Information and Software Technology 115 (2019) 119–130

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Test case selection-prioritization approach based on memoization dynamic programming algorithm

Ovidiu Banias

Politehnica University of Timisoara, Faculty of Automation and Computer Science, Automation and Applied Informatics Department, Parvan 2, Timisoara 300223, Romania

ARTICLE INFO

Keywords:
Software testing
Test case selection
Test case prioritization
Dynamic programming
Algorithms

ABSTRACT

Context: In the software industry, selection and prioritization techniques become a necessity in the regression and validation testing phases because a lot of test cases are available for reuse, yet time and project specific constraints must be respected.

Objective: In this paper we propose a dynamic programming approach in solving test case selection-prioritization problems. We focus on low memory consumption in pseudo-polynomial time complexity applicable in both selection and selection-prioritization problems over sets of test cases or test suites. In dynamic programming optimization solutions, huge amounts of memory are required and unfortunately the memory is limited. Therefore, lower memory consumption leads to a higher number of test cases to be involved in the selection process.

Method: Our approach is suited for medium to large projects where the required memory space is not higher than the order of tens of GBytes. We employed both objective methods as the dynamic programming algorithm and subjective and empiric human decision as defining the prioritization criteria. Furthermore, we propose a method of employing multiple project specific criteria in evaluating the importance of a test case in the project context.

Results: To evaluate the proposed solution relative to the classical dynamic programming knapsack solution, we developed a suite of comparative case studies based on 1000 generated scenarios as close as possible to real project scenarios. The results of the comparative study reported the proposed algorithm requires up to 400 times less memory in the best-case scenarios and about 40 times less memory in average.

Conclusion: The solution delivers optimal results in pseudo-polynomial time complexity, is effective for amounts of test cases up to the order of millions and compared with the classical dynamic programming methods leads to higher number of test cases to be involved in the selection process due to reduced memory consumption.

1. Introduction

In the last decades conclusive results have been achieved in test case selection and prioritization methodologies, topics in the focus of both research field and software industry. Usually, selection and prioritization become a necessity in the regression and validation testing phases when a lot of test cases and test suites are available for reuse, yet a specific time constraint must be respected for exercising the available test cases. In circumstances when the cumulated runtime of the available test cases exceeds the available project specific time constraint, decisions have to be made regarding the selection of a subset of test cases or test suites from the w hole set of available test cases. The scope of test case selection and prioritization methodologies is to maximize the fault detection rate in respect to the runtime constraint and project specific criteria such as code coverage, novelty of system under test or the novelty of the test case, the number of previously discovered defects, complexity of the requirements, and others.

According to the extensive surveys and literature reviews [1–6] on the test case selection approaches there are a plethora of researched techniques as: integer programming, data-flow analysis, dynamic slicing, graph walk, textual difference, path analysis, modification-based, firewall-based, cluster-based, design-based and others. To the best of our knowledge, there is no optimal approach solved by dynamic programming algorithms. In this paper we continue our previous study on dynamic programming practicability in resolving test case selection problems [7]. Therefore, we propose an optimal solution, as a dynamic programming algorithm approach, with efficient memory consumption.

In software industry it is common for projects to face situations when there are hundreds of thousands or even millions of available test cases that would require a total execution time of months order, yet the execution time is constrained to couple of days due to project specific reasons. Based on the input from our collaborators from the industry, we address the test case selection problem in the context of numerous test cases having a runtime measured in seconds and a total available testing runtime

E-mail address: ovidiu.banias@aut.upt.ro

https://doi.org/10.1016/j.infsof.2019.06.001
Received 11 October 2018; Received in revised form 22 May 2019; Accepted 14 June 2019
Available online 15 June 2019
0950-5849/© 2019 Elsevier B.V. All rights reserved.

up to the megaseconds (weeks) order. Because optimization methods require huge amounts of memory in the form of two-dimensional data structures for generating the optimal solutions and because the allocable memory is an issue that limits the number of test cases to be involved in the selection process, we focused on reducing the memory consumption and consequently increasing the number of usable test cases.

The majority of the selection methods are even based on code coverage or based on code modification criteria [1–3,8]. We consider these approaches very specific and restrictive by employing only one selection as long as software testing and test case selection decision could be influenced by a multitude of criteria at the same time [9,10]. We aim the generalization of the test case selection applicability by including in the selection process of multiple criteria, statically or dynamically defined, based on discussions and feedback from our software industry collaborators. The most employed criteria in test case selection will be presented and included in the selection strategy. Furthermore, we consider human experts' subjective opinion should strongly weight in the test case selection process as long as each project has specific constraints and criteria imposed by the customer.

Test case selection solutions support project managers in the decision-making process by minimizing the risks of excluding important test cases, or in other words by maximizing the potential fault detection rate. We consider project managers play an important role in decision making and no mathematic formula can decide and replace totally the subjective human opinion based on experience and specific project data that cannot be measured. We also consider the mix between computer optimization methods and human experts' decision making as the best solution at the moment when fully automated testing in large scale projects is not yet feasible. Thus, the proposed test case selection optimization algorithm is a mix between both human subjective opinions based on experience (defining the project importance criteria), and the objective optimization algorithm. The proposed solution could be applied in any of the test phases of the development cycle where optimization is needed, regardless of the test case features. Moreover, the solution is implementable in the same programming language as the potential project testing framework, having the benefit of being integrable with the testing framework and available for customization based on specific project demands.

The rest of the paper is organized as follows. In Section 2 the context and the related work will be presented in relation with both the test case selection methodologies and the dynamic programming optimization methods applied to computer science. In Section 3 the test case selection criteria will be formalized, and the selection cost will be defined, while in Section 4 the proposed algorithm will be formalized and presented as pseudocode. To evaluate the benefits of the proposed solution against the classical dynamic programming knapsack solution, an extensive and realistic comparative case study is presented in Section 5. In the final section conclusions will be made on the novelty of the proposed solution and the benefits brought to the test managers decision making process.

2. Background and related work

In this section we will present briefly the related work regarding test case prioritization and selection methodologies, discussing the context in which these methodologies are mandatory. Furthermore, we will formalize three types of techniques as prioritization, selection and selection-prioritization and will introduce the dynamic programming algorithmic approach.

Agile and Kanban [11,12] software development methodologies are extensively used today. Following these methodologies implies that testing and development are closely interconnected and the need for test case selection becomes a must in situations when running the whole set of test cases is not an option due to the insufficiency of available time. The testing team can face situations when the software is still under development or the development of a certain feature is ready for testing. Usually, running the integration, regression or validation test cases is needed but constrained by time and other specific project criteria [1–3,13–16]. In the early stages of the project development there is no need for selection methods because the total number of test cases is not numerous. The cumulated runtime of the whole set of test cases would not exceed the maximum allocated time constraint and consequently all the test cases would be executed with no time constraint. As the project development advances in time, the project features are growing and the lines of code are increasing along with the number of the test cases. Due to client specific requirements and project timelines the allocated time for running the test cases is limited and often not enough for running all the test cases. Therefore, project managers and test managers are facing the need for selecting from the available set of test cases the ones that are more prone to reveal defects in the available amount of time. Decisions must be made in the direction of running certain test cases with the risk of excluding others that may or may not reveal defects.

As mentioned briefly in the previous section, the need for reduction of the number of test cases that will be exercised in different testing phases is common in the software industry. The challenge of reducing the exercising set of test cases due to project time constraints is mostly met in the regression testing and validation testing phases when a lot of test cases are available for reuse. Yet there is a high chance just some of them could expose defects. The higher the complexity of the project, the increased number of test cases to be performed and the furthest the testing team is from the ideal case scenario in which all the test cases may be exercised without a time constraint. As the development, testing, debugging and fixing defects are continuous and interconnected processes in most of the software projects, reducing or increasing the allocated time in one of these phases could generate a domino effect in the other phases. Along with the customers-imposed deadlines, the project could follow a low-quality direction unless the allocated time for development, testing, debugging and fixing is wisely leveraged by the project managers. This issue is another fact that supports the general opinion of the researchers [1–4] to value the expertise of the human experts and suggest not to be excluded from the decision making in the software projects.

It is common in the software industry to group test cases having the same characteristics in test suites and thereupon solve selection problems over test suites as reduced sets of test cases [1,3,17]. Test suites selection problem is much easier to solve as the number of items to be selected is drastically reduced with the order of hundreds or even thousands. In test suite selection problems, all the test cases belonging to a certain test suite are even all exercised or none of them are exercised. The constraint of exercising all the test cases in a test suite eases the complexity in time and memory of the selection problem but has the drawback of dropping important test cases that belong to low priority test suites. The decision upon choosing between defining a test case selection problem or a test suite selection problem falls into the tasks of test managers and project managers and is influenced by different project specific criteria.

