STIPI: Using Search to Prioritize Test Cases Based on Multi-objectives Derived from Industrial Practice

Dipesh Pradhan[1(✉)], Shuai Wang[1], Shaukat Ali[1], Tao Yue[1,2],
and Marius Liaaen[3]

[1] Certus V&V Center, Simula Research Laboratory, Oslo, Norway
{dipesh,shuai,shaukat,tao}@simula.no
[2] University of Oslo, Oslo, Norway
[3] Cisco Systems, Oslo, Norway
marliaae@cisco.com

Abstract. The importance of cost-effectively prioritizing test cases is undeniable in automated testing practice in industry. This paper focuses on prioritizing test cases developed to test product lines of Video Conferencing Systems (VCSs) at Cisco Systems, Norway. Each test case requires setting up configurations of a set of VCSs, invoking a set of test APIs with specific inputs, and checking statuses of the VCSs under test. Based on these characteristics and available information related with test case execution (e.g., number of faults detected), we identified that the test case prioritization problem in our particular context should focus on achieving high coverage of configurations, test APIs, statuses, and high fault detection capability as quickly as possible. To solve this problem, we propose a search-based test case prioritization approach (named *STIPI*) by defining a fitness function with four objectives and integrating it with a widely applied multi-objective optimization algorithm (named Non-dominated Sorting Genetic Algorithm II). We compared *STIPI* with random search (RS), Greedy algorithm, and three approaches adapted from literature, using three real sets of test cases from Cisco with four time budgets (25 %, 50 %, 75 % and 100 %). Results show that *STIPI* significantly outperformed the selected approaches and managed to achieve better performance than RS for on average 39.9 %, 18.6 %, 32.7 % and 43.9 % for the coverage of configurations, test APIs, statuses and fault detection capability, respectively.

Keywords: Test case prioritization · Search · Configurations · Test APIs

1 Introduction

Testing is a critical activity for system or software development, through which system/software quality is ensured [1]. To improve the testing efficiency, a large number of researchers have been focusing on prioritizing test cases into an optimal execution order to achieve maximum effectiveness (e.g., fault detection capability) as quickly as possible [2–4]. In the industrial practice of automated testing, test case

prioritization is even more critical because usually there is a limited budget (e.g., time) to execute test cases, and thus executing all available test cases at a given context is infeasible [1, 5].

Our industrial partner for this work is Cisco System, Norway, who develops product lines of Video Conferencing Systems (VCSs), which enable high quality conference meetings [4, 5]. To ensure the delivery of high quality VCSs to the market, test engineers of Cisco continually develop test cases to test software of VCSs under various hardware or software configurations, statuses (i.e., states) of VCSs with dedicated test APIs. A test case is typically composed of the following parts: (1) setting up test configurations of a set of VCSs under test; (2) invoking a set of test APIs of the VCSs; and (3) checking the statuses of the VCSs after invoking the test APIs to determine the success or failure of an execution of the test case. When executing test cases, several objectives need to be achieved, i.e., covering the maximum number of possible configurations, test APIs, statuses and detecting as many faults as possible. However, given a number of available test cases, it is often infeasible to execute all of them in practice due to a limited budget of execution time (e.g., 10 h), and it is therefore important to seek an approach for prioritizing the given test cases to cover maximum number of configurations, test APIs, statuses and detect faults as quickly as possible.

To address the above-mentioned challenge, we propose a search-based test case prioritization approach named *Search-based Test case prioritization based on Incremental unique coverage and Position Impact* (*STIPI*). *STIPI* defines a fitness function with four objectives to evaluate the quality of test case prioritization solutions, i.e., Configuration Coverage (*CC*), test API Coverage (*APIC*), Status Coverage (*SC*) and Fault Detection Capability (*FDC*), and integrates the fitness function with a widely-applied multi-objective search algorithm (i.e., Non-dominated Sorting Genetic Algorithm II) [6]. Moreover, we propose two prioritization strategies when defining the fitness function in *STIPI*: (1) *Incremental Unique Coverage*, i.e., for a specific test case, we only consider the incremental unique elements (e.g., test APIs) covered by the test case as compared with the elements covered by the already prioritized test cases; and (2) *Position Impact*, i.e., a test case with a higher execution position (i.e., scheduled to be executed earlier) has more impact on the quality of a prioritization solution. Notice that both of these strategies are defined to help search to achieve high criteria (i.e., *CC*, *APIC*, *SC* and *FDC*) as quickly as possible.

To evaluate *STIPI*, we chose five approaches for the comparison: (1) Random Search (*RS*) to assess the complexity of the problem; (2) *Greedy* approach; (3) One existing approach [7] and two modified approaches from the existing literature [8, 9]. The evaluation uses in total 211 test cases from Cisco, which are divided into three sets with varying complexity. Moreover, four different time budgets are used for our evaluation, i.e., 25 %, 50 %, 75 % and 100 % (100 % refers to the total execution time of all the test cases in a given set). Notice that 12 comparisons were performed (i.e., three sets of test cases*four time budgets) for comparing *STIPI* with each approach, and thus in total 60 comparisons were conducted for the five approaches. Results show that *STIPI* significantly outperformed the selected approaches for 54 out of 60 comparisons (90 %). In addition, *STIPI* managed to achieve higher performance than RS for on

average 39.9 % (configuration coverage), 18.6 % (test API coverage), 32.7 % (status coverage), and 43.9 % (fault detection capability).

The remainder of the paper is organized as follows: Sect. 2 presents the context, a running example and motivation. *STIPI* is presented in Sect. 3 followed by experiment design (Sect. 4). Section 5 presents experiment results and overall discussion. Related work is discussed in Sect. 6, and we conclude the work in Sect. 7.

2   Context, Running Example and Motivation

Figure 1 presents a simplified context of testing VCSs (Systems Under Test (SUTs)), and Fig. 2 illustrates (partial) configuration, test API and status information for testing a VCS. First, one VCS consists of one or more configuration variables (e.g., attribute *protocol* of class VCS in Fig. 2), each of which can take two or more configuration variable values (e.g., literal *SIP* of enumeration *Protocol*). Second, a VCS holds one or more status variables defining the statuses of the VCS (e.g., *NumberofActiveCalls*), and each status variable can have two or more status variable values (e.g., *NumberofActiveCalls* taking values of 0, 1, 2, 3 and 4). Third, testing a VCS requires employing one or more test API commands (e.g., *dial*), each of which includes zero or more test API parameters (e.g., *callType* for *dial*). Each test API parameter can take two or more test API parameter values (e.g., *Video* and *Audio* for *CallType*).

