IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 18, Issue 3, Ver. IV (May-Jun. 2016), PP 130-141
www.iosrjournals.org

Regression Test Suite Prioritization Using Hill Climbing Algorithm

D. Vivekananda Reddy[1], Dr. A. Rama Mohan Reddy[2],
[1]*Assistant Professor, Department of Computer Science and Engineering, S V University College of Engineering, Tirupati, Andhra Pradesh,*
[2]*Professor, Department of Computer Science and Engineering, S V University College of Engineering, Tirupati, Andhra Pradesh,*

Abstract: *Regression testing is an expensive, but important action in software testing. Unfortunately, there may be bereft assets to acquiesce for the re-execution of all analysis cases during corruption testing. In this situation, analysis case prioritization techniques aim to advance the capability of corruption testing by acclimation the analysis cases so that a lot of benign are accomplished first. In this cardboard we adduce a new analysis case prioritization technique Using Hill climbing Algorithm. The proposed address prioritizes subsequences of the Original analysis apartment so that the new suite, which is run a time-constrained beheading environment, will accept the above amount of accountability apprehension if compared of randomly Prioritized analysis suites. This agreement analyzes Hill Climbing algorithm with attention to capability and time aerial by utilizing structurally-based archetype to accent test Cases. An Average Percentage of Faults Detected (APFD) metric is acclimated to actuate the capability of the new analysis case orderings.*

I. Introduction

Test case prioritization techniques adapt the analysis cases in a analysis suite, acceptance for an access in the capability of testing. One achievement goal, the accountability apprehension rate, is a measurement of how bound faults are detected during the testing process. A bigger amount of accountability apprehension can accommodate faster acknowledgment apropos the superior of the arrangement beneath test, but frequently, complete testing is too expensive. This is generally the case with corruption testing, the action of acceptance adapted software to ascertain whether new errors accept been alien into ahead activated cipher and to accommodate aplomb that modifications are correct. By accretion the all-embracing amount of accountability detection, a greater amount of errors can be begin added rapidly in the code .As common rebuilding and corruption testing accretion popularity, the charge for a time coercion acquainted prioritization address grows. New software development processes such as acute programming as well beforehand a abbreviate development and testing and common beheading of fast analysis cases [1]. Therefore, there is a bright charge for a prioritization address that has the abeyant for added capability if a analysis suite's accustomed beheading time is known, decidedly if that beheading time is short. This cardboard shows that if the best time allotted for beheading of the analysis cases is accepted in advance, added able prioritization can be produced. The time accountable analysis case prioritization botheration can be bargain to the NP-complete zero/one Knapsack problem [2, 3, 4] which can generally be efficiently approximated with a Hill climbing algorithm (HCA) is a heuristic optimal seek technique. Just as Hill Climbing algorithms accept been finer acclimated in added software engineering and programming Language problems such as analysis bearing [5], affairs transformation [6], and software maintenance resource allocation [7], this paper demonstrates that they also prove to be effective in creating time constrained test prioritizations. We present a technique that prioritizes regression test suites so that the new ordering (i) will always run within a given time limit and (ii) will have the highest possible potential for defect detection based on derived coverage information (iii) In will summary, have the highest important optimal contributions solution in of compare this paper all random areas Prioritization follows techniques.

II. Related Work

In this area we accommodate an overview of accompanying works of the analysis case prioritization. In recent Years several advisers accept addressed the analysis case prioritization botheration and presented techniques for acclamation it. Analysis case prioritization techniques appear in [8, 9] orders analysis cases such that the analysis cases with accomplished priority, according to some criterion, are accomplished first. Analysis case prioritization can abode a advanced array of objectives [10]. For example, advantage alone, testers ability ambition to agenda analysis cases in adjustment to accomplish cipher advantage at the fastest amount accessible in the antecedent appearance of corruption testing, to ability 100% advantage soonest or to ensure that the best accessible advantage is accomplished by some pre-determined cut–off point. In the Microsoft Developer

Network (MSDN) library, the accomplishment of able advantage after crumbling time is a primary application if administering corruption tests [11]. Furthermore, several testing standards crave annex able coverage, authoritative the accelerated accomplishment of advantage an important aspect of the corruption testing process. In the literature, abounding techniques for corruption analysis case prioritization accept been described. Most of these techniques are code–based, relying on advice analysis cases to advantage of cipher elements. In [12], Rothermel et al. advised several prioritizing techniques such as absolute account (or branch) advantage prioritization and added account (or branch) advantage prioritization that can advance the amount of accountability detection. In [9], Wong et al. Prioritized analysis cases according to the archetype of 'increasing amount per added coverage'. Acquisitive Algorithms are as well acclimated and are implemented in a apparatus called ATAC [22]. In [14], Srivastava and Thiagarajan advised a prioritization address that is based on the changes that accept been fabricated to the affairs and focused on the cold action of "impacted block coverage". Other non–coverage based techniques in the abstract cover fault–exposing–potential (FEP) prioritization [10], history–based analysis prioritization [15], and the assimilation of capricious analysis costs and accountability severities into analysis case prioritization [16, 12]. In [17] Zheng Li, Mark Harman, and Robert M. Hierons advised 5 seek techniques: two meta–heuristic seek techniques (Hill Climbing and Genetic Algorithms), calm with three acquisitive algorithms (Basic Greedy, Added Acquisitive and 2–Optimal Greedy) and accepted that Genetic Algorithms performed able-bodied in analysis case prioritization. Hyunsook Do, Gregg Rothermel and Alex Kinneer [18] accept advised and performed a controlled agreement analytical whether analysis case prioritization can be able on Java programs activated beneath Junit, and compared the after-effects to those accomplished in beforehand studies. Their analyses appearance that analysis case prioritization can decidedly advance the amount of accountability apprehension of Junit analysis suites. Saff and Ernst [19, 20, and 21] advised analysis case prioritization for Java in the ambience of connected testing, which acclimated additional CPU assets to continuously run corruption tests in the accomplishments as programmer codes. They accumulated the concepts of analysis abundance and analysis case prioritization, and appear that connected prioritized testing can abate decay of development time. Test case prioritization has as well been done based on the accordant slices. Most recently, Dennis Jeffry and Neelam Gupta [29] proposed a prioritization address based on the coverage requirements present in the relevant slices of the outputs of test cases. However, these prioritization techniques are based on different sources of information, such as history of recent or frequent errors and test cost, code coverage information, and have not considered test suite time. Hence in our proposed prioritization technique we consider test suite time along with coverage information.

