The Journal of Systems & Software 172 (2021) 110850

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Hansie: Hybrid and consensus regression test prioritization

Shouvick Mondal *, Rupesh Nasre

Department of Computer Science and Engineering, IIT Madras, Chennai 600036, India

ARTICLE INFO

Article history:
Received 26 December 2019
Received in revised form 14 September 2020
Accepted 21 October 2020
Available online 27 October 2020

Keywords:
Regression test prioritization
Priority awareness
Hybridization
Consensus
Permutation distances

ABSTRACT

Traditionally, given a test-suite and the underlying system-under-test, existing test-case prioritization heuristics report a permutation of the original test-suite that is seemingly best according to their criteria. However, we observe that a single heuristic does not perform optimally in all possible scenarios, given the diverse nature of software and its changes. Hence, multiple individual heuristics exhibit effectiveness differently. Interestingly, together, the heuristics bear the potential of improving the overall regression test selection across scenarios. In this paper, we pose the test-case prioritization as a rank aggregation problem from social choice theory. Our solution approach, named Hansie, is two-flavored: one involving priority-aware hybridization, and the other involving priority-blind computation of a consensus ordering from individual prioritizations. To speed-up test-execution, Hansie executes the aggregated test-case orderings in a parallel multi-processed manner leveraging regular windows in the absence of ties, and irregular windows in the presence of ties. We show the benefit of test-execution after prioritization and introduce a cost-cognizant metric (EPL) for quantifying overall timeline latency due to load-imbalance arising from uniform or non-uniform parallelization windows. We evaluate Hansie on 20 open-source subjects totaling 287,530 lines of source code, 69,305 test-cases, and with parallelization support of up to 40 logical CPUs.

1. Introduction

Software regression testing has been traditionally categorized into three types (Yoo and Harman, 2012): (i) test-selection, (ii) test-suite reduction (or minimization), and (iii) test-prioritization. Test-selection executes only those test-cases that traverse the modified parts of the system-under-test (SUT). While selection of regression tests is temporary respecting the changes introduced, test-suite reduction permanently removes test-cases. The latter approach has been sub-divided into adequate and inadequate approaches. Adequate reduction retains fault detection capability of the original test-suite by enforcing full coverage of requirements, for instance code-coverage, whereas inadequate approaches introduce some loss in fault-detection capability by satisfying subset of requirements.

It is to be noted that two test-cases t_1 and t_2 may have exactly the same code-coverage, even if their constituent input values may be different. Most approaches deal with identification of a smallest subset of test-cases, criteria being coverage of every code element at least once. In this situation, exactly one of t_1 and t_2, say t_1 is selected. Although t_1 exercises a modified code element and its outcome is *pass*, it is still possible that t_2 (not selected) reports *failure*. This is due to the fact that exact outcome (*pass* or *fail*) of a test-case is unknown before execution. For instance, consider the change $\{f(x) = 2 + x\} \rightarrow \{f'(x) = 2 * x\}$ for a numerical program. If $\{t_1(2), t_2(4)\}$ constitutes the full test-suite, and t_1 with input value 2 is the only regression test-case, then no failure is detected, as functionally $f(2) = f'(2) = 4$ is still intact. Failure (functional discrepancy) is observed only when at least t_2 with value 4 is executed.

In the absence of input-independent test-suite minimization (reduction), a careful coverage-based selection of regression test-cases is necessary. Thus, adequacy coupled with essential test-redundancy (Marijan and Liaaen, 2018) jointly play a major role in the safety of test-suite reduction.

The third approach to regression testing performs criteria-driven test-suite prioritization such that potential fault-revealing test-cases are scheduled earlier in the execution time-line. This ensures that even if testing is terminated (due to a limited time budget[1]) after having executed some test-cases, the most important code sections have indeed been exercised. This approach forms the underlying theme of our study from the viewpoint of employing individual, hybrid, and consensus test-case prioritization approaches.

* Corresponding author.
E-mail addresses: shouvick@cse.iitm.ac.in (S. Mondal), rupesh@cse.iitm.ac.in (R. Nasre).

[1] Tight release schedules often demand quick testing of the most important parts of the code. Sometimes, the most number of test-cases are expected to be run prior to a release — demanding a different prioritization order than coverage.

https://doi.org/10.1016/j.jss.2020.110850
0164-1212/

```c
// source.c
#include<stdio.h>
int f(int n)
{
  int result=-1;

  if(n % 2 == 0)
    result = n * n;
  else
    result = n + 3 * n;

  return(result);
}
int main()
{
  int n;
  scanf("%d", &n);
  printf("%d\n",f(n));
  return(0);
}
```

```c
// source_v1.c
#include<stdio.h>
int f(int n)
{
  int result=-1;

  if(n % 2 == 0)
    result = n + 2; //change
  else
    result= n + 3 * n;

  return(result);
}
int main()
{
  int n;
  scanf("%d", &n);
  printf("%d\n",f(n));
  return(0);
}
```

Σ: $\{t_1, t_2, t_3, ..., t_{100}\}$ (original test-suite)
$\{t_1, ..., t_{100}\}$: (contains randomly generated integer in [0..99])
$\{0, 4, 16, 50, 2\}$: (integers present in $\{t_1, t_3, t_4, t_{15}, t_{99}\}$)
All other test-cases $\in \Sigma$ contain odd numbers in [0..99]

Fig. 1. Original C source (*source.c*) and its modified revision (*source_v1.c*). Test-suite description appears below the code. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

1.1. Motivation for consensus test prioritization

We begin by motivating the need for consensus in test-prioritization using an example and visualization of test-failures along the sequential execution timeline.

By default, test-cases are run in the order of their appearance (e.g., sorted by name in a script). More sophisticated testing teams prioritize test-cases in the decreasing order of their coverage. More variations exist: for instance, one may want a test-case execution order based on the decreasing order of the *remaining coverage*, or in the increasing order of test-case execution time, or a combination. In test-prioritization, researchers have found that there is no optimal heuristic for all the cases in testing regressions, given the diverse nature of software and its underlying modifications (Chen et al., 2018).

Conventional heuristics target individual (Yoo and Harman, 2012) criterion, instead of a weighted hybridization (Shin et al., 2019). Hybridization may boost effectiveness of prioritization when at least two different individual heuristics are employed. The underlying compound heuristic assigns weighted scores due to individual heuristics. The normalized weights are empirically chosen such that they sum to one. As priority scores are determined statically, these prioritization approaches are on-the-fly hybrid. To the best of our knowledge, no prioritization approach performs consensus of its individual prioritized permutations.

In this paper, we propose Hansie, which performs: (i) hybrid (on-the-fly), and (ii) consensus (post-individual) regression test prioritization. In the first variant, weighted-sum-hybridization is employed. For any given instance of hybridization, exactly two individual heuristics are used. Test-cases are sorted *only after the weighted score* is computed. In the second variant, *individual permutations are first computed* (offline phase), which is then followed by computation of a consensus permutation respecting preference lists of individual prioritization approaches, to the extent possible.

Fig. 1 shows an example of code revision: (`source.c` → `source_v1.c`), and its original full test-suite (Σ). Fig. 2 illustrates a scenario where consensus prioritization is fruitful as we explain below.

σ: $\{t_1, t_3, t_4, t_{15}, t_{99}\}$ (regression test-suite)
p_1: $<t_3, t_1, t_4, t_{99}, t_{15}>$ (prioritization due to heuristic #1)
p_2: $<t_4, t_{99}, t_3, t_{15}, t_1>$ (prioritization due to heuristic #2)
p_3: $<t_{15}, t_1, t_4, t_3, t_{99}>$ (prioritization due to heuristic #3)
p^*: $<t_4, t_3, t_1, t_{15}, t_{99}>$ (consensus prioritization)
p_{opt}: $<t_3, t_4, t_{15}, t_1, t_{99}>$ (optimal prioritization)

Fig. 2. Regression consensus test-prioritization for the code revision in Fig. 1. Failed test-cases are highlighted in blue. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

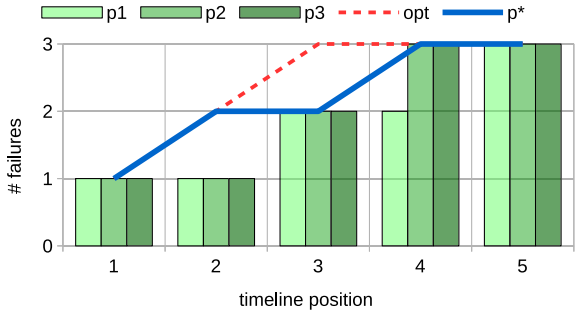

Fig. 3. Cumulative observance of test failures (for Fig. 2) due to individual (p_1, p_2, p_3), consensus (p^*), and optimal (p_{opt}) prioritizations. *y*-axis label indicates cumulative failures up to each *x*, and assumes no ordering among p_is. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Due to the code-change in line 8, the regression test-suite σ (having five test-cases) is selected from the original test-suite Σ (having hundred test-cases). Then, the individual prioritizations (p_1, p_2, and p_3) are specified due to respective heuristics. The permutation p^*: $<t_4, t_3, t_1, t_{15}, t_{99}>$ comprises of the final consensus ordering as specified by a standard Kemeny–Young method (Press, 2012). In this instance of test-case prioritization, let the failed test-cases be $\{t_3, t_4, t_{15}\}$, which occur as failures along the timeline of test-execution. For p^*, the failure positions are $(1, 2, 4)$. Had individual prioritizations been employed, one would observe the following failure positions: $p_1(1, 3, 5)$, $p_2(1, 3, 4)$, $p_3(1, 3, 4)$. The optimal position vector should be $p_{opt}(1, 2, 3)$, in which case failures are observed consecutively from the beginning. Note that there could be multiple optimal prioritizations, but the optimal position vector is fixed.

Cumulative observance of failures along the execution timeline for Fig. 2 is depicted in Fig. 3. The positions along the timeline are shown along *x*-axis. If viewed in terms of area under the curve, the consensus prioritization (shown in **blue**) turns out to be the closest to the optimal (shown in **red** dashes), area of two unit-triangles away. In Fig. 3, this deficit area is represented with coordinates $(2, 2)$ and $(4, 3)$ as lower-left and upper-right corners, respectively. All other prioritization have smaller areas. This indicates that the test-case ordering due to consensus is able to find failures earlier.

Based on a recently conducted empirical study (Torres et al., 2019), a dense distribution of failing test-cases observed near the beginning of the timeline is desirable. Such prioritizations more effectively cluster failures compared to a peer technique detecting the same number of faults with a greater spread with respect to the position of the failed test-cases.

At the end of prioritized execution, an ideal prioritization should comprise of the following scenario when graphically viewed on the *xy*-plane (along *x*-axis: timeline position, along *y*-axis: #failures). With $c > 0$ failures, cumulative failures have

been observed following the trend equation $y = x$, such that $x > 0$ and $y > 0$, with $y = c$ ($c > 0$) whenever $x \geq c$. For our example, we observe that the failure pattern of p^* (consensus prioritization) is the closest (in terms of Manhattan distance) to the optimal prioritization. It is to be noted that this visualization is valid only for sequential test-execution where each position is associated with a unique time-stamp. Parallel test-case execution poses more challenges as tests are executed in a parallel window (batch), all having the same parallel position.

1.2. Definitions and preliminaries

In this subsection, we define key terms and review background information necessary for the rest of this paper.

Rank aggregation problem: In social choice theory, the problem of rank aggregation (Terzi, 2015; Dwork et al., 2001) is defined as follows. Given n individual ranked lists: $\{p_1, \ldots, p_n\}$ of m items: $\{t_1, \ldots, t_m\}$, find a single ranking p^* that is in agreement (to the extent possible) with each of the existing input rankings. Further, p^* is an optimal permutation if it has a minimal sum-total of disagreements with individual rankings. Each ranked list can be viewed as an expert opinion and p^* as a consensus ordering. Although, ties (Fagin et al., 2004) may be present in each p_i and p^*, w.l.o.g., we treat each p_i as a strictly specified ordering i.e., a permutation and allow ties only in the consensus ordering, p^*. The rank aggregation problem exhibits two variants defined as follows:

1. *Priority-aware aggregation* (hybridization): The scores assigned to objects by individual rankings are known in advance. In this situation we want to find an aggregated score per object. This is what we say as on-the-fly hybrid aggregation, which in our case, turns out to be weighted sum of priorities.
2. *Priority-blind aggregation* (consensus): In this variant, all heuristics have specified their opinion (prioritized permutation) and we do not study the underlying mechanism employed. Instead we only have access to individual orderings. In this situation, we are required to combine all rankings based on an optimization criteria involving the notion of distance between rankings or permutation distances.

Strength of prioritization: A test-suite ordering is said to be strong or strictly specified if no two test-cases have the same rank or priority score, otherwise it is said to be weak or loosely specified. A strong prioritization is a total order, and a weak prioritization is a partial order.

Stability of prioritization: A test-prioritization is said to be *stable* if the underlying sorting algorithm employed to shuffle or re-order the test-cases, is stable. In a stable prioritization, a contiguous segment of test-cases tied for the same priority score will have their relative positions retained even after prioritization. In this scenario, test-case identifiers form a monotonically increasing sub-sequence.

Test-case segment: A group of independent test-cases arranged in arbitrary order of test-case identifiers or labels. Test-case segments are especially useful for strengthening a subset of a loosely specified test-suite permutation.

Parallelization window (Mondal and Nasre, 2019): A batch or window of multiple threads intended to execute individual test-cases in an independent parallel fashion.

Strengthening/weakening/respecting a prioritization: A weak test-permutation may have irregularly sized tied test-segments at arbitrary places. In this scenario, the prioritization may be *respected* by executing only tied[2] segments using non-uniform (different segment lengths) parallel windows (Mondal and Nasre, 2019), and sequentially otherwise. Alternatively, a weak permutation may be *strengthened* by disregarding ties present in the ordering and treating the prioritization as a strictly specified one. Yet another alternative is to further *weaken* an initially strong or weak prioritization by employing uniformly sized parallelization windows, except of course the last window which may be fragmented. There are multiple ways to weaken an arbitrary prioritization which may contain ties or no ties. In this work, we weaken such prioritizations by first finding a strong total order determined by stability of sorting. Thereafter, we slice (partition) this ordering with parallel windows of uniform size, resulting in a weaker partial ordering than the initial prioritization.

Test-execution timeline (Mondal and Nasre, 2019): A test-execution timeline is a sequence of test-cases scheduled for sequential or parallel execution. The schedule is either strictly ordered for sequential execution, or segment-wise parallel in the presence of windows. In the latter case, windows are scheduled sequentially while within a window parallelism still persists.

Window latency (cost): Latency (cost) of a parallel execution window is the end-to-end time taken by a group of independently executing test-cases. Latency takes into consideration the synchronization costs (if any) among test-cases, and at window barriers.

Overall timeline latency: Overall execution latency of a timeline is the sum total of latencies of its individual components. A component may be a single test-case or a parallel window of test-cases.

Local monopolization: A heavy test-case is one that has a significantly-long execution time compared to others members of the same window. Such test-cases are said to be locally monopolizing with respect to that window. Test-monopolization is the major cause of load-imbalance inside parallelization windows.

Global monopolization: A locally monopolizing test-case which also dominates the overall timeline latency is said to be a globally monopolizing test-case. Another form of global monopolization occurs due to the presence of heavy windows along the timeline.