Figure 3 illustrates the key steps of a test case for testing VCSs. First, a test case configures one or more VCSs by assigning values to configuration variables. For example, the test case shown in Fig. 3 configures the configuration variable *protocol* with *SIP* (Line 1). Second, a test API command is invoked with appropriate values
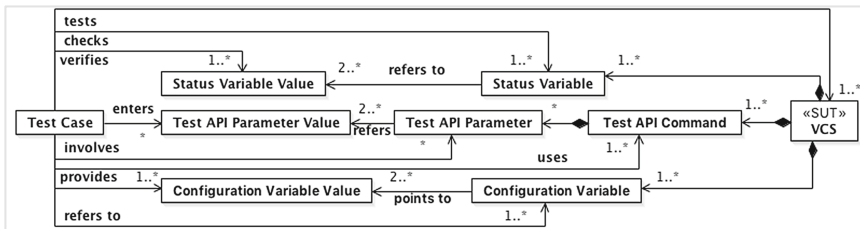

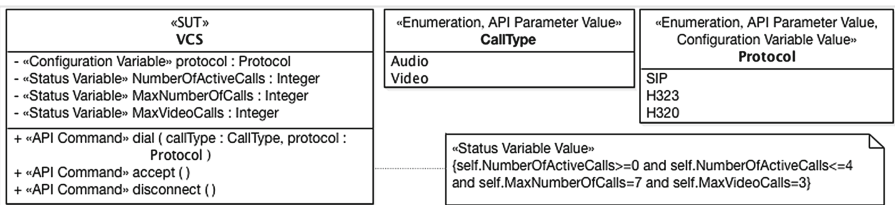**Fig. 1.** A simplified context of testing VCSs

Fig. 2. Partial configuration, status and test API information for testing a VCS

```
1. protocol = SIP   //Configure the configuration variable
2. dial(Video, SIP) //Employ test API command dial and assigning
                     values to parameters: callType and protocol
3. accept            //Employ test API command with no parameters
4. assert (NumberOfActiveCalls=1,MaxNumberOfCalls=1,
           MaxVideoCalls =1) //Check values of the status variables
5. disconnect        //Employ test API command with no parameters
6. assert(NumberofActiveCalls=0) //Check status
```

Fig. 3. An excerpt of a sanitized and simplified test case

assigned to its input parameters, if any. For example, the test case in Fig. 3 invokes the test API command *dial* consisting of the two test API parameter values: *Video* for *callType* and *SIP* for *protocol*) (Line 2). Third, the test case checks the actual statuses of VCSs. For example, the test case in Fig. 3 checks the status of the VCS to see if *NumberOfActiveCalls* equals to 1 (Line 4).

In the context of testing VCSs, test case prioritization is a critical task since it is practically infeasible to execute all the available test cases within a given time budget (e.g., 5 h). Therefore, it is essential to cover maximum configurations (i.e., configuration variables and their values), test APIs (i.e., test API commands, parameters and their values) and statuses (i.e., status variables and their values), and detect faults as quickly as possible. For instance, Table 1 lists five test cases $(T_1 \ldots T_5)$ with the information about configurations, test APIs and statuses. The test case in Fig. 3 is represented as T_1 in Table 1, which (1) sets the configuration variable *protocol* as *SIP*; (2) uses three test API commands: *dial* with two parameters (*callType*, *protocol*), *accept* and *disconnect*; and (3) checks values of three status variables (e.g., *MaxVideoCalls*).

Notice that the five test cases in Table 1 can be executed in 325 orders (i.e., $C(5, 1) \times 1! + C(5, 2) \times 2! + \ldots + C(5, 5) \times 5!$). When there is a time budget, each particular order can be considered as a prioritization solution. Given two prioritization solutions $s_1 = \{T_5, T_1, T_4, T_2, T_3\}$, $s_2 = \{T_1, T_3, T_5, T_2, T_4\}$, one can observe that s_1 is better than s_2 since the first three test cases in s_1 can cover all the configuration variables and their values, test API commands, test API parameters, test API parameter values, status variables and status variable values, while s_2 needs to execute all the five test cases to achieve the same coverage as s_1. Therefore, it is important to seek an

Table 1. Illustrating test case prioritization*

Test case	Configuration	Test API				Status		
	Protocol	Dial		Accept	Disconnect	SV_1	SV_2	SV_3
		callType	Protocol					
T_1	SIP	Video	SIP	✔	✔	0, 1	1	1
T_2	SIP	Audio	SIP	✔	✔	0, 1	1	0
T_3	SIP	Audio	SIP	✔		1	1	0
T_4	H323	Audio	H323	✔		0, 1, 2	2	0
T_5	H320	Audio	H320	✔		1	1	1

*SV_1: *NumberOfActiveCalls*, SV_2: *MaxNumberOfCalls*, SV_3: *MaxVideoCalls*.

efficient approach to find an optimal order for executing a given number of test cases to achieve high coverage of configurations, test APIs and statuses, and detect faults as quickly as possible, which forms the motivation of this work.

3   *STIPI*: Search-Based Test Case Prioritization Based on Incremental Unique Coverage and Position Impact

This section presents the problem representation (Sect. 3.1), four defined objectives, fitness function (Sect. 3.2) and solution encoding (Sect. 3.3).

3.1   Basic Notations and Problem Representation

Basic Notations. We provide the basic notations as below used throughout the paper.

$T = \{T_1, T_2 \ldots T_n\}$ represents a set of n test cases to be prioritized.

$ET = \{et_1, et_2 \ldots et_n\}$ refers to the execution time for each test case in T.

$CV = \{cv_1, cv_2 \ldots cv_{mcv}\}$ represents the configuration variables covered by T. For each cv_i, CVV_i refers to the configuration variable values: $CVV_i = \{cvv_{i1} \ldots cvv_{icvv}\}$. $mcvv$ is the total number of unique values for all the configuration variables, which can be calculated as: $mcvv = \left| \left(\bigcup_{i=1}^{mcv} CVV_i \right) \right|$.

$AC = \{ac_1, ac_2 \ldots ac_{mac}\}$ represents a set of test API commands covered by T. For each ac_i, AP_i denotes the test API parameters: $AP_i = \{ap_{i1} \ldots ap_{iap}\}$. map is the total number of unique test API parameters, calculated as: $map = |\left(\bigcup_{i=1}^{mac} AP_i \right)|$. For each ap_i, AV_i refers to the test API parameter values: $AV_i = \{av_{i1} \ldots av_{iav}\}$. mav is the total number of unique test API parameter values, i.e., $mav = |\left(\bigcup_{i=1}^{map} AV_i \right)|$.

$SV = \{sv_1, sv_2 \ldots sv_{msv}\}$ represents a set of status variables covered by T. For each sv_i, SVV_i refers to the status variable values: $SVV_i = \{svv_{i1} \ldots svv_{isvv}\}$. $msvv$ is the total number of unique status variable values, calculated as: $msvv = \left| \left(\bigcup_{i=1}^{msv} SVV_i \right) \right|$.

$Effect = \{effect_1 \ldots effect_{neffect}\}$ defines a set of effectiveness measures.

$S = \{s_1, s_2 \ldots s_{ns}\}$ represents a set of potential solutions, such that $ns = C(n, 1) \times 1! + C(n, 2) \times 2! + \ldots + C(n, n) \times n!$. Each solution s_j consists of a set of prioritized test cases in T: $s_j = \{T_{j1} \ldots T_{jn}\}$, where $T_{ji} \in T$ refers to the test case with the execution position i in the prioritized solution s_j. Note that it is possible for the maximum number of test cases in s_j (i.e., jn) to be less than the total number of test cases in T, since only a subset of T is prioritized during limited budget (e.g., time).

Problem Representation. We aim to prioritize the test cases in T in two contexts: (1) 100 % time budget and (2) less than 100 % time budget (i.e., time-aware [1]). Therefore, we formulate the test case prioritization problem as follows: (a) search a

solution s_k with nk test cases from the total number of ns solutions in S to obtain the highest effectiveness; and (b) a test case T_{jr} in a particular solution (e.g.,s_j) with a higher position p has more influence for *Effect* than the test case with a lower position q.