III. Challenges in Time based Prioritization

Test prioritization schemes about actualize a individual reordering of the analysis apartment that can be accomplished after wards abounding consecutive changes to the affairs beneath analysis [23, 3]. Re-ordering of analysis apartment can be added able at award faults if testing have to be concluded beforehand [3]. In this area we present the challenges in time based prioritization. For example, accept that corruption analysis apartment T contains six analysis cases with the antecedent acclimation {T1, T2, T3, T4, T5, T6} as declared in Figure 1(a). For the purposes of motivation, this archetype assumes a priori ability of the faults detected by T in the affairs P. As apparent in Figure 1(b), analysis case T1 can acquisition seven faults, {f1, f2, f4, f5, f6, f7, f8} in nine minutes, T2, finds one fault, f1, in one minute, and T3 isolates two faults, { f1, f5} in three minutes. Test Cases T4, T5, and T6 anniversary acquisition three faults in four minutes, {f2, f3, f7}, {f4, f6, f8}, and {f2, f4, f6}, respectively

Figure 1.(A)

FAULTS / TESTCASES	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F9**	**F10**
T1	S	S	S	F	F	F	F	S	S	S
T2	S	F	S	S	S	S	F	S	S	S
T3	S	S	S	S	F	S	F	S	F	S
T4	S	F	F	F	F	F	S	S	S	S
T5	F	F	S	S	F	S	S	S	F	S
T6	S	S	S	S	S	F	F	S	F	S
T7	F	S	S	S	S	S	S	S	S	S
T8	F	S	F	S	S	S	S	F	F	F

Figure 1. (B)

Test Cases	No of Faults	Execution Time (mins)	Avg faults/min
T1	7	9	0.77
T2	1	1	1.00
T3	2	3	0.66
T4	3	4	0.75

T5	3	4	0.75
T6	3	4	0.75
T7	4	5	0.8
T8	5	6	0.83

Figure (A) and (B). Sample Test cases, Faults identified and its Execution time .Suppose that the time budget for regression testing is twelve minutes. Because we want to find as many faults as possible early on, it would seem intuitive to order the test cases by only considering the number of faults that they can detect. Without a time budget, the test case order *T1, T4, T5, T6, T3, T2* would execute. Out of this, only the test case *T1* would have time to run when under a twelve minute time constraint and would find only a total of seven faults, as noted in Figure 2. Since time is a principal concern, it may also seem intuitive to order the test cases with regard to their execution time. Consider the test case orders TC1 = {*T1*}, TC2 = {*T2, T3, T4, T5*}, TC3 = {*T2, T1*}, TC4 = {*T5, T 4, T3*}, TC5 = {T6,T7}, TC6 = {T7,T8} for execution. In the time constrained environment, a time-based prioritization the test case order TC2 could be executed and find eight defects, as described in Figure 2. Another option would be to consider the time budget and accountability advice together. To do this, we could adjustment the analysis cases according to the boilerplate percent of faults that they can ascertain per minute. Under the time constraint, the analysis case adjustment TC3 would be accomplished and acquisition a absolute of seven faults. If the time account and the accountability advice are both advised intelligently, that is, in a way that accounts for overlapping accountability detection, the analysis cases could be bigger prioritized and appropriately access the all-embracing amount of faults begin in the adapted time period. In this example, the analysis cases would be intelligently reordered so that the analysis case adjustment TC4 would run absolute eight errors in beneath time than TC2. Also, it is bright that TC4 can acknowledge added defects than TC1 and TC3 in the defined testing time. Finally, it is important to agenda that the aboriginal two analysis cases of TC2, T2 and T3, acquisition a absolute of two faults in four account admitting the aboriginal analysis case in TC4, T5, detects three defects in the aforementioned time period. Therefore, the "good" prioritization, TC4, is advantaged over TC2 because it is able to ascertain added faults beforehand in the beheading of the tests.TC5,T6, detects 9 defects in the aforementioned time period for prioritization of test case and TC6,T7, detects 11defects in the budget time of 12 minutes time period. Therefore the "excellent" prioritization was TC6. Where within the time budget high faults was detected in this case.

	Time Limit:12 minutes					
	Fault TC1	Fault TC2	Fault TC3	Fault TC4	APFD TC5	Excellent TC6
	T1	T2 T3 T4 T5	T2 T1	T5 T4 T3	T6 T7	T7 T8
Total Faults	7	8	7	8	7	9
Total Time	9	12	10	11	9	11

Figure 2. Comparison of Prioritization

IV. Proposed Prioritization Technique

The proposed prioritization technique is based on the fitness both testing values Time of test and suites , testing potential fault time highest fitness value of the Detection information to intelligently reorder a test suite using Hill Climbing Algorithm. Our and potential fault detection information to intelligently reorder a test suite using Hill Climbing prioritization algorithm reorders the tests in any sequence that maximizes the Isolate defects. In section 4.1 we present an overview of the proposed prioritization technique and in section 4.2 we present the proposed prioritization technique.

4.1 Overview

We activated Hill Climbing algorithm in the proposed prioritization technique. We aboriginal recorded the beheading time of anniversary analysis case. Because time coercion could be actual short, analysis case Execution times have to be exact in adjustment to appropriately prioritize. Only the beheading time of the analysis case is included in the recorded time and not that of loading. Timing advice is additionally includes any initialization and abeyance time appropriate by a test. Inclusion of antecedent time and abeyance time is all-important because these operations can abundantly access the beheading time appropriate by the analysis case. The affairs P and anniversary Analysis case in a analysis apartment are ascribe into the Hill Climbing algorithm, forth with the afterward user defined parameters:

Test apartment – T

Collection of all permutations of elements of T - perms (2T)

Number of analysis suites to be created per abundance - s

Time account - tmax

Two functions from permutations to the absolute numbers – time and fit.

