Arabian Journal for Science and Engineering (2019) 44:9599–9625
https://doi.org/10.1007/s13369-019-03817-7

RESEARCH ARTICLE - COMPUTER ENGINEERING AND COMPUTER SCIENCE

Performance Comparison of Multi-objective Algorithms for Test Case Prioritization During Web Application Testing

Munish Khanna[1] · Achint Chaudhary[2] · Abhishek Toofani[1] · Anil Pawar[1]

Received: 21 July 2018 / Accepted: 14 March 2019 / Published online: 27 March 2019

Abstract
Test case prioritization (TCP) is a widely accepted and extensively used strategy during regression testing. TCP is the permutation of test cases to enhance efficiency in achieving performance goals. These goals can belong to the category of single objective problem or multi-objective problem. This empirical study focuses on three objectives wherein two objectives are to be maximized and the remaining one minimized. During this study, three websites and various versions were created on which non-dominated sorting genetic algorithm-II and variant of non-dominated sorting artificial bee colony algorithm were applied to prioritize sequence of test cases. The problem size varies from small-size fault matrix (34 × 27) to mid-size fault matrix (157 × 128). Performance of the two algorithms was measured on various parameters and also verified on the basis of statistical testing. An alternate approach for solving this multi-objective problem, based on dynamic programming, is also proposed in this study, and it is concluded that performance of this algorithm is at par with other suggested ones.

Keywords Multi-objective optimization · NSGA-II · NSABC · Dynamic programming · Test case prioritization · Search-based software engineering

1 Introduction

In the current scenario, E-commerce is spreading wings throughout the world, in both developed countries and under-developed countries, and the number of web-based systems and their users are increasing at a fast pace. To remain sustainable in this highly competitive era and retain the customer base by meeting their endless expectations, web-based applications (dynamic websites in our case) have to undergo technology enhancement and frequent updates. Regression testing is performed to ensure high quality and to validate such updates without interrupting the provided services of end products/deliverables.

✉ Munish Khanna
Munishkhanna.official@rocketmail.com

Achint Chaudhary
Chaudharyachint08@gmail.com

Abhishek Toofani
abhishektoofany@gmail.com

Anil Pawar
anilpawar25@gmail.com

1 Hindustan College of Science and Technology, Mathura, India

2 Indian Institute of Sciences, Bengaluru, India

Regression testing [1–3] is an inevitable and expensive process implemented during the maintenance phase of the evolving software so as to attain best quality. It makes up a significant percentage of the overall cost of software development and maintenance. It is undertaken when updates are made to an existing software system. The ultimate purpose of regression testing is to provide best quality assurance practices and to ensure that the updated features do not influence the behavior of the existing, unchanged part of the original flawless software. Different strategies are followed for the implementation of regression testing, test case prioritization being one of them.

TCP [4] is formulated to find new permutation of test cases, T' belonging to a set of permutations SP such that the value for $f(T')$ would be greater than or equal to any other permutation T'' belonging to SP, i.e., to find $T' \in \mathrm{SP}$ such that $(\forall T'')(T'' \in \mathrm{SP})\ (T'' \neq T')|f(T') \geq f(T'')$ where f is a function when applied to any such permutation would yield an award value for that permutation. This prioritized permutation of test cases should execute in such a fashion that the test case having the highest award value as per the given testing criteria would be executing earliest, followed by test cases having lesser award value. The testing criteria can be uni-objective, where one parameter has to be maximized/minimized, or multi-objective where there are multiple parameters out of which some are to be minimized and the remaining ones maximized. Therefore, TCP problem can be categorized into a single objective optimization problem or a multi-objective optimization problem, similar to other software engineering problems. In the presented study, three parameters are taken into consideration which are: minimization of testing cost (execution time of test cases), maximization of severity detection per test case execution (early detection of severe faults) and maximization of fault severity detected per unit of test cost.

The efficacy of prioritized test cases is measured in terms of average percentage of parameter detection where the parameter can be faults, branch, loops, blocks or statements. The focus of this study is on faults detection; hence, efficiency is measured in terms of average percentage of faults detection (APFD). While calculating, APFD execution time of each test case was considered unit time; at the same time, severity of each fault was also considered to be unity. These values are not practically feasible in the real-life scenario. Rothermal et al. [4,5] improved this formula, by introducing cost cognizant average percentage of faults detection ($\mathrm{APFD_C}$), which is more viable and practical. It is the measure of unit of fault severity detected per unit of test cost.

The $\mathrm{APFD_C}$ for the test suite is calculated using Eq. 1.

$$\mathrm{APFDc} = \frac{\sum_{\mathrm{i}=1}^{\mathrm{m}} \left(f_{\mathrm{i}} \times \left(\sum_{\mathrm{j}=\mathrm{TF_i}}^{\mathrm{n}} T_{\mathrm{j}} - \frac{1}{2} T_{\mathrm{TF_i}} \right) \right)}{\sum_{\mathrm{i}=1}^{\mathrm{n}} T_{\mathrm{i}} \times \sum_{\mathrm{i}=1}^{\mathrm{m}} f_{\mathrm{i}}} \tag{1}$$

where T is a test suite consisting of n test cases whose costs are $T_1, T_2, \ldots, T_\mathrm{n}$, respectively, F is a set of m faults exposed by T with severities $f_1, f_2, \ldots, f_\mathrm{m}$, TFi is the first test case of an ordering T' of T that exposed fault i, m is the total number of faults and n is the total number of test cases.

Harman et al. [6] discussed various parameters and their coverage within the program such as block coverage, decision coverage and statements coverage by different categories of algorithms. In the presented work, focus is laid on the coverage of faults since their coverage would cover blocks, decisions and statements proxy wise. In this empirical study, faults belonging to diverse categories were manually transplanted at random locations in the dynamic website under test. Relevant test cases having the capability to expose these manually seeded faults were generated using Selenium testing tool.

In the present work, two state-of-the-art multi-objective algorithms [non-dominated sorting genetic algorithm-II (NSGA-II) and variant of non-dominated sorting artificial bee colony algorithm (vNSABC)] are applied to solve the above-stated multi-objective optimization problem. The results generated on a range of size instances of different versions of various dynamic websites from both algorithms are compared. Meanwhile, performances of these two algorithms are also measured on other parameters such as time taken to solve the problem and how many iterations are required to compute the best results. Statistical testing is also performed to compare these two algorithms. Concurrently, one non-incremental approach, the Dynamic Programming-based approach, is also discussed for solving the problem. In this approach, various parameters are presented including count of recursive calls and count of dynamic table look up for solving the problem. A complete code of dynamic programming, written by the authors in Python, is presented in the "Appendix" section. Moreover, a diagrammatic presentation of two parameters is also presented in the sections dealing with results and discussions. The proposed work is an attempt to answer the discrete combinatorial multi-objective optimization problem having contradictory objectives.

Academic contribution of this experimental study can be considered as solving the following questions:

Q1 Is vNSABC able to generate better results, within less time and less number of iterations, in comparison with NSGA-II, in solving the suggested problem? and

Q2 Is there any other non-incremental algorithm-based approach to solve the problem in feasible exponential time?

This experimental study provides three different solutions to the tester fraternity. They can make use of it according to requirements and objectives.

The rest of the paper is organized as follows: Sect. 2 presents prior studies related to TCP, TCP in multi-objective environment, usability of NSGA-II and NSABC. Section 3 gives a brief description of implemented algorithms and suggested objectives (proposed approach). Section 4 presents in detail the experimental setup and experiments conducted. Section 5 illustrates results, analysis and discussion. Section 6 presents the conclusion, future scope and threats to validity. In Sect. 7, the "Appendix" section, an algorithm based on dynamic programming approach, memo (created in random access memory) table and a detailed Python code for dynamic programming implementation are provided.

2 Related Works

This section has been divided into three subsections. First subsection focuses on prior published studies on TCP. Next subsection showcases previous studies which present practitioners efforts in solving various problems using NSGA-II. Finally, the last subsection presents earlier studies which give researcher's efforts for solving various problems using NSABC.

2.1 Prior Works Related to TCP

Diverse attempts have been made by the researcher's fraternity to solve TCP, some prominent work being discussed below.

Elbaum et al. [7] introduced two factors, varying test costs and severity, while prioritizing test sequence so as to enhance rate of fault detection, i.e., verifying how fast the given test set exposes faults during the testing process. They also proposed a new metric for analyzing rate of fault detection of a prioritized test set that incorporates varying test cases and fault costs.

Another benchmark study by Jiang et al. [8] proposed a coverage-based adaptive random testing test case prioritization technique which is supported by white-box coverage information and tested on empirical study. Authors justified that their proposed approach is superior to random ordering for earlier detection of failures.

Tanzeem et al. [9] proposed a quality metric that estimates quality of test cases using their similarity to previously failing test cases. The experimentation process executed on five software's reveals that anticipated similarity-based quality measure is appreciably more effective for prioritizing fault-revealing test cases compared to existing test case quality measures (or traditional history-based approach). Authors also revealed that their quality metrics is superior to supplementary test quality metrics, e.g., coverage and test size in prioritizing tests.

The next prioritization approach presented by Arafeen et al. [10] investigated whether efficiency of test case prioritization can be improved if requirement-based clustering approach (incorporating traditional code analysis information) is applied. They also proved that results may vary by cluster sizes, and by grouping similar type of requirements, efficiency can be improved. They have used requirements to group test cases—on two projects and their versions, rather than focusing on utilizing source code information.

Laali et al. [11], in their recent published study, focus on online test cases prioritization technique (using feedback, i.e., localization of fault), rather than classical offline test case prioritization, and to justify the innovativeness, they applied the proposed approach on Siemens 7 programs data set.

In a recently published experimental study, Wang et al. [12] focus on limitations of the traditionally followed coverage-based test case prioritization and proposed a new QTEP, quality aware test case prioritization technique, which leverages two dominant code inspection techniques, i.e., a typical statistic defect prediction model and a typical static bug finder, to weight source code in terms of fault proneness. They experimented using 16 variants of the proposed technique on various versions of seven open source Java projects to show the improvement in results over coverage-based test case prioritization technique.

Mei et al. [13] proposed a new approach named as JUPTA, JUnit test case prioritization techniques operating in the absence of coverage information, which specifically works on the Java programs which are tested under JUnit framework. The proposed approach understands the static call graphs of JUnit test cases and the program under test to guess the capability of each test case to attain code coverage and then schedules the sequence of these test cases based on these estimates.

Stephen et al. [14] in their empirical work concluded that the presented static black-box TCP technique performs better than existing static black-box TCP techniques and has comparable or better performance than two existing execution-based TCP techniques.

2.2 Previous Works Completed Using NSGA-II

There exists prior studies which conclude that NSGA-II has been successfully applied by researcher's fraternity working in various domains [15,16], of engineering, not limited to software engineering. Some studies related to optimization problems, of software engineering attempted by NSGA-II, are reported below.

Zhang et al. [17] reported the suitability of NSGA-II for solving a multi-objective next released problem under the class of requirement engineering.

Ruiz et al. [18] in their experimental study included three conflicting objectives: development time, cost and productivity, to support software project managers in taking decisions in a multi-objective perspective. It was reported that NSGA-II algorithm was able to converge quickly to a set of optimal solutions.

Wong et al. [19] presented a solution for multi-objective optimal test resource allocation problem, considering reliability of the system, testing cost and total testing resources consumed as parameters with the help of NSGA-II and harmonic distance-based multi-objective evolutionary algorithm (HaD-MOEA).

Chaudhary et al. [20] considered two conflicting objectives, uniform distribution of test cases over the given range and maximization of code coverage, while solving multi-objective automatic test data generation

Mondal et al. [21] investigated multi-objective test case selection problem with the help of NSGA-II, with three conflicting objectives maximizing code coverage, maximizing diversity among the selected test cases and minimizing test execution time.

Yoo et al. [22] attempted to solve the multi-objective test case selection problem considering fault coverage, code coverage and cost, using additional greedy algorithm, NSGA-II and its variant vNSGA-II.

Of late, researcher's fraternity has turned its attention to solving test case reduction problem in a multi-objective environment, with objectives such as code coverage, requirement coverage and execution time [23], rather than in the classical single objective scenario. Authors of the presented study propose multi-objective test case reduction (MORE) technique and apply NSGA-II for validation of the proposed technique on multiple Java programs. The generated results were promising and show that there is scope of improvement.

Zheng et al. [24] applied classical greedy, NSGA-II, MOEA/D and variant of MOEA/D on SIR repository for solving multi-objective problem, with conflicting objectives, code coverage vs. execution time, test case minimization problem, during regression testing.

Canfora et al. [25], in a recently published empirical work, consider maximum effectiveness and minimum cost as objectives which are to be optimized while solving multi-objective logistic regression problem and multi-objective decision trees problem using formulated defect prediction.

In another published study by Marchetto et al. [26], 21 java applications were considered as the subject for test case prioritization in a multi-objective scenario where source code coverage, requirements and test case execution time were the objectives. Authors proved that NSGA-II outperforms other competitive algorithms on the basis of APFD.

Thus, the literature survey covered above reveals that no major significant work has been presented by researcher's community in test case prioritization in the multi-objective scenario using NSGA-II. This motivates us to solve the suggested problem using the said algorithm.

2.3 Previous Attempts Successfully Accomplished Using ABC and NSABC

Regarding usability and motivation behind ABC and NSABC for solving the problem in hand, a small literature survey is presented below in which some prior relevant studies are discussed where similar types of problems of software engineering/testing have been solved. Moreover, the discussion presented below also proves that no key work has been published in solving the suggested problem with selected parameters using NSABC. This gives us the motivation for solving the same using NSABC, as there are indications in some prior studies, like [27], that the algorithm has the capability to solve similar category of problems.