(1) With 100 % time budget:

$$\forall_{i=1\,to\,n\,effect}\forall_{j=1\,to\,ns}\,Effect\,(s_k,\,effect_i) \geq Effect\,(s_j, effect_i)$$
$$\lor\,effect_i(T_{jr},p) > \forall_{q \geq (p+1)}effect_i(T_{jr},q).$$

where $effect_i(T_{jr},p)$ and $effect_i(T_{jr},q)$ refer to the effectiveness measure i for a test case T_{jr} at position p and q, respectively for a particular solution s_j. $Effect(s_k,effect_i)$ and $Effect(s_j,effect_i)$ returns the effectiveness measure i for solutions s_k, s_j respectively.

(2) With a time budget tb less than 100 % time budget:

$$\forall_{i=1\,to\,neffect}\forall_{j=1\,to\,ns}\,Effect\,(s_k,\,effect_i) \geq Effect\,(s_j,\,effect_i)$$
$$\lor\sum_{l=1}^{nk} ET_l \leq tb, effect_i(T_{jr},p) > \forall_{q \geq (p+1)}effect_i(T_{jr},q).$$

3.2   Fitness Function

Recall that we aim at maximizing the overall coverage for configuration, test API and status, and detect faults as quickly as possible (Sect. 2). Therefore, we define four objective functions for the fitness function to guide the search towards finding optimal solutions, which are presented in details as below.

Maximize Configuration Coverage (*CC*). *CC* measures the overall configuration coverage of a solution s_j with jn number of test cases, which is composed of Configuration Variable Coverage (*CVC*) and Configuration Variable Values Coverage (*CVVC*). We can calculate *CVC* and *CVVC* for s_j as: $CVC_{s_j} = \frac{\sum_{i=1}^{jn} UCV_{T_{ji}} \times \frac{n-i+1}{n}}{mcv}$, $CVVC_{s_j} = \frac{\sum_{i=1}^{jn} UCVV_{T_{ji}} \times \frac{n-i+1}{n}}{mcvv}$, where mcv and $mcvv$ represent the total number of unique Configuration Variables (*CV*) and Configuration Variable Values (*CVV*) respectively covered by the total test cases in T (e.g., in Table 1 $mcvv = 3$). Moreover, we propose two prioritization strategies for calculating *CVC* and *CVVC*. The first one is *Incremental Unique Coverage*, i.e., $UCV_{T_{ji}}$ and $UCVV_{T_{ji}}$ representing the number of incremental unique *CV* and *CVV* covered by T_{ji} (Sect. 3.1). For example, in Table 1, for one test case prioritization solution $s_1 = \{T_5, T_1, T_4, T_2, T_3\}$, $UCVV_{T_5}$ is 1 since T_5 is in the first execution position and covers one *CVV* (i.e., *H320*). $UCVV_{T_1}$ and $UCVV_{T_4}$ are at the second and third position, and cover one *CVV* each (i.e., *SIP*, *H323*). However, $UCVV_{T_2}$ and $UCVV_{T_3}$ are 0, since they are already covered by $UCVV_{T_1}$. This strategy is defined since test case prioritization in our case concerns how many configurations, test APIs, and statuses can be covered rather than how many times they can be covered. The second prioritization strategy is *Position Impact*, which is calculated as $\frac{n-i+1}{n}$, where n is the total number of test cases, and i is a specific execution position in a prioritization

solution. Thus, test cases with higher execution positions have higher impact on the quality of a prioritization solution, which fits the scope of test case prioritization that aims at achieving higher criteria as quickly as possible. For instance, using this strategy, $CVVC$ for s_1 is: $CVVC_{s_1} = \frac{1 \times \frac{5}{5} + 1 \times \frac{4}{5} + 1 \times \frac{3}{5} + 0 \times \frac{2}{5} + 0 \times \frac{1}{5}}{3} = 0.8$. Moreover, CC for s_j is represented as: $CC_{s_j} = \frac{CVC_{s_j} + CVVC_{s_j}}{2}$. A higher value of CC shows a higher coverage of configuration.

Maximize Test API Coverage ($APIC$). $APIC$ measures the overall test API coverage of a solution s_j with jn number of test cases. It consists of three sub measures: Test API Command Coverage (ACC), Test API Parameter Coverage (APC), and Test API parameter Value Coverage (AVC). ACC, APC and AVC can be calculated as below:

$$ACC_{s_j} = \frac{\sum_{i=1}^{jn} UAC_{T_{ji}} \times \frac{n-i+1}{n}}{mac}, APC_{s_j} = \frac{\sum_{i=1}^{jn} UAP_{T_{ji}} \times \frac{n-i+1}{n}}{map}, AVC_{s_j}$$
$$= \frac{\sum_{i=1}^{jn} UAV_{T_{ji}} \times \frac{n-i+1}{n}}{mav}.$$

Similarly, the same two strategies (i.e., *Incremental Unique Coverage* and *Position Impact*) are applied for calculating ACC, APC and AVC, where $UAC_{T_{ji}}$, $UAP_{T_{ji}}$ and $UAV_{T_{ji}}$ denotes the number of unique test API commands (AC), test API parameters (AP), and test API parameter values (AV) respectively covered by T_{ji} (Sect. 3.1). They are measured similar as for $UCVV_T$ in $CVVC$. mac, map, and mav refer to the total number of unique AC, AP, and AV covered by the total number of test cases as explained for $mcvv$ in $CVVC$. The $APIC$ for s_j is represented as: $APIC_{s_j} = \frac{ACC_{s_j} + APC_{s_j} + AVC_{s_j}}{3}$. A higher value of $APIC$ shows a higher coverage of test APIs.

Maximize Status Coverage (SC). SC measures the total status coverage of a solution s_j. It consists of two sub measures: Status Variable Coverage (SVC) and Status Variable Value Coverage ($SVVC$), calculated as follow: $SVC_{s_j} = \frac{\sum_{i=1}^{jn} USV_{T_{ji}} \times \frac{n-i+1}{n}}{msv}$, $SVVC_{s_j} = \frac{\sum_{i=1}^{jn} USVV_{T_{ji}} \times \frac{n-i+1}{n}}{msvv}$. Similarly, $USV_{T_{ji}}$ and $USVV_{T_{ji}}$ are the number of unique Status Variables (SV) and Status Variable Values (SVV) respectively covered by T_{ji} (Sect. 3.1), which are measured similar as $UCVV_T$ in $CVVC$. msv and $msvv$ represent the total number of unique SV and SVV respectively measured similar as for $mcvv$ in $CVVC$. The SC for s_j is represented as: $SC_{s_j} = \frac{SVC_{s_j} + SVVC_{s_j}}{2}$, with a higher value indicating a higher status coverage, and therefore representing a better solution.

Maximize Fault Detection Capability (FDC). In the context of Cisco, FDC is defined as the detected number of faults for test cases in a solution s_j [4, 5, 10–12]. The FDC for a test case T_{ji} is calculated as: $FDC_{T_{ji}} = \frac{Number\ of\ times\ that\ T_{ji}\ found\ a\ fault}{Number\ of\ times\ that\ T_{ji}\ was\ executed}$. Notice that the FDC of T_{ji} is calculated based on the historical information of executing T_{ji}. For example, if tc_i was executed 10 times, and it detected fault 4 times, the FDC for tc_i is 0.4. We calculate FDC for a solution s_j as: $FDC_{s_j} = \frac{\sum_{i=1}^{jn} FDC_{T_{ji}} \times \frac{n-i+1}{n}}{mfdc}$. $FDC_{T_{ji}}$ denotes

the *FDC* for a T_{ji}, *mfdc* represents the sum of all *FDC* of test cases, and a higher value of *FDC* implies a better solution. Notice that we cannot apply the *incremental unique coverage* strategy for calculating FDC_{s_j} since the relations between faults and test cases are not known in our case (i.e., we only know whether the test cases can detect faults after executing it for a certain number of times rather than having access to the detailed faults detected).

