-oriented test data generation using adaptive random testing techniques,” 2015 2nd International Conference on Knowledge-Based Engineering and Innovation.KBEI, pp. 510-513, 2015.

[3] X. Bao, Z. Xiong, N. Zhang, J. Qian, B. Wu, W. Zhang, “Path-oriented test cases generation based adaptive genetic algorithm,” PloS one, vol. 12, no. 11, pp. 1-7, 2017.

[4] H. Itti, C. and Rajender, “A Review: Study of Test Case Generation Techniques,” International Journal of Computer Applications, vol. 107, no. 16, pp. 33-37, 2014.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 03, March-2022

[5] P. Annibale, F. M. Kifetew, T. Paolo, "Automated Test Case Generation as a Many-Objective Optimization problem with Dynamic selection of Targets," IEEE Transaction on Software Engineering, vol. 44, no. 2, 122-158, 2018.

[6] G.Grano, C. Laaber, A. Panichella, S. Panichella, "An Adaptive Approach to performance-Aware Test Case Generation," IEEE Transaction on Software Engineering, vol. 47, no. 11, pp. 2332-2347, 2019.

[7] B. Korel, "Automated software test data generation," IEEE TSE, vol. 16, no. 8, pp. 870–879, 1990.

[8] K. Lakhotia, P. McMinn, M. Harman, "An empirical investigation into branch coverage for C programs using CUTE and AUSTIN," The Journal of Systems and Software, vol. 8, no.12 , pp. 1085–1110, 2010.

[9] M. Harman, P. Mcminn, "A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search," IEEE Transactions on Software Engineering, 226–247, 2010.

[10] J. Wegener, A. Baresel, H. Sthamer, "Evolutionary test environment or automatic structural testing," Information & Software Technology, vol.43, no. 14, pp. 841–854, 2001.

[11] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," proceedings of the 5th International Conference on Computer and Games, Lecture Notes in Computer Science. LNCS), pp. 72-83, 2006.

[12] B. Arneson, R. B. Hayward, P. Henderson, "Monte Carlo Tree Search in Hex," IEEE Transactions on Computational Intelligence and AI in Games, vol.2, no. 4, pp. 251–258, 2010.

[13] B. Cameron, E. J. Powley, D. Whitehouse, M.S. Lucas, P. I. Cowling, P. Rohlfshagen, Tavener, D. Perez, S. Samothrakis, S. Colton, "A survey of Monte Carlo tree search methods," IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, pp. 1-43, 2012.

[14] P. I. Cowling, E. J. Powley, D. Whitehouse, "Information Set Monte Carlo Tree Search.," IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 2, pp. 122-134, 2012.

[15] G. Fraser, A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," ACM Transaction on Software Engineering and Methodology(TOSEM), vol.24, no. 2, pp. 1-42, 2014.

[16] P. Annibale, F. M. Kifetew, T. Paolo, "Automated Test Case Generation as a Many-Objective Optimization problem with Dynamic selection of Targets," IEEE Transaction on Software Engineering, vol. 44, no. 2, pp. 122-158, 2018.

[17] G. Grano, C. Laaber, A. Panichella, S. Panichella, "An Adaptive Approach to performance-Aware Test Case Generation.," IEEE Transaction on Software Engineering, vol. 47, no. 11, pp. 2332-2347, 2019.

[18] C. Paduraru, M. Melemciuc, "An Automatic Test Data Generation Tool using Machine Learning," 13th International Conference on Software Technologies, pp. 506-515, 2018.

[19] G. Fraser, A. Arcuri, "Whole Test Suite Generation," EEE Transaction on Software Engineering, vol. 39, no.2, pp. 276-291, 2013.

[20] M. Harman, P. Mcminn, "A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search," IEEE Transactions on Software Engineering, pp. 226–247, 2010.

[21] K. Lakhotia, N. Tillmann, M. Harman, J. de Halleux, "Search Based Floating Point Constrain Solving for Symbolic Execution," Testing Software and Systems. ICTSS 2010. Lecture Notes in Computer Science book auth. A. Petrenko, J.C. Maldonado, Springer, Berlin Heidelberg, pp. 142-157, 2010.

[22] S. M. Poulding, R. Feldt, "Generating Structured Test Data with Specific Properties using Nested Monte-Carlo Search," Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation. New York, NY, USA, Association for Computing Machinery, pp. 1279–1286, 2014.

[23] S. Harnam , P. Preet, "Software Reliability Testing using Monte Carlo Methods," International Journal of Computer Applications, vol. 69, no. 4, pp. 0975 – 8887, 2013.

[24] T. Cazenave, J. Mehat, "Combining UCT and nested Monte Search for Single Player general game playing," IEEE Transactions on Computational Intelligence and AI in Games. IEEE, vol. 2, no. 4, pp. 271-277, 2010.

[25] S. Muhammad, S. Ibrahim, M. N. Mohd, "A Study on Test Coverage in Software Testing," 2011 International Conference on Telecommunication Technology and Applications. IACSIT Press, Singapore, 2011.

[26] A. H. Sultan, G. Ahmed, E. E Mohammed, "The limitations of genetic algorithms in software testing," ACS/IEEE International Conference on Computer Systems and Applications- AICCSA, pp. 1-7, 2010.

[27] P. Auer, N. Cesa-Bianchi, P. Fischer P, "Finite-time analysis of the multi armed bandit problem," Machine Learning, vol. 47, no. 3, pp. 235- 256, 2002.