1.3. Contributions

To the best of our knowledge, no existing prioritization approach computes a consensus permutation, given individual permutations formed due to two or more heuristics. Currently, the state-of-the-art consensus approach in social choice theory is Kemeny–Young method (Kemeny, 1959), but it has a high complexity (Press, 2012). Alternative is Borda-Count method (de Borda, 1781) but it does not satisfy the Condorcet criteria (Young and Levenglick, 1978; Truchon, 1998), which is viewed as a fairness of the overall agreement. Hence, we need a new mechanism from that can be applied to the regression test-prioritization problem such that the Condorcet property is met and the time complexity of computing the consensus is also low. In case of multiple consensuses arising from ties, we leverage parallelization windows (Mondal and Nasre, 2019) to execute tied test-cases under the assumption (Mondal and Nasre, 2019) that all unit tests are independent, and can be safely executed in parallel.

The main contributions of this paper are as follows.

[2] Presence of ties is a natural phenomenon in coverage-based test-case prioritizations (Eghbali and Tahvildari, 2016; Eghbali et al., 2019).

- We perform a study on *hybrid* (on-the-fly), and *consensus* (post-individual) regression test prioritization techniques.
- We introduce Hansie *distance* between two prioritizations (with and without ties) to measure the *quality* of the final consensus prioritization.
- We introduce EPL_{ω}, a *cost-cognizant metric* for quantifying *effectiveness* of prioritization, when execution is driven by *size-varying* test-parallelization windows (Mondal and Nasre, 2019) of *unequal load* distribution.

The rest of this paper is organized as follows. Sections 2, and 3 present detailed explanations of the hybrid and consensus techniques, respectively. Section 3.2 explains our proposed permutation metric namely Hansie distance, in detail. Section 4 presents experimental evaluation of Hansie. Section 4.1.7 explains existing measurements of prioritization effectiveness and introduces our latency-cognizant effectiveness metric, EPL_{ω}. Section 5 compares with and contrasts against the relevant related work. Finally, Section 6 mentions our findings and states directions for future work.

2. Priority-aware aggregation: Hybridized approach

Individual prioritization strategies employed in our framework, Hansie, are specified in Table 1. The first column specifies the name of the strategy. Brief description of the heuristic is mentioned in the next column. The third column specifies whether hybridization is involved. The last column states papers introducing the heuristics. Strategies greedy total and additional have been widely regarded as two state-of-the-art coverage-based prioritization mechanisms. We note that the original definitions of these two heuristics are involved with *statement* coverage and full test-suites (i.e., not only the selected regression test-cases).

In Hansie, on the other hand, we have tailored these mechanisms to consider only the affected *basic-block* coverage (LLVM IR) for only the selected regression tests. On an average, this makes implementation of these prioritization strategies not only more precise in our framework, but also faster than their original proposals. The precision comes from fewer tests to be considered, i.e., regression test prioritization instead of RetestAll prioritization (which runs all the test-cases). The speedup is due to the coarser granularity of basic-blocks vis-a-vis a statement, as a basic-block comprises of multiple statements (IR level instructions). These together reduce the total number of entities (tests as well as code elements) to be processed.

To observe the benefit of basic-block granularity, note that in case of our running example, `source.c` (Fig. 1), the corresponding LLVM IR (Fig. 4) contains only 5 basic-blocks as compared to 29 statements at the intermediate representation (10 statements at the C source-code level). The choice of the particular optimization level, `-O0`, is necessary to not miss: (i) refactoring changes, and (ii) unintended optimizations, which the compiler (`clang`) may otherwise perform before running the change-identification. This optimization level is however applied to both the versions before code-differencing is performed, to ensure uniform comparison.

For cost-only prioritization, cost of a test-case (in old version) is approximated by the cost model employed in Yoo and Harman (2007), Epitropakis et al. (2015). This model associates cost with the number of I-cache references made (event `Ir`) while executing a test-case via `cachegrind` (Valgrind Developers, 2019). In hybridized model of aggregation, individual priority scores due to two heuristics (Mondal and Nasre, 2019) relevance and confinedness, are known before computing their aggregation as a weighted sum. The *weight of relevance* is specified by the factor α, expressed as a percentage, and the residual factor $(1-\alpha)$

```
;source.ll
...
define i32 @f(i32 %n) #0 {
entry:                          ; basic-block #2: (f,entry)
  %n.addr = alloca i32, align 4
  %result = alloca i32, align 4
  store i32 %n, i32* %n.addr, align 4
  store i32 -1, i32* %result, align 4
  %0 = load i32, i32* %n.addr, align 4
  %rem = srem i32 %0, 2
  %cmp = icmp eq i32 %rem, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:                        ; basic-block #3: (f,if.then)
  %1 = load i32, i32* %n.addr, align 4
  %2 = load i32, i32* %n.addr, align 4
  %mul = mul nsw i32 %1, %2
  store i32 %mul, i32* %result, align 4
  br label %if.end

if.else:                        ; basic-block #4: (f,if.else)
  %3 = load i32, i32* %n.addr, align 4
  %4 = load i32, i32* %n.addr, align 4
  %mul1 = mul nsw i32 3, %4
  %add = add nsw i32 %3, %mul1
  store i32 %add, i32* %result, align 4
  br label %if.end

if.end:                         ; basic-block #5: (f,if.end)
  %5 = load i32, i32* %result, align 4
  ret i32 %5
}
...
define i32 @main() #0 {
entry:                          ; basic-block #1: (main,entry)
  %retval = alloca i32, align 4
  %n = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  %call = call i32 (i8*, ...) @__isoc99_scanf(...)
  %0 = load i32, i32* %n, align 4
  %call1 = call i32 @f(i32 %0)
  %call2 = call i32 (i8*, ...) @printf(...)
  ret i32 0
}
...
```

Fig. 4. LLVM IR (opt. level `-O0`): {`source.c`}→{`source.ll`}. Comments begin with ';' and are highlighted in **red**. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

weighs the other heuristic, *confinedness*. The linear combination is mathematically formulated as:

$$relcon(t,\alpha) = \alpha \times \frac{|cov(t) \cap \Delta|}{|\Delta|} + (1-\alpha) \times \frac{1}{|cov(t) - \Delta|} \quad (1)$$

The first component of summation, $\alpha \times \frac{|cov(t)\cap\Delta|}{|\Delta|}$, is the weighted-*relevance* of which assigns a priority score equal to the normal value of the number of affected basic-blocks covered. The second component is weighted-*confinedness*, $(1-\alpha) \times \frac{1}{|cov(t)-\Delta|}$, which assigns remaining weight to priority score equal to inverse of the number of unaffected basic-blocks. The combination is translated as follows: schedule a test-case earlier in test-execution if it covers affected blocks as much as possible without considering the coverage of unaffected blocks. Independently also prefer a test-case if it executes as few as possible of unaffected blocks. The latter heuristic treats two test-cases equally if they cover exactly same amount of unaffected code segments even if they have a differing amount in covering affected ones. To achieve a combined affect and besides being *heuristically independent*, we utilize the addition operator '+' introduced by Mondal and Nasre (2019). In this paper, we modify their equation by: (i) normalizing the relevance value so that both relevance and confinedness scores lie in the interval [0, 1], and (ii) assign relevance weightage α and linearly combined the two scores. In this formulation, test-cases with higher weighted-sums are preferred early in execution schedule.

Note that in Eq. (1) the parameter α is primarily intended to assign more weight to relevance and less to confinedness. This

Table 1
Individual and hybrid prioritization strategies in our evaluation.

Strategy	Heuristic	Type	References
relevance (greedy total)	largest affected (Δ) coverage ($cov(t)$) first	Single	Rothermel et al. (2001)
{*rel*()}	$rel(t) = \frac{\lvert cov(t) \cap \Delta \rvert}{\lvert \Delta \rvert}$ cover *affected* code as *much* as possible		Elbaum et al. (2002) Mondal and Nasre (2019)
confinedness {*con*()}	largest undeviated (confined) coverage first $con(t) = \lvert cov(t) - \Delta \rvert^{-1}$ cover *unaffected* code as *less* as possible	Single	Mondal and Nasre (2019)
cost-only	Shortest execution time first	Single	Chen et al. (2018)
greedy additional	Largest remaining coverage first	Single	Rothermel et al. (2001) Elbaum et al. (2002)
relcon (10,90)	$0.1 * rel(t) + 0.9 * con(t)$[a]	Hybrid	This paper
relcon (20,80)	$0.2 * rel(t) + 0.8 * con(t)$	Hybrid	This paper
relcon (30,70)	$0.3 * rel(t) + 0.7 * con(t)$	Hybrid	This paper
relcon (40,60)	$0.4 * rel(t) + 0.6 * con(t)$	Hybrid	This paper
relcon (50,50) (*relevance and confinedness*)	$0.5 * rel(t) + 0.5 * con(t)$	Hybrid	Mondal and Nasre (2019)
relcon (60,40)	$0.6 * rel(t) + 0.4 * con(t)$	Hybrid	This paper
relcon (70,30)	$0.7 * rel(t) + 0.3 * con(t)$	Hybrid	This paper
relcon (80,20)	$0.8 * rel(t) + 0.2 * con(t)$	Hybrid	This paper
relcon (90,10)	$0.9 * rel(t) + 0.1 * con(t)$	Hybrid	This paper

[a]Largest weighted-score first.

is needed because test-selection precedes prioritization. A test-case is relevant (selected) if it exercises at least one affected block irrespective of whether it is confined to the changes. So, a test-case can be selected based solely on relevance. However, this is not the case with confinedness in isolation, where test-cases need to be relevant a priori, and then the reordering is performed solely based on the confinedness scores. When hybridized, this relationship between the two metrics still remains intact.

Algorithm 1: Hybridized Prioritization

Input: Heuristics: $\{h_1, ..., h_k\}$, Weights: $\{w_1, ..., w_k\}$ Selected-tests: $\sigma = \{t_1, ..., t_n\}$
Output: Prioritized-permutation due to hybrid scoring p_*
1 p_* := `initialize`();
2 **for** $t \in \sigma$ **do**
  /* `generalized hybridization function` */
3 $score(p_*.t)$:= `hybridization`($score(h_1.t)$, ..., $score(h_k.t)$);
4 **end**
/* `sort test-cases based on hybrid score` */
5 p_* := `reorder-tests-by-score`(p_*);
6 **return** p_*;

The process of generalized hybrid test prioritization is presented as Algorithm 1 which accepts as input individual heuristic functions h_k for scoring, and respective weights w_k to be given to each score. The weights w_k sum to one. Line 1 initializes the hybrid permutation p_* by ordering all the selected test-cases in increasing order of their identifiers. It also sets priority score of each test-case to zero. We consider this computation as a preprocessing step which takes $O(n)$ time to execute, where n is the number of test-cases. Main computation begins with the **for** loop at lines 2–4 which iterates n times over all the selected test-cases in σ. For each iteration, line 3 executes in $O\left(\sum_{i=1}^{k} h_i^c + f^c\right)$ time, where h_i^c is the cost of finding score due to heuristic h_i, and f_c is the cost of `hybridization` function. The sorting by scores in line 5 executes in $O(n \log_2 n)$ time. Overall, the time complexity of Algorithm 1 turns out to be:

$$O\left(n \times \left(\sum_{i=1}^{k} h_i^c + f^c + \log_2 n\right)\right) \quad (2)$$

In this paper, we consider a specialization of parameters for $k = 2$ heuristics: (i) h_1 = *relevance* weighing α, and (ii) h_2 = *confinedness* weighing $(1 - \alpha)$, with `hybridization` function as specified in Eq. (1).

An example prioritization due to relevance-and-confinedness (relcon) in: (i) isolation, (ii) unweighted, and (iii) weighted {(80%, 20%), (20%, 80%)} hybridization, is illustrated in Fig. 5. The diagrams depict forward mapping from {test-cases}→{basic-blocks}. This data structure stores basic-block coverages of an entire suite of test-cases. The map is represented in memory as an array of linked-lists, similar to adjacency list representation of a graph. The selected set of regression test-suite consists of ten test-cases $\{t_1, t_2, \ldots, t_{10}\}$, due to the change-set, $\Delta = \{b_1, b_4, b_7\}$. For a coverage map, linked lists are of unequal lengths due to non-uniform simple paths traversed by different test-cases.

It is to be noted that back edges in the associated control flow graph may cause a basic-block to be encountered more than once in the trace of a test-case. We discard such back edges by keeping only the first visit to each basic-block. This ensures that unique blocks appear in the order they are executed. For t_1, the visit order is $\{b_1, b_2, b_3, b_4, b_7\}$ corresponding to the old version. We employ this technique in Hansie for computing all coverage-based prioritizations, as shown in Table 1. In Fig. 5, affected blocks appearing as part of test-coverage maps, are highlighted in red. At the end of each linked list, priority scores are displayed in parenthesis, except the unordered case. The higher the score of a test-case, higher is its priority. For the tied scores, the test-case identifier breaks the tie. For this example, it is simply the order in which test-cases appear in the original test-suite i.e., $\{t_1, t_2, t_3, \ldots\}$.

In Fig. 5, prioritized permutation for each heuristic is presented at the bottom of the associated coverage (forward) map. Prioritization by relevance gives more priority to test-cases such as $\{t_1(1), t_2(1)\}$ which cover three affected blocks as compared to $\{t_3(0.67), t_{10}(0.67)\}$ which cover two affected blocks. However, confinedness treats them exactly in the opposite manner: $\{t_3(1), t_{10}(1)\}$ versus $\{t_1(1), t_2(1)\}$, because the latter two test-cases cover two ($\{b_2, b_3\}$) and five ($\{b_2, b_3, b_5, b_8, b_9\}$) unaffected blocks, respectively even though they both cover all the affected blocks. On the other hand, t_3 and t_{10} cover only one unaffected block besides covering two affected blocks. In this aspect, they are more confined than t_1 and t_2.

The combination of relevance and confinedness brings together flavors of both. It is to be noted that the unweighted combination brings together all four test-cases $\{t_3, t_{10}, t_1, t_2\}$ as the four most favored tests in that order. However, these four can still be prioritized among themselves as reflected by assigning 80% weightage (say) to relevance that brings *more relevant* test-cases $\{t_1, t_2\}$ earlier in the permutation. On the other hand, giving 80% weightage to confinedness brings $\{t_3, t_{10}\}$ earlier than $\{t_1, t_2\}$. In this case t_2 appears second last because confinedness (individually) with 100% weightage specifies t_2 as least preferred. The gain in position of t_2 by 1 is due to a residual weightage of 20% to relevance. Thus, by varying relevance weightage α, we can obtain a hybridization spectrum of relevance and confinedness.