Karaboga et al. [27] presented a comprehensive survey of the applications of ABC in a choice of engineering problems from various fields such as industrial engineering, mechanical engineering, electrical engineering, electronics engineering, control engineering, civil engineering, software engineering image processing, data mining, sensor networks, protein structure and many more.

Karaboga et al. [28] proposed combinatorial ABC for solving traveling salesman problem which falls under the category of NP-Hard combinatorial optimization problem.

Lam et al. [29] compare ABC with other algorithms such as ant colony optimization (ACO) and genetic algorithm (GA), while proposing automatic generation of feasible independent paths followed by test suite optimization.

Chong et al. [30] applied ABC algorithm for solving job shop scheduling problem which also falls under the category of NP-hard problems. The results generated by ABC were promising, and they were compared with the performance of ACO and tabu search to evaluate the performance of ABC.

Kaur et al. [31] proposed ABC algorithm for regression test suite prioritization. Average percentage of conditions covered was the criteria to measure efficacy of the proposed algorithm. Maximum code coverage was used for explaining functioning of the algorithm.

Srikanth et al. [32] applied ABC to generate optimal number of test cases to be executed on the software, i.e., generation of optimized test suite. Full path coverage has been assured by the approach.

Joseph et al. [33] blended particle swarm optimization with ABC and named it as particle swarm artificial bee colony algorithm (PSABC) for test case optimization. PSABC prioritizes test cases to reduce time and cost of regression testing. Maximum statement and fault coverage with minimum execution time was the main objective of the work.

Mala et al. [34] proposed a framework for software test suite optimization using ABC approach. Dahiya et al. [35] applied ABC for structural testing of ten real-world programs. Konsaard et al. [36] made use of ABC algorithm to prioritize test suites based on code coverage.

Aghdam et al. [37] applied ABC algorithm for solving the issue of test data generation where branch coverage was the criteria as a fitness function. The process was implemented on seven standard programs, and the generated results were outstanding when compared with other competitive soft computing techniques.

Li et al. [38] presented ABC algorithm for multi-objective optimization problem. Simulation results on multi-objective test functions verified the validity of the presented algorithm.

Amarjeet et al. [39], in their empirical study, try to solve software module clustering problem using a variant of many-objective artificial bee colony algorithm. They compare the proposed algorithm with other four algorithms over seven clustering problems to prove better performance.

Mann et al. [40] made use of ABC algorithm for path-specific approach for automatic test case generation. Based on the findings, authors conclude that ABC outperforms other suggested soft computing-based algorithms.

3 Proposed Approach

Description of the algorithmic details is presented in the first half of this section, while a discussion on suggested parameters is presented in the second half.

3.1 Implemented Algorithms

Three algorithms have been shortlisted for implementation in the present study for performance evaluation while solving the problem in hand.

As mentioned in the previous section, NSGA-II has been widely accepted by the researcher's community worldwide and has been used not only in computer science but also in other fields of engineering such as civil, mechanical, electrical and system engineering [15,16]. The algorithm is used to solve problems where care needs to be taken for more than one objective simultaneously and; moreover, these objectives may conflict with each other in many instances. It is basically a selection technique which helps us to select best solutions (with respect to all objectives) among all solutions, and this selection is done by non-dominated sorting. Fronts are created in this algorithm on the basis of dominance; Pareto front (first front) consists of solutions that are non-dominated to each other; no other solution is dominating it nor are of major interest. Thus, Pareto optimal fronts are created in this algorithm, where solutions in one front dominate solutions of other fronts, and solutions in the same front are non-dominated. Within a front, we calculate crowding distance to find diversity between solutions in any front. It is implemented by incorporating special fast non-dominated sorting technique into the genetic algorithm. Crowding distance is used to estimate versatility, density of solutions surrounding any particular solution. It is used to preserve the shape of the front and to remove redundant solutions in the high-density region. During this work, non-dominated sorting functionality and crowding distance functionality of NSGA-II are implemented as presented in relevant prior studies [41,42]. Meanwhile, implementation information, at the abstract level, about NSGA-II and genetic algorithm (GA) is shown below. Certain modifications have been made in available NSGA-II which can be considered as author's contribution. After the completion of the last iteration of GA, three best solutions from the first front (for the suggested three parameters) are ascertained with application of linear search among the stored solutions. These three solutions satisfy maximum unit of fault severity detected per unit of test cost, maximum severity detection and minimum test case execution time to expose all the faults, respectively.

The NSGA-II algorithm, at abstract level, is presented below.
NSGA-II algorithm

1.	Initialize population size
2.	Start loop till max iteration
3.	Calculate fitness of each individual based on Pareto front and crowding distance [41]
4.	Generate new population (children) using selection crossover and mutation technique
5.	Select best population among previous (parents) and new (child) populations
6.	Terminate loop if reach max iteration
7.	Select best *k* solutions from population

Pareto front (and crowding distance) is not discussed in a detailed manner because it is computed in similar fashion for related problems. Previous published study [41] can be further referred to generate Pareto fronts and to calculate crowding distance required in the algorithm.

The genetic algorithm, at abstract level, is as follows.
Genetic algorithm

1.	Generate $2n$ number of initial solutions randomly, where n is the problem size (test cases in our case)
2.	Apply tournament selection procedure for parent's selection
3.	Apply crossover operation on selected parents
4.	Implement mutation process on the children generated during step 3
5.	Retain the best 50% solutions on the basis of dominance. Generate remaining 50% solutions randomly
6.	If the number of iterations performed is less than fifty times the problem size, goto step 2, otherwise exit

The practical usability of ABC and NSABC in research area is already presented in the previous section. The variant of NSABC (vNSABC) algorithm is presented below. Similar to NSGA-II, non-dominated sorting functionality and crowding distance functionality are implemented as mentioned in relevant prior studies [41,42]. The ABC algorithm is implemented as presented below along with detailed explanation of each and every step since a few modifications are suggested by the authors within the traditional ABC algorithm. Since modifications are recommended in the ABC algorithms, it is termed as variant ABC, by introducing a function called pool (), which is author's contribution.

ABC algorithm

```
Input to algorithm: test cases versus fault matrix (TCVFM), severity
  matrix (SM) and test case execution time matrix (TCETM)
1.  Create the initial solutions randomly whose count is twice the
    number of test cases; moreover, every test case will appear
    only once in these solutions. One constraint that has been
    compulsory implied on the solutions is that every test case
    will be given an opportunity to occupy first position in the
    solution; remaining positions can be filled randomly by the
    remaining $n - 1$ test cases. This implies that in the first and
    second solutions, the first position is occupied by the first test
    case, $T_1$. Similarly, in third and fourth solution, first posi-
    tion is occupied by second test case $T_2$ and so on. Thus,
    finally, we have 2n number of solutions where in $2n - 1$
    and $2n$th solutions first position is occupied by $T_n$th test
    case
2.  Functionality of employee bees
    (a) The solutions, generated during the above step, are supplied
    to the employee bees.
    (b) The employee bees having solution $X(i)$ with them start
    searching for the neighbor of this solution, named as $X(k)$, to
    produce a new solution $Y(i)$ in accordance with Eq. (2) where Ø
    is a random number between 0 and 1
    $Y(i) = X(i) + Ø^*(X(i) - X(k))$      (2)
    All the solutions will play the role of $X(i)$ once; in the meantime
    $X(k)$ is selected out of the remaining ones, randomly in such a
    manner that $X(i)$ and $X(k)$ are dissimilar. A pool function has
    been devised, our contribution, for the implementation of Eq.
    (2). In depth explanation of the function pool is presented after
    this algorithm
    (c) Calculate the fitness of the original one $X(i)$ and the newly
    generated solution $Y(i)$. On the basis of their finesses, select the
    better choice and reject the poorer one
3.  Functionality of onlooker bees
    (a) Sort all the sequences on the basis of dominance
    (b) Discard lower half, 50%, of the solutions
    (c) Now onlooker bee will again search and select the neighbor
    randomly from these selected solutions and try to generate a new
    solution $Y(i)$, using Eq. (2)
4.  Functionality of scout bees
    (a) Sort all the sequences on the basis of dominance. Best prof-
    itable 50% of the solutions will be retained on the basis of
    dominance and the remaining ones will be discarded
    (b) Produce new solutions randomly, equal to the number of
    solutions discarded; the imposed restriction mentioned in step
    1 is non-applicable on these newly generated random solu-
    tions
    (c) These solutions will become food source for the employee
    bees and process switches toward Step 2 (a)
5.  Termination criteria
    (a) This procedure will be repetitive utmost up to five times the
    number of test cases
```

Function pool () is implemented as follows

Pool function

```
1. Randomly select the solution $Y$ as a neighbor of $X$ such that $X \neq Y$.
   Let $X_1$ and $Y_1$ be the first test cases of the sequences $X$ and $Y$,
   respectively. Append the test case, having the highest parameter,
   AOC, value (Eq. 2), in the initial new empty test sequence which is
   to be generated; left one will become a part of the initially created
   empty pool
2. Select second test case, $X_2$ and $Y_2$, respectively, from $X$ and $Y$,
   compare $X_2$, $Y_2$ and the test cases which are part of pool on the basis
   of factor $f$. Choose one, out of these, which has the highest value of
   factor $f$ at this point of time. Factor $f$, of any arbitrary test case, $i$ is
   calculated as the sum of severity of all the undetected faults exposed
   by test case $i$ divided by execution time of the test case. Add the
   unexecuted test case(s) to the pool
3. If two of more candidate test cases have equal and highest factor $f$,
   select one test case randomly to come out of the tie situation. This
   selected test case will get appended to the new solution. Revise
   TCVFM on the basis of this selection. If all members of the pool
   along with $Xi$ and $Yi$ have equivalent factor $f$, then add Xi and Yi
   into the pool then select $Xi + 1$ and $Yi + 1$ and compare these with
   pool members. Repeat this step till the tie condition breaks down
4. If all the faults are exposed, but some test cases are still left in the
   pool or have yet not become part of the pool, select all the remaining
   test cases and append these into the solution in a random fashion
```

Dynamic programming is another classical algorithmic approach to solve many classical problems, not only related to computer science but also in other fields of engineering [43]. This motivates the authors to apply this algorithm for solving the problem in hand. The algorithm is devised for the suggested multi-objective problem. Detailed explanation of the algorithm along with code and performance generated by the algorithm is compared with the vNSABC in the upcoming sections. For the reader's understanding, Table 1 has been compiled which lists the acronyms (abbreviations) along with the expanded forms.

3.2 Multi-objective Functions Setup

Three parameters are selected in these studies which are to be either minimized or maximized. The first parameter that is to be maximized is unit of fault severity detected per unit of test cost, also known as APFD_C (Eq. 1). This is a standard parameter followed worldwide by researchers and testers fraternity for calculating efficiency of prioritized test sequences. We have thoroughly analyzed the behavior of how the APFD_C actually works, i.e., how the area of the graph which is to be maximized is filled. This is explained below. Let there be following relevant notations:

$\sum$cost: total cost of all the test cases.
SEV[i]: severity exposed by ith test case/$\sum$sev.
Cost[i]: cost of ith test case/$\sum$cost.
Undet: undetected faults.
FRC: fraction of remaining cost.
$\sum$sev: total severity of all the faults.

Table 1 Acronyms of various terms used in this empirical study

Acronym	Expanded form	Acronym	Expanded form
GA	Genetic algorithm	$APFD_C$	Cost cognizant average percentage of fault detection
vNSABC	Variant of non-dominated sorting artificial bee colony algorithm	NSGA-II	Non-dominated sorting genetic algorithm-II
TCVFM	Test cases versus fault matrix	SM	Severity matrix
TCETM	Test case execution time matrix	Unexec	Unexecuted test cases
Undet	Undetected faults	FRS	Fraction of remaining severity
FRC	Fraction of remaining cost	$\sum$cost	Total cost of all the test cases
$\sum$sev	Total severity of all the faults	AOR	Area of this region
US	Undetected severity	SDT	Severity detected by ith test case, out of not yet detected
PT	Position of the ith test case in test sequence	n	Number of test cases in test sequence
CDS	Current detected severity	RSD	Rate of severity detection
WiVj	jth version of ith website		

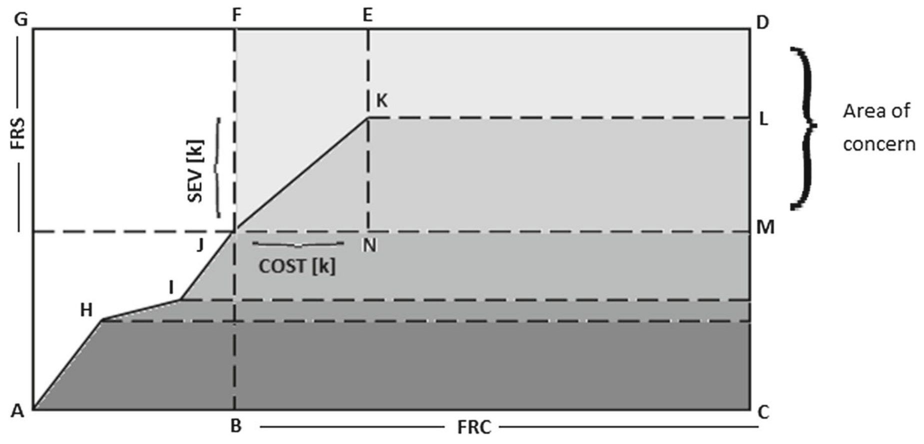

Fig. 1 Filling up of graph region per execution of test case, where cost and severity are the concern

sev[j]: severity of jth fault/$\sum$sev.
Unexec: unexecuted test cases.
FRS: fraction of remaining severity.