Extensive research has been made on different regression testing approaches [1–3,13,14,18] with notable results that could be classified in three major categories: minimization, selection and prioritization. Minimization and selection methodologies are based on the same principles of selecting a subset of test cases from the available set. The minimization method is meant to permanently eliminate test cases from test suites, while the selecting method is meant to temporary eliminate test cases from the current test execution. Prioritization methodologies [1–3,13] are in the focus of researchers in the last period more than minimization and selection methodologies as the software industry has a higher demand in this direction. A test case prioritization method orders a set of test cases based on specific criteria with the final scope of increasing the potential fault detection rate by running first the test cases most prone to expose faults. In practice, selection and prioritization methods are interconnected. The selection method could become a selection-prioritization method if we add to the time constraint, the ordering of test cases constraint. At the same time, a prioritization method could become a selection-prioritization method if we add the time constraint

to the ordering constraint that is specific to prioritization. Even though a lot of techniques were proposed by researchers in the past decades [1–3,13,17], there is a common statement about existence of no algorithms that can solve the prioritization and selection problems without mixing them with human expertise and online human decision making.

2.1. Test case selection

The reason for applying a test case selection technique is to select from the set of available test cases a subset of test cases having the property that the cumulated runtime does not exceed the allocated time constraint for the exercising of the test cases. The advantage of employing a selection technique is that if not all the test cases could be executed in the imposed time constraint, at least a part of the test cases will be selected and executed based on their probability to reveal defects. The tradeoff that comes along with the selection techniques involves project safety and efficiency [17]. High safety criteria may require exercising of more test cases on one side, striving for efficiency criteria may sacrifice safety on the other side.

We present below the formalization of the selection problem.

Given: a set of test cases $TC = \{1, 2, \dots, card(TC)\}$, the estimated execution time of each test case $Time_i$, and T_{max} the total available time for exercising the test cases,

Problem: find a subset S of test cases, $S \subset TC$ so that

$$\sum_{i=1}^{card(S)} Time_i \leq T_{max} \tag{1}$$

based on given criteria and costs values relative to each of the test cases.

In specific contexts, selection could also be considered prioritization because the subset S could be considered having a higher priority than the subset $TC - S$. Selection could become a selection-prioritization method if we add the ordering of test cases constraint to the time constraint as formalized in one of the next subsections.

2.2. Test case prioritization

Test case prioritization techniques are employed for ordering a set of available test cases based on specific criteria such as the degree of importance and the potential degree of fault detection. The goal is to exercise those test cases in the exact order as produced by the prioritization technique and if there are tests to fail, they should fail fast, at the beginning of the test cases exercising.

We present below the formalization of the prioritization problem.

Given: a set of test cases $TC = \{1, 2, \dots, card(TC)\}$,

Problem: find a permutation $P_{TC}(k)$ of the set TC, that maximizes the objective function f so that

$$f(P_{TC}(k)) \geq f\left(P_{TC}(i)\right) \mid \forall i, k = 1 \dots \ card(TC)! \ and \ i \neq k \tag{2}$$

where:

- $P_{TC} = \{P_1, \dots, P_{card(TC)!}\}$ is the set of all the permutations of the test cases from the set TC,
- $f : P_{TC}(i) \rightarrow \mathbb{R}, \ \forall i = 1 \dots card(TC)!$ is the objective function,
- i is the index of the permutation P_i from the set P_{TC},
- k is the index of the permutation P_k from the set P_{TC}.

Multiple approaches have been researched regarding prioritization. According to [1], one classification of these approaches could be but not restricted to: coverage based, requirement based, risk based, search based, fault based, history based, cost aware based, Bayesian Networks based, model based, multi-criteria based, and similarity based.

2.3. Test case selection-prioritization

The selection-prioritization technique is a mix of selection and prioritization techniques. In the software industry, this mixed approach is very common as both time constraint and the ordering constraint are present in various project scenarios of decision making. A selection technique produces as output an unordered list of test cases based on given importance criteria of each of the test cases. By ordering the selected list of test cases by the importance criteria and executing the test cases in this order, it can be observed that the more important test cases will be executed first. Therefore, the selection-prioritization techniques could be perceived as selection techniques with an ending of extra ordering of the test cases based on already used importance criteria.

We present below the formalization of the selection-prioritization problem.

Given: a set of test cases $TC = \{1, 2, \dots, card(TC)\}$, the estimated execution time of each test case $Time_i$, and T_{max} the total available time for exercising the test cases,

Problem: find a subset U of test cases, $U \subset TC$ so that

$$\sum_{i=1}^{card(U)} Time_i \leq T_{max} \tag{3}$$

and find a permutation $P_U(k)$ of the subset U, that maximizes the objective function f so that

$$f(P_U(k)) \geq f\left(P_U(i)\right), \ \forall i, k = 1 \dots card(U)!, \ i \neq k \tag{4}$$

where:

- $P_U = \{P_1, \dots, P_{card(U)!}\}$ is the set of all the permutations of the test cases t from the subset U,
- $f : P_U(i) \rightarrow \mathbb{R}, \ \forall i = 1 \dots card(U)!$ is the objective function.

2.4. Dynamic programming applied to test case selection problem

In this paper we refer to solving problems by dynamic programming method as a computer science algorithm and not as a mathematical optimization solution [19]. Dynamic programming method was developed by Richard Bellman in 1952 [20] as a mathematical optimization method for solving specific problems that admit solutions solvable by recurrence formulas. The principles of optimization were imported into computer science as an algorithmic technique and were implemented by writing lines of code in a programming language [21–23]. In computer science, dynamic programming algorithms are known for solving optimization problems by recursively splitting them into easier to solve sub-problems until the sub-problems are solvable. Therefore, the solution to the initial optimization problem is composed by combining of the results of the already solved sub-problems. In the current situation of test case selection, the problem admits solutions solvable by either mathematical dynamic programming approaches or dynamic programming algorithms. For the current goal of the paper of resolving the test case selection problem, we have chosen an algorithmic dynamic programming approach. Compared to the mathematical approach it is more customizable, leading to fine-tuned optimizations in space and time complexity thorough the power of writing lines of code in a certain programming language. In the context of the current paper we decided to compare the proposed solution as a dynamic programming algorithm to the classical knapsack solution. The reason we have made this choice is that the classical knapsack approach is solvable and produces the same results by both dynamical programming mathematical solution and computers science algorithmic solution.

In Section 4 we focus on solving the selection problem by means of computer science as a dynamic programming algorithm. We break the available set of test cases in smaller subsets recursively until the optimization sub-problems could be solved. After resolving the sub-problems, we combine the partial (sub-optimal) solutions backwards for solving the initial test case selection problem.

3. Test case selection criteria

In the first part of this section the test case selection criteria are discussed along with the process of evaluating the degree that a test

case satisfies a specific criterion. In the second part of the section we define the cost function of prioritization/importance of a given test case in both contexts: the local criteria context and global project context.

3.1. Selection criteria

The scope of the test case selection method is to choose the test cases more prone to reveal defects and exercise them without exceeding a given time constraint. To be able to objectively evaluate the importance of each of the test cases in the project context, the weight/importance of a certain test case is computed based on both the importance of the test case relative to a specific criterion, and the importance of the criterion in the scope of the project. Before initiating the selection process, the importance criteria should be defined based on project specific needs. The criteria could be classified in two categories: objectively calculated by means of software tools and subjectively defined based on human experience. Our collaborators report multiple criteria of high efficiency in the test case selection process [7]:

- Novelty of system under test (snippet of code tested by the given test case/test suite) – the more recent the code was implemented the highest the priority of the test case for execution;
- Novelty of the test case – even if the system under test components were not implemented recently, latest developed test cases have higher priority than older developed test cases because new test cases unexecuted yet are prone to discover new defects;
- Complexity of the requirements – the more complex the requirement linked with a test case the higher the priority of the test case;
- Number of defects discovered in the last test suite execution – the higher the number of the discovered defects in a certain test suite execution, the higher the priority of the test cases belonging to the test suite;
- Client unsolved/unanswered clarifications that are strictly linked with the requirements – in projects where there is a link between test cases and requirements, the test cases that are related to unanswered clarifications are more prone to produce defects;
- Importance of the safety of the end user – if the test case is related to functionality that concerns the safety or the health of the end user, then the test case has higher priority than test cases unrelated to safety functionalities;
- Importance of the efficiency in time – usually represented by smoke tests;
- Complexity of the under-test software component – the higher the cyclomatic complexity [24], the higher the risk of producing defects.