3.3   Solution Representation

The test cases in *T* are encoded as an array $A = \{v_1, v_2 \ldots v_n\}$, where each variable v_i represents one test case in *T*, and holds a unique value from 0 to 1. We prioritize the test cases in *TS* by sorting the variables in *A* in a descending order from higher to lower, such that 1 is the highest, and 0 is the lowest order. Initially, each variable in *A* is assigned a random value between 0 and 1, and during search our approach returns solutions with optimal values for *A* guided by the fitness function defined in Sect. 3.2. In terms of time-aware test case prioritization (i.e., with a time budget less than 100 %), we pick the maximum number of test cases that fit the given time budget. For example, in Table 1 for $TS = \{T_1 \ldots T_5\}$ with *A* as $\{0.6, 0.2, 0.4, 0.9, 0.3\}$ and the execution time (recorded as minutes) as $ET = \{4, 5, 6, 4, 3\}$, the prioritized test cases are $\{T_4, T_1, T_3, T_5, T_2\}$ based on our encoding way for test case prioritization. If we have a time budget of 11 min, the first two test cases (in total 8 min for execution) are first added to the prioritized solution s_j, and there are 3 min left, which is not sufficient for executing T_3 (6 min). Thus, T_3 is not added into s_j, and the next test case is evaluated to see if the total execution time can fit the given time budget. T_5 with 3 min will be added into s_j, since the inclusion of T_5 will not make the total execution time exceed the time budget. Therefore, the new prioritized solution will be $\{T_4, T_1, T_5\}$.

Moreover, we integrate our fitness function with a widely applied multi-objective search algorithm named Non-dominated Sorting Genetic Algorithm (NSGA-II) [6, 13, 14]. The tournament selection operator [6] is applied to select individual solutions with the best fitness for inclusion into the next generation. The crossover operator is used to produce offspring solutions from the parent solutions by swapping some of the parts (e.g., test cases in our context) of the parent solutions. The mutation operator is applied to randomly change the values of one or more variables (e.g., in our context, each variable represents a test case) based on the pre-defined mutation probability, e.g., 1/(total number of test cases) in our context.

4   Empirical Study Design

4.1   Research Questions

RQ1: Is *STIPI* effective for test case prioritization as compared with *RS* (i.e., random prioritization)? We compare *STIPI* with *RS* for four time budgets: 100 % (i.e., total execution time of all the test cases in a given set), 75 %, 50 % and 25 %,

to assess the complexity of the problem such that the use of search algorithms is justified.

RQ2: Is *STIPI* effective for test case prioritization as compared with four selected approaches, in the contexts of four time budgets: 100 %, 75 %, 50 % and 25 %?

RQ2.1: Is *STIPI* effective as compared with the Greedy approach (a local search approach)?

RQ2.2: Is *STIPI* effective as compared with the approach used in [7] (named as *A1* in this paper)? Notice that we chose *A1* since it also proposed a strategy to give higher importance to test cases with higher execution positions.

RQ2.3: Is *STIPI* effective as compared with the modified version of the approach proposed in [8] (named as *A2* in this paper)? We chose *A2* since it combines the Average Percentage of Faults Detected (*APFD*) metric and NSGA-II for test case prioritization without considering time budget. We modified it by defining Average Percentage of Configuration Coverage (*APCC*), Average Percentage of test API Coverage (*APAC*) and Average Percentage of Status Coverage (*APSC*) (Sect. 4.3) for assessing the quality of prioritization solutions for configurations, test APIs and statuses.

RQ2.4: Is *STIPI* effective as compared with the modified version of the approach in [9] (named as *A3* in this paper)? We chose *A3* since (1) it combines the *ADFD* with cost (*APFD$_c$*) metric and NSGA-II for addressing time-aware test case prioritization problem. We revised *A3* by defining Average Percentage of Configuration Coverage with cost (*APCC$_c$*), Average Percentage of test API Coverage with cost (*APAC$_c$*) and Average Percentage of Status Coverage with cost (*APSC$_c$*). For illustration, we provide a formula for Average Percentage of Configuration Variable Value Coverage with cost (*APCVVC$_c$*) that is a sub-metric for *APCC$_c$* as:

$$APCVVC_c = \frac{\sum_{i=1}^{mcvv}(\sum_{k=TCVV_i}^{jn} et_k - \frac{1}{2}et_{TCVV_i})}{\sum_{k=1}^{jn} et_k \times mcvv}.$$ For a solution s_j with *jn* test cases, $TCVV_i$ is the first test case from s_j that covers CVV_i (i.e., the ith configuration variable value), *mcvv* is the total number of unique configuration variable value, and et_k is the execution time for k^{th} test case. Notice that the detailed formulas for *APCC$_c$*, *APAC$_c$* and *APSC$_c$* can be consulted in our technical report in [15].

We also compare the running time of *STIPI* with all the five chosen approaches, since *STIPI* is invoked very frequently (e.g., more than 50 times per day) in our context, i.e., the test cases require to be prioritized and executed often. Therefore, it would be practically infeasible if it takes too much time to apply *STIPI*.

4.2 Experiment Tasks

As shown in Table 2 (*Experiment Task* column), we designed two tasks (T_1, T_2) for addressing *RQ1–RQ2*. The task T_1 is designed to compare *STIPI* with *RS* for the four time budgets (i.e., 100 %, 75 %, 50 % and 25 %) and three sets of test cases (i.e., 100, 150 and 211). Similarly, the task T_2 is designed to compare *STIPI* with the other four test case prioritization approaches, which is divided into four sub-tasks for comparing *Greedy*, *A1*, *A2* and *A3*, respectively.

Table 2. Overview of the experiment design

RQ	Experiment task		# test cases	Time budget %	Evaluation metric (*EM*)	Quality indicator	Statistical test
1	T_1: *STIPI* vs. *RS*		100 150 211	100	*APCC, APAC, APSC*	–	Vargha and Delaney $\hat{A}_{12}$ Mann-Whitney U test
				25, 50, 75	$APCC_p, APAC_p, APSC_p, MFDC$	–	
2	$T_{2.1}$	*STIPI* vs. *Greedy*		100	*APCC, APAC, APSC*	–	
				25, 50, 75	$APCC_p, APAC_p, APSC_p, MFDC$	–	
	$T_{2.2}$	*STIPI* vs. *A1*		100	*APCC, APAC, APSC*	*Hypervolume (HV)*	
				25, 50, 75	$APCC_p, APAC_p, APSC_p, MFDC$		
	$T_{2.3}$	*STIPI* vs. *A2*		100	*APCC, APAC, APSC*		
				25, 50, 75	$APCC_p, APAC_p, APSC_p$		
	$T_{2.4}$	*TIPI* vs. *A3*		100	*APCC, APAC, APSC*		
				25, 50, 75	$APCC_p, APAC_p, APSC_p$		

Moreover, we employed 211 real test cases from Cisco for evaluation by dividing it into three sets with varying complexity (*#Test Cases* column in Table 2). For the first set, we used all the 211 test cases. For the second set, we used 100 random test cases from the 211 test cases. Finally, for the third set, we used the 150 test cases by choosing 111 test cases not selected in the second set (i.e., 100) and 39 random test cases from the second set. Notice that the goal for using three test case sets is to evaluate our approach with test datasets with different complexity.