For cost-only prioritization, let us model (only in this example) the cost of a test-case by cardinality of its distinct basic-block coverage. In Fig. 5, it is the length of the linked list associated with a test-case. For this example, the cost vector can be

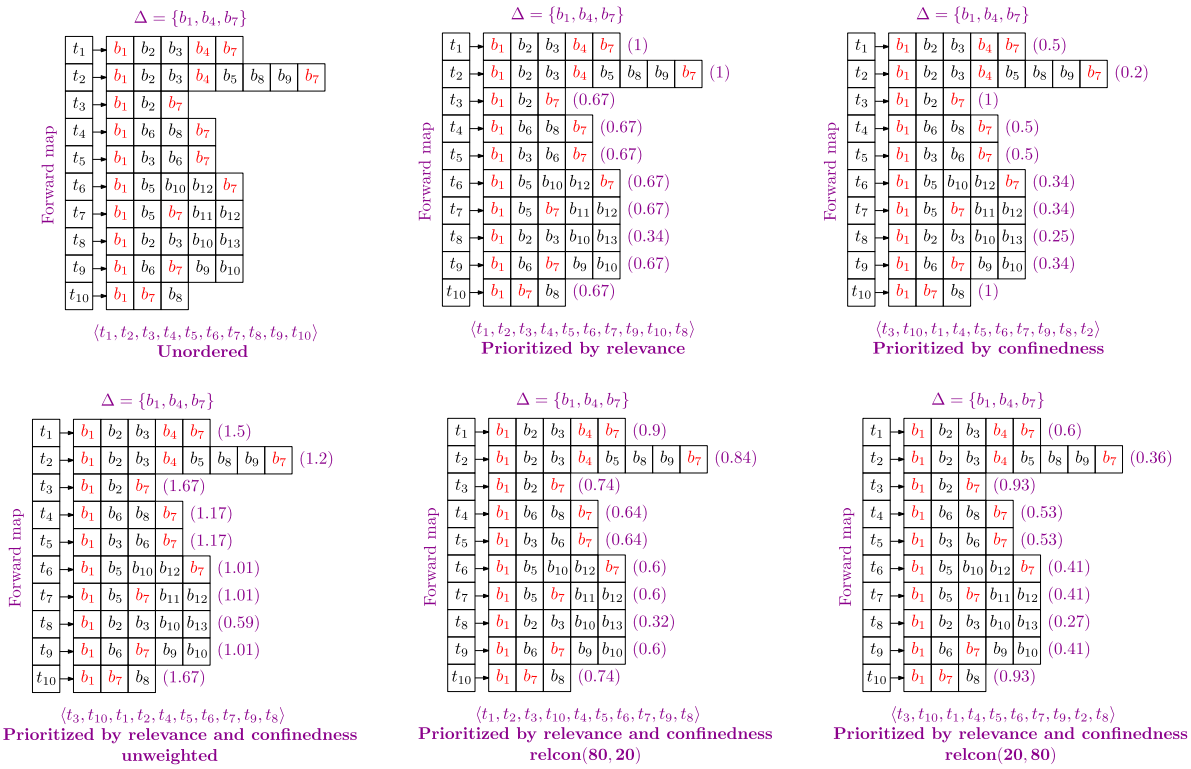

Fig. 5. Illustration of relevance and confinedness (Mondal and Nasre, 2019): unordered, isolated (relevance-only and confinedness-only), unweighted, and weighted {(80%, 20%), (20%, 80%)} *hybridization* (this paper). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

represented as: $\overline{c} = \langle 5_1, 8_2, 3_3, 4_4, 4_5, 5_6, 5_7, 5_8, 5_9, 3_{10}\rangle$ for t_1 through t_{10}, c_i being the cost of t_i. After sorting, the vector $\overline{c}$ turns out to be: $\overline{c'} = \langle 3_3, 3_{10}, 4_4, 4_5, 5_1, 5_6, 5_7, 5_8, 5_9, 8_2\rangle$. The corresponding permutation for cost-only prioritization is: $\langle t_3, t_{10}, t_4, t_5, t_1, t_6, t_7, t_8, t_9, t_2\rangle$.

For greedy additional prioritization, test-cases are ordered based on decreasing amount of remaining basic-block coverage after having selected one with the largest coverage. The final ordering for greedy additional heuristic after 10 iterations turns out to be: $\langle t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_9, t_{10}, t_8\rangle$, which coincides with the ordering determined by 100% relevance alone.

3. Priority-blind aggregation: Consensus approach

In social choice theory (Truchon, 1998), the notion of consensus corresponds to maximization of an objective, namely agreement. Equivalently, this can also be viewed as minimization of disagreement. This calls for computation of an aggregated preference ordering that minimizes sum total of individual disagreements. In rank aggregation problem, this demands computing distance between test-case orderings and the problem leads to minimizing permutation distances. Existing literature models this distance as *Kendall-tau (KT) distance* (Kendall, 1938, 1948) between two rankings. This distance between two permutations is a metric which follows the axioms of non-negativity, identity, symmetry, and triangle-inequality.

3.1. Kendall-tau distance

We now recall definition of permutation distances (metrics) to determine distance between strong (*totally ordered*) or weak (*partially ordered*) prioritizations.

KT-distance without ties (Kendall, 1938, 1948; Kemeny, 1959): The Kendall-tau rank distance quantifies the total number of pairwise disagreements between two permutations (totally ordered lists). Disagreement for a pair (i, j)[3] with $(i < j)$ is one (100%) if order of their appearance in two prioritizations (ranked lists) are opposite. Otherwise, the disagreement is zero (0%). This quantification takes into account only the relative ordering and not how far they are separated. Our customized mathematical formulation for strictly specified permutations is as follows:

$$\begin{aligned} &KT(a, b) = |\{(i, j) : (i < j) \wedge (A \vee B)\}| \\ &A : \{\rho(a_i) < \rho(a_j)\} \wedge \{\rho(b_i) > \rho(b_j)\} \\ &B : \{\rho(a_i) > \rho(a_j)\} \wedge \{\rho(b_i) < \rho(b_j)\} \end{aligned} \tag{3}$$

where $\rho(a_i)$ denotes the rank of element i in the ordering represented by a. The lower the ρ value, the earlier is the occurrence of i in a. $KT(a, b)$ is 0 if a and b are identical. The maximum value of KT is $\frac{n_1(n_1-1)}{2}$ where n_1 is the size of both a and b. This occurs when a is the reverse of b. This maximum value is often used in normalizing the distance in [0, 1]. For instance, the KT distance between $\langle t_{15}, t_3, t_1, t_{99}, t_4\rangle$ and $\langle t_1, t_{99}, t_{15}, t_3, t_4\rangle$ is 4 because four pairs: (t_1, t_3), (t_1, t_{15}), (t_3, t_{99}), and (t_{15}, t_{99}), appear in opposite order when compared. For this example, size of the test-suite is $n_1 = 5$. As there are 10 unordered pairs in total $(5 \times (5-1)/2)$, the normalized value becomes 0.4.

KT-distance with ties (Kemeny, 1959): We resort to this measure when at least one of our input prioritization is weakly specified.[4] In this version an additional disagreement of 0.5 (50%) is contributed for an unordered pair of elements (i, j) if one ordering indicates a preference, say either $i \rightarrow j$ $(\rho(a_i) < \rho(a_j))$ or $j \rightarrow i$ $(\rho(a_j) < \rho(a_i))$, and the other indicates a tie

[3] Throughout this paper, these form a pair of test-case identifiers which belong to the set of natural numbers.

[4] A prioritization with tied test-cases is a *partial ordering* and does not strictly adhere to the technical definition of permutation.

Table 2
Computation of *KT*-distance in the presence and absence of ties.

Prioritization #1	$\langle t_{15}, t_3, t_1, t_{99}, t_4\rangle$	$\langle [t_1, t_{15}], [t_{99}, t_3], t_4\rangle$	$\langle t_1, [t_{15}, t_{99}], t_4, t_3\rangle$
Prioritization #2	$\langle t_1, t_{99}, t_{15}, t_3, t_4\rangle$	$\langle [t_{15}, t_1], [t_3, t_{99}], t_4\rangle$	$\langle [t_1, t_4], t_3, [t_{15}, t_{99}]\rangle$
Formulation	*Eq.* (3)	*Eq.* (4)	*Eq.* (4)
Pair	**Disagreement**	**Disagreement**	**Disagreement**
(t_1, t_3)	1	0	0
(t_1, t_4)	0	0	0.5
(t_1, t_{15})	1	0	0
(t_1, t_{99})	0	0	0
(t_3, t_4)	0	0	0
(t_3, t_{15})	0	0	1
(t_3, t_{99})	1	0	1
(t_4, t_{15})	0	0	1
(t_4, t_{99})	0	0	1
(t_{15}, t_{99})	1	0	0
Σ	4	0	4.5
Norm. Σ	**0.40**	**0.00**	**0.45**

$(\rho(b_i) = \rho(b_j)$. Similar explanation holds when roles of a and b are interchanged. For pairs of elements appearing untied in both a and b, the formulation stated in Eq. (3) can be followed. It is to be noted that the normalizer for KT_{ties} is chosen as $\frac{n_2(n_2-1)}{2}$ where n_2 denotes the total number of test-cases. Note that choice of subscripts in n_1 and n_2 is only to demarcate between tied and untied formulations. For this example, $n_2 = 5$ due to the test-suite $\{t_1, t_3, t_4, t_{15}, t_{99}\}$. Our customization of KT_{ties} in combination with Eq. (3), is mathematically formulated as follows:

$$KT_{ties}(a,b) = KT(a,b) + 0.5 \times |\{(i,j) : (i<j) \wedge (C \vee D)\}|$$
$$C : [\{\rho(a_i) < \rho(a_j)\} \vee \{\rho(a_i) > \rho(a_j)\}] \wedge \{\rho(b_i) = \rho(b_j)\}$$
$$D : \{\rho(a_i) = \rho(a_j)\} \wedge [\{\rho(b_i) < \rho(b_j)\} \vee \{\rho(b_i) > \rho(b_j)\}] \quad (4)$$

An example computation of $KT(a,b)$ and $KT_{ties}(a,b)$ is illustrated in Table 2. The first row specifies the prioritizations between which the distance is to be calculated. The next row specifies the formulation used for computing the overall disagreement (distance). Starting from third row, the first column specifies all unordered pairs as specified in the orderings. For the next two columns, a tie in the ordering (first row) is specified in square brackets. For instance, the third column is associated with computation of Kendall-tau distance between two prioritizations with ties, in which $[t_1, t_4]$ (in permutation $\langle [t_1, t_4], t_3, [t_{15}, t_{99}]\rangle$) indicates t_1 and t_4 tied for the first rank. The second last row (with label 'Σ') sums up the entries starting from fourth row for each column.

The normalizing factor in all cases happens to be 10 for this example, and normalized distance is displayed in the last row. Note that the second column in Table 2 depicts two identical prioritizations in the presence of ties, hence, their normalized *KT*-distance is zero. If these two loosely specified prioritizations are strengthened by removing the square brackets, the orderings are no longer identical, which will result in a distance of 0.2. This is due to two disagreements: (i) $(t_1 \rightarrow t_{15})$ versus $(t_{15} \rightarrow t_1)$, and (ii) $(t_{99} \rightarrow t_3)$ versus $(t_3 \rightarrow t_{99})$, which were otherwise treated as equivalents in the presence of ties (enclosed in '[]').

3.2. Hansie distance

Kendall-tau distance can be misleading because of its binary nature of disagreement (0 or 1) in the absence of ties. Such a formulation does not capture the *magnitude of disagreement*. In practice, there exist cases wherein the relative orders of test-cases is the same in two prioritizations but either their positions or their mutual separation are different across the prioritizations. This is not captured in *KT*-distance.

Thus, for the pair (t_1, t_{100}) and test-cases $\{t_1, \ldots, t_{100}\}$, the sequence $\{t_{100}, t_2, t_3 \ldots, t_{99}, t_1\}$ should be treated differently from $\{t_2, t_3, \ldots, t_{99}, t_{100}, t_1\}$. For (t_1, t_{100}), *KT*-distance (disagreement) value between the two test-cases is 0, as their relative order is the same in two prioritizations. However, the separation is 99 positions in the first prioritization, and 1 position in the second. To mitigate this effect, we propose Hansie distance between two prioritizations. Unlike Kendall-tau, the formulation of Hansie distance is invariant to occurrence of ties in prioritizations. Given two prioritizations a and b, Hansie distance is defined as follows:

$$HD(a,b) = \sum_{(i,j):\ i<j} \big(|\rho(a_i) - \rho(b_i)| + |\rho(a_j) - \rho(b_j)|\big) \quad (5)$$

The first summand $|\rho(a_i) - \rho(b_i)|$ measures the absolute difference (in terms of rank separation) in ranks of element i (test-case t_i) in both lists. Similarly, the second summand measures the disagreement for j (test-case t_j). These values are summed over all distinct unordered pairs drawn from the set of selected test-cases.

3.2.1. Hansie distance as a metric

As consensus involves minimizing sum total of permutation distances of the final consensus ordering from each of the participating individual permutations, we need a distance measure comparable with the classical Kendall-tau metric. We now show that our proposed Hansie distance Eq. (5), is a metric. Let a, b denote two prioritizations of regression test-cases σ, and Π^σ denote the set of all possible permutations of σ. To show a measure to be a metric, we need to show that it exhibits the following four properties.

- *Non-negativity*: For $a, b \in \Pi^\sigma$, with $a \neq b$, the value of $HD(a,b) \in [0, \infty)$ due to non-negativity of $|\cdot|$ and $+$. Therefore, $HD(a,b) \geq 0$.
- *Symmetry*: For $a, b \in \Pi^\sigma$, with $a \neq b$, $HD(a,b) = H(b,a)$ due to commutativity of $|\cdot|$ and $+$.
- *Identity*: For $a \in \Pi^\sigma$, $HD(a,a) = 0$. Therefore, identity of indiscernibles is satisfied.
- *Triangle-inequality*: For $a, b, c \in \Pi^\sigma$, with $a \neq b \neq c$, we need to show whether $HD(a,c) \leq HD(a,b) + HD(b,c)$ is satisfied. We begin by expanding the L.H.S. as follows:

  $$HD(a,c) = \sum_{(i,j):\ i<j} \big(|\rho(a_i) - \rho(c_i)| + |\rho(a_j) - \rho(c_j)|\big)$$

  We now proceed with the R.H.S.: $HD(a,b) + HD(b,c)$ which can be re-written as follows:

  $$\sum_{(i,j):\ i<j} \big(|\rho(a_i) - \rho(b_i)| + |\rho(a_j) - \rho(b_j)|\big) +$$
  $$\sum_{(i,j):\ i<j} \big(|\rho(b_i) - \rho(c_i)| + |\rho(b_j) - \rho(c_j)|\big)$$

  or, in a more compact form, the template for summand is:

  $$|\rho(a_i) - \rho(b_i)| + |\rho(a_j) - \rho(b_j)| + |\rho(b_i) - \rho(c_i)| + |\rho(b_j) - \rho(c_j)|$$

  If $|\cdot|$ is removed, there are sixteen combinatorial cases depending on which of the '$\geq$'s are changed to '$<$'. The calculation for some cases (underlined wherever '$\geq$' changed to '$<$' appears) is shown as follows:
  case 1: If $\rho(a_i) \geq \rho(b_i)$, $\rho(a_j) \geq \rho(b_j)$, $\rho(b_i) \geq \rho(c_i)$, $\rho(b_j) \geq \rho(c_j)$, the R.H.S. can be written as:

  $$\sum_{(i,j):\ i<j} \big(\rho(a_i) - \rho(c_i) + \rho(a_j) - \rho(c_j)\big)$$

  or, simply L.H.S. = R.H.S.

Table 3
Computation of pairwise disagreements for Kendall-tau, and Hansie distance between $\langle t_{15}, t_3, t_1, t_{99}, t_4\rangle$ and $\langle t_1, t_{99}, t_{15}, t_3, t_4\rangle$.