In practice, the weight of a certain test case regarding the relative importance to the project is defined as a value objectively computed or subjectively defined in respect to the project priorities. The criteria mentioned above are dynamically defined by the testing team and project managers during the lifecycle of the project and are related to specific project necessities and customer requirements. We propose the following incremental process of evaluating the degree that a test case satisfies a criterion:

- The set of importance criteria is defined by the project managers relative to the project context.
- For each criterion a specific percentage of importance is empirically set and represents the importance of the criterion in the project context. For example, if one of the defined criterion would be “novelty of the written code (system under test)” and the project under test is facing an influx of new developers, then the project managers might give a higher importance for a certain period of time to this criterion because the new developers in the project might produce at the beginning more defects than the seasoned developers. Therefore, test cases that would test newly written code would have a higher importance in the general context due to higher importance of the “novelty of the written code” criterion.
- For each of the defined project criteria, every test case available to the process of selection will be given a nominal value representing the importance of the test case relative to the given criterion. The nominal values are even empirically set by the project managers before initiating the selection process or are dynamically set or imported from software analysis tools.
- At test case level, for each available criterion, two parameters would be available: the test case importance in the local context of a certain criterion and the importance of the criterion in global context of the project. Based on these two parameters, the importance of a test case in the global project context is evaluated. In the next section we will present further details and how the test case importance should be calculated in real project scenarios.

3.2. Selection cost

In order to formalize the optimization problem, we define the cost function for each test case as a nominal value representing the importance of selecting the test case from the total available set of test cases. Without losing the degree of generality, we define the cost function of prioritization/importance of a given test case as a function of: 1) the weight of the test case relative to a specific criterion and 2) the weight/importance of the criterion in the project context.

The formulation of the test case weight function relative to the defined criteria discussed in previous subsection is presented in the Table 1.
where:

- $W_i(j)$ is the weight/importance of the *TestCase* i in respect to the *Criterion* j,
- *TestCase* i is the test case number i, $i = 1 \dots n$,
- *Criterion* j is the criterion number j, $j = 1 \dots m$,
- C_k is the weight of the criterion number k, $k = 1 \dots m$, and represents the importance of the criterion in the project context,
- n is the total number of test cases,
- m is the total number of prioritization criteria.

Test case selection criteria are project specific, some of the criteria are human subjective and empirically defined, the others are measurable and computed objectively. We formulate further in the Eq. (5) the cost function for a given test case as the importance of the test case in the project context:

$$Cost_i = f\left(W_i(j), C_j\right) \tag{5}$$

where:

- $Cost_i$ is the cost/importance of the test case number i,
- f is the project specific defined cost function,
- $W_i(j)$ is the weight/importance of the test case number i in respect to the criterion number j,
- C_j is the weight/importance of the criterion number j in respect to the project context,
- $\sum_{j=1}^{m} C_j = 1$.

In Eq. (5a) we propose a useful and easy implementable cost function for a given test case as a sum of products between the importance of each criterion in the project context and the importance of the test case in the criterion context:

$$Cost_i = \sum_{j=1}^{m} W_i(j) * C_j \tag{5a}$$

with nominal values for $W_i(j)$ and C_j as percentage.

The majority of the software development projects today follow the first and second Agile principles from the Agile manifesto [25], thus continuous changing of the requirements and of the project goals during

Table 1
Weight/importance of test cases.

	Criterion #1 (C_1%)	Criterion #2 (C_2%)	. . .	Criterion #m (C_m%)
TestCase #1	$W_1(1)$	$W_1(2)$		$W_1(m)$
TestCase #2	$W_2(1)$	$W_2(2)$		$W_2(m)$
…				
TestCase #n	$W_n(1)$	$W_n(2)$		$W_n(m)$

development is not unexpected, on the contrary it is welcomed. Therefore, we consider it is important that the decisional team members in the project development should be involved in defining and continuously adjusting the test case selection criteria and importance formalized in this section, in strict relation with the changing project context.

4. Dynamic programming algorithm

In the current section the proposed algorithm will be formalized based on the definition of the prioritization cost function described in the previous subsection. Furthermore, the pseudocode representation of the algorithm will be presented and rigorously described.

The dynamic programming algorithm will be presented in three steps: 1) the optimization problem statement; 2) the recurrence formula; 3) the backward algorithm for computing the maximal cost and the related list of test cases. Further discussions will be made in the next section on the advantages of the proposed solution.

4.1. The optimization problem

Given:

- $TC = \{1, 2, \dots, card(TC)\}$ as the set of available test cases,
- $Cost_i$ as the importance/priority of the test case number i as a nominal value,
- $Time_i$ as the estimated execution time of the test case number i,
- T_{max} as the time constraint for exercising partially or totally the set of given test cases,

Problem: find a subset S of test cases (descending sorted by $Cost_i$), $S \subset TC$ that maximizes: subject to:

$$\sum_{i=1}^{card(S)} Time_i \leq T_{max} \tag{7}$$

4.2. The recurrence formula

In order to solve the optimization problem described in the previous section, in this section we propose and formalize a dynamic programming algorithm approach.

For each available test case, the algorithm searches incrementally the sub-optimal solutions that the current test case could be attached to and thereupon build new sub-optimal solutions. We consider a sub-optimal solution as a solution that solves the selection problem over a reduced set of test cases. In the context of the current algorithm, a reduced set of test cases always contains all the test cases in the initial set until a certain element and does not contain the rest of test cases from that element on.

In Eq. (8) we define the recurrence formula for calculating sub-optimal solutions that may or may not include the current test case based on previously calculated sub-optimal solutions from the first $i-1$ test cases. We define $O[j + Time_i]$ as the optimum/maximum cost of selecting a subset of test cases from the set of the first i test cases having the property that the sum of execution time is $j + Time_i$ seconds. In Eq. (8a) we define a data structure as a set for memorizing the test cases that compose the sub-optimal solutions that are related to the selection costs defined in Eq. (8).

$$O\left[j + Time_i\right] = \max\left(O\left[j + Time_i\right],\ O[j] + Cost_i\right) \tag{8}$$

$$S\left[j + Time_i\right] = S[j] \cup \{i\}, \tag{8a}$$

where:

- i is the test case iterator, $i = 1 \dots card(TC)$,
- j is the cumulated execution time in seconds of the test cases that compose the sub-optimal solution with the cost $O[j]$,
- $O[j]$ is the optimum/maximum cost of a subset of test cases executable in j seconds. $O[j] = 0$ when there is no subset of test cases that have a cumulated execution time of j seconds,
- $S[j]$ is the subset of test cases that compose the sub-optimal solution with the cost $O[j]$,
- $j + Time_i \leq T_{max}$.

We defined the recurrence formula in Eq. (8) based on the following observations:

- the optimal selection between the first i test cases requires a decision from a maximum of 2^{i-1} sets based on all the possible combinations of selecting between the first $i-1$ test cases (i.e. {}+{i}, {1}+{i}, …, {i-1}+{i}, {1,2}+{i}, {1,3}+{i}, …, {1,i-1}+{i}, …, {1,2,…,i-1}+{i}),
- the sub-optimal selection from the set of first i test cases, $i = 1 \dots$ TC, could be computed based on the sub-optimal selections from the set of first $i-1$ test cases by either including the test case number i in the optimal solution or excluding it. This observation reduces significantly the number of possible combinations to be covered and also matches the dynamic programming paradigm of decomposing a problem in similar ones of reduced complexity,
- before including the test case number i into the optimization process, multiple sub-optimal solutions, assembled from the first $i-1$ test cases (with a cumulated execution time of j seconds, $j \in [1, T_{max}]$, and sub-optimal costs O[j]) are available through the positive elements of the O array,
- in order to attach the test case number i to the sub-optimal already existing solutions obtained from previous $i-1$ test cases, the following condition should be met: the sum of costs, $Cost_i$ (the cost/importance of test number i) and O[j] (the cost of optimal selection from previous $i-1$ test cases with a total execution time of j seconds) should not be lower than the already existing cost O[$j + Time_i$] related to the sum of execution time $Time_i$ (the execution time of test case number i) and j (the execution time of sub-optimal selection from previous $i-1$ test cases).

4.3. The algorithm

The algorithm requires for input the set of test cases, the execution time $Time_i$ of each test case and the importance $Cost_i$ of each test case. The algorithm produces at output the optimum/maximum selection cost $O[j]$ and the associated sorted subset of test cases $S[j]$.

Input: $= \{1, 2, \dots, card(TC)\}$, $Time_i$, $Cost_i$, $T_{max} \mid i = 1 \dots card(TC)$
Output: $\max(O[j])$ and associated $S[j] \mid j = 1 \dots T_{max}$

The algorithm is presented in pseudocode form in Fig. 1 and is described thoroughly as follows. The algorithm starts in line number 1 by iterating through the set of test cases, from the first test to the last one. At each step i, it is assessed if adding the test case number i to the already existing sub-optimal solutions (subsets of testcases

```
1: for i = 1 to nrTests do
2:   for j = crtMaxTime downto 0 do
3:     if (j+time[i] <= tMax) AND (O[j+time[i]] < O[j]+cost[i]) then
4:       O[j+time[i]] = O[j]+cost[i];
5:       S[j+time[i]] = S[j] + {i};
6:       if (crtMax < j+time[i]) then crtMaxTime = j+time[i] end if
7:     end if
8:   end for
9: end for
10: max = 0; j = 0;
11: for t = crtMaxTime downto 0 do
12:   if (O[t] > max) then max = O[t]; j = t; end if
13: end for
14: return SortByImportance(S[j])
```

Fig. 1. The proposed algorithm pseudocode.

formed from the previous $i-1$ testcases) would produce a better sub-optimal solution. In line number 2, the variable j iterates from the current maximum cumulated execution time of existing sub-optimal solutions down to 0. If the conditions in line number 3 are met, in line number 4 new sub-optimal solutions are built from 1) already existing sub-optimal solutions composed from a subset of fist $i-1$ test cases, with cost $O[j]$ and total execution time of j seconds and 2) the test case number i with execution time $Time_i$ and importance $Cost_i$. Also, if the conditions in line number 3 are met, in line number 5 the subset of testcases that compose the new sub-optimal solution is saved and computed as the reunion between the test case number i and the subset of test cases related to the sub-optimal solution executable in j seconds.