4.3   Evaluation Metrics

To answer the *RQ*s, we defined in total seven *EMs* (Table 3). Six are used to assess how fast the configurations, test APIs and statuses can be covered: (1) Average Percentage Configuration Coverage (*APCC*), (2) Average Percentage test API Coverage (*APAC*), (3) Average Percentage Status Coverage (*APSC*), (4) Average Percentage Configuration Coverage that penalizes missing configuration ($APCC_p$), (5) Average Percentage test API Coverage that penalizes missing test API ($APAC_p$) and (6) Average Percentage Status Coverage with penalization for missing status ($APSC_p$). We defined *APCC, APAC* and *APSC* for test case prioritization with 100 % time budget based on the *APFD* metric [8, 16]. For example, for a solution s_j with jn test cases and total number of test cases n from T (a given number of test cases), TCV_1 is the first test case from s_j that covers CV_1 for the sub metric *APCVC* in Table 3 (Sect. 3.1). Notice that n and jn are equal when there is 100 % time budget.

When there is a limited time budget, it is possible that not all the configurations, test APIs and statuses can be covered. Therefore, we defined $APCC_p$, $APAC_p$, and $APAC_p$ to give penalty to missing configurations, test APIs, and statuses for time-aware prioritization (i.e., 25 %, 50 % and 75 % time budget) based on the variant of *APFD*

Table 3. Different metrics for evaluating the approaches*

EC	Time budget %	EM	Sub metric		Formula
			Name	Formula	
Con	100	APCC	APCVC	$1 - \frac{TCV_1 + TCV_2 + ... + TCV_{mcv}}{n \times mcv} + \frac{1}{2n}$	$APCC = \frac{APCVC + APCVVC}{2}$
			APCVVC	$1 - \frac{TCVV_1 + TCVV_2 + ... + TCVV_{mcvv}}{n \times mcvv} + \frac{1}{2n}$	
	25 50 75	$APCC_p$	$APCVC_p$	$1 - \frac{\sum_{cv=1}^{mcv} reveal(cv,s_j)}{jn \times mcv} + \frac{1}{2jn}$	$APCC_p = \frac{APCVC_p + APCVVC_p}{2}$
			$APCVVC_p$	$1 - \frac{\sum_{cvv=1}^{mcvv} reveal(cvv,s_j)}{jn \times mcvv} + \frac{1}{2jn}$	
API	100	APAC	APACC	$1 - \frac{TAC_1 + TAC_2 + ... + TAC_{mac}}{n \times mac} + \frac{1}{2n}$	$APAC = \frac{APACC + APAPC + APAVC}{3}$
			APAPC	$1 - \frac{TAP_1 + TAP_2 + ... + TAP_{map}}{n \times map} + \frac{1}{2n}$	
			APAVC	$1 - \frac{TAV_1 + TAV_2 + ... + TAV_{mav}}{n \times mav} + \frac{1}{2n}$	
	25 50 75	$APAC_p$	$APACC_p$	$1 - \frac{\sum_{ac=1}^{mac} reveal(ac,s_j)}{jn \times mac} + \frac{1}{2jn}$	$APAC_p = \frac{APACC_p + APAPC_p + APAVC_p}{3}$
			$APAPC_p$	$1 - \frac{\sum_{ap=1}^{map} reveal(ap,s_j)}{jn \times map} + \frac{1}{2jn}$	
			$APAVC_p$	$1 - \frac{\sum_{av=1}^{mav} reveal(av,s_j)}{jn \times mav} + \frac{1}{2jn}$	
Stat	100	APSC	APSVC	$1 - \frac{TSV_1 + TSV_2 + ... + TSV_{msv}}{n \times msv} + \frac{1}{2n}$	$APSC = \frac{APSVC + APSVVC}{2}$
			APSVVC	$1 - \frac{TSVV_1 + TSVV_2 + ... + TSVV_{msvv}}{n \times msvv} + \frac{1}{2n}$	
	25 50 75	$APSC_p$	$APSVC_p$	$1 - \frac{\sum_{sv=1}^{msv} reveal(sv,s_j)}{jn \times msv} + \frac{1}{2jn}$	$APSC_p = \frac{APSVC_p + APSVVC_p}{2}$
			$APSVVC_p$	$1 - \frac{\sum_{svv=1}^{msvv} reveal(svv,s_j)}{jn \times msvv} + \frac{1}{2jn}$	
FDC	25,50,75	MFDC	-	-	$MFDC = \frac{\sum_{j=1}^{jn} FDC_{T_j}}{\sum_{k=1}^{n} FDC_{T_k}} \times 100\%$

*EC: Evaluation Criteria, Con: Configuration, API: Test API, Stat: Status.

metric used for time-aware prioritization [1, 16]. For example, for a solution s_j with jn test cases $reveal(cv, s_j)$ gives the test case from s_j that covers cv for $APCVC_p$ in Table 3. If s_j does not contain a test case that covers cv, $reveal(cv, s_j) = jn + 1$. Notice that in our context, we only have information about how many times in a given period (e.g., a week) a test case was successful in finding faults. Therefore, it is not possible to use the *APFD* metric to evaluate *FDC*. Hence, we defined a metric: Measured Fault Detection Capability (*MFDC*) to measure the percentage of fault detected for time budget of 25 %, 50 % and 75 %.

4.4   Quality Indicator, Statistical Tests and Parameter Settings

When comparing the overall performance of multi-objective search algorithms (e.g., NSGA-II [6]), it is common to apply quality indicators such as hypervolume (*HV*). Following the guideline in [10], we employ *HV* based on the defined *EM*s to address *RQ2.2–RQ2.4* (i.e., tasks $T_{2.2}$–$T_{2.4}$ in Table 2). *HV* calculates the volume in the objective space covered by members of a non-dominated set of solutions (i.e., Pareto front) produced by search algorithms for measuring both convergence and diversity [17]. A higher value of *HV* indicates a better performance of the algorithm.

The Vargha and Delaney $\hat{A}_{12}$ statistics [18] and Mann-Whitney U test are used to compare the EMs (T_1 and T_2), and HV ($T_{2.2}$–$T_{2.4}$), as shown in Table 2 by following the guidelines in [19]. The Vargha and Delaney $\hat{A}_{12}$ statistics is a non-parametric effect size measure, and Mann-Whitney U test tells if results are statistically significant [20]. For two algorithms A and B, A has better performance than B if $\hat{A}_{12}$ is greater than 0.5, and the difference is significant if p-value is less than 0.05.

Notice that *STIPI, A1, A2* and *A3* are all combined with NSGA-II. Since tuning parameters to different settings might result in different performance of search algorithms, standard settings are recommended [19]. We used standard settings (i.e., population size = 100, crossover rate = 0.9, mutation rate = 1/(number of test cases)) as implemented in jMetal [21]. The search process is terminated when the fitness function has been evaluated for 50,000 times. Since *A2* does not support prioritization with a time budget, we collect the maximum number of test cases that can fit a given time budget.