Maximum iterations - dmax
Percent of absolute analysis apartment time – pt

4.2 Hill Climbing Algorithm

Hill Climbing is a mathematical optimization technique which belongs to the family of *local search*. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found.
For example, hill climbing can be applied to the travelling salesman problem. It is easy to find an initial solution that visits all the cities but will be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much shorter route is likely to be obtained.

Hill climbing is good for finding a local optimum (a solution that cannot be improved by considering a neighboring configuration) but it is not necessarily guaranteed to find the best possible solution (the global optimum) out of all possible solutions (the search space). In convex problems, hill-climbing *is* optimal. Examples of algorithms that solve convex problems by hill-climbing include the simplex algorithm for linear programming and binary search .The characteristic that only local optima are guaranteed can be cured by using restarts (repeated local search), or more complex schemes based on iterations, like iterated local search, on memory, like reactive search optimization or memory-less stochastic modifications, like simulated annealing.
The relative simplicity of the algorithm makes it a popular first choice amongst optimizing algorithms. It is used widely in artificial intelligence, for reaching a goal state from a starting node. Choice of next Node and starting node can be varied to give a list of related algorithms. Although more advanced algorithms such as simulated annealing or tabu search may give better results, in some situations hill climbing works just as well. Hill climbing can often produce a better result than other algorithms when the amount of time available to perform a search is limited, such as with real-time systems. It is an anytime algorithm: it can return a valid solution even if it's interrupted at any time before it ends.

4.2.1 Travelling Salesman Problem:

Consider an asymmetric traveling salesman problem with n cities to visit. Let c_{ij} be the distance from city i to city j. Moreover, let x_{ij} be 1 if the salesman travels from city i to city j and 0, otherwise. Then, the asymmetric traveling salesman problem can be formulated as follows:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} C_{ij} X_{ij}$$

$$\text{s.t (1)} \quad \sum_{i=1}^{n} X_{ij=1,\,j=1,\ldots,n}$$

$$(2) \quad \sum_{i=1}^{n} X_{ij=1,\,j=1,\ldots,n}$$

$$(3) \quad \sum_{i\epsilon U,\, j\epsilon U,\, i\neq j} X_{ij \le |U|-1,}\ \forall_U \subset \{1,\ldots,n\},\ 2 \le |U| \le n-2$$

$$(4) \quad X_{ij} \epsilon \{0,1\},\ i=1,\ldots..n,\ j=1\ldots.n,$$

Interestingly, Bellmore and Malone [25] have shown that the asymptotic probability that the optimal solution of the assignment relaxation problem is a Hamiltonian cycle is *e/n*, where e denotes the natural logarithmic base. That is, the number of infeasible solutions with sub tours dominates that of directed Hamiltonian tours. For an n-city asymmetric traveling salesman problem, the number of infeasible solutions with two groups of sub-tours, *f(n, 2),* can be found as below. Let *h(i,n=i), i ≤ [n/2]* denote the number of infeasible solutions in which one sub-tour contains *i* nodes and the other contains *n − i* nodes, where 'a' denotes the largest integer that does not exceed a. Then

$$f(n,2) = \sum_{i=1}^{[n/2]} h(i, n\text{-}i),$$

Where

$$h(i, n-i) = \begin{cases} \dfrac{n!}{i(n-i)} & \text{if } i \neq n/2, \\ \dfrac{n!}{2i(n-i)} & \text{if } i = n/2, \end{cases}$$

Note that when $i = n/2$, the number of infeasible solutions with $n/2$ groups of sub-tours is double counted. Thus, $h(n/2, n/2)$ requires a separate consideration. Generalizing the above approach to three or more groups, we obtain the following expression for $f(n, g)$, the number of infeasible solutions with g groups of subtours. Let $h(i_1, i_2, \ldots .. i_{g-1}, n\text{-}s_{g-1})$ be the number of infeasible solutions in which g subtours contain $i_1 \leq i_2 \leq, \ldots \leq i_{g-1} \leq n - s_{g-1}$ nodes, where

$s_k = \sum_{t=1}^{k} i_t$ and $s_0 = 0$. Then

$$f(n,g) = \sum_{i_1=2}^{[n/g]} \sum_{i_2=i_1}^{[(n-s_1)/(g-1)][(n-s_2)/(g-2)]} \sum_{i_3=i_2}^{} \cdots\cdots\cdots \sum_{i_{g-1}=i_{g-2}}^{[(n-s_{g-2})/2]} h(i_1, i_2, \ldots . i_{g-1}, n\text{-}s_{g-1})$$

$$h(i_1, i_2, \ldots . i_{g-1}, n\text{-}s_{g-1}) = \prod_{k=1}^{g-1} (n\text{-}s_{k-1}) C_{ik} \; X \; \prod_{t=1}^{g-1} (i_t\text{-}1)! \; X \; (n\text{-}s_{g-1}\text{-}1)! \; / \; \prod_{x} N_x!$$

$$= \frac{n!}{i_1, i_2, \ldots . i_{g-1}, n\text{-}s_{g-1} \; \prod_{x} N_x !}$$

and N_x denotes the number of repeated patterns of α-city subtours in $i_1, i_2, \ldots, i_{g-1}, n - s_{g-1}$. Here, ${}_aC_b$ denotes the number of combinations, i.e. ${}_aC_b = a!/(a - b)!b!$. The above formulae for f(n, g) and $h(i_1, i_2, \ldots, i_{g-1}, n - s_{g-1})$ are obtained by first considering all possible combinations of subtours in an n-city asymmetric traveling salesman problem and then by using induction and recursion.