In line number 3 it is assessed if the test number i is suitable to be combined to a sub-optimal existing solution composed from a subset of previous $i-1$ test cases and characterized by a cumulated execution time of j seconds. First, the cumulated execution time j plus the execution time $Time_i$ of the test case number i should not exceed the time constraint T_{max} because it would not comply with the requirements. Second, building a new sub-optimal solution that includes the test case number i and an existing sub-optimal solution with cost $O[j]$ would require the current cost $O[j+Time_i]$ of the test cases that are executable in $j+Time_i$ seconds to be lower than the cumulated costs of the new potential sub-optimal solution. Line number 6 is used for reducing the complexity in time of the algorithm by providing for each test case i the maximum limit in seconds reached in the previous steps of the algorithm. Furthermore, searching for sub-optimal solutions outside this limit it would be ineffective. In the lines 11–14 the solution with maximum cost is selected and in line number 14 the list of test cases is returned as a sorted list based on the importance criteria.

Studying the pseudocode presented in the Fig. 1 it can be observed the two repetitive cycles in the first two lines influence the most the complexity in time of the algorithm as the cycle in the last lines has much less iterations. Because the variable *crtMaxTime* may reach a maximum value equal to T_{max} in worst case scenarios, we conclude the proposed algorithm has a complexity in time of $O(card(TC)^*T_{max})$. Almost the same complexity in memory space of $O(card(TC)^*T_{max}/2)$ is obtained in the worst-case scenario when the sub-optimal solutions grow fast from one element to nearly all the test cases available for selection.

5. Comparative case study

In this section we present three realistic case studies in order to evaluate the proposed solution (PA) relative to the classical 0–1 knapsack [26] solution (CA). We will briefly present the recurrence formula for the classical knapsack solution (the algorithm and details could be reviewed in [7]), then we discuss the differences between the two approaches regarding the complexity in memory space, and finally we present results based on multiple test case selection scenarios.

5.1. Preliminary discussion

The recurrence formula for the CA [7] is presented in Eq. (9):

$$Q[i][t] = \begin{cases} Q[i-1][t], \ if \ Q[i-1][t] > \\ Q[i-1]\left[t - Time_i\right] + Cost_i \\ \quad or \\ Q[i-1]\left[t - Time_i\right] + Cost_i, \ else \end{cases} \tag{9}$$

where $Q[i][t]$ is the optimum cost of selecting a subset of the first i test cases having the sum of execution time t seconds.

As presented and discussed in the previous section, the complexity in time of the proposed algorithm is $O(card(TC)^*T_{max})$, the algorithm being quadratic in time and dependent on the number of test cases and the time constraint. After finding the optimum selection cost, the next step is to generate the solution as a list of selected test cases. To be able to generate the solution, for each updated sub-optimal cost $O[j + Time_i]$ a list of test cases that compose the sub-optimal solution should be kept in memory. The advantage of saving in memory the sub-optimal solution for each element in the vector O is that the final solution as a list of test cases can be generated immediately in $O(card(TC))$ time complexity through a simple iteration over the solution set. Without memorizing the solutions as presented above, generating the solution as a list of test cases for a given $j | j \in [1, T_{max}]$ would require a recursive and exponential approach. As usually in the software industry, the number of test cases is large, and an exponential approach would not be feasible in time. The drawback of keeping the partial solutions in memory as a list of test case indexes is that to the already necessary memory of complexity $O(card(TC))$ for optimum cost generation, a maximum of $O(card(TC)^*T_{max}/2)$ would add up for fast generation of the final solution.

Usually the runtime of the test cases is measured in seconds. For higher values of T_{max} (e.g. 10^5 to 10^7 s) combined with huge amounts of test cases (e.g. 10^6) the required memory size could exceed 1 TByte ($1Tbyte = 10^{12} bytes$) and the complexity in time could reach the order of hours. Although the runtime of the order of hours could be acceptable in certain scenarios, allocating 1 TByte of RAM without parallelization is not an option due to operation systems limitations. The memory complexity of CA is $O(card(TC)^*T_{max})$, thus based on the Eq. (9) the required memory in bytes is given by the Eq. (10):

$$Mem_{CA} = card(TC) * T_{max} * mem(Q[i][t]) \tag{10}$$

where $mem(x)$ is the memory required by the variable x.

As the values of the test cases importance $Cost_i$ are calculated and manageable based on human defined criteria, the $mem(Cost_i)$ values should not exceed 2 bytes in normal case scenarios. Given the fact that in the current case study we consider the number of test cases of up to 10^5, the memory required by each of the elements of the two-dimensional vector Q is $mem(Q[i][t]) = 4 \ bytes$. Thus, the total memory required by CA is:

$$Mem_{CA} = card(TC) * T_{max} * 4 \ bytes. \tag{11}$$

The memory complexity of PA is $O(card(TC)^*T_{max}/2)$, thus based on the Eq. (8) the required memory in bytes is given by the Eq. (12):

$$Mem_{PA} = T_{max} * mem(O[j]) + \sum_{j=1}^{T_{max}} mem(S[j]) \tag{12}$$

where:

- $j \in [1, T_{max}]$,
- $S[j]$ is the list of test cases for the sub-optimal solution with the cost $O[j]$.

The memory required by the $O[j]$ elements of PA solution is similar with the memory required by the $Q[i][t]$ elements of CA solution, $mem(O[j]) = 4 \ bytes$, in the same conditions: the number of test cases up to 10^5 and $mem(Cost_i) = 2 \ bytes$. One of the advantages of the PA solution is that even considering the impossible scenario that $S[j] = \{1, 2, \ldots, card(TC)\}, \ \forall j = 1 \ldots T_{max}$, because the elements of the set S require a maximum of 2 bytes of memory, the total memory required by the PA is reduced to at least a half of the memory required by CA. Taking in consideration that the total allocable memory is limited, only the reducing of the required memory to at least a half is a major improvement compared to the CA as the number of test cases that could be involved in the selection process could be at least doubled. In the context of test case selection problems, the difference between being able to select from 10^5 test cases or between 2^*10^5 test cases in the same memory space, is unneglectable. Thus, the maximum memory required by the PA is:

$$Mem_{PA} = T_{max} * 4 + \ card(TC) * T_{max} * 2 \ bytes. \tag{13}$$

From the Eq. (11) it can be observed that regardless of number of test cases $card(TC)$ and the time constraint T_{max}, the memory size required by the CA algorithm is the same for every scenario with no relation to the selection solution. On the other hand, from Eq. (12) it can be observed that the memory size required by the PA algorithm is strictly related to the dimensions of the sub-optimal solutions and consequently related to the selection solution. In practice, the dimensions of the sub-optimal solutions are much smaller than the total number of test cases $card(TC)$ and consequently the required memory size of the PA algorithm is much smaller. In the following case studies we investigate how the dimensions of the sub-optimal solutions influence the required memory size of the PA algorithm.

5.2. Case studies

For evaluating the PA memory requirements, we developed scenarios with T_{max} ranging from 1 day $\cong 10^5$ seconds to 1 month $\cong 10^6$ seconds and the number of test cases up to 10^5. We defined three types of scenarios:

- the exercising runtime of all the test cases much greater than the time constraint T_{max} ($\sum_{i=1}^{card(TC)} Time_i \gg T_{max}$),
- the exercising runtime of all the test cases is close to double the time constraint T_{max} ($\sum_{i=1}^{card(TC)} Time_i \cong 2 * T_{max}$),
- distinct cost values for each of the test cases.

We generated a total of 1000 scenarios classified by the three criteria mentioned above and we detailed it further in the following subsections. We defined a scenario as a function of four parameters: the number of test cases available for selection, the time constraint, the range of values for each test case importance and the range of values for each test case runtime:

$$Scen_i\left(card(TC), T_{max}, \left[cost_{min}, cost_{max}\right], \left[time_{min}, time_{max}\right]\right), \tag{14}$$

where:

- $card(TC) \in \{i*10^3 | i = 1 \ldots 10\} \cup \{i*10^4 | i = 1 \ldots 10\}$,
- $T_{max} \in \{i*10^4 | i = 1 \ldots 10\} \cup \{i*10^5 | i = 1 \ldots 10\}$,
- $Cost_i \in [cost_{min}, cost_{max}], \ i = 1 \ldots card(TC), cost_{min} \geq 1, cost_{max} \leq 10^5$, randomly generated,
- $Time_i \in [time_{min}, time_{max}], \ i = 1 \ldots card(TC), time_{min} \geq 10, time_{max} \leq 10^4$.