Pair	$\{\rho(a_i), \rho(a_j)\}$	$\{\rho(b_i), \rho(b_j)\}$	H-dist.	KT-dist.	KT mislead
(t_1, t_3)	{3, 2}	{1, 4}	4	1	none
(t_1, t_4)	{3, 5}	{1, 5}	2	0	low
(t_1, t_{15})	{3, 1}	{1, 3}	4	1	none
(t_1, t_{99})	{3, 4}	{1, 2}	4	0	high
(t_3, t_4)	{2, 5}	{4, 5}	2	0	low
(t_3, t_{15})	{2, 1}	{4, 3}	4	0	high
(t_3, t_{99})	{2, 4}	{4, 2}	4	1	none
(t_4, t_{15})	{5, 1}	{5, 3}	2	0	low
(t_4, t_{99})	{5, 4}	{5, 2}	2	0	low
(t_{15}, t_{99})	{1, 4}	{3, 2}	4	1	none
Σ	–	–	**32**	**4**	–

case 2: If $\rho(a_i) < \rho(b_i)$, $\rho(a_j) \geq \rho(b_j)$, $\rho(b_i) \geq \rho(c_i)$, $\rho(b_j) \geq \rho(c_j)$, the R.H.S. can be written as:

$$\sum_{(i,j):\ i<j} \big(2\rho(b_i) - \rho(a_i) - \rho(c_i) + \rho(a_j) - \rho(c_j)\big)$$
$$> \sum_{(i,j):\ i<j} \big(\rho(a_i) - \rho(c_i) + \rho(a_j) - \rho(c_j)\big)$$

or, simply L.H.S. < R.H.S.

case 3: If $\rho(a_i) \geq \rho(b_i)$, $\rho(a_j) < \rho(b_j)$, $\rho(b_i) \geq \rho(c_i)$, $\rho(b_j) \geq \rho(c_j)$, the R.H.S. can be written as:

$$\sum_{(i,j):\ i<j} \big(\rho(a_i) - \rho(c_i) + 2\rho(b_j) - \rho(a_j) - \rho(c_j)\big)$$
$$> \sum_{(i,j):\ i<j} \big(\rho(a_i) - \rho(c_i) + \rho(a_j) - \rho(c_j)\big)$$

or, simply L.H.S. < R.H.S.

cases 4–15: These are combinatorially similar cases which satisfy the following inequality:

L.H.S. < R.H.S.

case 16: If $\rho(a_i) < \rho(b_i)$, $\rho(a_j) < \rho(b_j)$, $\rho(b_i) < \rho(c_i)$, $\rho(b_j) < \rho(c_j)$, the R.H.S. can be written as:

$$\sum_{(i,j):\ i<j} \big(\rho(c_i) - \rho(a_i) + \rho(c_j) - \rho(a_j)\big)$$

or, simply L.H.S. = R.H.S.

Considering all cases L.H.S. ≤ R.H.S. Therefore, sub-additivity or triangle-inequality is satisfied and we conclude that Hansie distance is a metric.

3.2.2. Hansie distance versus Kendall-tau distance

In the presence of ties, the disagreement of 0.5 in Kendall-tau distance increases the precision by only 1 by revising the distance formulation as ternary {0, 0.5, 1}. Recall that the formulation of *KT*-distance only counts whether a pair is inverted, which is binary, i.e., 0 (same order) or 1 (opposite order) in two given permutations. The number of inverted pairs (# of 1s) accumulates to form the *KT*-distance. In comparison, Hansie distance (HD) provides a greater precision by formulating the notion of disagreement as absolute separation in ranks.

We now illustrate a scenario where Kendall-tau distance is misleading and this situation is remedied by Hansie distance. Let us consider a regression test-suite $\{t_1, t_3, t_4, t_{15}, t_{99}\}$ and its two prioritizations: $a = \langle t_{15}, t_3, t_1, t_{99}, t_4\rangle$, and $b = \langle t_1, t_{99}, t_{15}, t_3, t_4\rangle$. No two test-cases in a and b are tied for a given rank, the rank sequence being $\langle 1, 2, 3, 4, 5\rangle$.

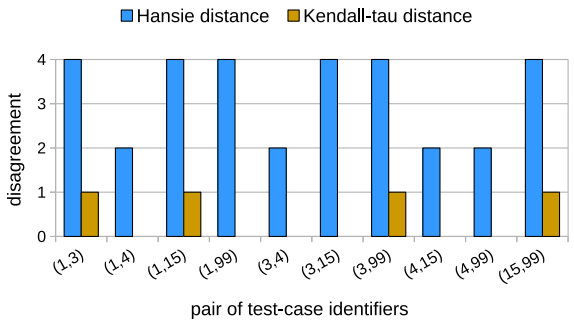

Fig. 6. Visual depiction of pairwise disagreement for Kendall-tau vs. Hansie distance between $\langle t_{15}, t_3, t_1, t_{99}, t_4\rangle$ and $\langle t_1, t_{99}, t_{15}, t_3, t_4\rangle$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 4
Computation of Hansie distance in the presence of ties between $\langle t_1, [t_{15}, t_{99}], t_4, t_3\rangle$ and $\langle [t_1, t_4], t_3, [t_{15}, t_{99}]\rangle$.

Pair	$\{\rho(a_i), \rho(a_j)\}$	$\{\rho(b_i), \rho(b_j)\}$	H-dist.	KT-dist.
(t_1, t_3)	{1, 4}	{1, 2}	2	0
(t_1, t_4)	{1, 3}	{1, 1}	2	0.5
(t_1, t_{15})	{1, 2}	{1, 3}	1	0
(t_1, t_{99})	{1, 2}	{1, 3}	1	0
(t_3, t_4)	{4, 3}	{2, 1}	4	0
(t_3, t_{15})	{4, 2}	{2, 3}	3	1
(t_3, t_{99})	{4, 2}	{2, 3}	3	1
(t_4, t_{15})	{3, 2}	{1, 3}	3	1
(t_4, t_{99})	{3, 2}	{1, 3}	3	1
(t_{15}, t_{99})	{2, 2}	{3, 3}	2	0
Σ	–	–	**24**	**4.5**

The associated computation is shown in Table 3. The first column specifies all possible unordered pairs. The second column lists the indices of occurrences of t_i and t_j in a. Likewise, the third column lists the pairs of indices in b. The next two columns respectively reflect the Hansie and the Kendall-tau distances. The last column linguistically specifies the amount by which *KT* distance is misleading. For the first pair (t_1, t_3), we do no observe any issue as both the distances specify non-zero disagreement. However, for pairs (t_1, t_{99}), and (t_3, t_4), {Hansie, Kendall-tau} distances are {4, 0} and {2, 0}, respectively. This shows that Hansie offers better precision than Kendall-tau distance.

Fig. 6 provides a visual depiction of variation in disagreement between Hansie and Kendall-tau distances, as shown in Table 3 (columns 4 and 5).

3.2.3. Hansie distance in the presence of ties

The formulation of Hansie distance remains the same in case of both the presence and the absence of ties. The only difference is in quantification of variables for tied prioritization: (i) $\rho(a_i)$ can be equal to $\rho(a_j)$, (ii) $\rho(b_i)$ can be equal to $\rho(b_j)$. We illustrate this using an example. Let us again consider the regression test-suite $\{t_1, t_3, t_4, t_{15}, t_{99}\}$ with two prioritizations: (i) $a = \langle t_1, [t_{15}, t_{99}], t_4, t_3\rangle$ with t_{15} and t_{99} tied at rank 2, and (ii) $b = \langle [t_1, t_4], t_3, [t_{15}, t_{99}]\rangle$ with t_1 and t_4 tied at rank 1, and t_{15} and t_{99} tied at rank 3. Respective rank sequences are $\langle 1, 2, 3, 4\rangle$, and $\langle 1, 2, 3\rangle$. Note that if this sequence happens to be test-prioritization with ties, parallelization can be effectively utilized to execute tied test-cases such that their priorities are respected. Otherwise, enforcing a strict sequential ordering would essentially demarcate their relative importance.

The computation is illustrated in Table 4. We observe that for pair (t_1, t_4), and (t_{15}, t_{99}), either $\rho(a_i) = \rho(a_j)$, or $\rho(b_i)$ can be equal to $\rho(b_j)$, or both. This is because the first pair (t_1, t_4) is fully ordered in a at ranks 1 and 3, but appears tied in b for 1st rank.

Table 5
Illustration of non-uniqueness in Hansie distance normalization.

	$HD(\langle 1,2,3,4,5,6\rangle, \langle 6,5,4,3,2,1\rangle)$			$HD(\langle 1,2,3,4,5,6\rangle, \langle 4,5,6,1,2,3\rangle)$		
Pair	$\{\rho(\pi_i^1), \rho(\pi_j^1)\}$	$\{\rho(\pi_i^2), \rho(\pi_j^2)\}$	**HD**	$\{\rho(\pi_i^1), \rho(\pi_j^1)\}$	$\{\rho(\pi_i^3), \rho(\pi_j^3)\}$	**HD**
(1, 2)	{1, 2}	{6, 5}	8	{1, 2}	{4, 5}	6
(1, 3)	{1, 3}	{6, 4}	6	{1, 3}	{4, 6}	6
(1, 4)	{1, 4}	{6, 3}	6	{1, 4}	{4, 1}	6
(1, 5)	{1, 5}	{6, 2}	8	{1, 5}	{4, 2}	6
(1, 6)	{1, 6}	{6, 1}	10	{1, 6}	{4, 3}	6
(2, 3)	{2, 3}	{5, 4}	4	{2, 3}	{5, 6}	6
(2, 4)	{2, 4}	{5, 3}	4	{2, 4}	{5, 1}	6
(2, 5)	{2, 5}	{5, 2}	6	{2, 5}	{5, 2}	6
(2, 6)	{2, 6}	{5, 1}	8	{2, 6}	{5, 3}	6
(3, 4)	{3, 4}	{4, 3}	2	{3, 4}	{6, 1}	6
(3, 5)	{3, 5}	{4, 2}	4	{3, 5}	{6, 2}	6
(3, 6)	{3, 6}	{4, 1}	6	{3, 6}	{6, 3}	6
(4, 5)	{4, 5}	{3, 2}	4	{4, 5}	{1, 2}	6
(4, 6)	{4, 6}	{3, 1}	6	{4, 6}	{1, 3}	6
(5, 6)	{5, 6}	{2, 1}	8	{5, 6}	{2, 3}	6
Σ	–	–	**90**	–	–	**90**

The second pair, (t_{15}, t_{99}), appears tied in both prioritization, the only difference being tied for rank 2 in a, and rank 3 in b. The total (unnormalized) Hansie distance turns out to be 24. In comparison, the KT-distance computation in the presence of ties is shown in the fifth column, where a partial disagreement (50%) occurs for the test-pair (t_1, t_4), leading to overall KT-distance of 4.5.

3.2.4. Normalization of Hansie distance

The normalizing factor for Hansie distance is the unnormalized Hansie distance between the label permutation and its reverse (mirror image). A label permutation is a prioritized test-sequence such that identifiers of test-cases are arranged in increasing order, possibly with some test-cases pruned due to test-selection. For instance, if originally there is a full test-suite: $\{t_1, \ldots, t_{10}\}$, and test-selection happens to be odd numbered test-cases with even numbered ones removed, we are left with the unique label permutation $\{1, 3, 5, 7, 9\}$. This corresponds to the prioritized regression test-suite $\{t_1, t_3, t_5, t_7, t_9\}$, and its associated mirror image happens to be $\{t_9, t_7, t_5, t_3, t_1\}$ (reverse prioritization). We denote the label permutation by λ and it reverse by λ^{-1}. In terms of distance, the normalizer is $HD(\lambda, \lambda^{-1})$. Eq. (5) in now normalized as follows:

$$\overline{HD}(a, b) = \frac{HD(a, b)}{HD(\lambda, \lambda^{-1})} \tag{6}$$

The normalized Hansie distance between a and b (with or without ties) can represent as $\overline{HD}(a, b)$ which lies in $[0, 1]$. The choice of normalizer (denominator in Eq. (6)) is due to maximum distance corresponding to two prioritizations being exactly reverse of each other. Note that from the viewpoint of test-case prioritization, λ and λ^{-1} are maximally opposing test-schedules. However, from the perspective of permutation distances, there exists a family of permutations, F_{norm}, representing the set of all candidate permutations λ_c such that $HD(\lambda, \lambda^{-1}) = HD(\lambda, \lambda_c)$.

$$F_{norm} = \{\lambda_c : HD(\lambda, \lambda_c) = HD(\lambda, \lambda^{-1})\} \tag{7}$$

This means non-existence of unique normalizing factor for Eq. (6). This is due to the symmetrical nature of '$|\cdot|$' function which forms the basis of Hansie distance. We illustrate this with an example next.

Let us consider three permutations of first six natural numbers: $\pi^1 = \langle 1, 2, 3, 4, 5, 6\rangle$, $\pi^2 = \langle 6, 5, 4, 3, 2, 1\rangle$, and $\pi^3 = \langle 4, 5, 6, 1, 2, 3\rangle$. The unique sequence of ranks being $\overrightarrow{\rho} = \langle 1, 2, 3, 4, 5, 6\rangle$. Computation of $HD(\pi_1, \pi_2)$, and $HD(\pi_1, \pi_3)$ is shown in Table 5. We observe that the normalizing distance of 90 occurs for both $HD(\pi^1, \pi^2)$ (columns 1–3 in Table 5), and $HD(\pi^1, \pi^3)$ (columns 4–6 in Table 5). However, to maintain uniformity, in Eq. (6), we choose $HD(\lambda, \lambda^{-1})$ as the only normalizing factor with respect to permutations a and b.

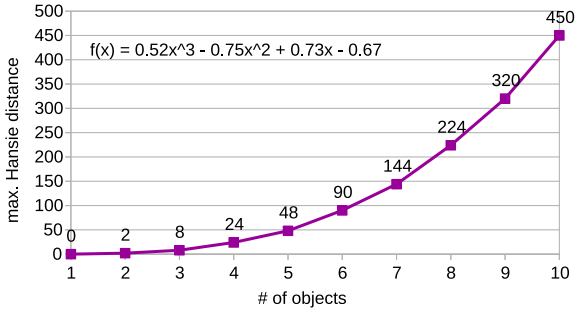

Fig. 7. Variation of normalizing factor for Hansie distance.

The normalizing factor for Hansie distance between two permutations depends on the number of unique members present in them. Fig. 7 shows the variation of normalizer with increasing number of objects (members). The corresponding trend can be mathematically expressed as $f(x) = 0.5x^3 - 0.75x^2 + 0.73x - 0.67$, where x is the distinct number of members/objects/test-cases.

3.2.5. Distance computation and consensus qualities

For two ranked lists a and b, if n is the number of candidates to be ranked, then the total number of distinct pairs (i, j) such that $i < j$ is $O(n^2)$. For each such pair, the ranking takes $O(n)$ time to search with possible occurrence of ties. Therefore, the overall time complexity of $KT(a, b)$, $KT_{ties}(a, b)$, and $HD(a, b)$ turns out to be as high as $O(n^3)$.

More efficient implementations of KT-distance exist (Knight, 1966; Ionescu, 2011) which take $O(n \log_2 n)$ time by modifying `merge-sort()`. However, this efficiency comes at an $O(n^2)$ preprocessing cost which mandates labeling a as an identity permutation $a = \langle 1, 2, 3, \ldots\rangle$ and b as its shuffle ($b := \sigma(a)$), where $\sigma(\cdot)$ is the shuffling function whose range constitutes the permutation space. Note that additional overhead may be incurred when ties are involved either in creating multiple instances of the same label, or during the `merge()` operation of `merge-sort()`, which counts the number of inversions in b with respect to a.