The analysis above indicates that there are strong chances that the optimal solution obtained from the assignment problem relaxation of (ATSP) can include several subtours. For instance, consider the number of infeasible solutions with three four-city subtours, two three-city subtours, and one two-city subtour in a 20-city asymmetric traveling salesman problem. In this case, $N_{10} = \cdots = N_5 = 0$; $N_4 = 3$; $N_3 = 2$, and $N_2 = 1$. Thus

$$h(4, 4, 4, 3, 3, 2) = \frac{{}_{20}C_4 \; X \; {}_{16}C_4 \; X \; {}_{12}C_4 \; X \; {}_8C_3 X {}_5C_3 X \; (4\text{-}1)! \; X \; (4\text{-}1)! \; X \; (4\text{-}1)! \; X \; (3\text{-}1)! \; X \; (3\text{-}1)! \; X \; (2\text{-}1)!}{3! \; X \; 2! \; X \; 1!}$$

$$= 1.7599117536 \; X \; 10^{14}$$

Moreover, considering all possible con4gurations of subtours (2, 18), (3, 17). . . . . , (10, 10), (2, 2, 16), (2, 3, 15), ... , (6, 6, 8), ... , (2, 2, 2,. . . ., 2), we 4nd the number of infeasible solutions with subtours in a 20-city asymmetric traveling salesman problem to be

$$\sum_{g=2}^{10} f(20, g) = 7.7337 \; X \; 10^{17}.$$

4.2.2 Selection Methods

First, a pair of organisms is selected for breeding. Selection, as line is biased towards elements of the initial generation which have better fitness, though it is usually not so biased that poorer elements have no chance to participate. This prevents the solution set from converging too early to a local solution in hopes that it will eventually lead to points in the search space with an ultimately higher payo [22]. While there are many well-defined organism selection methods, roulette wheel selection and tournament selection are the most typical [9]. Both of these methods may also be combined with an elite selection strategy which allows some of the

better organisms from the first generation to carry over to the second, unaltered. In tournament selection, a specified group of two or more individuals is chosen at random from the population. The single individual in the chosen group with the highest fitness is then selected as the winner of the tournament and goes on to reproduce. This type of selection is especially useful when only a partial order exists among the individuals in the population.

In roulette wheel selection, individuals are selected with a probability proportional to their fitness. After fitness values are assigned, they are normalized in relation to the rest of the population by multiplying each by a fixed number. Next, the population is sorted by descending fitness values, and accumulated normalized fitness values are calculated. A random number R 2 [0; 1] is then chosen, and the first individual whose accumulated normalized value is greater than or equal to R is selected. While candidate solutions with a higher fitness will be less likely to be eliminated there is still a chance that they may be. Thus, Roulette wheel selection is often combined with the elitist strategy in order to guarantee that the best individual in a population is kept [11, 22].

4.2.3 Terminating Conditions

These processes ultimately result in a second generation pool of chromosomes that is different from the initial generation. Generally the average degree of fitness will have increased by this procedure for the second generation pool, since mainly the best organisms from the first generation are selected for breeding. The algorithm is repeated until an organism is produced which gives a solution that is "good enough". The terminating condition, mentioned in line 11 of Figure 4.1 usually consists of either satisfying a problem-specific success predicate or completing a specified number of generations. A success predicate depends on the goal of the algorithm or on the nature of the problem. For example, success may consist of reaching a certain threshold level, or for certain problems, it may be possible to recognize an 100% correct solution to the problem.

While the computational setting of a Hill Climbing algorithm is much simplified when compared with the natural world, Hill Climbing algorithms are capable of evolving complex and interesting structures. Individuals within a population can be used to represent solutions to problems, visual images, or strategies for computer games; Hill Climbing algorithms are appropriate for use in studying and modeling evolution in social and cognitive systems. One example of a problem for which solutions calculated by Hill Climbing algorithms are appropriate is the 0/1 knapsack problem [20, 41], for which the research presented in this thesis has a strong. Although almost all components of a Hill Climbing algorithm are unspecified by the process in to the known general description of a HCA, there are three main components of all Hill Climbing algorithms ,as described by Forrest [9]:Independent sampling is provided by large populations that are initialized randomly .High- fitness individuals are preserved through selection, which biases the sampling process toward regions of high fitness. While the computational setting of a Hill Climbing algorithm is much simplified when compared with the natural world, Hill Climbing algorithms are capable of evolving complex and interesting structures. Individuals within a population can be used to represent solutions to problems, visual images, or strategies for computer games; Hill Climbing algorithms are appropriate for use in studying and modelling evolution in biological, social, and cognitive systems. One example of a problem for which solutions calculated by Hill Climbing algorithms are appropriate is the 0/1 knapsack problem [20, 41], for which the research presented in this thesis has a strong.

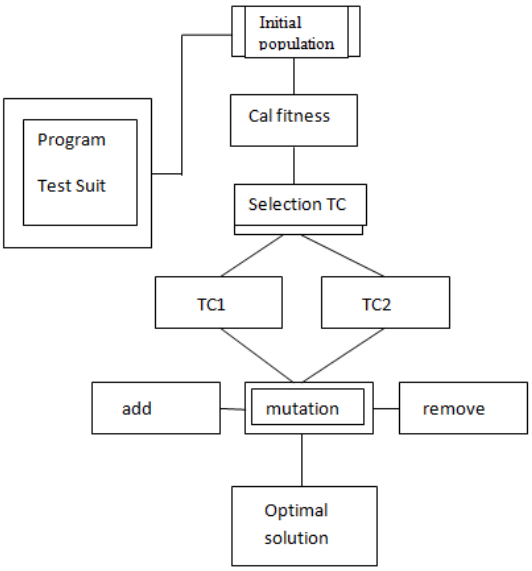

Figure 3. Proposed Hill Climbing

4.2.4 Fitness Function

The CalcFitness(P, TSj, pT, tc, w) method uses fit (P, TSj,, tc, w) to calculate fitness. The fitness function, represented by fit, assigns each test suite a fitness based on (i) the percentage of code covered in P by that test suite and (ii) the time at which each test covers its associated code in P. It is then appropriate to divide fit into two parts such that fit(P, TS_j, tc, w) = $F_{primary}$(P, TS_j, tc, w) + F_{second}(P, TS_j, tc). The primary fitness $F_{primary}$ is calculated by measuring the code coverage cc of the entire test suite TSj. Because the overall coverage of the test suite is more important than the order in which the coverage is attained, $F_{primary}$ is weighted by multiplying the percent of code covered by the program coverage weight, w. The selection of w's value should be sufficiently large so that when $F_{primary}$ and F_{second} [0, 1] are added together, $F_{primary}$ dominates the result. Formally, for some TS_j perms(2^T),