Suppose $K-1$ test cases are executed and that next chance is given to Kth test case for which the cost, COST[K], will be incurred and possibly new severity SEV[K] (out of remaining one) will be exposed. In other words, SEV[K] will be exposed/deducted from FRS and COST[K] is deducted from the FRC. Region shown in Fig. 1 is guaranteed to account for $APFD_C$.

Area of this region (AOR, refer Fig. 1) can be calculated as

$$\begin{aligned}\text{Parameter (AOR)} &= (\text{FRC}^*\text{SEV[K]}) \\ &\quad -(1/2^*\text{COST[K]}^*\text{SEV[K]}) \\ &= \text{SEV[K](FRC} - 1/2\text{COST[K])} \qquad (3)\end{aligned}$$

The value obtained from Eq. (1) is equal to summation of parameter (AOR) resulting from the first test case to the last test case of the test sequence. That is, summation of the area is covered by first test case through last test case. The ultimate objective of TCP is to find the test sequence for which the generated area would be maximum.

A test case, next in order, will be selected for execution.

The algorithm given below, with complexity n^2, will generate total $APFD_C$ from the test sequence.

```
1.  FRS=FRC=1 , APFD_C=0 and given sequence.
2.  While (FRS!=0)
    {
          a.  Select next test case from sequence.
          b.  FRS = FRS-SEV  [of Selected test case]
          c.  FRC=FRC-COST[of Selected test case]
          d.  APFD_C =APFD_C + AOR contribution of executed test case.
    }
```

The next suggested parameter which is related to severity of the faults is rate of severity detection per execution of test case which is to be maximized. Authors have devised the equation for this, which is given below:

$$\text{Sev} = \sum_{\text{i}=1}^{\text{n}} \frac{\text{US}(\%) \times \text{SDT}}{\text{PT}} \tag{4}$$

where

US undetected severity;
SDT severity detected by ith test case, out of not yet detected;
PT position of the ith test case in test sequence;
n number of test cases in test sequence.

The value of severity of the fault varies with the stakeholder's perspective. Four stakeholders are taken into considerations which are designer, coder, tester and end user. A survey has been conducted among these stakeholders having sufficient relevant experience, and severity has been finalized on the basis of their responses.

Another question which comes to mind: is it really worth to consider severity as a parameter?

This question is justified by the scenario given below. Suppose there exist three test cases, T_1, T_4 and T_7, which have equal candidature for next execution. T_1 detects six faults of three severities each; T_4 detects eight faults of four severities each; and T_7 exposes seven faults of five severities each. When maximization of fault detection per execution of test case is the objective, then T_4 will be the first choice; but in case of maximization of severity detection per execution of test case, T_7 will be the premium choice. Hence, it can be said that the ultimate purpose of parameter "rate of severity detection" is earlier detection of more severe faults. Similar to the discussion on the above parameter, we have also devised the equation and finally drawn the graph for this parameter also. The summation of the covered area of Fig. 2 is equal to summation in Eq. (4) (Sev).

Let there be following relevant notations

RSD rate of severity detection;
US undetected severity;
CDS current detected severity;

Initially, since no test case is executed, undetected severity is equal to summation of all the elements of the severity array. US multiplied by CDS gives area of the rectangle formed inside the region. Factor "position" is divided to put more concern on earlier detection of faults with same severity. The equation of RSD is given below (Eq. 5)

$$\text{RSD} = \frac{\sum_{\text{position}}^{\text{length}} \frac{\text{US}\times\text{CDS}}{\text{position}}}{\text{Total\_Severity}} \tag{5}$$

Let us consider a scenario for which Fig. 2 has been built as shown above. In this scenario, there are three test cases,

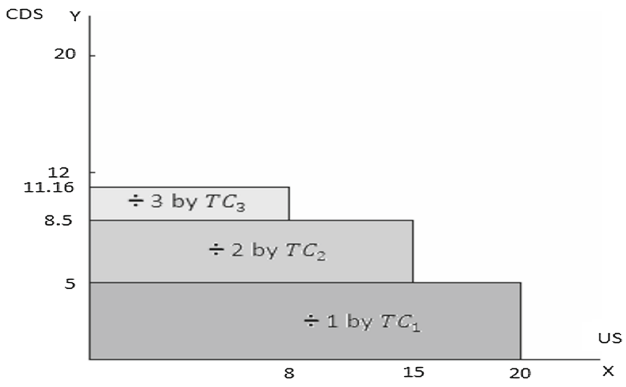

Fig. 2 Filling up of graph region per execution of test case, where severity is the concern

TC_1, TC_2 and TC_3, which can detect fault whose summation of severity is 5, 7 and 8, respectively. Initially, the total undetected severity is equal to summation of all severities, which is 20 in this case. When the first test case is executed, faults are detected whose summation of severity is 5, the covered area, by TC_1 of the graph, will be $((20\times05)/1)$. Now, the second test case TC_2 is going to be executed for which US value reduces to 15 from the initial value of 20; area covered by TC_2 is $((15 \times 07)/2)$; now, the severity to be detected would be 8. Finally, the remaining undetected severity is 8. TC_3 is going to be executed and will detect the whole remaining severity; meanwhile, the area covered by this test case will be $((8 \times 8)/3)$. The value of summation of these three areas would be equal to the value computed from Eq. (4).

The third objective is minimization of test case execution time so as to expose all the faults. It should be noted here that the sequence of selected test cases does not matter. The ultimate purpose is so select the minimum number of test cases which can expose all the faults.

4 Experimental Setup

Four jsp-based dynamic websites, consisting of 50–100 pages and 5000–11,500 lines of code, were selected for testing purpose. Various versions of these websites were created by addition/deletion/modification at code and functionality levels. Three websites belong to major projects of postgraduate students while the remaining one belongs to an inventory company that was built by IT professionals. As previously mentioned, faults were manually injected into the websites; up to 157 faults were introduced in the websites to make the system faulty [44]. These faults belonged to various categories and were manually injected at random locations in different versions. For exposing these faults, test cases were created using Selenium testing and replay tool. It was installed on the client side and used to capture/replay user interactions with the system (website). For calculation of execution time of test cases, EMMA and Selenium test tools

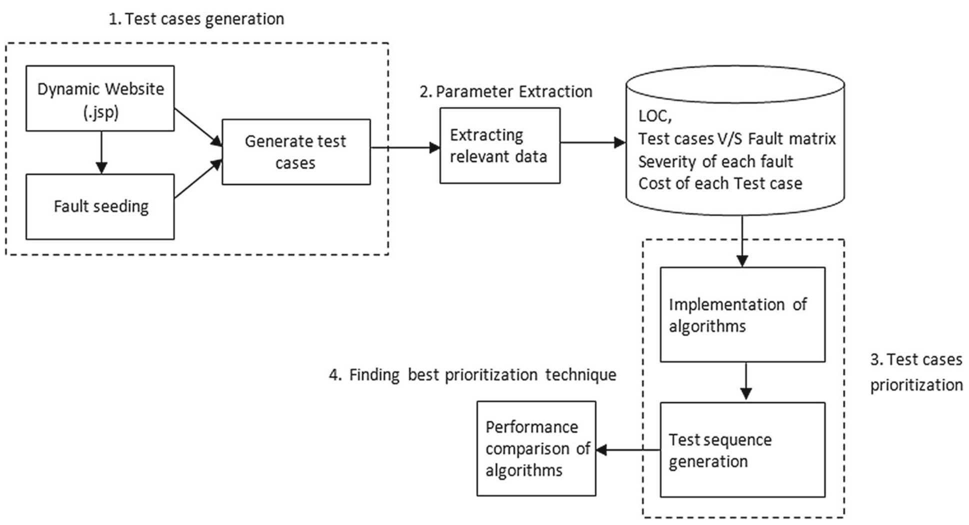
1. Test cases generation
Dynamic Website (.jsp)
Fault seeding
Generate test cases
2. Parameter Extraction
Extracting relevant data
LOC, Test cases V/S Fault matrix Severity of each fault Cost of each Test case
Implementation of algorithms
Test sequence generation
3. Test cases prioritization
4. Finding best prioritization technique
Performance comparison of algorithms

Fig. 3 Framework of the proposed model

Table 2 Version data of the five websites used in test case prioritization

Websites and their versions	w_1v_1	w_1v_2	w_1v_3	w_1v_4	w_2v_1	w_2v_2	w_2v_3	w_3v_1	w_3v_2	w_3v_3	w_3v_4	w_4v_1	w_4v_2	w_4v_3	w_4v_4
KLOC	5.93	7.76	9.20	8.98	8.20	8.99	10.78	6.87	6.10	7.28	7.34	11.76	12.07	12.92	14.03
Test cases	34	41	61	74	52	58	78	89	90	126	134	69	84	97	157
Faults	27	23	25	35	23	49	36	64	73	109	105	31	45	52	128

Table 3 Result matrix depicting performances in achieving all the parameters by NSGA-II and vNSABC when applied on all versions of all the websites under test

Version	Matrix size	NSGA-A ($APFD_C$)	NSABC-A ($APFD_C$)	NSGA-E (execution time)	NSABC-E (execution time)	NSGA-S (severity)	NSABC-S (severity)
W_1V_1	34 × 27	0.930979	0.930979	52,128.571	52,128.571	48.5815	48.5815
W_1V_2	41 × 23	0.868163	0.870644	45,071.428	41,071.428	48.9428	48.9428
W_1V_3	61 × 25	0.961182	0.961198	51,400	52,014.285	66.7787	66.7787
W_1V_4	74 × 35	0.964828	0.964828	70,585.714	70,585.714	60.6035	60.6035
W_2V_1	52 × 23	0.95603	0.958206	45,414.285	41,157.142	55.9158	55.9158
W_2V_2	58 × 49	0.898458	0.90204	101,857.14	99,900	66.3265	66.3265
W_2V_3	78 × 36	0.956201	0.956201	69,157.142	69,157.142	71.9676	71.9676
W_3V_1	89 × 64	0.897991	0.899895	117,000	114,985.71	58.6413	58.7132
W_3V_2	90 × 73	0.930515	0.931094	161,914.28	159,200	93.3363	93.3874
W_3V_3	126 × 109	0.927944	0.932738	248,171.42	237,714.28	134.1887	134.6791
W_3V_4	134 × 105	0.941938	0.943005	239,528.57	238,242.85	135.9258	135.9258
W_4V_1	69 × 31	0.957867	0.95813	59,028.571	58,385.714	65.4502	65.4502
W_4V_2	84 × 45	0.951509	0.951509	86,328.571	84,842.857	84.1923	84.2278
W_4V_3	97 × 52	0.942733	0.945702	104,242.85	102,500	71.3426	71.3426
W_4V_4	157 × 128	0.94679	0.949908	315,657.14	282,757.14	149.1875	150.1198

were used. Test cases were executed several times on various platforms. Thereafter, the average of these values was taken. On the other side, a survey was conducted to decide severity of the manually seeded faults. Five students of postgraduate level were permitted to go through the system (websites) thoroughly. After this, they were asked to introduce faults anywhere randomly so that the system became faulty and it was shared with the tester. The tester could execute test cases randomly or follow the prioritized test sequence generated by the suggested algorithms. Technique of creating system faulty by manual fault injection has been widely practiced by researcher's community, and various studies have been published which used the same methodology for performance evaluation rationale [44–47].

Table 4 Result matrix depicting the performance in terms of number of iterations and time to solve the problem in case of NSGA and vNSABC when applied on all versions of the websites under test

Version	Matrix size	Smallest iteration for finding best APFD$_C$ in NSGA	Smallest iteration for finding best APFD$_C$ in vNSABC	Smallest iteration for finding best TIME in NSGA	Smallest iteration for finding best TIME in vNSABC	Smallest iteration for finding best SEVERITY in NSGA	Smallest iteration for finding best SEVERITY in vNSABC
W_1V_1	34 × 27	68 (2.119)	58 (1.9188)	233 (7.264)	37 (1.224)	175 (5.455)	322 (10.666)
W_1V_2	41 × 23	152 (5.91)	42 (1.715)	106 (4.122)	15 (0.613)	50 (1.944)	155 (6.34)
W_1V_3	52 × 23	82 (3.95)	34 (1.902)	32 (1.543)	29 (1.710)	20 (0.9664)	5 (0.29)
W_1V_4	58 × 49	268 (16.51)	43 (2.9968)	268 (16.508)	137 (9.451)	2682 (165.08	357 (24.628)
W_2V_1	61 × 25	699 (51.4)	30 (2.695)	210 (15.456)	565 (50.837)	76 (5.593)	29 (2.60)
W_2V_2	69 × 31	73 (7.25)	25 (3.267)	89 (8.851)	112 (14.757)	646 (64.24)	165 (21.56)
W_2V_3	74 × 35	305 (43.89)	49 (7.316)	592 (67.29)	429 (64.05)	298 (33.875)	218 (32.581)
W_3V_1	78 × 36	738 (39.79)	110 (18.65)	492 (64.3191)	35 (5.934)	492 (64.301)	139 (22.2)
W_3V_2	84 × 45	3470 (518)	590 (113.2)	2567 (383.79)	245 (47.116)	3471 (518.9)	855 (164.09)
W_3V_3	89 × 64	1878 (295)	416 (79.02)	1221 (192.17)	104 (18.75)	2948 (463.9)	2928 (444.80)
W_3V_4	90 × 73	3702 (628)	554 (130.8)	2826 (479.65)	1413 (282.8)	8043 (1365)	976 (189.388)
W_4V_1	97 × 52	1109 (268)	150 (40.57)	959 (230.29)	339 (103.43)	948 (227.8)	387 (118.162)
W_4V_2	126 × 109	6286 (1519)	700 (199.3)	4272 (749.90)	720 (238.76)	5444 (929.9)	1478 (299.21)
W_4V_3	134 × 105	8612 (2081)	3011 (849)	3686 (684.76)	1239 (399.4)	8945 (1548)	5009 (1009)
W_4V_4	157 × 128	6802 (1643)	956 (274.5)	2591 (489.56)	756 (249.98)	5161 (894.9)	1397 (287.78)

Bracketed value represents the execution time taken by the algorithm(s) to execute mentioned number of iterations

Table 5 Average performance of NSGA-II and vNSABC algorithms on all the parameters while testing all the versions of all the websites under test

	NSGA-A	vNSABC-A	NSGA-E	vNSABC-E	NSGA-S	vNSABC-S
APFDc	0.93554	0.93707	–	–	–	–
Severity	–	–	–	–	80.758	80.864
Cost	–	–	117,832	113,642	–	–
Number of iterations required	2283	451	1343	412	2627	961
Time taken to solve the problem (in s)	474.85	115.123	226.364	99.254	419.323	175.553

Proposed framework of the presented work is depicted using Fig. 3. A short description is as follows. In the first phase, error-free websites are shortlisted for testing and later on, faults are transplanted into them manually. In the next phase, all the required matrices are generated on the basis of observation, conducted survey and software tools. In the final phase, all the algorithms are applied on the matrices and the resultant test sequences generated from these algorithms are evaluated on various parameters. Table 2 depicts the lines of code, faults introduced and corresponding test cases in all versions of all the websites under test.