3.3. Consensus prioritization strategies

In general, consensus is *post-process* in the sense that it is performed after solutions due to individual algorithms are available. We then aggregate these solutions without accessing individual underlying mechanisms. For test-case prioritization, this translates to computing a consensus permutation after individual prioritization heuristics have computed respective permutations. The underlying criteria of prioritization may be cost, coverage, and risk, to name a few. Regarding minimization of total distance (disagreement), the objective is to minimize sum total of permutation distances of consensus ordering from each of the individual permutations. The consensus prioritization problem can be mathematically stated as follows. Given a permutation distance metric $d(p_i, p_j)$ and prioritized permutations $\{p_1, p_2, \ldots, p_n\}$ as input, find an optimal permutation p^* such that:

$$p^* = \arg\min_{p} \sum_{i=1}^{n} d(p_i, p) \tag{8}$$

Multiple optimal prioritizations may exist in the presence of ties in the consensus rankings. However, input prioritizations are strictly ordered such that ties are absent. It is due to this nature of ties in the consensus that a consensus test-schedule may be respected by executing tied test-cases within parallel execution windows, and sequentially otherwise.

3.3.1. Prioritization by Kemeny-Young consensus

The Kemeny–Young (KY) consensus (Kemeny, 1959; Young and Levenglick, 1978; Dwork et al., 2001; Lv, 2014; Fagin et al., 2003, 2004; Betzler et al., 2008; Kumar and Vassilvitskii, 2010) is a classical method of rank aggregation used in voting systems, where voters specify their preference of candidates as a priority list, which is collectively called a profile of rankings or preferences.

In general, the KY-consensus allows each preference list to be partial in the sense that voters need not specify all the candidates in their individual preferences. It also allows a voter to indicate ties at the same level of preference.

The KY-consensus is a Condorcet method[5] (Young and Levenglick, 1978; Davenport and Kalagnanam, 2004) that follows majority rule and assigns first rank to most preferred candidate (Condorcet winner), second rank to the next most preferred candidate, until assigning last rank to the least preferred candidate (Condorcet loser).

The computation of consensus can be formulated as in Eq. (8) by substituting the distance function, d, by Kendall-tau (KT) distance. Alternatively, the KY-consensus calls for computation of sequence of candidates that has the highest overall ranking score, wherein the score of a sequence is the sum total of pairwise preference votes. For instance, if candidate i comes before j in the consensus, the score of $i \rightarrow j$ (prefer i to j) denotes how many voters support this preference. In this sense, the consensus ordering is the one which has most supporting pairwise preferences. Ties may occur for a pair (i, j) in the majority preference, if the number of voters preferring $i \rightarrow j$ exactly equals the count of dis-preferring it, i.e., preferring $j \rightarrow i$. Tie-breaking amounts to selecting an ordering from among those with same overall score. Thus, in the final consensus ordering, i precedes j if and only if majority ($> 50\%$) of voters prefer $i \rightarrow j$.

Note that a Kemeny–Young ordering does not take into account the distance between pairs for which preference is to be calculated, only the relative ordering is considered to distinguish a preference from a dis-preference. This is analogous to the formulation of Kendall-tau distance.

We now cast this approach into consensus test-case prioritization where voters correspond to prioritization heuristics, and candidates correspond to selected test-cases. It is to be noted that each heuristic ranks all test-cases, hence, all preference lists (individual prioritizations) are complete lists, as opposed to partial lists occurring in conventional preferential voting. We also assume that all individual prioritizations are strictly ordered where ties are absent.

The KY-method has been found to be NP-hard even for 4 voters (Dwork et al., 2001), so we leverage an existing implementation (Press, 2012) that employs a quick heuristic for KY-consensus computation that occasionally uses randomization. This mechanism which maximizes ranking score or agreement score of KY-consensus is explained as follows. Given a profile of prioritized permutations, the initial approximation is obtained by assigning to each test-case the arithmetic mean of the ranks specified by individual prioritizations. This score of this approximated ordering by mean ranks is further refined in N independent iterations, each of which attempts to refine the ordering by randomly shuffling a subsequence (of size M) of the initially approximate ordering and re-evaluating its ranking score. This shuffling window of size M is slided along the entire sequence and the iteration is terminated when it is no longer possible to further find a sequence of test-cases with a better overall agreement score.

[5] The Condorcet principle states that if any element wins by majority votes in head-to-head comparisons against all others, then this element should be ranked first in consensus ordering (Dwork et al., 2001).

Table 6
Preference counts for profile $R = \{\langle t_{15}, t_3, t_1, t_{99}, t_4\rangle, \langle t_1, t_{99}, t_{15}, t_3, t_4\rangle\}$.

Pair (x, y)	$x \rightarrow y$	$y \rightarrow x$	$\lvert x \rightarrow y\rvert + \lvert y \rightarrow x\rvert$
(t_1, t_3)	1	1	2
(t_1, t_4)	2	0	2
(t_1, t_{15})	1	1	2
(t_1, t_{99})	2	0	2
(t_3, t_4)	2	0	2
(t_3, t_{15})	0	2	2
(t_3, t_{99})	1	1	2
(t_4, t_{15})	0	2	2
(t_4, t_{99})	0	2	2
(t_{15}, t_{99})	1	1	2

Multiple orderings may have the same maximum agreement score. In such cases, the final ordering is determined by averaging ranks obtained for each test-case across all iterations that lead to optimal orderings. Time complexity of this approach is $O(N \times 2^M)$. Typical ranges of parameters we observed in our empirical evaluation are: $0 \leq N \leq 200$, and $1 \leq M \leq 7$.

Example

We now illustrate this with an example. Let us consider a regression test-suite $\{t_1, t_3, t_4, t_{15}, t_{99}\}$ and a profile $R = \{p_1, p_2\}$ consisting of two prioritizations: $p_1 = \langle t_{15}, t_3, t_1, t_{99}, t_4\rangle$, and $p_2 = \langle t_1, t_{99}, t_{15}, t_3, t_4\rangle$. The KY-consensus (Press, 2012) gives rise to multiple Kemeny optimal orderings, two of which are $p^* = \{\langle t_1, t_{15}, t_{99}, t_3, t_4\rangle, \langle t_{15}, t_1, t_3, t_{99}, t_4\rangle\}$ both having the maximum agreement score of 16. In terms of Kendall-tau distance, the minimized sum (disagreement)[6] amounts to 4.

Pairwise preference counts are shown in Table 6. In the table, '$x \rightarrow y$' denotes that test-case x is preferred over test-case y. The last column represents the grand total of supporting and opposing votes for a pair. Hence, for each row, the entry corresponding to the last column reflects the total number of voters, which for this example is 2.

The calculation of the agreement score for $\langle t_1, t_{15}, t_{99}, t_3, t_4\rangle$, where $|t_i \rightarrow t_j|$ denotes the count of votes for the associated preference.

$$
\begin{aligned}
&\text{Agreement score}\\
&= |t_1 \rightarrow t_{15}| + |t_1 \rightarrow t_{99}| + |t_1 \rightarrow t_3| + |t_1 \rightarrow t_4|\\
&+ |t_{15} \rightarrow t_{99}| + |t_{15} \rightarrow t_3| + |t_{15} \rightarrow t_4|\\
&+ |t_{99} \rightarrow t_3| + |t_{99} \rightarrow t_4|\\
&+ |t_3 \rightarrow t_4|\\
&= (1 + 2 + 1 + 2) + (1 + 2 + 2) + (1 + 2) + 2 = 16
\end{aligned}
$$

The corresponding disagreement score is as follows:

$$
\begin{aligned}
&\text{Disagreement score}\\
&= KT(\langle t_1, t_{15}, t_{99}, t_3, t_4\rangle, \langle t_{15}, t_3, t_1, t_{99}, t_4\rangle)\\
&+ KT(\langle t_1, t_{15}, t_{99}, t_3, t_4\rangle, \langle t_1, t_{99}, t_{15}, t_3, t_4\rangle)\\
&= 3 + 1 = 4
\end{aligned}
$$

The calculation of agreement score for $\langle t_{15}, t_1, t_3, t_{99}, t_4\rangle$, where $|t_i \rightarrow t_j|$ denotes the count of votes for the associated preference.

$$
\begin{aligned}
&\text{Agreement score}\\
&= |t_{15} \rightarrow t_1| + |t_{15} \rightarrow t_3| + |t_{15} \rightarrow t_{99}| + |t_{15} \rightarrow t_4|\\
&+ |t_1 \rightarrow t_3| + |t_1 \rightarrow t_{99}| + |t_1 \rightarrow t_4|\\
&+ |t_3 \rightarrow t_{99}| + |t_3 \rightarrow t_4|\\
&+ |t_{99} \rightarrow t_4|\\
&= (1 + 2 + 1 + 2) + (1 + 2 + 2) + (1 + 2) + 2 = 16
\end{aligned}
$$

[6] This score is called Kemeny score (Kemeny, 1959; Lv, 2014).

The corresponding disagreement score is as follows:

Disagreement score
$= KT(\langle t_{15}, t_1, t_3, t_{99}, t_4\rangle, \langle t_{15}, t_3, t_1, t_{99}, t_4\rangle)$
$+ KT(\langle t_{15}, t_1, t_3, t_{99}, t_4\rangle, \langle t_1, t_{99}, t_{15}, t_3, t_4\rangle)$
$= 1 + 3 = 4$

We now consider another ordering $\langle t_{15}, t_3, t_1, t_4, t_{99}\rangle$ and calculate its agreement and disagreement scores.

The calculation of agreement score for $\langle t_{15}, t_3, t_1, t_4, t_{99}\rangle$, where $|t_i \to t_j|$ denotes the count of votes for the associated preference.

Agreement score
$= |t_{15} \to t_3| + |t_{15} \to t_1| + |t_{15} \to t_4| + |t_{15} \to t_{99}|$
$+ |t_3 \to t_1| + |t_3 \to t_4| + |t_3 \to t_{99}|$
$+ |t_1 \to t_4| + |t_1 \to t_{99}|$
$+ |t_4 \to t_{99}|$
$= (2 + 1 + 2 + 1) + (1 + 2 + 1) + (2 + 2) + 0 = 14\ (\downarrow\ \text{by } 2)$

The corresponding disagreement score is as follows:

Disagreement score
$= KT(\langle t_{15}, t_3, t_1, t_4, t_{99}\rangle, \langle t_{15}, t_3, t_1, t_{99}, t_4\rangle)$
$+ KT(\langle t_{15}, t_3, t_1, t_4, t_{99}\rangle, \langle t_1, t_{99}, t_{15}, t_3, t_4\rangle)$
$= 1 + 5 = 6\ (\uparrow\ \text{by } 2)$

The discrepancies (by value 2) are due to the fact that none of the individual rankings prefers t_4 to t_{99}, as denoted by a zero count in Table 6. Hence, $\langle t_{15}, t_3, t_1, t_4, t_{99}\rangle$ is not a Kemeny optimal ordering. Six optimal orderings are possible in this example: $\langle t_1, t_{15}, t_3, t_{99}, t_4\rangle$, $\langle t_1, t_{15}, t_{99}, t_3, t_4\rangle$, $\langle t_{15}, t_1, t_3, t_{99}, t_4\rangle$, $\langle t_{15}, t_1, t_{99}, t_3, t_4\rangle$, $\langle t_1, t_{99}, t_{15}, t_3, t_4\rangle$, and $\langle t_{15}, t_3, t_1, t_{99}, t_4\rangle$, with optimal agreement and disagreement scores of 16 and 4, respectively. In general, the Kemeny–Young consensus gives *an* optimal ordering rather than *the* optimal ordering. The latter is possible only when there are no ties in pairwise majority preference counts or in other words, for any pair (i, j), the inequality: $|i \to j| \neq |j \to i|$ holds.

Any KY-ordering is a strictly specified ordering or is a total order (Dwork et al., 2001; Young and Levenglick, 1978) because it additionally satisfies the extended Condorcet criterion (Truchon, 1998). The extension can be stated in terms of consensus test-prioritization as follows. If a consensus ordering p^* is partitioned into left and right sub-orderings: p^*_l and p^*_r, then for all the test-cases $t_i \in p^*_l$ and $t_j \in p^*_r$, t_i can precede t_j in the consensus only when it wins by majority of individual prioritizations that prefer it that way.

3.3.2. Prioritization by Borda-count method

Kemeny–Young consensus has a high time complexity and is favorable when regression test-suites are small in size i.e., in hundreds. From the scalability perspective, a more time-efficient quick consensus approach is the Borda-count method (de Borda, 1781; Dwork et al., 2001; Lv, 2014; Davenport and Kalagnanam, 2004), This method assigns to each test-case t the number of test-cases it defeats in each prioritization in R. This amounts to number of remaining test-cases occurring to the right of t, in each ranking. This score is calculated for all the test-cases and the final ordering is determined by test-cases arranged in decreasing order of the Borda-count score. It has been shown that the Borda-count method does not satisfy the Condorcet principle (Young and Levenglick, 1978; Davenport and Kalagnanam, 2004). The Borda-count method is outlined in Algorithm 2.

The algorithm takes as input the profile R of m individual prioritizations, and the selected regression test-suite σ containing n test-cases. Line 1 initializes the consensus permutation p^* by setting BC-scores of all the test-cases to zero. We consider this computation as a preprocessing step which takes $O(n)$ time to execute. The main computation begins with the **for** loop of lines 2–4 which iterates n times over all the selected test-cases in σ. For each iteration, line 3 executes in $O(mn + m)$ time, where the breakdown is as follows. BC-score of each test-case is determined in $O(n)$ time for each of the m prioritizations. The summation is subsequently computed in $O(m)$ time. The sorting by scores in line 5 executes in $O(n \log_2 n)$ time. Overall, the time complexity of Algorithm 2 turns out to be $O(mn^2 + mn + n \log_2 n)$.

Algorithm 2: Borda-count Consensus Prioritization

Input: Prioritized-permutations: $R = \{p_1, ..., p_m\}$, Selected-tests: $\sigma = \{t_1, ..., t_n\}$
Output: A Prioritized-permutation p^*, consensus of $\{p_1, ..., p_m\}$
1 p^* := initialize();
2 **for** $t \in \sigma$ **do**
  /* sum of Borda-counts across all prioritizations in the profile R */
3 BC-score $(p^*.t)$:= $[\{n - \rho(p_1.t)\} + ... + \{n - \rho(p_m.t)\}] \times (-1)$;
4 **end**
/* sort test-cases based on BC-scores */
5 p^* := reorder-tests-by-BC-scores(p^*);
6 **return** p^*;

For our example regression test-suite of five test-cases: $\{t_1, t_3, t_4, t_{15}, t_{99}\}$ and a profile $R = \{p_1, p_2\}$ consisting of two prioritizations: $p_1 = \langle t_{15}, t_3, t_1, t_{99}, t_4\rangle$, and $p_2 = \langle t_1, t_{99}, t_{15}, t_3, t_4\rangle$. the BC-score is computed as follows:

$$\text{BC-score}(t_1) = 2 + 4 = 6$$
$$\text{BC-score}(t_3) = 3 + 1 = 4$$
$$\text{BC-score}(t_4) = 0 + 0 = 0$$
$$\text{BC-score}(t_{15}) = 4 + 2 = 6$$
$$\text{BC-score}(t_{99}) = 1 + 3 = 4$$
$$\textit{Borda} - \textit{count consensus} = \langle [t_1, t_{15}], [t_3, t_{99}], t_4\rangle$$

For t_1 the BC-score from p_1 is 2 as it defeats two test-cases $\{t_{99}, t_4\}$. From p_2 the BC-score is 4 as it defeats all remaining four test-cases. Therefore, the total BC-score for t_1 in R turns out to be 6 (2+4). The final consensus ordering due to Borda-count method contains ties for rank 1 ($[t_1, t_{15}]$) and rank 2 ($[t_3, t_{99}]$).