For evaluating the PA algorithm over an extensive set of scenarios, the importance of the test cases in the project context were randomly generated. Not to lose the degree of generality of the current case studies we did not calculate the importance of a test cases based on any subjective criteria. Defining the selection criteria, the importance/weight of a test case in the local criteria context and the importance/weight of the criteria in the global project context as described in Eq. (5), should be defined by the end user relative to the project specific needs. The PA algorithm was run for each of the generated scenarios and the required memory size was measured based on Eq. (13). Considering the Eq. (11) formula, the ratio Mem_{opt} was calculated:

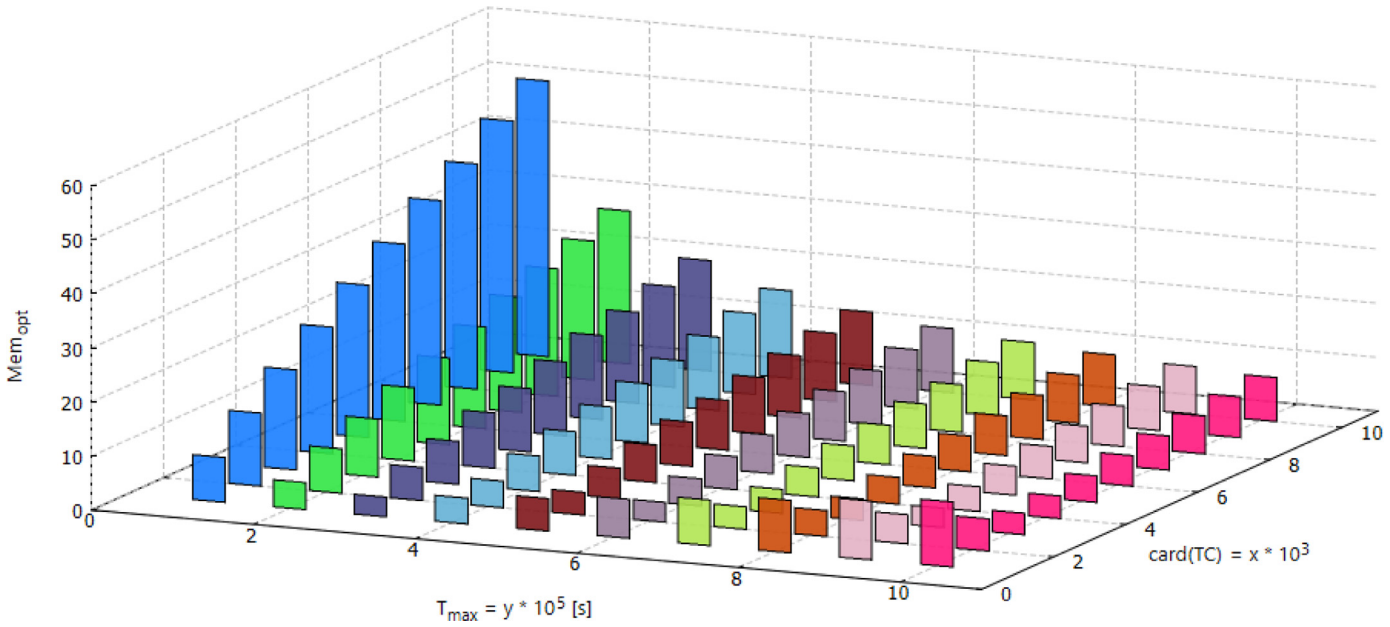

Fig. 2. Memory optimization in $\{Scen_1, \ldots, Scen_{100}\}$.

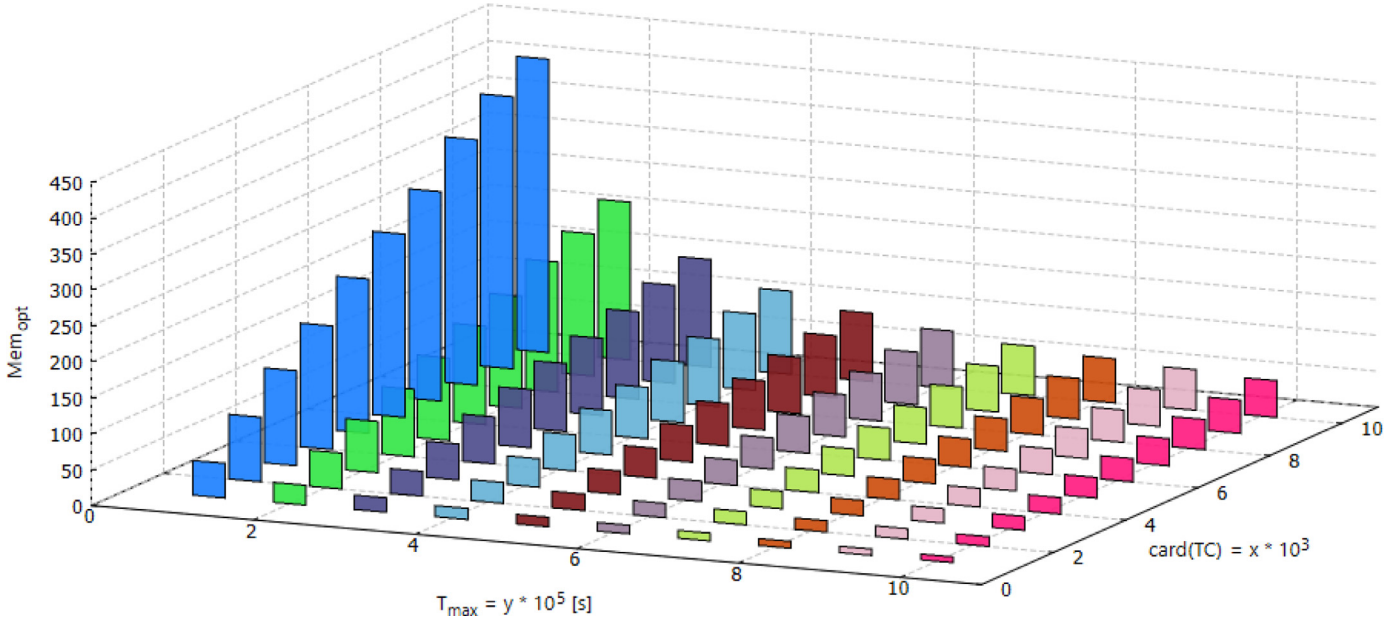

Fig. 3. Memory optimization in $\{Scen_{101}, \ldots, Scen_{200}\}$.

$$Mem_{opt} = \frac{Mem_{CA}}{Mem_{PA}} \tag{15}$$

The results were compacted and plotted for groups of 100 scenarios each as ($card(TC)$, T_{max}, Mem_{opt}) 3D charts.

5.2.1. Group of scenarios 1

We generated 400 scenarios fulfilling the requirement that the exercising runtime of all the test cases must be much greater than the time constraint T_{max}. We considered the following criteria:

- $\sum_i^{card(TC)} Time_i \gg T_{max}$,
- $card(TC) \in \{i{*}10^3 | i = 1 \ldots 10\}$,
- $T_{max} \in \{i{*}10^5 | i = 1 \ldots 10\}$,
- $Cost_i \in [1, 10^2],\ i = 1 \ldots card(TC)$, randomly generated,
- $Time_i \in [10^2, 5{*}10^3],\ i = 1 \ldots card(TC)$, randomly generated and divided in four intervals for each of the studied categories presented below.

The generated scenarios were divided in four categories:

- $\{Scen_1, \ldots, Scen_{100}\},\ Time_i \in [10^2, 5{*}10^2],\ i = 1 \ldots card(TC)$,
- $\{Scen_{101}, \ldots, Scen_{200}\},\ Time_i \in [3{*}10^2, 1.5{*}10^3],\ i = 1 \ldots card(TC)$,
- $\{Scen_{201}, \ldots, Scen_{300}\},\ Time_i \in [7.5{*}10^2, 3.5{*}10^3],\ i = 1 \ldots card(TC)$,
- $\{Scen_{301}, \ldots, Scen_{400}\},\ Time_i \in [10^3, 5{*}10^3],\ i = 1 \ldots card(TC)$.