The codes of NSGA-II and NSABC were implemented using C language and were executed on the same hardware platform which is Windows platform, Intel (R) Core(TM) i3-4150 CPU @ 3.5 GHz. However, the dynamic programming-based approach was coded in Python language and implemented on the same hardware platform.

5 Results, Discussion and Analysis

In this part, under the first section, results generated by NSGA-II and vNSABC and analysis on the same are reported. In the last Sect. 5.1, a thorough discussion on every aspect related to dynamic programming is presented.

Results generated by both the algorithms Tables 3 and 4 on all versions of the website are presented below.

Depending upon the scenario and requirement, the tester may be interested in finding a solution fulfilling one of the objectives such as finding the test sequence having highest fault detection capability, or finding the test sequence which takes least amount of time to detect all the faults in case of hard deadline of product delivery during maintenance phase. The generated Tables 3, 4 and 5 present solutions generated by vNSABC performing better or equivalent than that of NSGA-II, either in single objective scenario or multi-objective scenario. Results Tables 3, 4 and 5 clearly indicate that the vNSABC has the capability to generate better (some time equivalent also) solutions as compared with NSGA-II in all versions of all the websites. All the solutions which are part of the first front are equally important and non-dominating to each other; if linear search is applied, we can get the best among these with respect to suggested objectives. These algorithms are called NSGA-E and vNSABC-E, if linear search on the first front is applied for execution time; similarly, for the remaining parameters, they are called NSGA-A/vNSABC-A (for APFD_C) and NSGA-S/vNSABC-S (for severity as parameter).

Table 5 draws attention toward average performance of the suggested algorithms during all versions of all the websites, which shows the results in the form of values from which comparison can be made. Table 5 clearly indicates that NSGA-A (sorted for APFD_C) was not able to outperform vNSABC average wise also. Similarly, vNSABC-E and vNSABC-S perform better average wise than NSGA-E and NSGA-S, respectively. If comparison is made on the basis of the parameters "Number of iterations required" and "Time taken to solve the problem," vNSABC performs far better than NSGA-II. For example, in case of comparison between NSGA-A and vNSABC-A, the latter takes less than 25% of the total iterations required by NSGA-A algorithm. Similarly, in case of execution time, vNSABC-A takes less than 24% of the total time taken by NSGA-A. Hence, the algorithms clearly outperform in terms of result generated, number of iterations required to generate the result and time taken to solve the problem.

With the help of programs, log files of each and every version of websites under test were created so as to analyze the behavior of vNSABC algorithm toward reaching the final result (maximization or minimization, whatever may be the objective). Iteration-wise values generated by all the algorithms were recorded for all suggested objectives. Figure 4 represents visualization of this recorded behavior in the form of graphs. Result values for all the objectives are normalized between 0 and 1 so as to represent in the scale of 0 to 1 on the graph's Y axis. Similarly, X axis represents the normalized number of iterations required, as some of the parameters become saturated/converge very fast and reach the optimal value earlier than the remaining ones. Hence, it can be said that these graphs are the visual representation of the working of vNSABC during each iteration and illustrate how parameters converge iteration wise.

As a part of the analysis, we have also performed t test (statistical testing) between NSGA and vNSABC to compare vNSABC vs. NSGA-II.

Above Tables 6a–c depict the results of paired one-tail t test. In this problem, we have considered three hypotheses, separate hypothesis for each parameter, i.e., APFD_C, execution time and severity.

HA0 = NSGA (APFD_C) and NSABC (APFD_C) give same result, and there is no significant difference between them.
HE0 = NSGA (execution time) and NSABC (execution time) generate same results and there is no significant difference between them.
HS0 = NSGA (severity) and NSABC (severity) generate same results and there is no significant difference observed.

If any hypothesis is rejected, then we can conclude that there are significant differences between results; to show which result is best, we have used average value.

To analyze hypothesis, we use significance level value 0.05.

As we can see in Table 6a, b, one pair one-tail P value (0.0011, 0.036) is less than the significant value (0.05), which means the first two null hypotheses are rejected. This means there is significant difference between results of NSGA and NSABC for the first two cases. If we compare the average value of NSGA–APFDc (0.935541867) with NSABC–APFDc (0.9370718), we find that NSABC–APFDc gives better results compared to NSGA–APFDc. In the same way, we found that NSABC-execution time gives better results compared to NSGA-execution time.

With the help of Table 6, we observe that the hypothesis is not rejected which means there is no significant difference between results. The reason is that out of 15, 10 versions have same severity of NSGA and NSABC value. But if we see the other five values, we find that NSABC outperforms NSGA in severity. Thus, we can say NSABC severity gives better result than NSGA severity.

5.1 Finding Optimal Objective Function of Test Case Execution in Feasible Worst Case Exponential Time and Space Complexity

Finding optimal solution of a permutation-based problem can be naively tackled by finding all permutations. But this will take up a large amount of time since time rises to $O(N!)$ complexity. Based on previous execution performed on above-stated hardware, it was observed that to find all permutations of 10 test cases in 30 min, from which evaluation of 15 permutations will be done would take approximately 20 years (practically unfeasible).

A test case execution sequence consists of two parts; the first part contains test cases that detect fault upon execution; the second part consists of redundant test cases that are not capable of detecting any new fault. When all faults are detected, further execution of test cases and their order would make no effect on the performance of the testing system.

So, finding entire permutations will be of no use and will be just waste of valuable time and resources. But the sequence

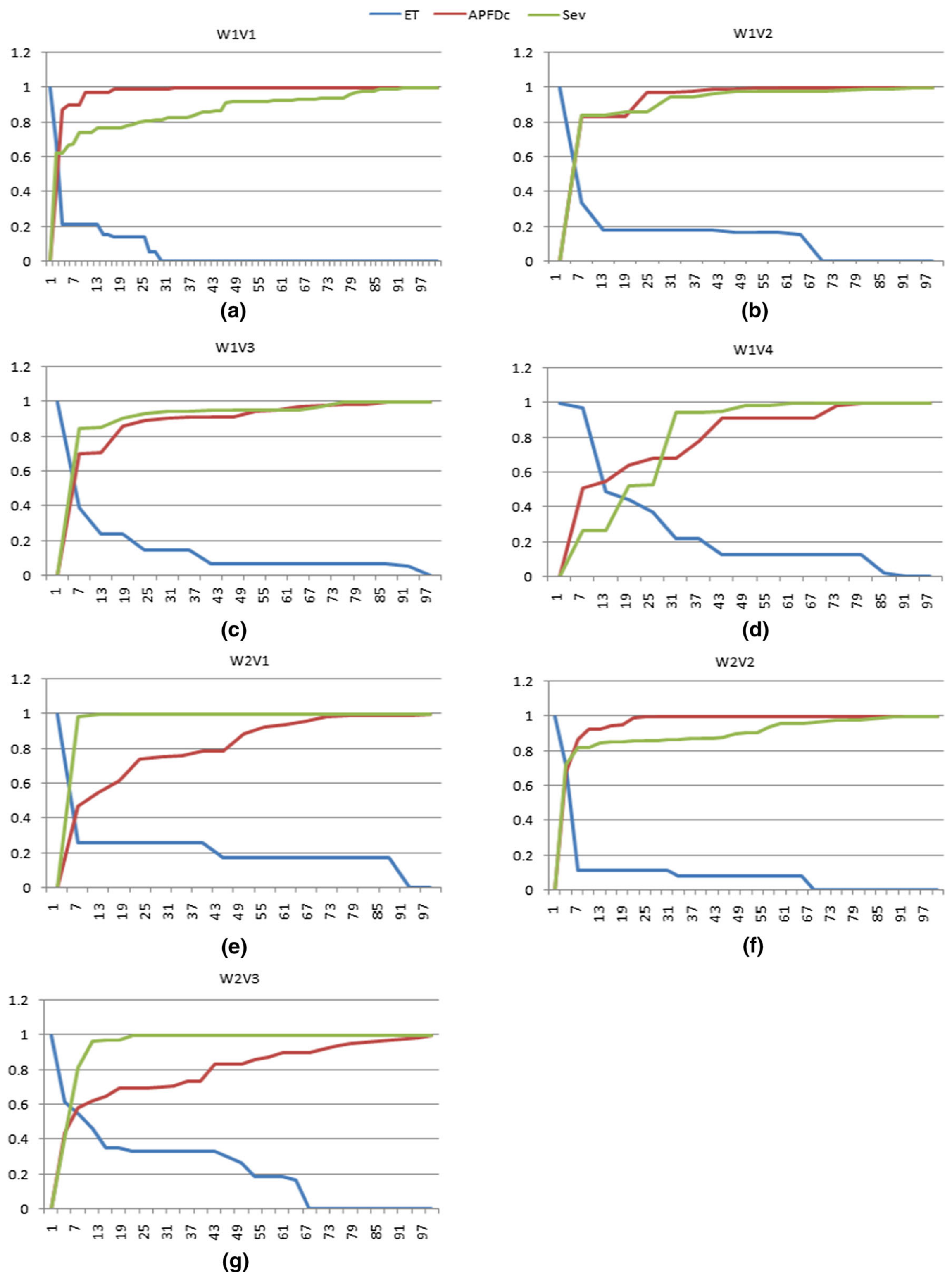

Fig. 4 Graphical representation of log files of Websites 1 and 2 where wi represent website number and vj represents version number of wi

up to which all faults are detected, further called as truncated sequence can be of variable length depending upon the elements in it.

So, we have developed an approach to find all such truncated sequences using recursion. We begin with an empty sequence at root, adding each available test case into that

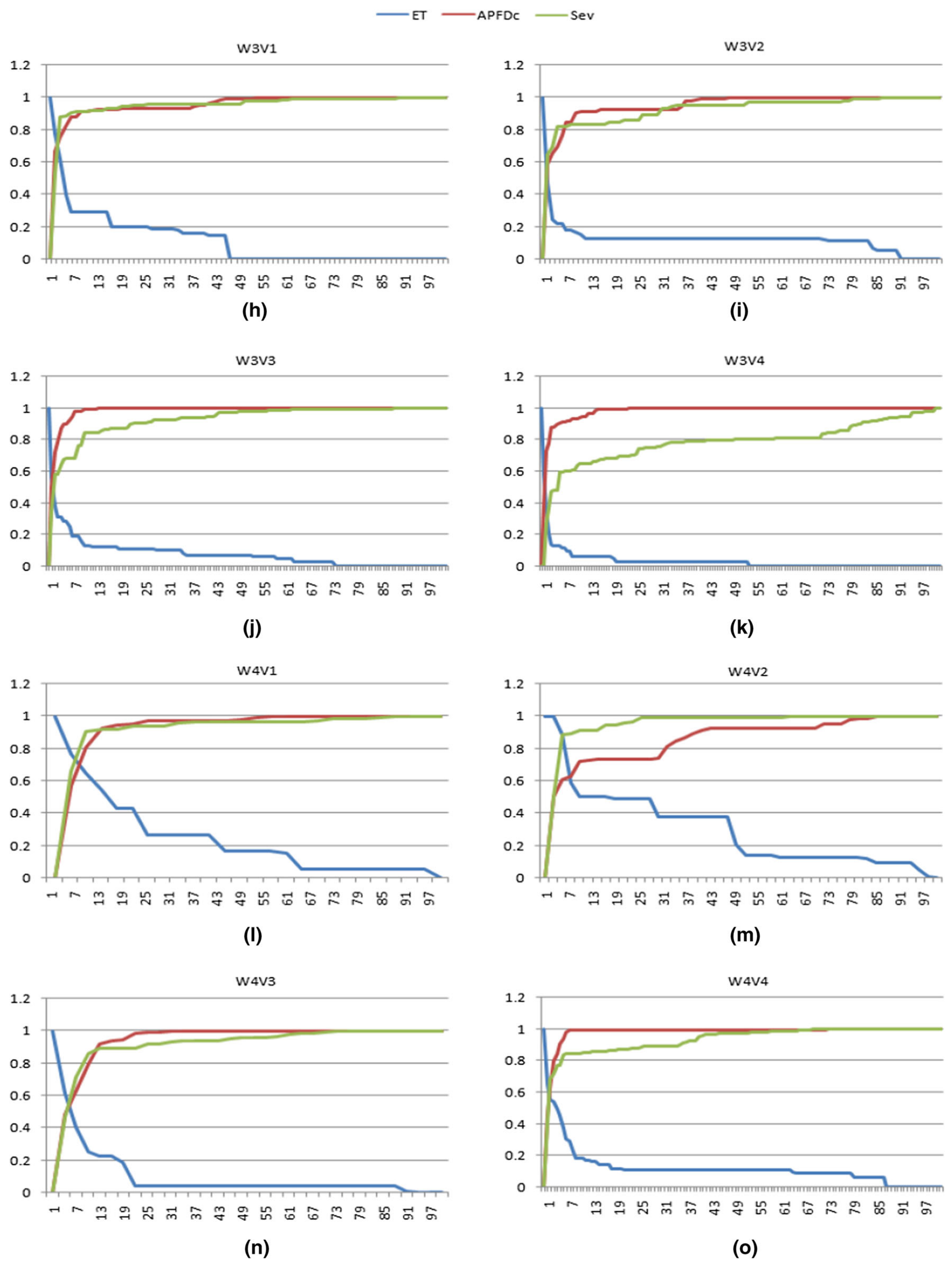

Fig. 4 continued

sequence one at a time. Further recursive calls are made with this partial sequence built. Recursion terminates when we are left with no more faults to be detected. Solutions obtained at the leaves will be returned back level by level to the root and can be evaluated for optimality.