5   Results, Analyses and Discussion

5.1   *RQ1*: Sanity Check (*STIPI* vs. *RS*)

Results in Tables 4 and 5 show that on average *STIPI* is higher than *RS* for all the *EM*s across the three sets of test cases. Moreover, for the three test sets using four time budgets, *STIPI* managed to achieve higher performance than RS for on average 39.9 % (configuration coverage), 18.6 % (test API coverage), 32.7 % (status coverage), and 43.9 % (*FDC*). In addition, results of the Vargha and Delaney statistics and the Mann Whitney U test show that *STIPI* significantly outperformed *RS* for all the *Em*s since all the values of $\hat{A}_{12}$ are greater than 0.5 and all the p-values are less than 0.05.

Table 4.   Average values of the *EM*s with 100 % and 75 % time budget*

# T	100 % time budget							75 % time budget						
	EM	RS	Gr	A1	A2	A3	STI	EM	RS	Gr	A1	A2	A3	STI
100	CC	0.7	0.76	0.75	0.77	0.75	0.77	CC_p	0.63	0.71	0.73	0.74	0.73	0.74
150		0.68	0.84	0.8	0.79	0.75	0.79		0.60	0.81	0.69	0.72	0.73	0.77
211		0.74	0.83	0.83	0.85	0.81	0.85		0.67	0.76	0.79	0.80	0.79	0.81
100	AC	0.83	0.74	0.85	0.85	0.84	0.86	AC_p	0.78	0.70	0.83	0.82	0.84	0.83
150		0.78	0.64	0.83	0.86	0.85	0.86		0.72	0.57	0.75	0.81	0.83	0.84
211		0.82	0.67	0.85	0.89	0.89	0.89		0.77	0.56	0.83	0.87	0.87	0.88
100	SC	0.73	0.65	0.76	0.82	0.76	0.82	SC_p	0.67	0.60	0.73	0.79	0.79	0.81
150		0.74	0.62	0.8	0.85	0.83	0.85		0.68	0.56	0.71	0.80	0.81	0.83
211		0.78	0.64	0.79	0.85	0.82	0.85		0.72	0.56	0.79	0.84	0.85	0.86
100	-	-	-	-	-	-	-	MF	0.78	0.79	0.91	-	-	0.89
150	-	-	-	-	-	-	-		0.79	0.80	0.70	-	-	0.87
211	-	-	-	-	-	-	-		0.77	0.63	0.91	-	-	0.90

*T: Test Case, Gr: *Greedy*, CC: APCC, AC: APAC, SC: APSC, CC_p: $APCC_p$, AC_p: APACp, SC_p: APSCp, MF: MFDC, STI: STIPI.

Table 5. Average values of the *EM*s with 25 % and 50 % time budget*

EM	# T	25 % time budget						50 % time budget					
		RS	Gr	A1	A2	A3	STIPI	RS	Gr	A1	A2	A3	STIPI
$APCC_p$	100	0.37	0.30	0.55	0.51	0.62	0.66	0.52	0.65	0.65	0.67	0.70	0.73
	150	0.35	0.59	0.52	0.45	0.66	0.71	0.50	0.81	0.74	0.63	0.72	0.74
	211	0.42	0.43	0.63	0.56	0.69	0.71	0.52	0.53	0.65	0.67	0.70	0.73
$APAC_p$	100	0.56	0.26	0.70	0.61	0.74	0.70	0.71	0.61	0.79	0.77	0.81	0.81
	150	0.50	0.35	0.59	0.55	0.74	0.75	0.64	0.54	0.76	0.74	0.81	0.82
	211	0.58	0.33	0.71	0.65	0.77	0.75	0.71	0.52	0.79	0.81	0.85	0.85
$APSC_p$	100	0.42	0.14	0.59	0.55	0.70	0.66	0.57	0.51	0.68	0.72	0.76	0.76
	150	0.44	0.33	0.54	0.53	0.73	0.74	0.52	0.53	0.65	0.67	0.70	0.73
	211	0.48	0.24	0.66	0.62	0.78	0.77	0.63	0.52	0.74	0.78	0.84	0.85
MFDC	100	0.30	0.06	0.55	-	-	0.50	0.54	0.45	0.77	-	-	0.78
	150	0.30	0.19	0.40	-	-	0.63	0.55	0.74	0.75	-	-	0.76
	211	0.29	0.09	0.52	-	-	0.44	0.53	0.48	0.75	-	-	0.76

5.2   *RQ2*: Comparison with the Selected Approaches

We compared *STIPI* with *Greedy*, *A1*, *A2* and *A3* using the statistical tests (Vargha and Delaney statistics and Mann Whitney U test) for the four time budgets (25 %, 50 %, 75 % and 100 %), and the three sets of test cases (i.e., 100, 150, 211). Results are summarized in Fig. 4. For example, the first bar (i.e., *Gr*) in Fig. 4 refers to the comparison between *STIPI* and *Greedy* for the 100 % time budget where A = *STIPI* and B = *Greedy*. A > B means the percentage of *EM*s for which *STIPI* has significantly better performance than *Greedy* ($\hat{A}_{12} > 0.5 \,\&\&\, p < 0.05$), A < B means the opposite ($\hat{A}_{12} < 0.5 \,\&\&\, p < 0.05$), and A = B implies there is no significant difference in performance ($p \geq 0.05$).

RQ2.1 (***STIPI* vs. *Greedy*).** From Tables 4 and 5, we can observe that the average values of *STIPI* are higher than *Greedy* for 93.3 % (42/45)[1] *EM*s across the three sets of test cases with the four time budgets. Moreover, from Fig. 4, we can observe *STIPI* performed significantly better than *Greedy* for an average of 93.1 % for the four time budgets (i.e., 88.9 % for 100 %, 91.7 % for 75 %, 91.7 % for 50 %, and 100 % for 25 % time budget). Detailed results are available in [15].

RQ2.2 (***STIPI* vs. *A1*).** Based on Tables 4 and 5, we can see that *STIPI* has a higher average value than *A1* for 82.2 % (37/45) *EM*s, and *STIPI* performed significantly better than *A1* for an average of 76.4 % *EM*s across the four time budgets, while there was no difference in performance for 14.6 % from Fig. 4. Figure 5 shows that for *HV*, *STIPI* outperformed *A1* for all the three sets of test cases with the four time budgets, and such better results are statistically significant. Detailed results are in [15].

---

[1] An *EM* has one average value for one set of test case with one time budget (Tables 4 and 5). Thus, for 100 % time budget with 3 *EMs* there are 9 values, and 45 average values for 4 time budgets and 4 EMs for other 3 time budgets.
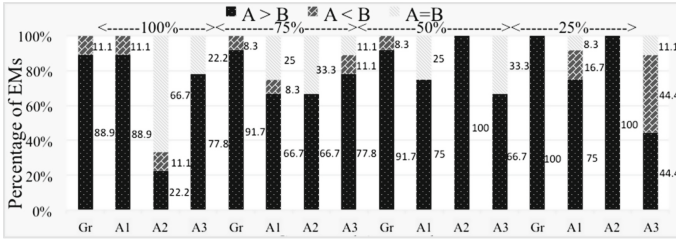
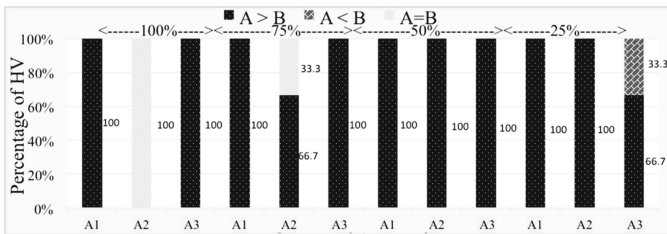**Fig. 4.** Results of comparing *STIPI* with *Greedy*, *A1*, *A2* and *A3* for *EM*s

Fig. 5. Results of comparing *STIPI* with *A1*, *A2* and *A3* for *HV*

RQ2.3 (STIPI vs. A2). *RQ2.3* is designed to compare *STIPI* with the approach *A2* (Sect. 4.1). Table 4 shows that the two approaches had similar average for *EM*s with 100 % time budget. Moreover, for 100 % time budget, there was no significant difference in the performance between *STIPI* and *A2* in terms of *EM*s and *HV* (Figs. 4 and 5). However, when considering the time budgets of 25 %, 50 % and 75 %, *STIPI* had a higher performance for 96.3 % (26/27) *EM*s (Tables 4 and 5). Furthermore, the statistical tests in Figs. 4 and 5 show that *STIPI* significantly outperformed *A2* for an average of 88.9 % *EM*s and *HV* values across the three time budgets (25 %, 50 %, 75 %), while there was no significant difference for 11.1 %.