The results for each of the four categories of scenarios were plotted in Figs. 2–5 respectively. Studying the results, we observed that in the worst-case scenario the memory used by the PA is 3.6 times lower than the memory used by the CA and in the best-case scenario the memory

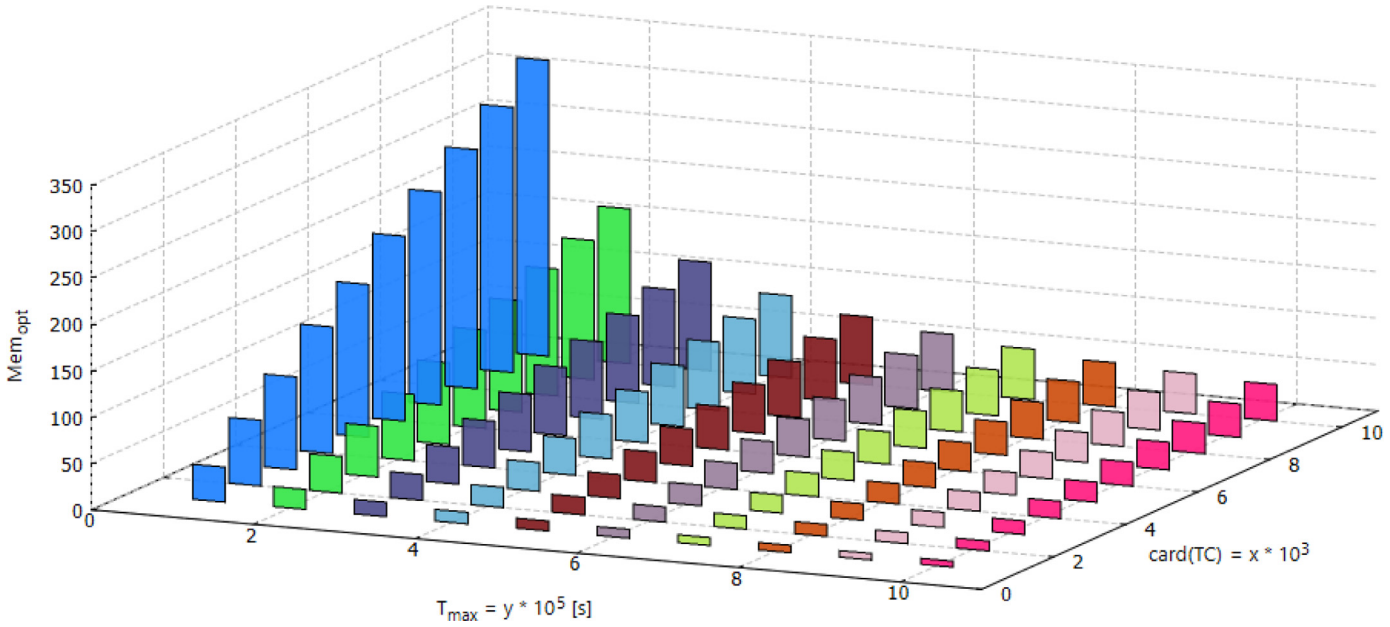

Fig. 4. Memory optimization in $\{Scen_{201}, \ldots, Scen_{300}\}$.

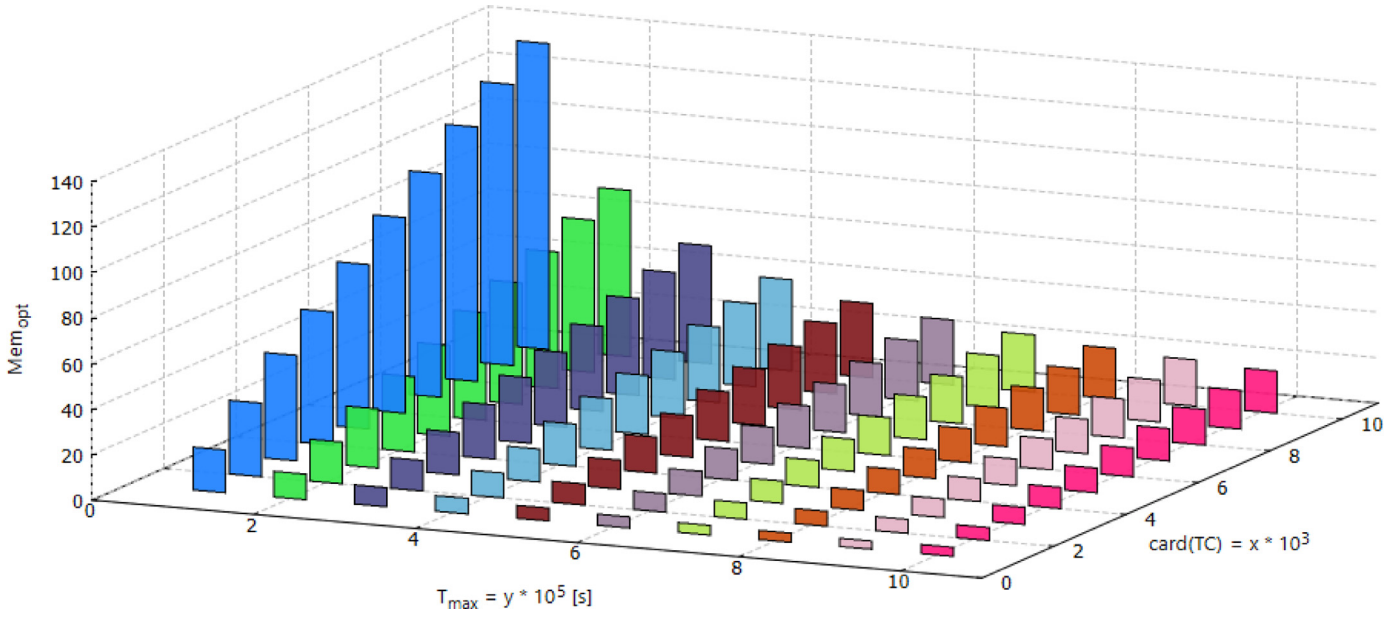

Fig. 5. Memory optimization in $\{Scen_{301}, \ldots, Scen_{400}\}$.

used by PA is 408 times lower. Regardless of the scenarios category we observe that the closer the values of $card(TC)$ and T_{max}, the lower the ratio Mem_{opt} (e.g. $Mem_{opt} = 3.6$) and the farther the values of $card(TC)$ and T_{max}, the higher the ratio Mem_{opt} (e.g. $Mem_{opt} = 408$). Comparing scenarios $\{Scen_1, \ldots, Scen_{100}\}$ with scenarios $\{Scen_{101}, \ldots, Scen_{200}\}$ we observe that the higher the $\sum_{i=1}^{card(TC)} Time_i$, the less memory is required by the PA.

Studying the Mem_{opt} values distribution in the scenarios $\{Scen_{101}, \ldots, Scen_{200}\}$, $\{Scen_{201}, \ldots, Scen_{300}\}$ and $\{Scen_{301}, \ldots, Scen_{400}\}$, we observed that the proportions between scenarios belonging to the same group of scenarios are the same for all the three groups. Also, we observed two scenarios with the same parameters except the $Time_i \in [time_{min}, time_{max}]$ produce different results and more exactly Mem_{opt} reaches higher values for smaller values $time_{max} - time_{min}$:

$$Mem_{opt} \propto \frac{1}{time_{max} - time_{min}} \tag{16}$$

5.2.2. Group of scenarios 2

We generated 200 scenarios fulfilling the requirement that the exercising runtime of all the test cases must be close to double the time constraint T_{max}. We considered the following criteria:

- $\sum_{i}^{card(TC)} Time_i \cong 2 * T_{max}$,
- $card(TC) \in \{i * 10^3 | i = 1 \ldots 10\}$,
- $T_{max} \in \{i * 10^5 | i = 1 \ldots 10\}$,
- $Cost_i \in [1, 10^2],\ i = 1 \ldots card(TC)$.

The generated scenarios were divided in two categories:

- $\{Scen_{401}, \ldots, Scen_{500}\},\ Time_i \in [15, 25],\ i = 1 \ldots card(TC)$,
- $\{Scen_{501}, \ldots, Scen_{600}\},\ Time_i \in [150, 250],\ i = 1 \ldots card(TC)$.