Table 6 t test of various parameters between NSGA-II and vNSABC

t test: paired two sample for means between APFDc of NSGA-II and vNSABC		
	NSGA (APFD$_C$)	NSABC (APFD$_C$)
(a): t test of APFD$_C$ between NSGA-II and vNSABC		
Mean	0.935541867	0.9370718
Variance	0.000768798	0.00072893
Observations	15	15
Pearson correlation	0.998670448	
Hypothesized mean difference	0	
df	14	
t Stat.	− 3.731686744	
$P(T \Leftarrow T)$ one-tail	0.001116407	
t critical one-tail	1.761310115	

t test: paired two sample for means between execution time of NSGA-II and vNSABC		
	NSGA (execution time)	NSABC (execution time)
(b): t test of execution time between NSGA-II and vNSABC		
Mean	117,832.3788	113,642.8555
Variance	7,259,810,463	6,266,240,757
Observations	15	15
Pearson correlation	0.99749027	
Hypothesized mean difference	0	
df	14	
t Stat.	1.933906893	
$P(T \Leftarrow T)$ one-tail	0.036800411	
t critical one-tail	1.761310115	

t test: paired two sample for means between severity of NSGA-II and vNSABC		
	NSGA (severity)	NSABC (severity)
(c): t test of severity between NSGA-II and vNSABC		
Mean	80.75874	80.86415333
Variance	1079.635022	1092.44196
Observations	15	15
Pearson correlation	0.999986075	
Hypothesized mean difference	0	
df	14	
t Stat.	− 1.565590907	
$P(T \Leftarrow T)$ one-tail	0.069881239	
t critical one-tail	1.761310115	

For example, when trying for a sequence of 10 test cases, recursion leads to 69,459 truncated sequences, which is much less as compared to seeking 10! (3,628,800) solutions.

Both approaches of finding optimal sequences, either by finding all permutations or doing recursion to find all truncated sequences will not take more than linear space.

But recursion itself accounts for exponential time complexity, making it non-feasible to extend this approach above sequence of 15 test cases. Neither of these approaches is much scalable, because we are using very large amount of CPU time, but paying no attention to use of available memory resources.

The objectives under consideration to be optimized are based on prefix–suffix optimization (this term will be discussed shortly), where dynamic programming (DP) technique can be greatly used. First, we will see what this property is, then have a look at how DP can be used to find optimal solution in least possible time.

5.1.1 Prefix–Suffix (P–S) Optimization Problem

In this class of problems, each solution can be divided into P&S, order of elements in P contributes something to objective function and same is applicable to S part too.

But order of elements in P&S and the contribution made by them to objective function are independent of each other. This means that a fixed P contributed to the objective function is independent of any order of elements in S; vice versa is also true.

5.1.2 Dynamic Programming Approach

As explained earlier with recursive approach, we move the tree in top–down manner to explore leaves. Suppose, we have created P and elements from the remaining test cases that contribute to an optimal S. Reaching up to leaf of tree, we get a sequence as P–S in which assume S to be discovered optimal. But assume that the order of elements in P does not lead to optimality; a different order in P can do so.

While traveling another path of recursion tree, when we reach the node, where elements of P are in certain order which can led to optimality, the remaining S part need to be concatenated with this new built P.

There is no use of reevaluating S part again. Our dynamic programming approach does exactly the same; it saves the optimal discovered S part, corresponding to the set of elements in a discovered P (which can be nonoptimal on some node).

When we reach a node where a different order of elements of P is discovered, we just simply concatenate the S part from the look-up table created during execution. In this manner, we save a lot of computation time involved in redundant evaluation of optimal S for all possible orders of elements in P.

Below are the total solutions generated by each explained approach from the beginning of the document for a test suite of ten test cases.

Finding all permutation = 10! (3,628,800)
Truncated sequence finding by Recursion = 69,459
Actual and Memo calls by dynamic programming = 944 and 1913, respectively.

It should be noted that actual calls are recursive calls where solutions (S parts) are built and, saved to look-up table for future usage. Memo call on the other hand denotes the number of times entries are taken from look-up table for concatenation of S.

Given below (Fig. 5) is the snap shot of the results generated, along with other parameters, by our Python code. The first time, recursion is applied, while in the other case, dynamic programming-based approach is applied. The generated results clearly indicate that dynamic programming-based approach is able to present optimal results for 10×10 matrixes.

A detailed example (Fig. 6) on 5×5 test cases versus fault matrix along with its running is given below.

Let us consider the given below test case versus fault matrix

$$\begin{matrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{matrix}$$

Suppose execution time (cost) array of test cases, in milliseconds, is given below

1842.857143 1471.428571 3200.000000 4371.428571 2942.857143

Severity of these five faults is as given below

6 9 4 3 3

It is manually verified that the results generated are optimal for all the three parameters. Here, in the pruned tree constructed ($T1$, $T2$ and $T3$ given below) for all the parameters A represents actual calls, D represents Dynamic call and "–" means the level of the tree.

Fig. 5 Snapshot of results generated through both approaches on running example of 10×10 matrix

```
Enter Folder Name10x10
        0         // Do not use Dynamic Programming, instead go for Recursion
        OPT_APFDC    (0.8188884108957796, 69459, 0, 0, 0.784543, [4, 0, 2, 3, 8])
        OPT_SEVERITY (0.8188884108957796, 69459, 0, 0, 36.284, [0, 4, 3, 8, 2])
        OPT_COST     (0.8188884108957796, 69459, 0, 0, 18528.5714, [0, 2, 3, 8, 4])

Enter Folder Name10x10
        1         //Go for Dynamic Programming Approach
        OPT_APFDC    (0.027552050663750615, 944, 1913, 895, 0.784543, [4, 0, 2, 3, 8])
        OPT_SEVERITY (0.027552050663750615, 944, 1913, 895, 36.284, [0, 4, 3, 8, 2])
        OPT_COST     (0.027552050663750615, 944, 1913, 895, 18528.5714, [0, 2, 3, 8, 4])
```

Fig. 6 Snapshot of results generated through dynamic programming on running example of 5 × 5 matrix

The result generated, along with other parameters, by dynamic programming is as mentioned below.
OPT_APFDC (0.8638361850167635, 28, 19, 16, **0.751818**, [3, 4, 2])
OPT_SEVERITY (0.8638361850167635, 28, 19, 16, **19.48**, [3, 2, 4])
OPT_COST (0.8638361850167635, 28, 19, 16, **10514.2857**, [2, 3, 4]).

Tree T1: APFD$_C$ parameter dynamic programming preorder pruned tree

```
A [] 0.0
A - [0] 0.336012
A -- [0, 1] 0.195248
A --- [0, 1, 2] 0.077355
A ---- [0, 1, 2, 3] 0.059339
A ----- [0, 1, 2, 3, 4] 0.012769
All faults detected 0.680723
A ---- [0, 1, 2, 4] 0.050702
A ----- [0, 1, 2, 4, 3] 0.025289
All faults detected 0.684607
A --- [0, 1, 3] 0.096364
A ---- [0, 1, 3, 2] 0.039421
D ---- [0, 1, 3, 2, 4] 0.012769
All faults detected 0.679814
A ---- [0, 1, 3, 4] 0.040537
A ----- [0, 1, 3, 4, 2] 0.013884
All faults detected 0.682045
A --- [0, 1, 4] 0.078471
A ---- [0, 1, 4, 2] 0.051818
D ---- [0, 1, 4, 2, 3] 0.025289
All faults detected 0.686839
A ---- [0, 1, 4, 3] 0.062314
D ---- [0, 1, 4, 3, 2] 0.013884
All faults detected 0.68593
A -- [0, 2] 0.090124
A --- [0, 2, 1] 0.139711
D --- [0, 2, 1, 4, 3] 0.075992
All faults detected 0.641839
A --- [0, 2, 3] 0.190909
A ---- [0, 2, 3, 4] 0.025537
All faults detected 0.642583
A --- [0, 2, 4] 0.063471
A ---- [0, 2, 4, 1] 0.088636
D ---- [0, 2, 4, 1, 3] 0.025289
All faults detected 0.603533
A ---- [0, 2, 4, 3] 0.105785
All faults detected 0.595393
A -- [0, 3] 0.283471
A --- [0, 3, 2] 0.05219
D --- [0, 3, 2, 4] 0.025537
All faults detected 0.697211
A --- [0, 3, 4] 0.053306
A ---- [0, 3, 4, 2] 0.026653
All faults detected 0.699442
A -- [0, 4] 0.09124
A --- [0, 4, 1] 0.144174
D --- [0, 4, 1, 2, 3] 0.077107
All faults detected 0.648533
A --- [0, 4, 2] 0.064587
D --- [0, 4, 2, 1, 3] 0.113926
All faults detected 0.605764
A --- [0, 4, 3] 0.198347
D --- [0, 4, 3, 2] 0.026653
All faults detected 0.652252
A - [1] 0.227231
A -- [1, 0] 0.297707
D -- [1, 0, 4, 2, 3] 0.155579
All faults detected 0.680517
A -- [1, 2] 0.373388
A --- [1, 2, 3] 0.080661
A ---- [1, 2, 3, 4] 0.02876
All faults detected 0.710041
A --- [1, 2, 4] 0.066694
A ---- [1, 2, 4, 3] 0.046612
All faults detected 0.713926
A -- [1, 3] 0.382479
A --- [1, 3, 2] 0.055413
D --- [1, 3, 2, 4] 0.02876
All faults detected 0.693884
A --- [1, 3, 4] 0.056529
A ---- [1, 3, 4, 2] 0.029876
All faults detected 0.696116
A -- [1, 4] 0.377851
A --- [1, 4, 2] 0.06781
D --- [1, 4, 2, 3] 0.046612
All faults detected 0.719504
A --- [1, 4, 3] 0.083636
D --- [1, 4, 3, 2] 0.029876
All faults detected 0.718595
A - [2] 0.424463
A -- [2, 1] 0.171694
D -- [2, 1, 4, 3] 0.113306
All faults detected 0.709463
A -- [2, 3] 0.244215
A --- [2, 3, 4] 0.041529
All faults detected 0.710207
A -- [2, 4] 0.079463
A --- [2, 4, 1] 0.12062
D --- [2, 4, 1, 3] 0.046612
All faults detected 0.671157
A --- [2, 4, 3] 0.159091
All faults detected 0.663017
A - [3] 0.639876
A -- [3, 2] 0.068182
D -- [3, 2, 4] 0.041529
All faults detected 0.749587
A -- [3, 4] 0.069298
A --- [3, 4, 2] 0.042645
All faults detected 0.751818
A - [4] 0.428926
A -- [4, 1] 0.176157
D -- [4, 1, 2, 3] 0.114421
All faults detected 0.719504
A -- [4, 2] 0.080579
D -- [4, 2, 1, 3] 0.167231
All faults detected 0.676736
A -- [4, 3] 0.251653
D -- [4, 3, 2] 0.042645
All faults detected 0.723223
```