It is to be noted that construction of the final consensus ordering obtained by KY and BC are performed in a different manner. While KY involves optimizing Kendall-tau distance, BC aggregates defeats by positions in individual rankings. While KY satisfies the Condorcet criterion, it is highly time-inefficient. On the other hand BC is comparatively time-efficient but does not satisfy the Condorcet criterion.

3.3.3. Consensus prioritization approximation by means

The time complexity of the Borda-count method is appreciable but different formulation of rank aggregation lies in the underlying aggregating function. Hansie utilizes an alternative aggregation by arithmetic means of ranks having same asymptotic runtime as Borda-count. It is to be noted that statistical means having similar running time may differ in the final consensus ordering. Hence, each aggregating function gives rise to a potentially different consensus ordering.

Consensus by arithmetic mean (AM) of ranks is outlined in Algorithm 3 which accepts as input the profile R of m individual prioritizations, and the selected regression test-suite σ containing n test-cases. Line 1 initializes the consensus permutation p^* by setting ranks of all test-cases to zero. We consider this computation as a preprocessing step which takes $O(n)$ time to execute.

Algorithm 3: Rank Aggregation by Arithmetic Mean

Input: Prioritized-permutations: $R = \{p_1, ..., p_m\}$, Selected-tests: $\sigma = \{t_1, ..., t_n\}$
Output: A Prioritized-permutation p^*, consensus of $\{p_1, ..., p_m\}$ which is approximate/sub-optimal

```
1 p* := initialize();
2 for t ∈ σ do
    /* arithmetic mean of ranks across all prioritizations in
       the profile R                                          */
3   ρ(p*.t) := arithmetic-mean(ρ(p_1.t), ..., ρ(p_m.t));
4 end
  /* sort test-cases based on revised rank                   */
5 p* := reorder-tests-by-rank(p*);
6 return p*;
```

The main computation begins with the **for** loop of lines 2–4 which iterates n times over all selected test-cases in σ. For each iteration, line 3 executes in $O(mn + m)$ time, where the breakdown is as follows. Rank of each test-case is determined in $O(n)$ time for each of the m prioritizations. The `arithmetic-mean` is subsequently computed in $O(m)$ time. The sorting by ranks in line 5 executes in $O(n \log_2 n)$ time. Overall, the time complexity of Algorithm 3 turns out to be $O(mn^2 + mn + n \log_2 n)$.

Initial approximation by other means: harmonic mean (HM), geometric mean (GM), and median (med) can be performed by changing the aggregation function in line 3. For our example profile R, the initial approximate consensus due to AM, GM, HM, and median turn out to be same. We show the computation only for aggregation by arithmetic mean.

$$\rho(t_1) = AM(3, 1) = 2$$
$$\rho(t_3) = AM(2, 4) = 3$$
$$\rho(t_4) = AM(5, 5) = 5$$
$$\rho(t_{15}) = AM(1, 3) = 2$$
$$\rho(t_{99}) = AM(4, 2) = 3$$

$$\textit{AM initial approximation} = \langle [t_1, t_{15}], [t_3, t_{99}], t_4 \rangle$$

For our example, the initial approximation contains ties for ranks 1 ($[t_1, t_{15}]$) and rank 2 ($[t_3, t_{99}]$).

Issue with sequential execution

A sequential test-case execution necessarily requires breaking ties. Existing modeling of the test execution effectiveness is unable to capture ties as ties. Thus, a given set of test-cases may have multiple optima, while the sequentiality enforces only one. Therefore, a different optimal sequence (respecting the ties) would be cast as different from *the* optimal sequence. This is clearly a limitation of the existing models. In our example, if the partial ordering $\langle [t_1, t_{15}], [t_3, t_{99}], t_4 \rangle$ is viewed as a template, we see that it satisfies: (i) 4 out of 6 Kemeny optimal final orderings Eq. (8), (ii) final consensus ordering computed by Borda-count method (Algorithm 2), and (iii) initial consensus approximation by AM (Algorithm 3), GM, HM, and median.

Strengthening of the template ordering leads to a compulsion of sequentiality in the final consensus ordering as ultimately one out of many candidate optimal orderings has to be selected (in a sequential test execution). The problem with such a selection is that different optimal orderings may exhibit differences in effectiveness of prioritization (Section 4.1.7) depending on the position of failure due to a chosen optimal ordering. This may give rise to observed discrepancies in effectiveness.

Parallelization windows for multiple consensuses

The above described situation can be remedied by executing the tied segments in a consensus using parallelization windows which respects tied ranks by giving them parallel ranks. In other words, respecting the template, we can execute multiple optimal orderings together. This turns out to be faster than the state-of-the-art consensus approach described Kemeny–Young, and a similar complexity as that of the Borda-count method. However, the latter is a position-based method which may not satisfy the Condorcet principle.

In fact, if any prioritization contains ties, they need not be broken as proposed in a recent work (Eghbali et al., 2019), although their approach is orthogonal to ours. For our example, we enforce social fairness to all individual prioritizations and at the same time to their consensus p^*, by employing parallel execution windows of size $(2, 2, 1)$, in that order. The tester may also in some situations, make a consensus ordering stronger by enforcing sequential execution, in which case the supervised tie-breaking approach stated in Eghbali et al. (2019) can also be employed. Other criteria for tie breaking may be the cost of a test-case, risk based, cache-friendliness, to name a few. There may also be scenarios where the tester may need to further weaken the consensus ordering to enjoy parallelization benefit.

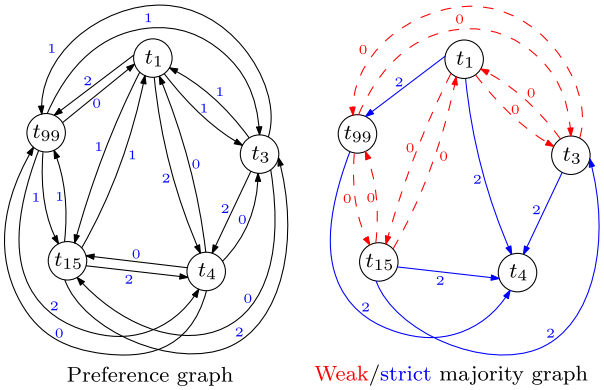

Fig. 8. Preference graph and its (weak/strict) majority graph for profile R containing $p_1 = \langle t_{15}, t_3, t_1, t_{99}, t_4 \rangle$, and $p_2 = \langle t_1, t_{99}, t_{15}, t_3, t_4 \rangle$. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Non-uniqueness of consensus prioritizations

We now describe some properties of ranking profiles that provide necessary and sufficient criteria toward non-existence of a unique consensus orderings. We begin with our example profile R of prioritizations, $p_1 = \langle t_{15}, t_3, t_1, t_{99}, t_4 \rangle$, and $p_2 = \langle t_1, t_{99}, t_{15}, t_3, t_4 \rangle$. A preference graph (PG) (Davenport and Kalagnanam, 2004) $G_R = (V_R, E_R)$ is a mathematical model of representing a profile of rankings R. The set of vertices, V_R, consists of distinct test-cases in regression test-suite. With $t_i, t_j \in V_R$, the directed edge $t_i \rightarrow t_j$ is weighted by the number of prioritizations preferring (voting) t_i to be executed before t_j. The reverse edge accounts for the number of oppositions vetoing this order.

Fig. 8 (left) represents the preference graph for our example prioritization profile, R. Note that $t_{15} \rightarrow t_4$ weighs 2 as both p_1 and p_2 prefer it that way. Associated reverse edge in this case happens to weigh zero. For $t_1 \rightarrow t_{15}$ one prefers but the other dis-prefers leading to edge weights of 1 in both directions. The corresponding strict majority graph (SMG) consists of the same set of vertices V but contains edges from $t_i \rightarrow t_j$ if and only if $|t_i \rightarrow t_j| > |t_j \rightarrow t_i|$, where $|\cdot|$ represents the edge weight. In Fig. 8 (right), these edges are highlighted in blue. A weaker form of majority graph relaxes the edge membership criteria to $|t_i \rightarrow t_j| \geq |t_j \rightarrow t_i|$. Thus, the weak majority graph (WMG) is a super-graph of the associated strict majority graph. Additional

edges are highlighted (red dashed) in Fig. 8 (right). All edges $t_i \rightarrow t_j$ belonging to majority graphs weigh $|t_i \rightarrow t_j| - |t_j \rightarrow t_i|$. In our example, we observe that in the WMG, t_1 and t_{15} are mutually 1-reachable from each other forming a cycle having edge weight zero. In the associated SMG, such edges cease to exist.

Let us now try to construct a consensus ordering based on the Condorcet principle and its extension. Condorcet winner can be identified by the vertex in SMG that has lowest in-degree of zero. This means that no other vertex (Condorcet alternative) is more preferred. The vertex with the highest in-degree of $n-1$ is the Condorcet loser, where n is the number of vertices. If such vertices are unique in the SMG in the sense that no two vertices have in-degree zero or $n-1$, then there exist unique Condorcet winner and loser (Davenport and Kalagnanam, 2004). Also, in the SMG, t_1 and t_{15} are mutually disconnected and there is a tie between whether t_1 should be preferred to t_{15} or the other way.

In our example, t_1, t_{15} both have first rank in the initial consensus construction as there is no unique Condorcet winner. t_4 occurs at the last position as it is the Condorcet loser. Thus, after stage 1, our consensus template is $p^* = \langle [t_1, t_{15}], X, t_4 \rangle$, where X is the sub-ordering that needs to be determined further. Stage 2 begins by removing t_1, t_4, t_{15} from the majority graph. The SMG now mutually disconnects t_3 and t_{99}, and once again a tie arises in their relative ordering in X. After stage 2 of reduction, the refined consensus is $p^* = \langle [t_1, t_{15}], [t_3, t_{99}], t_4 \rangle$.

The resulting consensus template is a partial order containing ties at ranks 1 and 2. We now state some general observations regarding ambiguities in consensus orderings (regression test-case prioritizations).

`Observation 1`: If at any stage in the reduction of a strict majority graph with n vertices, there exist more than one vertex such that their in-degrees are zero and they are mutually disconnected, then all such nodes are rank-tied at a given position in that stage.

- If at stage 1, multiple such nodes exist, there in no unique Condorcet winner.
- If at stage 1, exactly one such node exists, then it is the unique Condorcet winner.
- If at stage 1, there is exactly one node with in-degree equal to $(n-1)$, then it is the biggest Condorcet loser.

`Observation 2`: In the absence of cycles in SMG, ties may be present in the consensus template. In such a case, the corresponding WMG has cycles; see Fig. 8.

`Observation 3`: When cycles are present in SMG, Kemeny–Young consensus will break them (arbitrarily) and report a strict ordering as a consensus.

`Observation 4`: All the majority graphs that lead to no ties exhibit the following property: {All prioritizations rank all test-cases without ties} $\Leftrightarrow$ {Every pair of vertices in the undirected WMG is 1-reachable}.

`Observation 5`: Necessary and sufficient condition for no ties: {All prioritizations rank all the test-cases without ties, and WMG is a DAG with positive edge weights} $\Leftrightarrow$ {Ties are not present in the consensus template/initial approximation/rank of alternatives/final consensus}.

`Observation 6`: Relation between ties in ranks and cycles:

- {Cycles in WMG} $\Leftrightarrow$ {Ties in initial consensus approximation/finding ranks/consensus template}
- A Kemeny order is a strong order where no ties are present. Algebraically, it occurs when it is a total order.[7]

`Observation 7`: Properties of weak majority graphs: If there is an edge $t_i \rightarrow t_j$ in WMG, with weight zero, then that edge is *not absent* in the graph. Prima facie, such an edge may seem useless, but is a prerequisite for cycles and ties in determining a Condorcet alternative (test-case) at a particular position of the consensus. Cycles are broken in the graph by removing min-weighted feedback edge set which is NP-hard (Davenport and Kalagnanam, 2004). Equivalently, Kemeny–Young consensus does this by minimizing sum of Kendall-tau distances. The latter one also being NP-hard for even four voters (Davenport and Kalagnanam, 2004; Dwork et al., 2001) or prioritizations. Therefore, in practice, heuristics are applied for finding *approximate* Kemeny order, which is not unique. In this situation, with Hansie, we need not break cycles in WMG. We need only perform initial quick approximation by means of ranks (Algorithm 3), possibly containing ties.

This lightweight computation of a quick approximate consensus template is, in general, a weak (partial) ordering.[8] This partial order need not be refined (i.e., enforced into a total order which is inherently strict) further to arrive at a strongly ordered consensus. This design choice helps us avoid computational overheads of breaking ties arising out of consensus prioritization, finding preferences, stochastically computing sub-lists in search of refined permutations with better Kemeny scores, etc. improving overall performance.

4. Experimental results

We evaluated Hansie using twelve real-world programs from Software-artifact Infrastructure Repository (Dataset #1) (Do et al., 2005), used extensively in academic software testing research, and eight open-source projects from GitHub (Dataset #2). Our evaluation[9] addressed the following five research questions.

- **RQ1 (hybridization effectiveness)**: What is the impact of relevance-weight (α) on hybridization?
- **RQ2 (traditional versus consensus effectiveness)**: How effective are consensus prioritizations compared to traditional (isolated) prioritizations?
- **RQ3 (comparison with state-of-the-art)**: How does Hansie perform against existing state-of-the-art prioritization strategies?
- **RQ4 (execution parallelization)**: What are the effects of parallelization windows on consensus test-orderings?
- **RQ5 (consensus quality)**: How does Hansie distance affect the quality of consensus in terms of agreement?

4.1. Experimental setup

We evaluated Hansie on an Intel Xeon CPU E5-2640 v4 system with 20 physical cores (40 logical cores) clocked at 2.40 GHz, with 64 GB RAM that ran CentOS Linux release 7.5.1804 (Core) 64-bit operating system. The kernel version was 3.10.0–693.17.1.el7.x86_64.

The subjects in our evaluation were compiled using the `clang 3.9.0` frontend of `LLVM`. We instrumented the subject programs (for coverage map generation) by writing an LLVM pass. The implementation of Kemeny–Young consensus was obtained from Press (2012), and the rest of the framework was written in C++.

[7] There are cases when ties can be present in a Kemeny order when individual orderings have tied ranks for some test-cases. However, this situation does not arise in our case as individual prioritizations are considered to be totally ordered.

[8] Note that if this initial ordering happens to be strictly specified, then it is the unique Kemeny ordering or the only possible consensus.

[9] Replication package: https://doi.org/10.5281/zenodo.3988245.

Table 7
Characteristics of benchmarks from SIR (Dataset #1).

Benchmark	SLOC (total)	#Ver-pairs	#Tests	#Fun.	ACCN
flex	8737 (50579)	4	670	151	9.2
grep	7988 (55218)	5	809	144	12.6
space	6200 (35710)	5	13585	136	6.5
sed	5504 (34616)	4	370	83	15.2
gzip	4342 (29353)	5	214	30	10.0
replace	512 (3068)	5	5542	21	5.5
printtokens	475 (1967)	5	4130	18	5.6
printtokens2	401 (2170)	5	4115	19	5.3
totinfo	346 (1762)	5	1052	7	6.7
schedule2	297 (1604)	5	2710	16	4.0
schedule	292 (1749)	5	2650	18	3.0
tcas	135 (810)	5	1608	9	3.8

Table 8
Characteristics of projects from GitHub (Dataset #2).