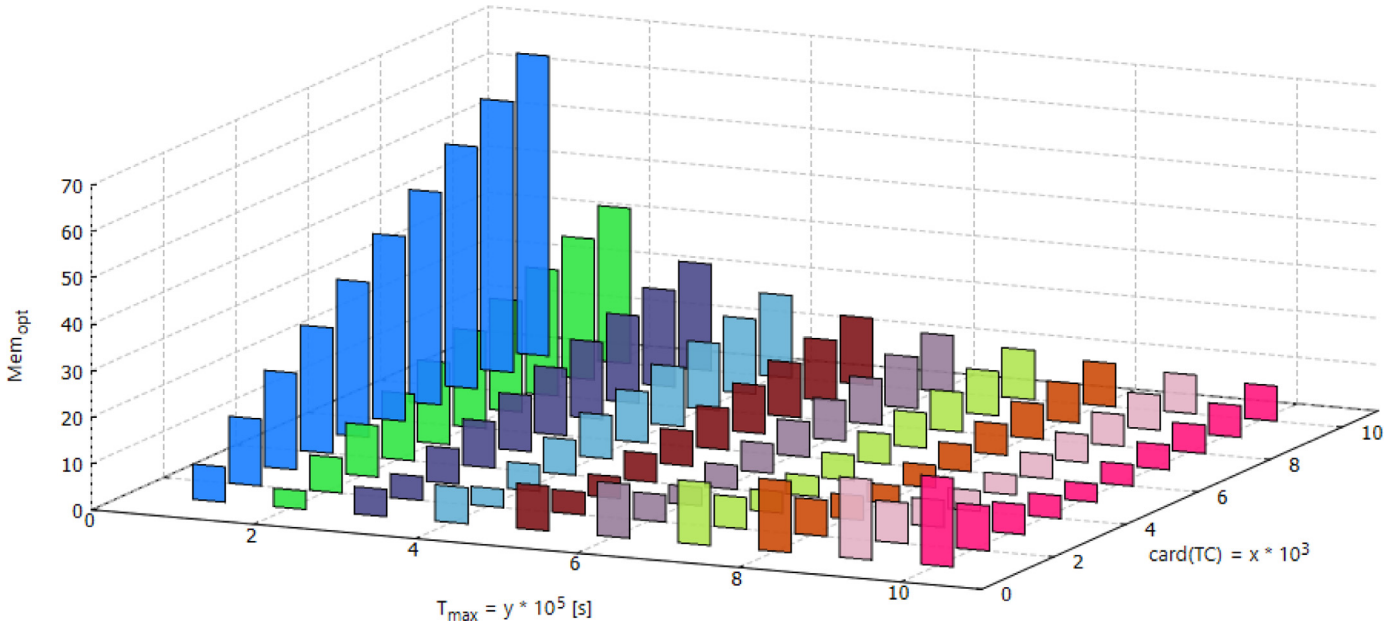

Fig. 6. Memory optimization in $\{Scen_{401}, \ldots, Scen_{500}\}$.

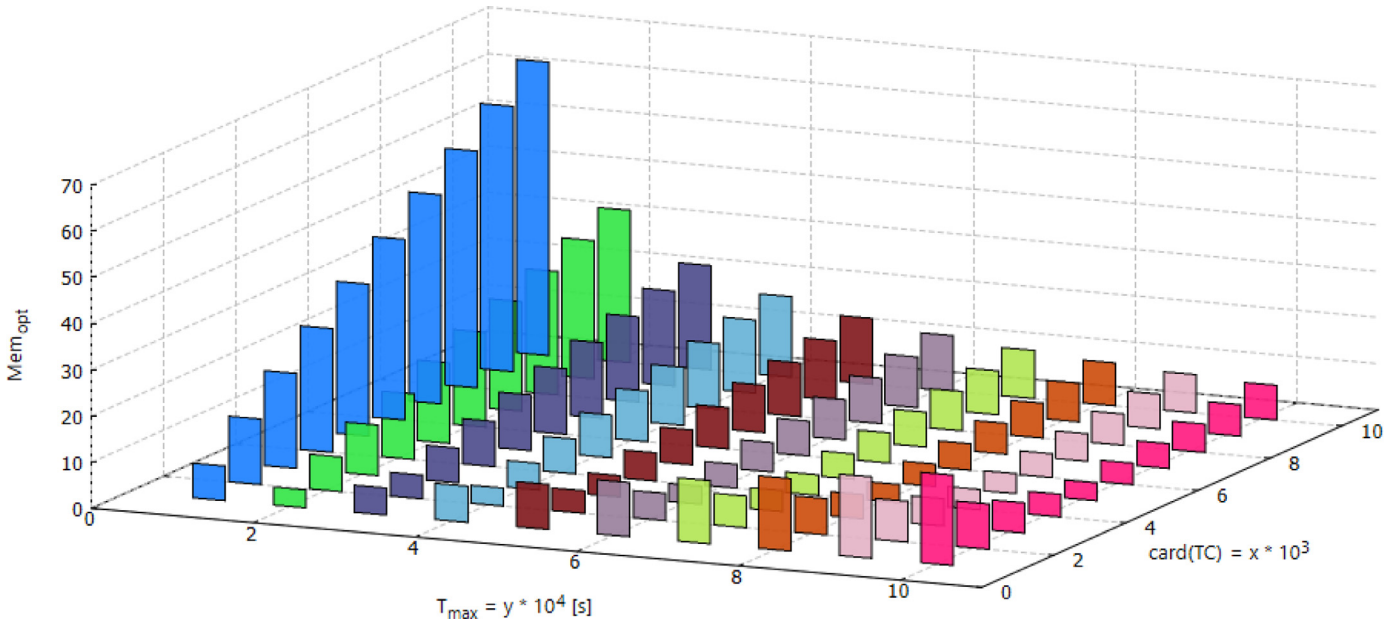

Fig. 7. Memory optimization in $\{Scen_{501}, \ldots, Scen_{600}\}$.

The results for each category of scenarios were plotted in Figs. 6 and 7. Comparing the results with the results in scenarios $\{Scen_1, \ldots, Scen_{400}\}$ we observed that even the Mem_{opt} ratio reaches values up to 60, the average ratio per group of scenarios is couple of time lower in the current scenarios $\{Scen_{401}, \ldots, Scen_{500}\}$ than the average ratio in scenarios $\{Scen_1, \ldots, Scen_{400}\}$. Also taking in consideration that scenarios $\{Scen_{101}, \ldots, Scen_{200}\}$ produce better results than scenarios $\{Scen_{401}, \ldots, Scen_{500}\}$ due to higher $\sum_{i=1}^{card(TC)} Time_i/T_{max}$ ratio, we conclude that the Mem_{opt} ratio is directly proportional with the $\sum_{i=1}^{card(TC)} Time_i/T_{max}$ ratio (Eq. (17)).

$$Mem_{opt} \propto \frac{\sum_{i=1}^{card(TC)} Time_i}{T_{max}} \tag{17}$$

4.2.3. Group of scenarios 3

We generated 400 scenarios fulfilling the requirements that the exercising runtime of all the test cases must be greater than the time constraint T_{max} and the cost values are divided in 4 categories. We considered the following criteria:

- $\sum_i^{card(TC)} Time_i \geq T_{max}$,
- $card(TC) \in \{i{*}10^3 | i = 1 \ldots 10\}$,
- $T_{max} \in \{i{*}10^5 | i = 1 \ldots 10\}$,
- $Time_i \in [10^3, 2{*}10^3],\ i = 1 \ldots card(TC)$.

The generated scenarios were divided in four categories:

- $\{Scen_{601}, \ldots, Scen_{700}\}$, $Cost_i \neq Cost_j,\ \forall i \neq j$,
- $\{Scen_{701}, \ldots, Scen_{800}\},\ Cost_i \in [10,\ 20],\ i = 1 \ldots card(TC)$,
- $\{Scen_{801}, \ldots, Scen_{900}\},\ Cost_i \in [10^2, 2{*}10^2],\ i = 1 \ldots card(TC)$,
- $\{Scen_{901}, \ldots, Scen_{1000}\},\ Cost_i \in [10^3, 2{*}10^3],\ i = 1 \ldots card(TC)$.

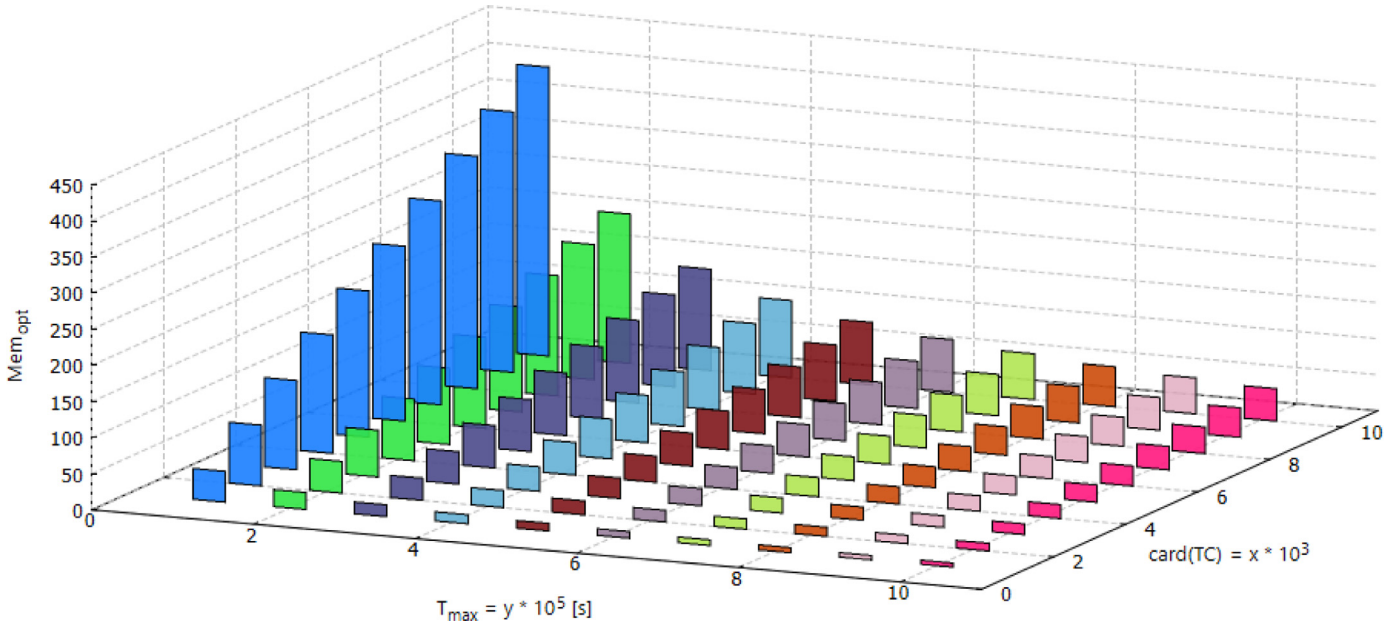

Fig. 8. Memory optimization in $\{Scen_{601}, \ldots, Scen_{1000}\}$.