$$F_{primary}(P, TS_j, tc, w) = cc(P, TS_j, tc) \times w \qquad (1)$$

The second component F_{second} considers the incremental code coverage of the test suite, giving precedence to test suites whose earlier tests have greater coverage. F_{second} is also calculated in two parts. First, $F_{s\text{-}actual}$ is computed by summing the products of the execution time *time*(*Ti*) and the code coverage *cc* of the subtest suite *TSj*{1,i} = (*T1 …. Ti*) for each test

case T_i TS_j . Formally, for some TS_j *perms*(2^T),

$$F_{s\text{-}actual}(P, TS_j, tc) = \sum_{i=1}^{|TS_j|} time\ T_i \times cc(P, TS_{j\{1,i\}}, tc) \qquad (2)$$

Fs-max represents the maximum value that $F_{s\text{-}actual}$ could take (i.e., the value of $F_{s\text{-}actual}$ if T_1 covered 100% of the code covered by T.) For a TS_j *perms*(2^T),

$$F_{s\text{-}max}(P, TS_j, tc) = cc(P, TS_j, tc) \times \sum_{i=1}^{|TS_j|} time\ T_i \qquad (3)$$

Finally, $F_{s\text{-}actual}$ and $F_{s\text{-}max}$ are used to calculate the secondary fitness F_{second} Specifically,

for TS_j *perms*(2^T),

$$F_{secondary}(P, TS_j, tc) = \frac{F_{s\ actual}(P, TS_j, tc)}{F_{s\ max}(P, TS_j, tc)} \qquad (4)$$

As an example of a fitness calculation, let the program coverage weight $w = 100$, P be a program, and tc be a test adequacy criterion. Suppose $TSj = (T1, T2, T3)$. Also, assume we have execution times $time(T1) = 5$, $time(T2) = 3$, and $time(T3) = 1$, and test suite code coverage $cc(P, TS_j, tc) = 0.20$. Then,
$F_{primary}(P, TS_j, tc, w) = 0.2 \times 100 = 20$

$F_{secondary}$ next gives preference to test suites that have more code covered early in execution. To calculate F_{second}, the code coverages of $TS_{j\ \{1,1\}} = (T_1)$, $TS_{j\ \{1,\ 2\}} = (T_1, T_2)$, and $TS_{j\{1,3\}} = (T_1, T_2, T_3)$ must each be measured. Suppose for this example that $cc(P, TS_{j\{1,1\}}, tc) = 0{:}05$, $cc(P, TS_{j\{1,\ 2\},}\ tc) = 0.19$, and, as already known, $cc(P, TS_{j\{1,3\}}, tc) = cc$
$(P, TS_j, tc) = 0.20$. F_{second} is calculated as follows,
$F_{s\text{-}actual}(P, TS_j, tc) = (5 \times 0.05) + (3 \times 0.19) + (1 \times 0.20) = 1.02$

$F_{s\text{-}max}(P, TS_j, tc) = 0.2(5 + 3 + 1) = 1.8$

$$F_{second}(P, TS_j, tc) = \frac{1.02}{1.8} = 0.567$$

Adding $F_{primary}$ and F_{second} gives the total fitness value F_j of TS_j . Therefore, in this example, $fit(P, TS_j, tc, w) = F_{primary}(P, TS_j, tc, w) + F_{second}(P, TS_j, tc) = 20 + 0.567 = 20.567$

If a test suite execution time *time*(*TS j*) is greater than the time budget *tmax*, *Fj* is automatically set to -1 by the *CalcFitness*(*P, TSj, pT, tc, w*) method. Because such a test suite violates the execution time constraint, it cannot be a solution and thus receives the worst fitness possible. While a test suite *TSj* with *Fj* = -1 could simply not be added to the next generation *Rd+1*, populations with individuals that have a fitness of -1 can actually be favorable. Since the "optimal" test suite prioritization likely teeters on the edge of exceeding the designated time budget, any slight change to a *TSj* with *Fj* = - 1 could create a new valid test suite. Therefore, some *TSj* 's with *Fj* = -1 are maintained in the next generation. If the test suite execution time $time(TS_j) <= t_{max}$, Equations 1-4 are used.

4.2.5. Mutation

The use of the *ApplyMutation*(*pM, TSj*) method also provides a way to add variation to a new population. The new test suite is identical to the prior parent suite except that one or more changes may be made to the new test suite's test cases. All test suites that are selected are first considered for crossover. Then they are subject to mutation at each test case position with a small user specified mutation probability *pM*. If a random number *r3* [0, 1] is generated such that r3 is less than *pM* for test case Ti, a new test not included in the current test suite is randomly selected from T to replace Ti, as demonstrated for T2 in Figure 5(a) . Figure 5(b) also shows that if there are no unused tests in T when T9 is chosen for mutation, the test suite is still mutated. Instead of replacing the test with a random test, the test to be mutated is swapped with the test case that succeeds it.