Tree T2: Severity parameter dynamic programming preorder pruned tree

```
A [] 0.0
A - [0] 9.0
A -- [0, 1] 1.92
A --- [0, 1, 2] 0.4
A ---- [0, 1, 2, 3] 0.28
A ----- [0, 1, 2, 3, 4] 0.072
All faults detected 11.672
A --- [0, 1, 2, 4] 0.21
A ----- [0, 1, 2, 4, 3] 0.128
All faults detected 11.658
A --- [0, 1, 3] 0.5333
A ---- [0, 1, 3, 2] 0.18
D ---- [0, 1, 3, 2, 4] 0.072
All faults detected 11.7053
A ---- [0, 1, 3, 4] 0.18
A ----- [0, 1, 3, 4, 2] 0.072
All faults detected 11.7053
A --- [0, 1, 4] 0.4
A ---- [0, 1, 4, 2] 0.21
D ---- [0, 1, 4, 2, 3] 0.128
All faults detected 11.658
A ---- [0, 1, 4, 3] 0.28
D ---- [0, 1, 4, 3, 2] 0.072
All faults detected 11.672
A -- [0, 2] 0.96
A --- [0, 2, 1] 1.04
D --- [0, 2, 1, 3, 4] 0.352
All faults detected 11.352
A --- [0, 2, 3] 1.7333
A ---- [0, 2, 3, 4] 0.09
All faults detected 11.7833
A --- [0, 2, 4] 0.52
A ---- [0, 2, 4, 1] 0.6
D ---- [0, 2, 4, 1, 3] 0.128
All faults detected 11.208
A ---- [0, 2, 4, 3] 1.0
All faults detected 11.48
A -- [0, 3] 3.2
A --- [0, 3, 2] 0.24
D --- [0, 3, 2, 4] 0.09
All faults detected 12.53
A --- [0, 3, 4] 0.24
A ---- [0, 3, 4, 2] 0.09
All faults detected 12.53
A -- [0, 4] 0.96
A --- [0, 4, 1] 1.04
D --- [0, 4, 1, 3, 2] 0.352
All faults detected 11.352
A --- [0, 4, 2] 0.52
D --- [0, 4, 2, 3] 1.0
All faults detected 11.48
A --- [0, 4, 3] 1.7333
D --- [0, 4, 3, 2] 0.09
All faults detected 11.7833
A - [1] 6.0
A -- [1, 0] 3.42
D -- [1, 0, 3, 2, 4] 0.7853
All faults detected 10.2053
A -- [1, 2] 4.56
A --- [1, 2, 3] 0.3733
A ---- [1, 2, 3, 4] 0.09
All faults detected 11.0233
A --- [1, 2, 4] 0.28
A ---- [1, 2, 4, 3] 0.16
All faults detected 11.0
A -- [1, 3] 4.94
A --- [1, 3, 2] 0.24
D --- [1, 3, 2, 4] 0.09
All faults detected 11.27
A --- [1, 3, 4] 0.24
A ---- [1, 3, 4, 2] 0.09
All faults detected 11.27
A -- [1, 4] 4.56
A --- [1, 4, 2] 0.28
D --- [1, 4, 2, 3] 0.16
All faults detected 11.0
A --- [1, 4, 3] 0.3733
D --- [1, 4, 3, 2] 0.09
All faults detected 11.0233
A - [2] 12.0
A -- [2, 1] 1.56
D -- [2, 1, 3, 4] 0.4633
All faults detected 14.0233
A -- [2, 3] 2.6
A --- [2, 3, 4] 0.12
All faults detected 14.72
A -- [2, 4] 0.78
A --- [2, 4, 1] 0.8
D --- [2, 4, 1, 3] 0.16
All faults detected 13.74
A --- [2, 4, 3] 1.3333
All faults detected 14.1133
A - [3] 19.0
A -- [3, 2] 0.36
D -- [3, 2, 4] 0.12
All faults detected 19.48
A -- [3, 4] 0.36
A --- [3, 4, 2] 0.12
All faults detected 19.48
A - [4] 12.0
A -- [4, 1] 1.56
D -- [4, 1, 3, 2] 0.4633
All faults detected 14.0233
A -- [4, 2] 0.78
D -- [4, 2, 3] 1.3333
All faults detected 14.1133
A -- [4, 3] 2.6
D -- [4, 3, 2] 0.12
D -- [4, 3, 2] 0.12
All faults detected 14.72
```

Tree T3: Execution time tree parameter dynamic programming preorder pruned tree

```
A [] 0
A - [0] 1842.8571
A -- [0, 1] 1471.4286
A --- [0, 1, 2] 3200.0
A ---- [0, 1, 2, 3] 4371.4286
A ----- [0, 1, 2, 3, 4] 2942.8571
All faults detected 13828.5714
A ---- [0, 1, 2, 4] 2942.8571
A ----- [0, 1, 2, 4, 3] 4371.4286
All faults detected 13828.5714
A --- [0, 1, 3] 4371.4286
A ---- [0, 1, 3, 2] 3200.0
D ---- [0, 1, 3, 2, 4] 2942.8571
All faults detected 13828.5714
A ---- [0, 1, 3, 4] 2942.8571
A ----- [0, 1, 3, 4, 2] 3200.0
All faults detected 13828.5714
A --- [0, 1, 4] 2942.8571
A ---- [0, 1, 4, 2] 3200.0
D ---- [0, 1, 4, 2, 3] 4371.4286
All faults detected 13828.5714
A ---- [0, 1, 4, 3] 4371.4286
D ---- [0, 1, 4, 3, 2] 3200.0
All faults detected 13828.5714
A -- [0, 2] 3200.0
A --- [0, 2, 1] 1471.4286
D --- [0, 2, 1, 3, 4] 7314.2857
All faults detected 13828.5714
A --- [0, 2, 3] 4371.4286
A ---- [0, 2, 3, 4] 2942.8571
All faults detected 12357.1429
A --- [0, 2, 4] 2942.8571
A ---- [0, 2, 4, 1] 1471.4286
D ---- [0, 2, 4, 1, 3] 4371.4286
All faults detected 13828.5714
A ---- [0, 2, 4, 3] 4371.4286
All faults detected 12357.1429
A -- [0, 3] 4371.4286
A --- [0, 3, 2] 3200.0
D --- [0, 3, 2, 4] 2942.8571
All faults detected 12357.1429
A --- [0, 3, 4] 2942.8571
A ---- [0, 3, 4, 2] 3200.0
All faults detected 12357.1429
A -- [0, 4] 2942.8571
A --- [0, 4, 1] 1471.4286
D --- [0, 4, 1, 2, 3] 7571.4286
All faults detected 13828.5714
A --- [0, 4, 2] 3200.0
D --- [0, 4, 2, 3] 4371.4286
All faults detected 12357.1429
A --- [0, 4, 3] 4371.4286
D --- [0, 4, 3, 2] 3200.0
All faults detected 12357.1429
A - [1] 1471.4286
A -- [1, 0] 1842.8571
D -- [1, 0, 2, 3, 4] 10514.2857
All faults detected 13828.5714
A -- [1, 2] 3200.0
A --- [1, 2, 3] 4371.4286
A ---- [1, 2, 3, 4] 2942.8571
All faults detected 11985.7143
A --- [1, 2, 4] 2942.8571
A ---- [1, 2, 4, 3] 4371.4286
All faults detected 11985.7143
A -- [1, 3] 4371.4286
A --- [1, 3, 2] 3200.0
D --- [1, 3, 2, 4] 2942.8571
All faults detected 11985.7143
A --- [1, 3, 4] 2942.8571
A ---- [1, 3, 4, 2] 3200.0
All faults detected 11985.7143
A -- [1, 4] 2942.8571
A --- [1, 4, 2] 3200.0
D --- [1, 4, 2, 3] 4371.4286
All faults detected 11985.7143
A --- [1, 4, 3] 4371.4286
D --- [1, 4, 3, 2] 3200.0
All faults detected 11985.7143
A - [2] 3200.0
A -- [2, 1] 1471.4286
D -- [2, 1, 3, 4] 7314.2857
All faults detected 11985.7143
A -- [2, 3] 4371.4286
A --- [2, 3, 4] 2942.8571
All faults detected 10514.2857
A -- [2, 4] 2942.8571
A --- [2, 4, 1] 1471.4286
D --- [2, 4, 1, 3] 4371.4286
All faults detected 11985.7143
A --- [2, 4, 3] 4371.4286
All faults detected 10514.2857
A - [3] 4371.4286
A -- [3, 2] 3200.0
D -- [3, 2, 4] 2942.8571
All faults detected 10514.2857
A -- [3, 4] 2942.8571
A --- [3, 4, 2] 3200.0
All faults detected 10514.2857
A - [4] 2942.8571
A -- [4, 1] 1471.4286
D -- [4, 1, 3, 2] 7571.4286
All faults detected 11985.7143
A -- [4, 2] 3200.0
D -- [4, 2, 3] 4371.4286
All faults detected 10514.2857
A -- [4, 3] 4371.4286
D -- [4, 3, 2] 3200.0
All faults detected 10514.2857
```

Eight running examples (Table 7), test cases versus fault matrix, are also generated for the evaluation of performances of dynamic programming while solving the problem in hand. We have created Python code for the implementation of dynamic programming-based approach for solving the problem. Three separate programs are created for evaluation of each parameter. The time taken to solve problem of different instances, for each parameter, is depicted Table 6. Meantime, it is also shown how many actual recursive calls are made and how many times the tables created by dynamic programming have been referred. After that, we have merged all the three programs into one. This complete program is shown in the "Appendix" section along with comments so that users can understand and reuse it as per their requirements. Results generated for all the three parameters by dynamic programming are at par with vNSABC.

A graph (Fig. 7) between the log of time and space taken versus instance size is shown below. The color coding is as below:

Before concluding this Section, we would like to answer two research questions that we have raised in earlier sections.

Q1 Is vNSABC is able to generate better results, within less time and less number of iterations,in comparison with NSGA-II, in solving the suggested problem?
Answer: Yes, while solving the suggested problem, on the dataset in hand, the generated results shown using Tables 2, 3 and 4 revealed that the vNSABC is able to generate better results when compared with NSGA-II. To support the statement, statistical testing is also performed, as shown in previous section, which also arrives at the same conclusion. Tables 2, 3 and 4 also clearly

Table 7 Presentation of data related to time to compute eight problem instances along with information related to actual calls and memo calls of dynamic programming

S. no.	Problem size (test case vs. fault matrix)	Time to compute "$APFD_C$	Time to compute "Severity"	Time to compute "Time"	Actual calls in dynamic programming	Memo calls (dynamic lookups) in dynamic programming
1	8 × 8	0.00570234	0.00575793	0.0042457	206	361
2	10 × 10	0.02925369	0.02728148	0.0204654	861	1947
3	12 × 12	0.12443800	0.12803244	0.0984199	3187	7402
4	14 × 14	0.93817187	0.97851678	0.7047538	14,982	48,513
5	16 × 16	3.42653176	3.63070763	2.7733889	59,392	195,729
6	18 × 18	18.1885335	19.3485216	15.900950	247,480	1,049,777
7	20 × 20	86.3752897	90.3063283	75.553968	1,012,500	4,499,453
8	22 × 22	338.217244	372.280950	281.36895	3,516,323	16,040,586

Table 8 Color coding for different parameters that are used in Fig. 5

Color	Parameter	Color	Parameter
Red	$APFD_C$	Blue	Severity
Green	Time	Yellow	Actual calls in dynamic programming
Black	Dynamic lookups		

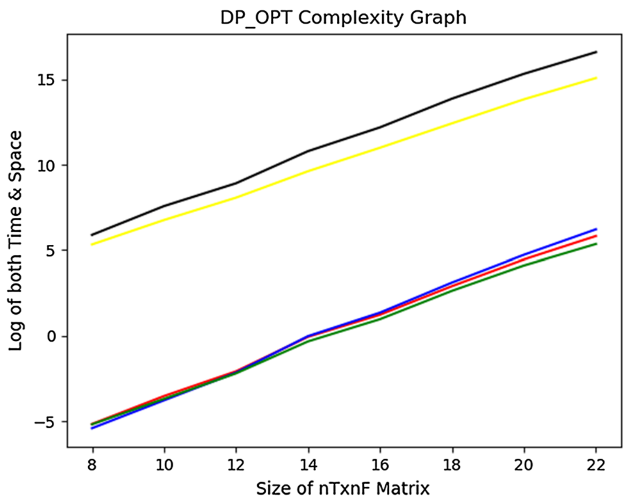

Fig. 7 Behavior of test matrices with log of time and space

depict that vNSABC is able to generate better results (or equivalent results in a few cases) in less iterations and in less interval of time.

Q2 Is there any other non-incremental algorithm-based approach to solve the problem in feasible exponential time?
Answer: Yes, there exists non-incremental algorithm-based approach, dynamic programming-based algorithm, which will be able to solve the suggested problem for smaller size instances. To strengthen the statement, the algorithm, its code and its full functioning are presented in Sect. 5 and "Appendix" section.

6 Conclusion and Future Scope

The presented study focuses on TCP under multi-objective scenario where there are three conflicting objectives. Many prior studies focus only on single objective. However, the presented study manages several conflicting objectives in parallel. TCP helps the software industry by saving software, hardware, human resources, time and effort at one end, while at the other end, it raises the standard of software quality assurance levels by finding severe faults earlier during testing practices so that they can be eliminated at the earliest. TCP supports the tester fraternity by prioritizing test cases with the support of suggested algorithms rather than running all the test cases blindly so that maximum optimization of the testing parameter takes place.

This study presents the usability of vNSABC over other algorithms based on same strategy or different strategy. The presented study compares the performance and relative effectiveness of NSGA-II with vNSABC while solving the discrete combinatorial test cases prioritization problem in multi-objective environment. The performance was evaluated on all the three objectives; moreover, in this work, we also tried to analyze how much time and how much iterations these algorithms take to generate the best results. Results show that vNSABC performs better than or equivalent to NSGA-II, most of the time, with respect to achieving all the three suggested objectives. Almost in all the cases and all the objectives, results generated by vNGSA were not inferior to the results generated by NSGA-II. Moreover, the results obtained by vNSABC take less time as well as less number of iterations with respect to NSGA-II. The average performance is also better, or sometimes equivalent, in case of vNSABC. Hence, it can be inferred from this study that vNSABC comes out as the better choice in a resource constrained environment. The resultant values of objectives generated by vNSABC play the role of upper bound for all the algorithms in hand. The vNSABC suggests the test cases which should be executed first so as to generate best APFDC, least execution time and highest severity detection rate.