Project	#Stars	SLOC (total)	#Ver-pairs	#Tests	#Fun.	ACCN
gravity[a]	3011	16312 (29423)	5	923	873	5.17
xxhash[b]	3778	3684 (12534)	5	15621	118	3.95
scd[c]	1015	2167 (9155)	5	9261	22	23.68
xc[d]	1203	1026 (6264)	5	800	13	19.92
mlisp[e]	584	715 (4238)	5	3721	58	3.27
slre[f]	458	688 (2274)	5	132	17	15.35
ckf[g]	233	472 (2719)	5	222	16	5.31
c4[h]	4897	462 (2317)	5	1170	4	64.75

[a] https://github.com/marcobambini/gravity.
[b] https://github.com/Cyan4973/xxHash.
[c] https://github.com/cr-marcstevens/sha1collisiondetection.
[d] https://github.com/lotabout/write-a-C-interpreter.
[e] https://github.com/rui314/minilisp.
[f] https://github.com/cesanta/slre.
[g] https://github.com/begeekmyfriend/CuckooFilter.
[h] https://github.com/rswier/c4.

When test-cases had the same priority score, ties were broken by the instability of (i) the C `qsort()` function for individual (stand-alone) prioritizations, and (ii) C++ STL `sort()` function for consensus prioritizations.

Hansie was compiled using g++ 5.3.1 with support for OpenMP. Parallelization was implemented via `#pragma omp parallel for` employed with k parallel iterations at a time, where k is the window size. Inside each iteration, a call to `system()` with appropriate command-line arguments was applied on a copy of the executable under test. Execution timing of each window took into account differences between in-code timers `gettimeofday()` enclosed outside the body of each iteration. This took into account overheads of implicit barrier synchronization preceding the last statement of each iteration. It is to be noted that all executions of parallel code were preceded by an explicit onset of thread pinning via the shell command: `export OMP_PROC_BIND=true`.

We also installed libraries for building 32-bit executables on 64-bit systems using `-m32` flag, as two benchmarks {`c4`, `xc`} were 32-bit applications.

Characteristics of benchmarks in our experiments, denoted as Dataset #1 and Dataset #2, are presented in Tables 7 and 8, respectively. In Table 7, the first column shows the benchmark name. Number of physical source lines of code (SLOC) associated with the base version with total SLOC across all versions in parentheses '(·)', number of version pairs[10] for each program, total number of test-cases in the test-pool, number of constituent functions in the program are listed in subsequent columns. The last column mentions the average cyclomatic complexity number (ACCN) (McCabe, 1976) for each benchmark which reports its code structure complexity. The rows are sorted by SLOC (base version). In Table 8, the additional column (#Stars) denotes the number of stars received by a project in GitHub, denoting its popularity in open-source contribution.

4.1.1. Properties of SIR subjects (Dataset #1)

Twelve real-world artifacts from SIR (Do et al., 2005) were chosen for experiments. Out of these nine subjects, five were of type *single-source multiple-versions*, where gold output was associated with the base version. The remaining four were *sequential versions* corresponding to various field releases. In this case, gold output was bound with clean releases and the associated faulty version was tested.

4.1.2. Properties of GitHub projects (Dataset #2)

We manually browsed the GitHub website for selecting pure C projects (language distribution: C=100%) for Dataset #2. The original search was performed by specifying projects where at least C language is used. The next refinement specified C as the predominant language, with search results sorted by "most-stars". As the result (before applying exclusion criteria) returned by GitHub was time varying, we observed that there were ≈275,000 available repositories.[11] The results of the query were grouped by 10 per page. We browsed the first ≈90 pages of search result (≈900 repositories) and further eliminated repositories: (i) which were not proper software projects, and (ii) use generalized build system such as `make`, `cmake`, etc. since these are currently unsupported in Hansie.

After this short-listing, projects with at least one of the following properties were eliminated: (i) very few test-cases (less than 100), (ii) absence of `main()` function in the program, (iii) usage of thread-level or process-level parallelism, and (iv) usage of non-deterministic functions such as usage of `rand()`,[12] and those that exhibited time-varying outputs.

We focused on projects that used a simple build system (that is, direct compilation with appropriate compilation flags), we identified 13 projects which interfaced well with our implementation of Hansie. Five out of these 13 had flaky tests, and inherent non-determinism, which was verified by applying Hansie on them. The remaining 8 projects were included in Dataset #2, and had *sequential versions* associated with repository commits. Starting from the latest revision, earlier commits involving code changes and successful builds, were hand-picked. Gold output corresponded to the oldest revision. Each test-case belonged to the test-suite of the latest revision.

4.1.3. End goal of our software testing strategy

We now state essential terminologies from existing works (Avizienis et al., 2004; Perez et al., 2017; Mondal and Nasre, 2019) that we adopted in Hansie.

- **Fault**: the underlying reason that causes an error.
- **Error**: a system status that may lead to a failure.
- ***Failure***: an event that indicates discrepancy between expected and observed outcome. Built on top of Mahtab (Mondal and Nasre, 2019), the nature of our test-oracle for Hansiewas detection of failures. This treatment was essential to maintain uniformity in effectiveness comparisons all of whose underlying assumption was *zero fault-knowledge*. Therefore, in this study, our goal of prioritization was to improve the *rate of failure observation*, unlike traditional fault detection which was pursued in Rothermel et al. (2001), Elbaum et al. (2002, 2001), Yoo and Harman (2007, 2012), Epitropakis et al. (2015).

[10] Number of version pairs (#Ver-pairs) is one less than that of versions.

[11] 288,504 by the same query on 18:14:27, Indian Standard Time, April 21, 2020.

[12] A function that returns a pseudo-random number, which may make the test-case flaky.

Table 9
Total failures, and faults associated with our subjects-under-test.

Dataset #1	Sel. (%)	#fails	#faults (ver.-wise)	Fault-multiplicity, type
flex	99.70	2649	62 (20+17+16+9)	Multiple, seeded
grep	3.13	69	57 (18+8+18+12+1)	Multiple, seeded
space	58.98	16938	5 (1+1+1+1+1)	Single, real
sed	99.00	501	18 (3+5+6+4)	Multiple, seeded
gzip	100.00	417	59 (16+7+10+12+14)	Multiple, seeded
replace	54.11	653	5 (1+1+1+1+1)	Single, seeded
printtokens	30.92	242	5 (1+1+1+1+1)	Single, seeded
printtokens2	28.09	1027	5 (1+1+1+1+1)	Single, seeded
totinfo	60.68	233	5 (1+1+1+1+1)	Single, seeded
schedule2	90.99	164	5 (1+1+1+1+1)	Single, seeded
schedule	66.54	704	5 (1+1+1+1+1)	Single, seeded
tcas	44.09	258	5 (1+1+1+1+1)	Single, seeded
Dataset #2				
gravity	5.33	35	Unknown	Unknown, real
xxhash	25.38	414	Unknown	Unknown, real
scd	100.00	1160	Unknown	Unknown, real
xc	11.18	1099	Unknown	Unknown, real
mlisp	100.00	2394	Unknown	Unknown, real
slre	30.58	14	Unknown	Unknown, real
ckf	100.00	249	Unknown	Unknown, real
c4	5.33	1369	Unknown	Unknown, real

4.1.4. Characterizing faults and failures in subject programs

Table 9 reports and compares failure and fault statistics for both Dataset #1 and #2. For Dataset #2, only the number of failures got revealed after execution of test-cases in a prioritized order. This synchronized with our evaluation setup which assumes zero fault-knowledge. The second column shows geometric mean test-selection (**Sel.**) across versions. In the parallel setting (RQ4), the test-selection ratio directly specifies the test-load in terms of end-to-end window execution time. Note that, combined with prioritization, the window configuration vector (size of consecutive windows) may give rise to load-imbalance during runtime.

In the third column, number of failures observed encompassing all versions shows that multiple test-cases failed due to a particular fault. The more closely-spread the failures occur (i.e., are observed), the better (Torres et al., 2019). This information may be utilized for fault-localization (Yoo and Harman, 2012).

The fourth column lists the total number of faults across all versions per subject. The version-wise break-up is shown alongside in parentheses. Each of the Siemens programs and `space` contained a single seeded fault per version. All the Unix utilities in our study were real-world corresponding to field releases and the associated multiple-faults[13] were seeded.

For the GitHub projects, no information was available for the number or type of real faults. The potential impact of fault-masking or subsumption, in which case a test did not fail in the presence of one or more faults simultaneously, was not pursed in this work.

Besides these statistics, other important observations for some benchmarks were as follows. For APFD, $APFD_c$, and EPS, zero values were observed in: 1 version for `grep` and `printtokens`; 2 versions for `mlisp`; and 3 versions for `xc`, `c4`. This means absence of failures was due to: (i) either the version was fault-free even after selective re-testing, (ii) or changes occurred in non-code artifacts with zero test-selection, in which case no basic-block was affected. For ECC, zero value was observed in: 1 version for `printtokens` and `xc`, and 2 versions for `c4`. This was again due to unchanged code with no test selection.

[13] Simultaneously enabled by C preprocessor macros in the header file, `FaultSeeds.h`, available from SIR.

Table 10
Disk-space overhead of associated full test-suites (Dataset #2).

Project	MB	KB	Avg.-MB-per-test	Avg.-KB-per-test
xxhash	510.25	522500.91	0.03	33.45
ckf	329.77	337688.86	1.49	1521.12
xc	22.66	23208.71	0.03	29.01
c4	20.52	21016.08	0.02	17.96
scd	18.77	19219.94	–	2.08
gravity	6.76	6920.58	0.01	7.50
mlisp	0.72	735.28	–	0.20
slre	0.01	12.91	–	0.10

Note. As our analyses in Hansie were fault-blind (failure aware), we did not assume existence of fault knowledge a priori before the testing exercise. To synchronize with this design decision, we did not distinguish between a version with faults and a version without faults. In some cases, changes definitely occurred in code components but they turned out not to be failure inducing, so no test-cases failed. Hence, APFD, APFDc, EPS remained zero for such scenarios.

4.1.5. Hindrance to effective parallelization

Both the datasets comprised of a few artificial characteristics responsible for causing sequential bottlenecks in parallelization (Section 4.5) due to disk-file processing. We discuss all such aspects below.

Disk-space consumed by test-suites: Dataset #1 was from the SIR repository where test-cases were *lightweight* in the sense that they did not consume much disk-space. The size reached a maximum of a few KBs. In Dataset #2, the sizes reached MBs. Table 10 (rows sorted by total test-suite size) presents a few relevant statistics. For subjects: {`scd`, `mlisp`, `slre`}, entries marked with (–) under **Avg.-MB-per-test** indicate that the consumed disk-space is too less to be counted in MBs.

Nature of subjects: Jointly considered, both the datasets had input formats: command-line only, command-line + file, and file only (purely file I/O). The first two modes were dominant in Dataset #1, whereas Dataset #2 consisted of only real-world files as input such as real-world C and LISP source-codes, PDF files, text files, other binary files, etc. Besides heaviness of test-suites (shown in Table 10), the other hindrance to parallelism arose from parallel execution of test-cases residing in the same file-system.

4.1.6. Compared state-of-the-art prioritization strategies

In isolation, we compare the hybridized and consensus prioritizations against three state-of-the-art prioritizations: (i) *adaptive random prioritization* (ART) (Jiang et al., 2009), (ii) *greedy additional*, and (iii) *greedy total*. It is to be noted that greedy additional and total are also employed as individual participants (stated in Table 1) in our consensus approaches.

The greedy total strategy was implemented in Mahtab (Mondal and Nasre, 2019) and in this work, as prioritization by *relevance*. The ART (Jiang et al., 2009) based prioritization is a randomized prioritization. This is augmented by the notion of Jaccard distances between test-coverages represented as sets.

Specifically in ART, we utilized the *max–min* approach to calculate distance between the candidate set and the already prioritized set of test-cases. This means the next candidate test-case picked from the candidate set was the one that has the largest minimum Jaccard distance from those in the prioritized set.

The prioritized set began as an empty set and was built incrementally. Following the algorithm presented in Jiang et al. (2009), the first test-case was randomly chosen. Thereafter, the candidate set was built as long as its size was within bounds or it was no longer possible to further increase the cumulative coverage of

the candidate set. We maintained the upper bound of 10 on the cardinality of the candidate set as suggested in prior studies (Chen et al., 2005; Jiang et al., 2009).

To combat noise due to randomization, our ART executions were repeated 30 times for each version-under-test, and we report the associated geometric mean values. Although empirically (Jiang et al., 2009) the ART-based prioritization achieves more effectiveness, the time complexity is $O(m^2)$ (best-case) and $O(m^3n)$ (worst-case), m and n being the number of test-cases and (changed) code elements, respectively. An extreme case was experienced for some of our benchmarks (9 out of 20) with either: (i) a lot of changes, or (ii) large test-suites ($m \approxeq n$), where the time complexity reached as high as $O(m^4)$. Therefore, we selectively report ART evaluations for some of the representative benchmarks.[14]

Additionally, to support our comparison, we qualitatively illustrate prioritization performance in terms of cumulative failure observation. Specifically, we present plots in two dimensions such as the one followed in Sánchez et al. (2014). Our differences from their work lie in: (i) keeping the axes unnormalized, and (ii) employing cumulative number of failures instead of faults. Although unnormalized, different prioritizations are simultaneously compared in the same scale for each plot. An advantage of using failure plots is to show the relative proximity of failing test-cases. The visualization shows how effectively a prioritization heuristic has ordered the spreading (Torres et al., 2019) of test-outcomes without prior fault information. In the optimal case, denoted as opt, failures are highly coalesced toward the beginning of the timeline.

4.1.7. Measuring effectiveness of prioritization

We now specify several metrics used for quantifying effectiveness of test prioritization both in the sequential and the parallel setting.

Average Percentage of Failures Detected (APFD)

APFD continues to be one of the standard and widely used (Miranda et al., 2018) metric to evaluate prioritization effectiveness. This is defined as follows:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2 \times n} \tag{9}$$

where n is the total number of test-cases, m is the number of failures, and TF_i is the position (in timeline) of the ith failure. APFD value ranges in [0..1]. A value closer to 1 denotes fruitful prioritization, indicating that the test-cases are ordered such that faults are detected early. The original definition of APFD specifies faults instead of failures. However, as the fault knowledge ceases to exist a priori, we followed the assumptions of Miranda et al. (2018), Chen et al. (2018), Yu et al. (2019), Mondal and Nasre (2019) by treating each failure as a different fault.

Cost-cognizant APFD ($APFD_c$)

APFD assumes all test-cases to have the same cost and all faults to have the same severity. However, a variant namely $APFD_c$ (Elbaum et al., 2001) takes into account cost of individual test-cases, and treats all faults as having uniform severity (Epitropakis et al., 2015). This is defined as follows:

$$APFD_C = \frac{\sum_{i=1}^{m} \left(\sum_{j=TF_i}^{n} C_j - \frac{C_{TF_i}}{2} \right)}{\left(\sum_{j=1}^{n} C_j \right) \times m} \tag{10}$$

where n is the total number of test-cases with costs $\{C_1, \ldots, C_n\}$, and m is the number of failures. TF_i is the position (in timeline) of the ith failure. $APFD_c$ value closer to 1 denotes fruitful prioritization.