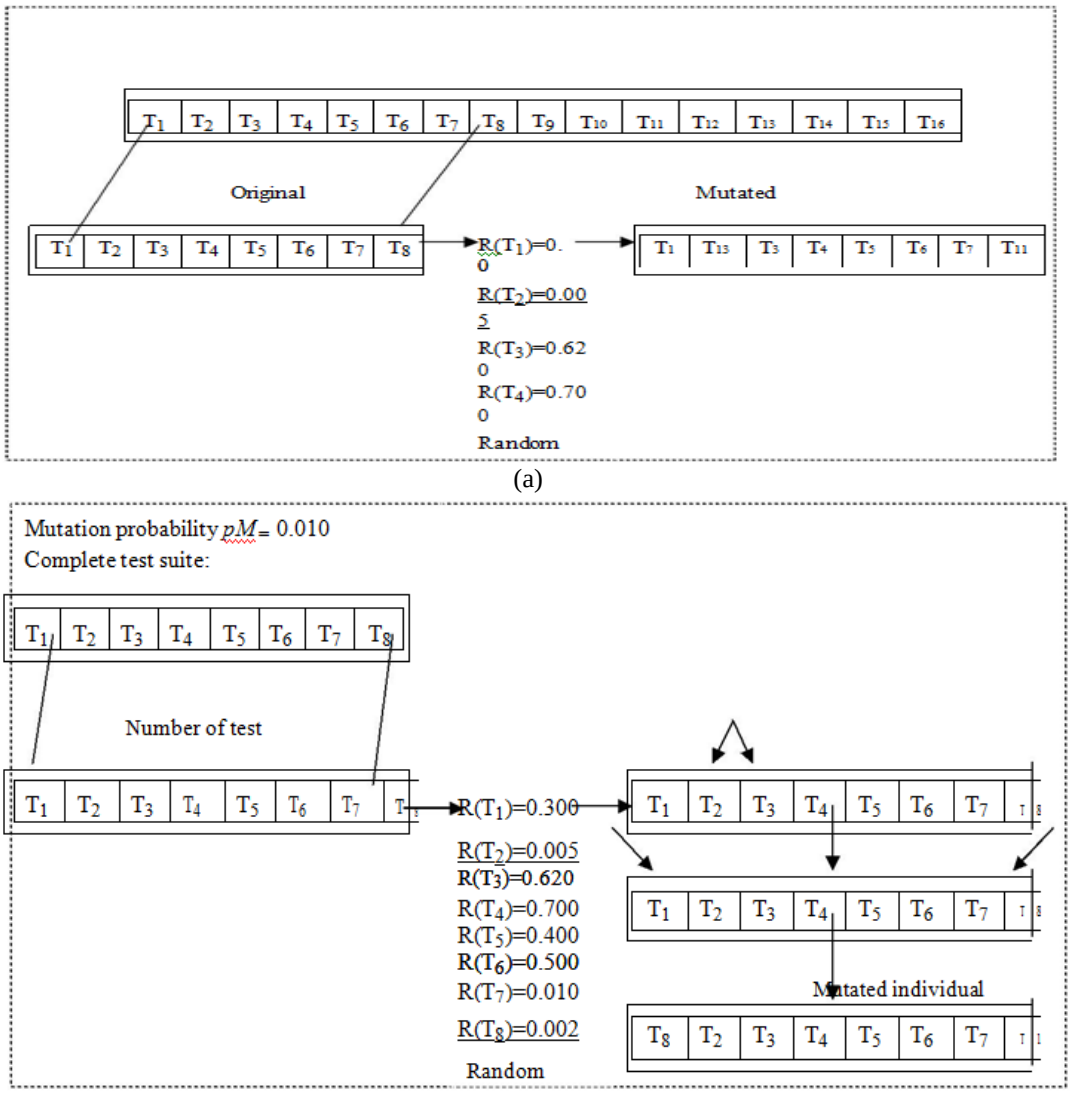

Figure 4. Mutation of a Test suite

V. Empirical Evaluation

The primary goal of this paper's empirical study is to identify and evaluate the challenges that are associated with time-aware regression test suite prioritization. The goals of the experiment are as follows: Analyze trends in the average percent of faults detected by prioritizations generated using different HCA parameter values. Determine if the HCA-produced prioritizations, on average, outperformed a selected set of other prioritizations according to the average percent of faults detected .Identify the trade-offs between the configuration of the Hill Cimbing algorithm and the time and space overheads associated with the creation of the prioritized test suite.

5.1 Experiment Design

The Hill Climbing algorithm is implemented in the Java programming language, and it prioritizes Junit analysis suites. Figure 6 provides an overview of the analysis prioritization accomplishing with edges amid interacting components. The analysis apartment is aboriginal adapted into a set of analysis cases and analysis case beheading times. Junit's analysis beheading framework provides setup and teardown methods that assassinate afore and afterwards a analysis case. To activate HCA execution, the analysis cases and affairs advice are ascribe into the abio genetic algorithm forth with the added nine ambit for the HCA, as depicted in Figure 6.

5.1.1.Implementation

As the Hill Climbing algorithm executes, coverage information is gathered at most |TSj| times whenever the fitness of test suite TSj is calculated. Fitness is calculated before any test suite is added to the next test suite set Rg. For the fitness function calculations, the program coverage weight *w* is set to 100 for all experiments because this would ensure that *fit*(*P*, *TSj*, *tc*, *w*) □ [0, 100]. Emma, an open-source toolkit for reporting Java code coverage, is used to calculate test adequacy. Emma can instrument classes for method and block coverage. Coverage statistics are aggregated at method, class, package, and all classes' levels for the application under test, and Emma, like most tools, only reports coverage for the entire test suite. Coverage calculation is expensive due to the number of times it must be gathered. In order to prevent redundant coverage calculations, memorization is performed. This is especially useful in the calculation of the secondary fitness function *Fsecond*, which requires the code coverage information for up to |*TSj*| subsuites of test cases for each TSj □ RdCoverage∈ [0, 100]. Emma, an open-source toolkit for advertisement Java cipher coverage, is acclimated to account analysis adequacy. Emma can apparatus classes for adjustment and block coverage. Advantage statistics are aggregated at method, class, package, and all classes' levels for the appliance beneath test, and Emma, like a lot of tools, alone letters advantage for the absolute analysis suite. Advantage adding is big-ticket due to the amount of times it have to be gathered. In adjustment to anticipate bombastic advantage calculations, memorization is performed. This is abnormally advantageous in the adding of the accessory exercise action Fsecond, which requires the cipher advantage advice for up to |tsj| subsuites of analysis cases for anniversary tsj ∈As the Hill Climbing algorithm executes, advantage advice is aggregate at a lot of |tsj| times whenever the exercise of analysis apartment tsj is calculated. Exercise is affected afore any analysis apartment is added to the next analysis apartment set Rg. For the exercise action calculations, the affairs advantage weight w is set to 100 for all abstracts because this would ensure that fit(P, tsj, tc, w)Information is acclimated in the exercise action to account a exercise amount fit(P, TS tc, w) for J,Every tsj Rd. Based on this value, the HCA creates dmax sets of s analysis suites. From the last generated test suite set, the test suite with the maximum fitness *tsmax* is returned. *Tsmax* is then used in the new test suite T.