RQ2.4 (STIPI vs. A3). Based on the results (Tables 4 and 5), *STIPI* held a higher average values for 75 % (27/36) *EM* values for the four time budgets and three sets of test cases. For 100 %, 75 %, and 50 %, we can observe from Fig. 4 that *STIPI* performed significantly better than *A3* for an average of 74.1 % *EM*s, while there was no significant difference for 22.2 %. For the 25 % time budget, there was no statistically significant difference in terms of *EM*s for *STIPI* and *A3*. However, when comparing the *HV* values, *STIPI* significantly outperformed *A3* for an average of 91.7 % across the four time budgets and three sets of test cases.

Notice that 12 comparisons were performed when comparing *STIPI* with each of the five selected approaches (i.e., three test case sets * four time budgets), and thus in total 60 comparisons were conducted. Based on the results, we can observe that *STIPI* significantly outperformed the five selected approaches for 54 out of 60 comparisons (90 %), which indicate that *STIPI* has a good capability for solving our test case prioritization problem. In addition, *STIPI* took an average time of 36.5, 51.6 and 82 s (secs) for the three sets of test cases. The average running time for the five chosen

approaches are: (1) RS: 18, 24.7 and 33.2 s; (2) *Greedy*: 42, 48 and 54 ms; (3) *A1*: 35.7, 42.8 and 65.5 s; (4) *A2*: 35.2, 42.2 and 55.4 s; and (5) *A3*: 8.9, 33.4 and 41.2 s. Notice that there is no practical difference in terms of the running time for the approaches except *Greedy*, however the performance of *Greedy* is significantly worse than *STIPI* (Sect. 5.2), and thus *Greedy* cannot be employed to solve our test case prioritization problem. In addition, based on the domain knowledge of VCS testing, the running time in seconds is acceptable when deployed in practice.

5.3   Overall Discussion

For *RQ1*, we observed that *STIPI* performed significantly better than *RS* for all the *EM*s with the three sets of test cases under the four time budgets. Such an observation reveals that solving our test case prioritization problem is not trivial, which requires an efficient approach. As for *RQ2*, we compared *STIPI* with *Greedy*, *A1*, *A2* and *A3* (Sect. 4.1). Results show that *STIPI* performed significantly better than *Greedy*. This can be explained that *Greedy* is a local search algorithm that may get stuck in a local space during the search process, while *STIPI* employs mutation operator (Sect. 4.4) to explore the whole search space towards finding optimal solutions. In addition, *Greedy* converted our multi-objective optimization problem into a single-objective optimization problem by assigning weights to each objective, which may lose many other optimal solutions that hold the same quality [22], while *STIPI* (integrating NSGA-II) produces a set of non-dominated solutions (i.e., solutions with equivalent quality).

When comparing *STIPI* with *A1*, *A2* and *A3*, the results of *RQ2* showed that *STIPI* performed significantly better than *A1*, *A2* and *A3* by 83.3 % (30/36). Overall *STIPI* outperformed the five selected approaches for 90 % (54/60) comparisons. That might be due to two main reasons: (1) *STIPI* considers the coverage of incremental unique elements (e.g., test API commands) when evaluating the prioritization solutions, i.e., only the incremental unique elements covered by a certain test case are taken into account as compared with the already prioritized test cases; and (2) *STIPI* provides the test cases with higher execution positions more influence on the quality of a given prioritization solution. Furthermore, *A2* and *A3* usually work under the assumption that the relations between detected faults and test cases are known beforehand, which is sometimes not the situation in practice, e.g., in our case, we are only aware how many execution times a test case can detect faults rather than having access to the detailed faults detected. However, *STIPI* defined *FDC* to measure the fault detection capability (Sect. 3.2) without knowing the detailed relations between faults and test cases, which may be applicable to the similar other contexts when the detailed faults cannot be accessed. It is worth mentioning that the current practice of Cisco do not have an efficient approach for test case prioritization, and thus we are working on deploying our approach in their current practice for further strengthening *STIPI*.

5.4   Threats to Validity

The *internal validity* threat arises due to using search algorithms with only one configuration setting for its parameters as we did in our experiment [23]. However, we

used the default parameter setting from the literature [24], and based on our previous experience [5, 10], good performance can be achieved for various search algorithms with the default setting. To mitigate the *construct validity* threat, we used the same stopping criteria (50,000 fitness evaluations) for finding the optimal solutions. To avoid *conclusion validity threat* due to the random variations in the search algorithms, we repeated the experiments 10 times to reduce the possibility that the results were obtained by chance. Following the guidelines of reporting the results for randomized algorithms [19], we employed the Vargha and Delaney test as the effect size measure and Mann-Whitney test to determine the statistical significance of results. First *external validity* threat is that one may argue the comparison performed only included RS, *Greedy*, one existing approach and two modified versions of the existing approaches, which may not be sufficient. Notice that we discussed and justified why we chose these approaches in Sect. 4.1, and it is also possible to compare our approach with other existing approaches, which requires further investigation as the next step. Second *external validity* threat is due to the fact that we only performed the evaluation using one industrial case study. We need to mention that we conducted the experiment using three sets of test cases with four distinct time budgets based on the domain knowledge of VCS testing.

6   Related Work

In the last several decades, test case prioritization has attracted a lot of attention and considerable amount of work has been done [1–3, 8]. Several survey papers [25, 26] present results that compare existing test case prioritization techniques from different aspects, e.g., based on coverage criteria. Followed by the aspects presented in [25], we summarize the related work close to our approach and highlight the key differences from the following three aspects: coverage criteria, search-based prioritization techniques (which is related with our approach) and evaluation metrics.

Coverage Criteria. Existing works defined a number of coverage criteria for evaluating the quality of prioritization solutions [2, 3, 26] such as branch coverage and statement coverage, function coverage and function-level fault exposing potential, block coverage, modified condition/decision coverage, transition coverage and round trip coverage. As compared with the state-of-the-art, we proposed three new coverage criteria driven by the industrial problem (Sect. 3.2): (1) Configuration coverage (*CC*); (2) Test API coverage (*APIC*) and (3) Status coverage (*SC*).