At the same time, we have also devised a dynamic programming-based algorithm to solve the suggested problem. Depth analysis of the functioning of the algorithm is also presented with respect to various parameters. Python code is also presented for the readers. It is also shown that for small-size instances of the problem, dynamic programming-based approach is able to replicate the same results as generated by vNSABC for all the objectives. However, due to exponential time complexity of the algorithm, the computation was not performed for large size instances.

Finally, the authors want to communicate that this study not only presents the best algorithm among suggested competitors, but also suggests three solutions to the tester community. They can make use of it on the basis of requirement, needs and priority.

A few parameters, such as requirement changes, priorities in requirements, coder experiences, can be still incorporated in the proposed model so that it can be converted into a many objectives optimization problem. Artificially implanted faults may be a threat to the model; however, they are symbolic of real-life faults. Matrices up to 150 sizes of faults are tested through the model; real-life test scenario may contain larger size problems. If the memo can be smartly managed (i.e., shrink/grow/optimized), large size problem instances can be solved using the proposed dynamic programming-based algorithm.

7 Appendix

Appendix section is divided into three parts. The first section presents the algorithm based on dynamic programming. Second section presents the memo required by dynamic programming for the given problem. Finally, the last section presents the whole Python code for dynamic programming algorithm along with a thorough discussion on amortized constant time complexity of dictionaries and set.

7.1 Dynamic Programming-Based Algorithm Combined for All the Three Parameters

```
OPT_PAR($S_1$,$S_2$,$S_3$,AP,SP,CP,DCT,RC,RS,L)
{ /*$S_1$: Prefix already generated for optimal $APFD_C$.
    $S_2$: Prefix already generated for optimal Severity parameter
    $S_3$: Prefix already generated for optimal Cost (Time in our case)
    AP: "$APFD_C$" evaluated for prefix S1
    SP:"Severity Parameter" evaluated for prefix S1
    CP:"Cost" evaluated for prefix S1
    DCT: Remaining test cases and their faults after execution of their prefix; there is only one DCT.
    RC: Remaining cost to be executed
    RS: Remaining severity to be detected
    L: Level/Depth in the tree/Number of elements in the prefix part
 */
        key =set of elements in PREFIX
        if(key in memo)
        {
                $v_1$ ,$v_2$, $v_3$=memo[key]
                /*
                        $v_1$ has optimal suffix for best $APFD_C$
                        $v_2$ has optimal suffix for best severity parameter
                        $v_3$ has optimal suffix for best cost.
                        They also have their contributions to respective parameter stored in them.
                        Therefore $v_1$[0]has contribution and $v_1$[1] is suffix itself.
                */
                $r_1$=(AP+$v_1$[0],$S_1$+$v_1$[1])
                $r_2$=(SP+v2[0],$S_2$+$v_2$[1])
                $r_3$=(CP+v3[0],$S_3$+$v_3$[1])
                return($r_1$,$r_2$,$r_3$)
        }
        else
        { //This part will execute if key is not in memo
                for test case,TC, in DCT
                {
                        //apfdc_con is contribution in $APFD_C$.
                        //severity_par_con is contribution in severity_parameter.
                        Find DCT' // New DCT after execution of TC.
                        RC'=RC-cost[TC]
                        RS'=RS-RS-exposed_severity[TC]
                        $t_1$,$t_2$,$t_3$=OPT_PAR($S_1$+TC,$S_2$+TC,$S_3$+TC,AP+apfdc_con[TC],SP+
                        severity_par_con[TC],CP+cost[TC],DCT'RC',RS',L+1)
                        if($t_1$ has high $APFD_C$ then all others)
                                {$T_1$=$t_1$}
                        if($t_2$ has high severity parameter then all previous)
                                {$T_2$=$t_2$}
                        if(t3 has low cost then previous sequences)
                                {$T_3$=$t_3$}
                }
                memo[key].insert($T_1$-$S_1$,$T_2$-$S_2$,$T_3$-$S_3$)
                // or it can be inserted as insert($T_1$[L:],$T_2$[L:],$T_3$[L:])
                return $T_1$,$T_2$,$T_3$
                }
        }
```

7.2 Memo (Stored in RAM) for APFDc Evaluation, Severity Evaluation and Execution time Evaluation

//Here, the first argument represents prefix, third argument represents suffix and second argument represents contribution by suffix.

Memo for $APFD_C$ parameter evaluation

{0, 1, 2, 3}	: 0.012769 [4]	{1,2,3}	: 0.02876 [4]
{0, 1, 2, 4}	: 0.025289 [3]	{1, 2, 4}	: 0.046612 [3]
{0, 1, 2}	: 0.075992 [4, 3]	{1, 2}	: 0.113306 [4, 3]
{0, 1, 3, 4}	: 0.013884 [2]	{1, 3, 4}	: 0.029876 [2]
{0, 1, 3}	: 0.054421 [4, 2]	{1, 3}	: 0.086405 [4, 2]
{0, 1, 4}	: 0.077107 [2, 3]	{1, 4}	: 0.114421 [2, 3]
{0, 1}	: 0.155579 [4, 2, 3]	{1}	: 0.492273 [4, 2, 3]
{0, 2, 3}	: 0.025537 [4]	{2, 3}	: 0.041529 [4]
{0, 2, 4}	: 0.113926 [1, 3]	{2, 4}	: 0.167231 [1, 3]
{0, 2}	: 0.216446 [3, 4]	{2}	: 0.285744 [3, 4]
{0, 3, 4}	: 0.026653 [2]	{3, 4}	: 0.042645 [2]
{0, 3}	: 0.079959 [4, 2]	{3}	: 0.111942 [4, 2]
{0, 4}	: 0.225 [3, 2]	{4}	: 0.294298 [3, 2]
{0}	: 0.36343 [3, 4, 2]	set()	: 0.751818 [3, 4, 2]

Memo for severity parameter evaluation

{0, 1, 2, 3}	: 0.072 [4]	{1, 2, 3}	: 0.09 [4]
{0, 1, 2, 4}	: 0.128 [3]	{1, 2, 4}	: 0.16 [3]
{0, 1, 2}	: 0.352 [3, 4]	{1, 2}	: 0.4633 [3, 4]
{0, 1, 3, 4}	: 0.072 [2]	{1, 3, 4}	: 0.09 [2]
{0, 1, 3}	: 0.252 [2, 4]	{1, 4}	: 0.4633 [3, 2]
{0, 1, 4}	: 0.352 [3, 2]	{1}	: 5.27 [3, 2, 4]
{0, 1}	: 0.7853 [3, 2, 4]	{2, 3}	: 0.12 [4]
{0, 2, 3}	: 0.09 [4]	{2, 4}	: 1.3333 [3]
{0, 2, 4}	: 1.0 [3]	{2}	: 2.72 [3, 4]
{0, 2}	: 1.8233 [3, 4]	{3, 4}	: 0.12 [2]
{0, 3, 4}	: 0.09 [2]	{3}	: 0.48 [2, 4]
{0, 3}	: 0.33 [2, 4]	{4}	: 2.72 [3, 2]
{0, 4}	: 1.8233 [3, 2]	set()	: 19.48 [3, 2, 4]
{0}	: 3.53 [3, 2, 4]		

Memo for execution time parameter evaluation

{0, 1, 2, 3}	: 2942.857142999993 [4]	{1, 2, 3}	: 2942.857143000001 [4]
{0, 1, 2, 4}	: 4371.428571 [3]	{1, 2, 4}	: 4371.428571000001 [3]
{0, 1, 2}	: 7314.285714 [3, 4]	{1, 2}	: 7314.285714000015 [3, 4]
{0, 1, 3, 4}	: 3200.0 [2]	{1, 3, 4}	: 3200.0 [2]
{0, 1, 3}	: 6142.857142999999 [2, 4]	{1, 3}	: 6142.857143000001 [2, 4]
{0, 1, 4}	: 7571.428570999995 [2, 3]	{1, 4}	: 7571.428571 [3, 2]
{0, 1}	: 10514.285714 [2, 3, 4]	{1}	: 10514.285714 [4, 3, 2]
{0, 2, 3}	: 2942.857143000001 [4]	{2, 3}	: 2942.857143000001 [4]
{0, 2, 4}	: 4371.428571 [3]	{2, 4}	: 4371.428571000001 [3]
{0, 2}	: 7314.285714000001 [4, 3]	{2}	: 7314.285714000015 [3, 4]
{0, 3, 4}	: 3200.0 [2]	{3, 4}	: 3200.000000000001 [2]
{0, 3}	: 6142.857143 [4, 2]	{3}	: 6142.857143000001 [2, 4]
{0, 4}	: 7571.428571 [2, 3]	{4}	: 7571.428571000001 [2, 3]
{0}	: 10514.285714000001 [2, 4, 3]	set()	: 10514.285714000001 [2, 3, 4]

7.3 Python's Amortized Constant Time Complexity of Dictionaries and Set

Mathematical notations of set provides its unordered nature and capability of having only a unique element. In Python, this element values must be *Hashable*, which means that they cannot be changed once placed in set. Entries placed in set are known as "Keys" which are said to be hashable. With dictionaries, we also have a data associated with each key, called its "Value." Sets only have collection of keys.

As the name hashable suggest, both sets and dictionaries utilize the concept of $O(1)$ time of basic operations of hash tables such as search, insert and delete. This speed is achieved using an open address hash table as the supporting data structure.

With all these pros of faster time taken in operations, there are also cons of their usage. They do take more amount of space in memory than ordered types; also, the speed of operation and amortized $O(1)$ time complexity depend upon the hashing function in use. Hash function should be fast to evaluate and probe each entry into slots with equal probability.

7.3.1 Working Process of Sets and Dictionaries

Both data types use open address hash table, which according to theory should provide for $O(1)$ time for basic operations. This is achieved by application of Hash function on key provided which generates a Hash value, which is used as index into list of hash values. To the user it seems to have access using an arbitrary key which can be any Hashable/Immutable data type

7.3.2 Searching and Insertion

The search begins with an empty open address hash table allocated in memory-Hash function is probed continuously until we get an empty slow where our key (and reference to value also in case of dictionary) is stored. Hash value obtained by key, is masked on lower bits, so that output numeric index would be the valid index of stored hash table.

Linear Probing is used with slight modification in Python, to eliminate clustering by providing a contribution of higher bits, which are removed during masking.

Choice of algorithm for hashing is open to user in Python, but it is important to hash into each slot with equal probability. Uniformity of data in hash table is related with load factor, which is relatable to entropy of hash function.

7.3.3 Deletion

It is a common practice in open addressing to place special sentinels in place of NULL, when an entry is deleted. This sentinel (special value) signifies further search using modified linear probing. These empty slots can be overwritten in future or can also be removed during resizing of hash table.

7.3.4 Resizing

One of the most important aspects of majority of mutable data types in Python is how we can let our data structure grow or shrink. Hash table should grow when it gets filled and more entries are coming for further insertion. There is a critical limit (generally 2/3 of maximum size) of storage; when that fraction of hash table is filled, a larger table is allocated and keys are rehashed into a new hash table, then further insertion takes place directly into the new hash table, and the previous one is de-allocated from memory.

This operation of rehashing is quite expensive during insertion of element which requires resizing (not on every element); it is proved to be amortized $O(1)$ times.

7.3.5 Effect of Faults/Testcase Density on Size of Memo

Our dynamic programming, based on recursive backtracking, does carry some inherent properties. Let Faults/Testcase be 'k,' if worst case of 'k' is 1, then depth of recursion tree will be equal to number of test cases. Hence, there will be maximum possible entries in memo. On the other hand, if 'k' is equal to number of faults, then depth of tree will be equal to one and memo will contain no entries at all.

On the basis of the above conclusion, we get to know that low values of 'k' favor high number of entries in memo and also more computation to be done.

In our experimental setup, we were limited by the memory resource available for storage of large memo and could reach maximum of 23×23.

As the last part of "Appendix" section, we present here the complete Python code for optimal parameter evaluation using dynamic programming.