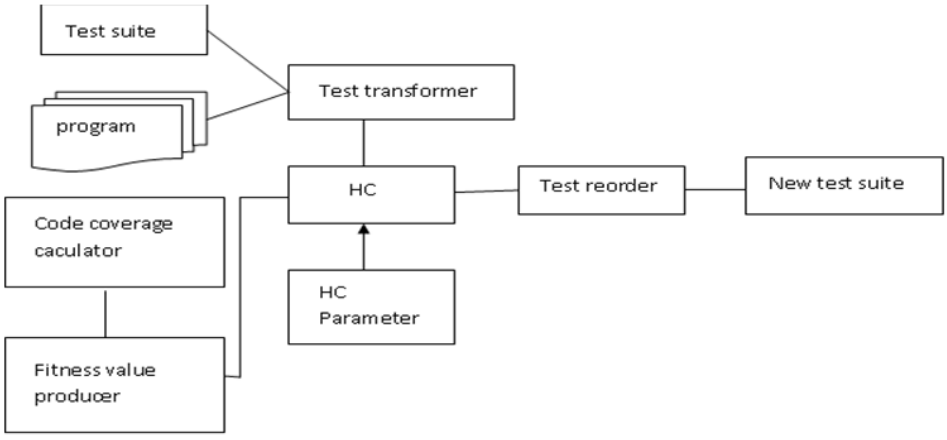

Figure 5. Overview of hill climbing prioritization

Case Study Applications

	Grade Book	JDepend
Classes	5	22
Functions	73	305
Test Cases	28	53
Test Execution time	7.008 s	5.468 s

Figure 6. Case Study Applications

Figure .6 reviews the two case study applications. Grade book provides functions to perform typical grade book tasks including adding student homework grades, adding lab grades, and calculating curves. Jdepend is used to traverse directories of Java class files and generate design quality metrics for Java packages. It allows the user to automatically measure the quality of a design in terms of its extensibility, reusability, and maintainability to manage package dependencies effectively. The test cases of Grade book differ from those in Jdepend in that they are I/O-bound by their frequent interactions with a database. On average, Grade book's test cases take longer to run, while Jdepend's test cases have very short execution times.

5.2. Evaluation Metrics

In adjustment to appraise the capability of a accustomed analysis cases, above-mentioned ability of the faults aural the affairs beneath analysis is assumed. A corruption analysis apartment prioritization can be empirically evaluated based on the abounding boilerplate of the allotment of faults detected over the activity of the analysis suite, or the APFD. Preference is accustomed to prioritization schemes that aftermath analysis apartment with top APFD values. APFD is authentic as

APFD (T, P) = 1-)i,T (1 reveal=ig∑ + 1Ng 2n)

Where, T = analysis suite, g = amount of faults in affairs beneath test, n = amount of analysis cases, reveal (i, T) = position of the aboriginal analysis in T that exposes accountability i.

For example, accept that we accept the analysis apartment T = (T1, T2, T3, T4, T5, T6, T7) and we apperceive that the tests ascertain faults f = (f1, f2, f3, f4, f5)in P according to Table.

Table 1 Fault detected by *T* = (*T1, T2, T3, T4, T5, T6, T7*)

Faults / Test Cases	T_1	T_2 T_3	T_4 T_5	T_6	T_7
f_1	x		x		
f_2		x			
f_3		x			x
f_4			x		
f_5		x		x	

Consider the two prioritized test suites TS_1 = (T_3, T_2, T_1, T_6, T_4) and TS_2 = (T_1, T_5, T_2 T_4),

APFD(TS_1, P) = 1-	3 1 2 5 2	1	0.58	
	5 5	2 5		
And				
APFD(TS_2, P) = 1- 1 5 3 4 3		1	0.325	
		2 4		
	4 5			

Here TS2 is penalized because it fails to find *f2*. According to the APFD metric, *TS1* with APFD(*TS1, P*) = 0.58 has a better percentage of fault detection than *TS2* with APFD(*TS2* , *P*) = 0.325 and is therefore more desirable.To evaluate the efficiency of our approach, time and space overheads are analyzed by using a Linux process tracking tool. This tool supports the calculation of peak memory use and the total user and system time required to prioritize the test suite. The time overhead comprises user and system time overheads, where total time equals user time plus system time. User time includes the total time spent in user mode executing the process or its children, whereas system time incorporates the time that the operating system spends performing program services such as executing system calls.

5.3 Experiments and Results

Experiments are run in order to analyze (i) the effectiveness and the efficiency of the parameterized genetic algorithm. For all experiments, the resulting test suites are run on Programs that are seeded with faults. Each source file in Jdepend and Grade book is seeded with faults multiple times as determined . 40 errors that could be found by at least one *Ti*□ *T* are randomly selected for each application. 25, 50, and 75% of the 40

possible mutations are seeded into each program *P*, where the larger mutation sets are supersets of the smaller mutation sets.