In Fig. 8 we have plotted the results only for the $\{Scen_{601}, \ldots, Scen_{700}\}$ as we observed the results in the other 3 categories were almost identical with the plotted results.

Comparing the results with the scenarios $\{Scen_{101}, \ldots, Scen_{200}\}$ we conclude that the cost values do not affect the performance of PA.

Regardless of the scenarios categories, we observed that for same values of $card(TC)$, the Mem_{opt} ratio is inversely proportional with the T_{max} and for same values of T_{max}, the Mem_{opt} ratio is directly proportional with the $card(TC)$.

6. Conclusions

In this paper we proposed a novel approach in solving the test case selection-prioritization problem by employing a computer science dynamic programming technique. Moreover, we generalized the test case selection applicability by including in the selection process of multiple selection criteria, compared with other selection techniques focused on only one selection criterion. We defined the optimization problem and we employed both the objective dynamic programming method and the subjective and empiric human decision for defining the prioritization criteria. We applied the proposed solution in realistic scenarios to measure and evaluate the behavior of the proposed algorithm regarding the required memory size.

One of the important findings of this work is that computer science dynamic programming algorithms with low memory consumption are applicable in test case selection problems and suitable for medium to large projects containing large amounts of test cases. The current study proved the proposed algorithm requires about 41.5 times less memory in average than a classical dynamic programming approach. The considerable reduction of the allocable memory size required by the proposed algorithm makes possible the increasing of the number of test cases involved in the selection-prioritization problem, otherwise impossible as the allocable memory is limited.

As long as the pseudo-polynomial complexity in time is acceptable in the project's perspective, the proposed solution is very useful as the required allocable memory does not represent an impediment even in scenarios characterized by numerous test cases of the orders of millions and time constraints of the order of months. Studying the results over multiple scenarios, we observed the effectiveness in memory consumption is inversely proportional with the time constraint and directly proportional with the number of test cases. Also, the effectiveness in memory consumption is directly proportional with the cumulated exercising time of the test cases and inversely proportional with the difference between the maximum exercising time and the minimum exercising time in the set of available test cases.

Future work will target studies on the relevance of the currently proposed optimal method in comparison with approximation methods in the context of the test case selection problem.

Conflict of Interest

The authors have no affiliation with any organization with a direct or indirect financial interest in the subject matter discussed in the manuscript.

References

[1] M. Khatibsyarbini, M.A. Isa, D.N.A. Jawawi, R. Tumeng, Test case prioritization approaches in regression testing: a systematic literature review, Inf. Softw. Technol. 93 (2018) 74–93. https://doi.org/10.1016/j.infsof.2017.08.014.
[2] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Softw. Test. Verif. Reliab. (2010) 67–120. https://doi.org/10.1002/stvr.430.
[3] S. Elbaum, A.G. Malishevsky, G. Rothermel, Test case prioritization: a family of empirical studies, IEEE Trans. Softw. Eng. 28 (2) (2002) 159–182. https://doi.org/10.1109/32.988497.
[4] M. Khatibsyarbini, M.A. Isa, D.N.A. Jawawi, R. Tumeng, Test case prioritization approaches in regression testing: a systematic literature review, Inf. Softw. Technol. 93 (2018) 74–93. https://doi.org/10.1016/j.infsof.2017.08.014.
[5] R. Kazmi, D.N.A. Jawawi, R. Mohamad, I. Ghani, Effective regression test case selection, ACM Comput. Surv. 50 (2) (2017) 1–32. https://doi.org/10.1145/3057269.
[6] E.N. Narciso, M.E. Delamaro, F.D.L.D.S. Nunes, Test case selection: a systematic literature review, Int. J. Softw. Eng. Knowl. Eng. 24 (04) (2014) 653–676. https://doi.org/10.1142/s0218194014500259.
[7] O. Banias, Dynamic programming optimization algorithm applied in test case selection, 2018 International Symposium on Electronics and Telecommunications (ISETC), 2018 https://doi.org/10.1109/isetc.2018.8583984.
[8] G.M. Kapfhammer, M.L. Soffa, Using coverage effectiveness to evaluate test suite prioritizations, in: Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007 – WEASELTech '07, 2007.
[9] J.A. Parejo, A.B. Sánchez, S. Segura, A. Ruiz-Cortés, R.E. Lopez-Herrejon, A. Egyed, Multi-objective test case prioritization in highly configurable systems: a case study, J. Syst. Softw. 122 (2016) 287–310. https://doi.org/10.1016/j.jss.2016.09.045.
[10] S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, M. Liaaen, Multi-objective test prioritization in software product line testing, in: Proceedings of the 18th International Software Product Line Conference on – SPLC '14, 2014.

[11] P. Measey, Agile Foundations: Principles, Practices and Frameworks, Publisher: BCS, 2015.
[12] E. Lander, J.K. Liker, The toyota production system and art: making highly customized and creative products the Toyota way, Int. J. Prod. Res. 45 (16) (2007) 3681–3698. https://doi.org/10.1080/00207540701223519.
[13] G. Rothermel, R.H. Untch, Chengyun Chu, M.J. Harrold, Test case prioritization: an empirical study, in: Proceedings IEEE International Conference on Software Maintenance – 1999 (ICSM'99). "Software Maintenance for Business Change" (Cat. No. 99CB36360), 1999, pp. 179–188. https://doi.org/10.1109/icsm.1999.792604.
[14] S. Elbaum, G. Rothermel, S. Kanduri, A.G. Malishevsky, Selecting a cost-effective test case prioritization technique, Softw. Qual. J. 12 (3) (2004) 185–210. https://doi.org/10.1023/b:sqjo.0000034708.84524.22.
[15] J. Kasurinen, O. Taipale, K. Smolander, Test case selection and prioritization, in: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement – ESEM '10, 2010.
[16] T. Byun, V. Sharma, S. Rayadurgam, S. McCamant, M.P.E. Heimdahl, Toward Rigorous Object-Code Coverage Criteria. 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), 2017 https://doi.org/10.1109/issre.2017.33.
[17] S. Elbaum, P. Kallakuri, A. Malishevsky, G. Rothermel, S. Kanduri, Understanding the effects of changes on the cost-effectiveness of regression testing techniques, Softw. Test. Verif. Reliab. 13 (2) (2003) 65–83. https://doi.org/10.1002/stvr.263.
[18] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, T. Xie, To be optimal or not in test-case prioritization, IEEE Trans. Softw. Eng. 42 (5) (2016) 490–505. https://doi.org/10.1109/tse.2015.2496939.
[19] R. Bellman, Dynamic Programming, Dover Publications, 2003.
[20] S. Dreyfus, Richard bellman on the birth of dynamic programming, Oper. Res. 50 (1) (2002) 48–51. https://doi.org/10.1287/opre.50.1.48.17791.
[21] H. Ney, The use of a one-stage dynamic programming algorithm for connected word recognition, Read. Speech Recognit. (1990) 188–196.
[22] E.L. Lawler, A dynamic programming algorithm for preemptive scheduling of a single machine to minimize the number of late jobs, Ann. Oper. Res. 26 (1) (1990) 125–133. https://doi.org/10.1007/bf02248588.
[23] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Stein C. Introduction to Algorithms, MIT Press, 2009.
[24] T.J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. SE-2 (4) (1976) 308–320. https://doi.org/10.1109/tse.1976.233837.
[25] M. Fowler, J. Highsmith, The agile manifesto, Softw. Dev. 9 (8) (2001) 28–35.
[26] D. Pisinger, P. Toth, Knapsack Problems. Handbook of Combinatorial Optimization, Kluwer Academic Publishers, 1998, pp. 299–428 https://doi.org/10.1007/978-1-4613-0303-9_5.