//**CODE FOR OPTIMAL PARAMTER EVALUATION USING DYNAMIC PROGRAMMING**

```
import os,math,time

st = input('Enter Folder Name')
flg1,flg2,flg3,flg_wr = [int(x) for x in input().strip().split()]
os.chdir(st)

matrixfile = '_'.join(st.strip().split('_')[:2])+'.txt'
TFmat = [[[int(i) for i in line.split()]] for line in open(matrixfile)]

nT = len(TFmat)
nF = len(TFmat[0][0])

for i in range(len(TFmat)):
        TFmat[i].insert(0,i)#Adding index values to matrix

f = open('arrays.h')
xctn_tm  = [ float(x) for x in f.readline().split() ]
severity = [  int(x)  for x in f.readline().split() ]
f.close()

remaining_severity = total_severity = sum(severity)
remaining_cost = total_cost = sum(xctn_tm)


dct = {} #Have Faults detectable by each testcase, created initially from TFmat Data-Structure, reference sent
recursively
for TC in TFmat:
   dct[TC[0]] = {x for x in range(nF) if TC[1][x]}


#FINDING OPTIMAL ALL PARAMETERS
memo = {} #Top-Down DP, entry has [reference_count, apfdc, sequence]
AC,DC = 0,0
#CHECKPOINT
def
optimal_PAR(seq1,seq2,seq3,apfdc_par,sev_par,cost_par,dct,remaining_cost,remaining_severity,level,val1,val2,val
3):

   global AC,DC
   if flg1:
         print('A','-'*level,seq1,round(val1/(total_cost*total_severity),6))
   if flg2:
         print('A','-'*level,seq2,round(val2/total_severity,4))
   if flg3:
         print('A','-'*level,seq3,round(val3,4))

   if not dct:
      if flg1:
            print( 'All Faults Detected',round(apfdc_par/(total_cost*total_severity),6) )
      if flg2:
            print( 'All Faults Detected',round(sev_par/total_severity,4) )
      if flg3:
            print( 'All Faults Detected',round(cost_par,4) )
      return [[apfdc_par+0,seq1+[]],[sev_par+0,seq2+[]],[cost_par+0,seq3+[]]]

   key = int(''.join(('0' if i in seq1 else '1')for i in range(nT)),2)

   if key not in memo:
      AC+=1
      #Sorted return Generator Expression to consume less memory
      choice = ((     TC,
            sum(severity[x] for x in dct[TC])*(remaining_cost - 0.5*xctn_tm[TC]),
            sum(severity[x] for x in dct[TC])*(remaining_severity)/(level+1) )      for TC in dct)

      bst_res1,bst_TC1,bst_res2,bst_TC2,bst_res3,bst_TC3 = [0,[]],None,[0,[]],None,[math.inf,[]],None
      for TC,PAR1,PAR2 in choice:
         # Computation of Local DCT to be sent
         tmp_dct = {}#;print('>',end='')
         for i in dct:
            val = dct[i].difference(dct[TC])
            if val:
               tmp_dct[i] = val
            #optimal_PAR(seq1,seq2,seq3,apfdc_par,sev_par,cost_par,dct,remaining_cost,remaining_severity,level)
            tmp_cost = remaining_cost - xctn_tm[TC]
            tmp_sev = remaining_severity - sum(severity[i] for i in dct[TC])

tmp_res1,tmp_res2,tmp_res3,=optimal_PAR(seq1+[TC],seq2+[TC],seq3+[TC],apfdc_par+PAR1,sev_par+PAR2,co
st_par+xctn_tm[TC],tmp_dct,tmp_cost,tmp_sev,level+1,PAR1,PAR2,xctn_tm[TC])

            if tmp_res1[0] > bst_res1[0]:
                bst_res1 = tmp_res1
                bst_TC1 = TC
            if tmp_res2[0] > bst_res2[0]:
                bst_res2 = tmp_res2
                bst_TC2 = TC
            if tmp_res3[0] < bst_res3[0]:
                bst_res3 = tmp_res3
                bst_TC3 = TC

        #Creating Entry for Best choice in MEMO
        ref = math.factorial(level)
        memo[key]=[ref]+[[bst_res1[0]-apfdc_par,bst_res1[1][level:]],[bst_res2[0]-
sev_par,bst_res2[1][level:]],[bst_res3[0]-cost_par,bst_res3[1][level:]]]

    else:
        DC+=1
        if flg1:
                print('D','-'*level,seq1+memo[key][1][1],round(memo[key][1][0]/(total_cost*total_severity),6))
                print( 'All Faults Detected',round((apfdc_par+memo[key][1][0])/(total_cost*total_severity),6) )
        if flg2:
                print('D','-'*level,seq1+memo[key][2][1],round(memo[key][2][0]/total_severity,4))
                print( 'All Faults Detected',round((sev_par+memo[key][2][0])/total_severity,4) )
        if flg3:
                print('D','-'*level,seq1+memo[key][3][1],round(memo[key][3][0],4))
                print( 'All Faults Detected',round((cost_par+memo[key][3][0]),4) )
        pass

    memo[key][0] -= 1

res=[[apfdc_par+memo[key][1][0],seq1+memo[key][1][1]],[sev_par+memo[key][2][0],seq2+memo[key][2][1]],[co
st_par+memo[key][3][0],seq3+memo[key][3][1]]]

    if not memo[key][0]:
        del memo[key]

    return res

init = time.clock()
res1,res2,res3 = optimal_PAR([],[],[],0,0,0,dct,remaining_cost,remaining_severity,0,0,0,0)
apfdc_par, seq_A = res1
sev_par  , seq_S = res2
cost_par , seq_T = res3
apfdc_par /= (total_cost*total_severity)
apfdc_par = round(apfdc_par,6)
sev_par/=total_severity
sev_par = round(sev_par,4)
cost_par = round(cost_par,4)
init = time.clock()-init
res   =   'OPT_APFDC'   +   str((init,AC,DC,len(memo),apfdc_par,seq_A))   +   '\n'   +\'OPT_SEVERITY'   +
str((init,AC,DC,len(memo),sev_par,seq_S)) + '\n' +\'OPT_COST' + str((init,AC,DC,len(memo),cost_par,seq_T))

print(res)

f1 = open('RESULT.h','a')
if flg_wr:
        f1.write('\n'+res)
f1.close()
```

References

1. Mathur, A.: Foundations of Software Testing, Seventh Impression. Pearson Education, London (2012)
2. Chauhan, N.: Software Testing Principles and Practices, 1st edn. Oxford University Press, Oxford (2010)
3. Singh, Y.: Software Testing, 1st edn. Cambridge University Press, Cambridge (2012)
4. Rothermal, G.; Untch, R.; Harrold, M.: Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. **27**(10), 929–948 (2001)
5. Malishevsky, A.G.; Ruthruff, J.R.; Rothermel, G.; Elbaum, S.: Cost-cognizant test case prioritization. Technical Report TR-UNL-CSE-2006-0004, Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln (2006)
6. Harman, M.; Li, Z.; Hierons, R.: Search algorithms for regression test case prioritization. IEEE Trans. Softw. Eng. **33**(4), 225–237 (2007)
7. Elbaum, S.; Malishevsky, A.; Rothermel, G.: Incorporating varying test costs and fault severities into test case prioritization. In: Proceedings of the 23rd International Conference on Software Engineering, pp. 329–338 (2001)
8. Jiang, B.; Zhang, Z.; Chan, W.K.; Tse, T.H.: Adaptive random test case prioritization. In: ASE2009, 24th IEEE/ACM International Conference on Automated Software Engineering, pp. 233–244 (2009)
9. Noor, T.B.; Hemmati, H.: A similarity-based approach for test case prioritization using historical failure data. In: 26th International Symposium on Software Reliability Engineering ISSRE, pp. 58–68 (2015)
10. Arafeen, M.J.; Do, H.: Test case prioritization using requirements-based clustering. In: 6th International Conference on Software Testing, Verification and Validation, pp. 312–321 (2013)
11. Laali, M.; Liu, H.; Hamilton, M.; Spichkova, M.; Schmidt, H.: Test case prioritization using online fault detection information. In: Ada-Europe International Conference on Reliable Software Technologies, pp. 78–93 (2016)
12. Wang, S.; Nam, J.; Tan, L.: QTEP: quality-aware test case prioritization. In: Foundations of Software Engineering, pp. 523–534. ACM, New York (2017)
13. Mei, H.; Hao, D.; Zhang, L.; Zhou, J.; Rothermel, G.: A static approach to prioritizing Junit test cases. IEEE Trans. Softw. Eng. **38**(6), 1258–1275 (2012)
14. Thomas, S.W.; Hemmati, H.; Hassan, A.E.; Blostein, D.: Static test case prioritization using topic models. Empir. Softw. Eng. **19**(1), 182–212 (2012)
15. Mohanty, R.; Suman, S.; Das, S.K.: Modelling the pull-out capacity of ground anchors using multi-objective feature selection. Arab. J. Sci. Eng. **42**(3), 1231–1241 (2017)
16. Shapiai, M.I.; Ibrahim, Z.; Adam, A.: Pareto optimality concept for incorporating prior knowledge for system identification problem with insufficient samples. Arab. J. Sci. Eng. **42**(7), 2697–2710 (2017)
17. Zhang, Y.; Harman, M.; Mansouri, S.A.: The multi-objective next release problem. In: GECCO'07, pp. 1129–1137. ACM, London (2007)
18. Ruiz, M.; Roderiguez, D.; Riquelme, J.; Harrison, R.: Multi-objective Simulation Optimization in Software Project Management. Oxford Brookes University, Oxford (2011)
19. Wang, Z.; Tang, K.; Yao, X.: Multi-objective approaches to optimal testing resource allocation in modular software systems. IEEE Trans. Reliab. **59**(3), 563–575 (2000)
20. Choudhary, K.; Purohit, G.: A Multi-objective optimization algorithm for uniformly distributed generation of test cases. In: IEEE International Conference on Computing for Sustainable Global Development, pp. 455–457 (2014)
21. Mondal, D.; Hemmati, H.; Durocher, S.: Exploring test suite diversification and code coverage in multi-objective test case selection. In: IEEE Conference on Software Testing, Verification and Validation, pp. 1–10 (2015)
22. Yoo, S.; Harman, M.: Pareto efficient multi-objective test case selection. In: ISSTA 2007, pp. 140–150. ACM, London (2007)
23. Marchetto, A.; Islam, M.; Scanniello, G.; Susi, A.: A multi-objective technique for test suite reduction. In: The 8th International Conference on Software Engineering Advances. IARIA (2013)
24. Zheng, W.; Hierons, R.; Li, M.; Liu, X.; Vinciotti, V.: Multi-objective optimization for regression testing. Inf. Sci. **334**, 1–16 (2015)
25. Canfora, G.; Lucia, A.D.; Penta, M.D.; Oliveto, R.; Panichella, A.; Panichella, S.: Defect prediction as a multi-objective optimization problem. Softw. Test. Verif. Reliab. **25**(4), 426–459 (2015)
26. Marchetto, A.; Islam, M.; Scanniello, G.; Asghar, W.; Susi, A.: A multi-objective technique to prioritize test cases. IEEE Trans. Softw. Eng. **42**(10), 918–940 (2016)
27. Karaboga, D.; Goremli, B.; Ozturk, C.; Karaboga, N.: A comprehensive survey: artificial bee colony (ABC) and applications. Artif. Intell. Rev. **42**(1), 21–57 (2014)
28. Karaboga, D.; Beyza, G.: A combinatorial artificial bee colony algorithm for travelling salesman problem. In: INISTA IEEE International Symposium, pp. 50–53 (2011)
29. Lam, S.S.B.; Raju, M.LH.P.; Kiran, U.M.; Swaraj, C.; Srivastava, P.R.: Automated generations of independent paths and test suite optimization using artificial bee colony. In: ICCTSD2011, Procedia Engineering 30, pp. 191–200 (2012)
30. Chong, C.S.; Low, M.Y.H.; Sivakumar, A.I, Lay.: A bee colony optimization for job shop scheduling. In: IEEE Proceedings of the 38th Conference on Winter Simulation, pp. 1954–1961 (2006)
31. Kaur, A.; Goyal, S.: A bee colony optimization algorithm for code coverage test suite prioritization. IJEST **3**(4), 2786–2795 (2011)
32. Srikanth; Kulkarni, N.J.; Naveen, K.V.; Singh, P.; Srivastava, P.R.: Test case optimization using artificial bee colony algorithm. In: International Conference on Advances in Computing and Communication, pp. 570–579 (2011)
33. Joseph, A.K.; RadhaMani, G.: A hybrid model of particle swarm optimization and artificial bee colony algorithm for test case optimization. IJCSE **3**(5), 459–471 (2011)
34. Mala, D.J.; Mohan, V.; kamalapriya, M.: Automated software test optimization framework and artificial bee colony optimization based approach. IET Softw. **4**(5), 334–348 (2010)
35. Dahiya, S.K.; Chhabra, J.K.; Kumar, S.: Application of artificial bee colony algorithm to software testing. In: 21st IEEE Australian Software Engineering Conference, pp. 149–154 (2010)
36. Konsaard, P.; Ramingwong, L.: Using artificial bee colony for code coverage based test suite prioritization. In: 2nd International Conference on Information Science and Security, pp. 1–4 (2015)
37. Aghdam, Z.K.; Arasteh, B.: An efficient method to generate test data for software structural testing using artificial bee colony optimization algorithm. Int. J. Softw. Eng. Knowl. Eng. **27**(6), 951–966 (2017)
38. Li, X.; Li, Z.; Lin, L.: An artificial bee colony algorithm for multi-objective optimization. In: 2nd International Conference on Intelligent Systems Design and Engineering Application, pp. 153–156 (2012)
39. Amarjeet, P.; Chhabra, J.K.: Many-objective artificial bee colony algorithm for large-scale software module clustering problem. Soft Comput. **22**(19), 6341–6361 (2017). https://doi.org/10.1007/s00500-017-2687-3

40. Mann, M.; Tomar, P.; Sangwan, O.P.: Bio-inspired meta heuristics: evolving and prioritizing software test data. Appl. Intell. **48**(3), 687–702 (2017). https://doi.org/10.1007/s10489-017-1003-3
41. Deb, K.; Pratap, A.; Agarwal, S.; Meyarivan, T.: A fast and elitist multi-objective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
42. Deb, K.: Multi-objective Optimization Using Evolutionary Algorithms, 1st edn. Wiley, London (2010)
43. Coreman, T.H.: Introduction to Algorithms, 2nd edn. PHI Printing Press, New York (2001)
44. Elbaum, S.; Rothermal, G.; Karre, S.; FisherII, M.: Leveraging user-session data to support web application testing. IEEE Trans. Softw. Eng. **31**(3), 187–202 (2005)
45. Elbaum, S.; Malishevsky, A.G.; Rothermal, G.: Test case prioritization: a family of empirical studies. IEEE Trans. Softw. Eng. **28**(2), 159–182 (2002)
46. Hutchins, M.; Foster, H.; Goradia, T.; Ostrand, T.: Experiments on the effectiveness of dataflow and control flow based test adequacy criteria. In: International Conference Software Engineering, pp. 191–200 (1994)
47. Wong, W.; Horgan, J.; London, S.; Mathur, A.: Effect of test set minimization on fault detection effectiveness. In: Proceedings 17th International Conference on Software Engineering, pp. 41–50 (1995)