Search-Based Prioritization Techniques. Search-based techniques have been widely applied for addressing test case prioritization problem [3–5, 10]. For instance, Li et al. [3] defined a fitness function with three objectives (i.e., Block, Decision and Statement Coverage) and integrated the fitness function with hill climbing and GA for test case prioritization. Arrieta et al. [7] proposed to prioritize test cases by defining a two-objective fitness function (i.e., test case execution time and fault detection capability) and evaluated the performance of several search algorithms. The authors of [7] also proposed a strategy to give higher importance to test cases with higher positions (to be executed earlier). A number of research papers have focused on addressing the

test case prioritization problem within a limited budget (e.g., time and test resource) using search-based approaches. For instance, Walcott et al. [1] proposed to combine selection (of a subset of test cases) and prioritization (of the selected test cases) for prioritizing test cases within a limited time budget. Different weights are assigned to the selection part and prioritization part when defining the fitness function followed by solving the problem with GA. Wang et al. [5] focused on the test case prioritization within a given limited test resource budget (i.e., hardware, which is different as compared with the time budget used in this work) and defined four cost-effectiveness measures (e.g., test resource usage), and evaluated several search algorithms (e.g., NSGA-II).

As compared with the existing works, our approach (i.e., *STIPI*) defines a fitness function that considers configurations, test APIs and statuses, which were not addressed in the current literature. When defining the fitness function, *STIPI* proposed two strategies, which include (1) only considering the unique elements (e.g., configurations) achieved; and (2) taking the impact of test case execution orders on the quality of prioritization solutions into account, which is not the case in the existing works.

Evaluation Metrics (*EM*s). *APFD* is widely used in the literature as an *EM* [2, 3, 8, 16]. Moreover, the modified version of *APFD* (i.e., $APFD_p$) using time penalty [1, 16] is usually applied for test case prioritization with a time budget. Other metrics were also defined and applied as *EM*s [9, 26] such as Average Severity of Faults Detected, Total Percentage of Faults Detected and Average Percentage of Faults Detected per Cost ($APFD_c$). As compared with the existing *EM*s, we defined in total six new *EM*s driven by our industrial problem for configurations, test APIs and statuses (Table 3), which include: (1) *APCC*, *APAC*, and *APSC*, inspired by *APFD*, when there is 100 % time budget; and (2) $APCC_p$, $APAC_p$, and $APSC_p$ inspired by $APFD_p$, when there is a limited time budget (e.g., 25 % time budget). Furthermore, we defined the seventh *EM* (*MFDC*) to assess to what extent faults can be detected when the time budget is less than 100 % (Table 3). To the best of our knowledge, there is no existing work that applies these seven *EM*s for assessing the quality of test case prioritization solutions.

7 Conclusion and Future Work

Driven by our industrial problem, we proposed a multi-objective search-based test case prioritization approach named *STIPI* for covering maximum number of configurations, test APIs, statuses, and achieving high fault detection capability as quickly as possible. We compared *STIPI* with five test case prioritization approaches using three sets of test cases with four time budgets. The results show that *STIPI* performed significantly better than the chosen approaches for 90 % of the cases. *STIPI* managed to achieve a higher performance than random search for on average 39.9 % (configuration coverage), 18.6 % (test API coverage), 32.7 % (status coverage) and 43.9 % (*FDC*). In the future, we plan to compare *STIPI* with more prioritization approaches from the literature using additional case studies with larger scale to further generalize the results.

References

1. Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S.: Timeaware test suite prioritization. In: Proceedings of 2006 International Symposium on Software Testing and Analysis, pp. 1–12 (2006)
2. Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Test case prioritization: an empirical study. In: Proceedings of International Conference on Software Maintenance (ICSM 1999), pp. 179–188 (1999)
3. Li, Z., Harman, M., Hierons, R.M.: Search algorithms for regression test case prioritization. IEEE Trans. Softw. Eng. (TSE) **33**, 225–237 (2007)
4. Wang, S., Buchmann, D., Ali, S., Gotlieb, A., Pradhan, D., Liaaen, M.: Multi-objective test prioritization in software product line testing: an industrial case study. In: International Software Product Line Conference, pp. 32–41 (2014)
5. Wang, S., Ali, S., Yue, T., Bakkeli, Ø., Liaaen, M.: Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search. In: ICSE, pp. 182–191 (2016)
6. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. TSE **6**, 182–197 (2002)
7. Arrieta, A., Wang, S., Sagardui, G., Etxeberria, L.: Test case prioritization of configurable cyber-physical systems with weight-based search algorithms. In: Genetic and Evolutionary Computation (GECCO), pp. 1053–1060 (2016)
8. Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Prioritizing test cases for regression testing. TSE **27**, 929–948 (2001)
9. Elbaum, S., Malishevsky, A., Rothermel, G.: Incorporating varying test costs and fault severities into test case prioritization. In: Proceedings of International Conference on Software Engineering (ICSE), pp. 329–338 (2001)
10. Wang, S., Ali, S., Yue, T., Li, Y., Liaaen, M.: A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering. In: ICSE, pp. 631–642 (2016)
11. Wang, S., Ali, S., Gotlieb, A.: Cost-effective test suite minimization in product lines using search techniques. J. Syst. Softw. **103**, 370–391 (2015)
12. Wang, S., Ali, S., Gotlieb, A.: Minimizing test suites in software product lines using weight-based genetic algorithms. In: Proceedings of 15th Annual Conference on Genetic and Evolutionary Computation, pp. 1493–1500 (2013)
13. Sarro, F., Petrozziello, A., Harman, M.: Multi-objective software effort estimation. In: ICSE, pp. 619–630 (2016)
14. Wang, S., Ali, S., Yue, T., Liaaen, M.: UPMOA: an improved search algorithm to support user-preference multi-objective optimization. In: International Symposium on Software Reliability Engineering (ISSRE), pp. 393–404 (2015)
15. Technical report (2016-06): https://www.simula.no/publications/stipi-using-search-prioritize-test-cases-based-multi-objectives-derived-industrial

16. Lu, Y., Lou, Y., Cheng, S., Zhang, L., Hao, D., Zhou, Y., Zhang, L.: How does regression test prioritization perform in real-world software evolution? In: Proceedings of 38th ICSE, pp. 535–546 (2016)
17. Nebro, A.J., Luna, F., Alba, E., Dorronsoro, B., Durillo, J.J., Beham, A.: AbYSS: adapting scatter search to multiobjective optimization. IEEE Trans. Evol. Comput. **12**, 439–457 (2008)
18. Vargha, A., Delaney, H.D.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J. Educ. Behav. Stat. **25**, 101–132 (2000)
19. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: 33rd International Conference on Software Engineering (ICSE), pp. 1–10 (2011)
20. Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. Ann. Math. Stat. **18**, 50–60 (1947)
21. Durillo, J.J., Nebro, A.J.: jMetal: a Java framework for multi-objective optimization. Adv. Eng. Softw. **42**, 760–771 (2011)
22. Konak, A., Coit, D.W., Smith, A.E.: Multi-objective optimization using genetic algorithms: a tutorial. Reliab. Eng. Syst. Safety **91**, 992–1007 (2006)
23. De Oliveira Barros, M., Neto, A.: Threats to validity in search-based software engineering empirical studies. Technical report 6, UNIRIO-Universidade Federal do Estado do Rio de Janeiro (2011)
24. Arcuri, A., Fraser, G.: On parameter tuning in search based software engineering. In: Cohen, M.B., Ó Cinnéide, M. (eds.) SSBSE 2011. LNCS, vol. 6956, pp. 33–47. Springer, Heidelberg (2011)
25. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. **22**, 67–120 (2012)
26. Catal, C., Mishra, D.: Test case prioritization: a systematic mapping study. Softw. Qual. J. **21**, 445–478 (2013)