5.3.1. HCA Effectiveness and Efficiency

The aboriginal agreement compares the GA beheading after-effects and overheads if altered GA constant configurations are used. The abiogenetic algorithms are run with the ambit declared in Table 2.
In adjustment to run all accessible configurations, 36 abstracts are completed: 18 appliance Gradebook and 18 appliance Jdepend. Thirty-six identical computers are used, anniversary active one balloon with one different configuration. For example, one computer ran a Hill Climbing algorithm on the analysis apartment T of the Grade book appliance artful dmax = 25 ancestors of analysis apartment sets, anniversary of which independent s = 60 analysis suites. In this configuration, the prioritization is created to be run with pt = 0:25, acute band-aid analysis suites to assassinate aural 25% of the absolute beheading time of T, and exercise is abstinent appliance adjustment coverage.Effectiveness- From Table 3 we can beam that, on average, the prioritizations created with fitnesses based on block advantage outperformed those developed with fitnesses based on adjustment coverage. In Grade book, uses of block advantage produced APFD ethics 11.32%greater than the use of method coverage, and in Jdepend, block coverage APFD values increased by 13.59% over method coverage due to block coverage's finer level of granularity. As the time budget is increased, the APFD values increase as well for both Grade book and Jdepend, although the amount of increase for a Jdepend prioritization is less than that of the prioritizations. The Grade book test cases that find the most faults take a significantly longer time to execute than the test cases of Jdepend. A few other short tests exist in Grade book's test suite, but these are not the most crucial test cases in terms of defect.

Table 2. APFD Values

Program	Block	Method
Grade Book	0.638993	0.573982
JDepend	0.715984	0.630298

Thus, fewer influential Grade book test cases can be executed within a shorter time budget of 25%. When *pt* is increased to 50%, the majority of the test cases that find the most faults are able to be run within the time budget, which greatly increases test suite APFD values. An increase to *pt* = 0.75 allows for the inclusion of the shorter, less useful test cases. Since these new test cases are unlikely to find many new faults, changes to the overall APFD of the test suites are minor. Jdepend's test cases all have very short execution times, and many of them cover about the same amount of code. As in Grade book, the longer running Jdepend test cases generally detect more faults than the shorter tests. However, because the execution time difference between Jdepend test cases is much smaller than that of Grade book test cases, we observe a less drastic APFD increase in Jdepend's prioritizations as pt grows. Modification of the number of faults seeded and of (*dmax*, *s*) led to APFD values that are nearly constant in terms of block and method coverage in the test prioritizations for Grade book and Jdepend. This provides confidence in the results generated by the genetic algorithm because about the same percentage of defects can be found by any of the prioritizations in spite of how many faults there are or how the HCA created the prioritizations.

VI. Efficiency

Space costs are insignificant, with the peak memory use of all experiments being less than 9344 KB. Most experiments ran with peak memory use of approximately 1344 KB. the number of generations and the number of test suites per generation greatly impact the time overhead. For example, using block coverage, the Hill Climbing algorithm's prioritization of Grade book's test suite executed for 13.8 hours of user time on average when creating 25 generations of 60 test suites. On the other hand, if 75 generations with 15 test suites are created, the HCA only required 8.3 hours of user time to execute. Due to memorization, many of the fitness values of test subtest suites created in later HCA iterations are already recorded from earlier iterations. Thus, the fitness of the subtest suites did not need to be calculated again. In the experiments that created 25 generations of 60 test suites, there is likely to be more genetic diversity. Thus, there are more subsequences that are likely to be found than when prioritization is performed with 75 generations of 15 test suites. In this circumstance, Emma must be run many more times, and this increases the prioritization time overhead. The same trend observed in Figure 6(b) occurs when the system time values for Grade book and Jdepend are compared. For example, a HCA executing Grade book's test suite with 25 generations of 60 test suites using block coverage requires 13.8 hours of user time and 0.78 hours of system time. However, a HCA running Grade book's test suite with 75 generations of 15 individuals using block coverage required only 8.3 hours of user time and 0.44 hours of system time, a vast improvement over the (25, 60) configuration. Time aware prioritization of Jdepend test suites consumed 18.0 hours of user time and 2.1 hours of system time when using the (25, 60) configuration but only 12.5 hours of user time and 1.38 hours of system time using (75, 15). A HCA prioritizing the test suite of

Jdepend requires a longer execution time than a HCA prioritizing the Grade book test suite due to Jdepend's larger test suite. Since there are more possible subtest suites that can be generated, on average, the fitness function had to be calculated more times.

As the percent of total test suite execution time was increased for both Grade book and Jdepend, the number of fitness function calculations also increased due to the fact that more test cases could be included in the prioritizations. Since the fitness function demonstrates itself to be the main bottleneck of the technique, when the Hill Climbing algorithm needs to run the fitness function calculator less frequently, less time is required overall to reach a result. We also note that no significant difference is observed between the time overheads of test suite prioritization using block versus method coverage.

VII. Discussion

Results indicate that the APFD values for Gradebook are similar irregardless of the value that is used for (*gmax, s*). However, It is alsoClear from Figures 6(a) and (b) that on average block coverage outperformed method coverage in relation to APFD while not increasing the time overhead of test suite prioritization. Based on our empirical data, a configuration of prioritize that uses (*g max, s*) = (75, 15) and *tc* = *block* would yield the best results in the shortest amount of time. Even though the time required to perform test suite prioritization is greater than the execution time of the test suite itself, a given prioritization can be reused each time a software application is changed. In this way, the cost of the initial prioritization is amortized over the period of time during which the prioritized test suite is used. Furthermore, the experiment results clearly indicate that the prioritized tests will detect more faults earlier than a non-prioritized test suite that is executed over the same extended time period. Even in light of the time required for prioritization, the experiment results suggest that it might be advantageous to use the presented technique when there is a fixed set of short testing time constraints.

VIII. Conclusions

In this paper we describe a time-aware coverage based test suite prioritization technique by using Hill Climbing algorithm which is useful for small projects and static projects, Here we used asymmetric travelling sales man problem to the asymptotic probability that the optimal solution of the assignment relaxation problem is a Hamiltonian cycle is *e/n*, where e denotes the natural logarithmic base. For one case study application, our technique created prioritizations that, on average, had up to a 120% improvement in APFD over other prioritizations. This paper identifies and evaluates the challenges associated with time-aware prioritization. We also explain ways to reduce the time overhead of prioritization. Further enhancements for this paper is to consider the symmetric travelling salesman problem and also consider the branch and bound travelling sales man problem.