Effectiveness of Change Coverage (ECC)

APFD and $APFD_c$ were originally defined to capture prioritization effectiveness when fault information is known. This situation occurs only in case of controlled experiments and not in the real setting. To overcome this issue, Mondal and Nasre (2019) introduced ECC, a variant of APFD which quantifies effectiveness based on faster coverage of changed code elements, and is free from the assumption of fault knowledge. Instead of requiring {test-cases}↔{faults} like APFD and $APFD_c$, ECC leverages {test-cases}↔{basic-blocks} mapping that is richly available when prioritization is employed alongside test-selection based on code-changes. The ECC metric is defined as follows:

$$ECC = 1 - \frac{\sum_{i=1}^{\delta} x_i}{n \times \delta} + \frac{1}{2 \times n} \tag{11}$$

where n and δ are total number of test-cases and the set of affected code-elements, respectively. x_i is the position (in timeline) of the first test-case, up to which all the affected elements are cumulatively covered. Similar to APFD and $APFD_c$, an ECC value closer to 1 implies better prioritization.

While APFD measures the rate of failure detection, ECC measures the rate of affected code coverage. Newly added basic-blocks fail to have any record of test-coverage corresponding to the older version (Mondal and Nasre, 2019). This is expected as these code elements are only associated with the new version-under-test. Further, a minor amount of newly added code would not affect the quantification of ECC significantly. However, major code revisions with new additions significantly affect ECC values. In this aspect, Mondal and Nasre (2019) mentioned that besides reflecting amount of affected-code coverage, ECC values additionally reflect the amount of new additions. A lower value says that a major revision has occurred and provides an indication to refresh the coverage map.

It is to be noted that alongside slower rate of affected code coverage, a lower value indirectly indicates amount of escaped (not executed) code changes. This can be translated as an indication to incorporate new test-cases besides existing regression tests.

Note. As per the findings of Mondal and Nasre (2019), disregarding change in globals may be safe for major revisions but not for minor changes. Since, in general, the scope of our study lied in reporting effectiveness of prioritization and not def-use analysis of global variables, our consistent mode of analysis happened to be `globals_off` (Mondal and Nasre, 2019) which took into account only the test-coverage mapping from {test-cases}→{basic-blocks}.

EPSilon (EPS) and EPSilon$_\omega$ (EPS$_\omega$)

The EPS metric (Mondal and Nasre, 2019) has been recently introduced for evaluating prioritization effectiveness completely in terms of position of failures along the execution timeline. This addresses two limitation of APFD: (i) assumption of prior fault-knowledge, and (ii) a strictly specified test-case ordering which in inherently sequential. The resulting metric also reduces space complexity required for computing the effectiveness.

Note that APFD mandatorily requires a fault matrix for computing effectiveness values. In EPS, failure events form vector of positions characterizing failed instants of the timeline. Similarity (complement of normalized Manhattan distance) of this vector with the optimal (consecutive failures starting from position one) prioritization quantifies effectiveness. EPS$_\omega$ is used in

[14] For others, the ART prioritization was terminated after a timeout of 1 h for the first version-pair. The overhead was due to our in-memory representation of test-coverage as a `set` of `strings` (basic-blocks), and potentially unoptimized set operations.

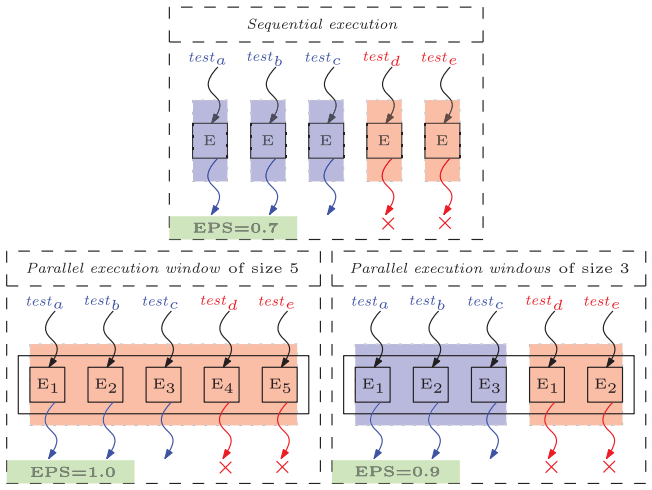

Fig. 9. An example (from Mondal and Nasre (2019)) illustrating boost in effectiveness (EPS) due to parallelization windows. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the presence of parallel test-execution windows. This metric is mathematically expressed as follows:

$$EPS_\omega = 1 - \left[\frac{\sum_{i=1}^{m} |P_i - P^+{}_i|}{\sum_{i=1}^{m} |P^-{}_i - P^+{}_i|} \right] \quad (12)$$

where P is the position vector of m failures, P^+ denotes best-case vector where failures occur consecutively from the beginning. The worst-case corresponds to P^- comprising of positions even beyond capacity of the full test-suite. The worst-case vector in the denominator normalizes the distance, which in turn is complemented in [0..1] for defining similarity with respect to the optimal vector P^+. EPS_ω value closer to 1 denotes fruitful prioritization.

Fig. 9 illustrates an example scenario where five test-cases $\{test_a, test_b, test_c, test_d, test_e\}$ are executed sequentially (top), using 5 cores with a window of size 5, (bottom left), and window of size 3 (bottom right). Similarity (EPS) with optimal prioritization in this example turns out to be 70% (sequential), 90% (window size 3), and 100% (window size 5). Note the fragmentation occurring for last window where two execution units E_1 and E_2 are reassigned $test_d$, and $test_e$. A window is marked *failed* if at least one residing test-case fails. Passing test-cases are marked in blue, whereas failed ones are shown in red (×). It is to be noted that EPS and EPS_ω are computed at the granularity of individual test-cases and batched mode (parallel window of test-cases), respectively.

For this example, parallelization windows help boost the effectiveness using appropriately chosen window size. We use the term *appropriate*, as a higher window size may still exhibit higher effectiveness due to parallelization but potential overheads of hardware resource contention, idle threads, hyper-threading in multi-processors, etc. are not accounted for in the EPS computation. This may bring down the benefit of parallelization with an increased end-to-end execution time, and be misleading if only EPS (EPS_ω) values are considered.

As an orthogonal metric, we subsequently propose EPsilon-Latency (EPL/EPL_ω) which considers such overheads and is intended to jointly reward prioritization effectiveness during parallel execution of prioritized test-cases.

EPsilon-Latency (EPL) and EPsilon-Latency$_\omega$ (EPL_ω)

While EPS_ω allows modeling parallel test-case execution as well as tied test-cases, it suffers from the limitation of not distinguishing between their costs. In practice, especially nearer a software release, it is not only the failure finding capability, but also the execution time of a test-case that is crucial. Within a limited time budget (which is not uncommon), product management would wish to choose test-cases based also on their execution cost.

EPS assumes that all the failing test-cases have uniform cost, whereas EPS_ω assumes all parallel windows to be perfectly load-balanced and that they have the same execution latency (load factor). To counter this limitation, we introduce a supplementary latency-cognizant metric (EPL_ω) that takes into account the cost of executed test-cases (windows) and quantifies overall timeline latency. It is calculated as diminishing log-weighted sum of test-case (window) costs executed at individual positions. Weights decrease as we move further along the timeline.[15]

It is to be noted that EPL is failure-blind unlike EPS/EPS_ω. Thus, EPL/EPL_ω, considers latency of all execution windows irrespective of whether a test-case fails inside a window. Thus, EPL operates at the granularity of a tie/window. Intuitively, the goal is to capture effectiveness of prioritization coupled with parallel execution from the perspective of end-to-end latency of the entire execution timeline that follows a predetermined schedule. This metric is defined as follows:

$$EPL_\omega = 1 - \left[\frac{c_1.\log_2(n+1) + c_2.\log_2 n + \cdots + c_n.\log_2 2}{c_{max}.\{\log_2(n+1) + \log_2 n + \cdots + \log_2 2\}} \right] \quad (13)$$

where n is the number of test-cases (windows), and c_i is the execution latency (cost) of test-case (window) at position i. With c_{max} as the cost of most heavily loaded test-case (window), the normalizing factor comprises of all test-cases (windows) having this worst-case load factor. The value $(n+1)$ is needed in $\log_2(n+1)$ to avoid discounting the last (right-most) window, which may be fragmented. It is to be noted that log-weights being constant for each position, the normalizing factor characterizes a timeline worse than the one with progressively decreasing load factors (c_is). The closer the normalized subtrahend is to zero, the lower the latency. Therefore, the final value is taken as a complement in [0..1]. EPL (EPL_ω) values closer to 1 denote fruitful prioritization having a balanced load distribution of the underlying execution timeline.

A test-case (window) with more load (imbalance) later in the timeline weighs less than the one appearing early in the timeline even if they have similar individual latencies. This penalizes the overall prioritization effectiveness when highly loaded tests (windows) execute too early in the timeline. This implies that two prioritizations p_x and p_y may have differing EPL (EPL_ω) values even if they have exactly the same EPS (EPS_ω) values. This is illustrated in Fig. 10 where two different permutations of ten selected regressions tests are executed using non-uniform and uniform parallel execution windows. Each permutation corresponds to a row, and EPS value remains 1 because all window failures (highlighted in red) are observed consecutively from the beginning of the timeline for each window configuration and its corresponding timeline. A window is marked "failed" if at least one residing test-case fails. Window sizes are specified at the top of the timeline. Prioritized permutation and obtained EPL and EPS values are presented at the bottom. Each E_i is a hardware execution unit, which may be viewed as cores in a multi-core processor.

At any given time, window size $1 \leq |\omega| \leq k$, k being the number of threads. Windows of size less than or equal to k can be re-assigned to cores where threads of a preceding window completed execution. For instance, first row corresponds to the permutation: $\langle t_1, t_3, t_5, t_2, t_8, t_7, t_4, t_9, t_{10}, t_6 \rangle$. Window sizes are

[15] We have particularly chosen logarithm as the weighing factor due to its short range that is not susceptible to arithmetic overflow.

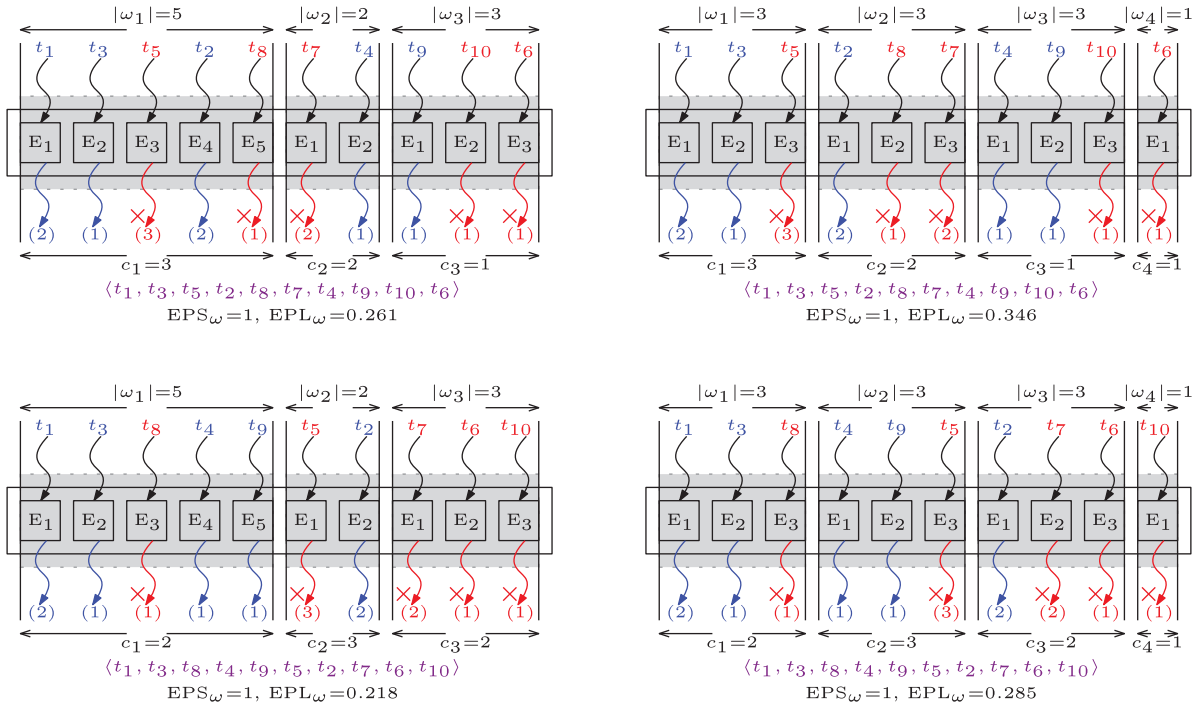

Fig. 10. A scenario where EPS_ω remains constant but EPL_ω varies due to different latencies arising from prioritizations, non-uniform, and uniform parallel execution windows. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

(5, 2, 3) for three parallel windows of non-uniform sizes. We see that ω_1 utilizes all five execution units, whereas the window ω_2 which should be executed immediately after ω_1 can be re-assigned two execution units E_1 (for t_7) and E_2 (for t_4) from window ω_1. Incoming threads to an execution unit is the executable system-under-test fed with a particular test-case as input. Outgoing threads correspond to execution termination with an output.

In Fig. 10, we annotate the outgoing arrowheads with the execution cost (latency) for that thread. Thus, for a parallel execution window, the latency is determined by the monopolizing (longest running) test-case. For ω_1 of first permutation (top left), the latency turns out to be 3 units, as rendered by the monopolizing test-case t_5. Passing test-cases are highlighted in blue whereas failing ones are highlighted in red with a cross-mark (×) near the arrowhead.

We observe that the window configuration (5, 2, 3) turns out to be better for the first prioritization: $\langle t_1, t_3, t_5, t_2, t_8, t_7, t_4, t_9, t_{10}, t_6 \rangle$. This is shown in Fig. 10 (first column). Overall timeline latency happens to be 6 units, disregarding barrier synchronization costs between successive windows with individual latencies (3, 2, 1). However, with the same set of window sizes, the second permutation exhibits an increased latency of 7 units. This is due to absence of windows of cost 1 which is otherwise present in the first permutation in the last window ω_3. Corresponding EPL values are 0.261 (first permutation) and 0.218 (second permutation). Similar explanation holds for Fig. 10 (second column), where uniform windows (except the last) of size (3, 3, 3, 1) are employed. Occasionally, the last window may appear fragmented when the number of test-cases in the schedule is not a multiple of the window size. In Fig. 10 (second column), the EPL values for the permutations are 0.346 and 0.285, respectively. For uniform windows, we observe two window latencies of cost 1 in the first case and one in the second. In general, the ideal approach would be to execute all test-cases of a particular latency inside one window, with lightly loaded windows appearing earlier.

Effect of shuffle on $APFD_c$ and EPL_ω

If we shuffle consecutive positions of a permutation, where test-cases are failing, APFD will remain the same but not $APFD_c$. This is because the latter additionally takes into account time to failure which can practically never be the same for any two test-cases. In EPL_ω, this can occur only when slicing (partitioning by windows) of the timeline happens in a way that splits the segment of consecutive positions into two consecutive windows. If shuffling the segment interchanges window members across the boundary resulting in changes of consecutive window latencies, the diminishing log-weighted overall latency for the timeline will be affected.

4.1.8. Measuring Consensus Quality

The *quality* of any consensus is determined by the underlying *distance* function used. The consensus problem may be geometrically viewed as a minimization problem in permutation spaces, P. In this aspect, the distance measures used in this work are indeed metrics in this space. The final consensuated ranking (solution point $p^* \in P$) is optimal if it is no longer possible to find another point $p' \in P$ with a shorter sum-total of distances ($dis(\cdot)$), such that the following inequality is satisfied.

$$dis(p', P) < dis(p^*, P), p' \neq p^* \tag{14}$$

If Inequality (14) is satisfied, then p^* in not an optimal consensus. If multiple, say f candidates $c_1, c_2, \ldots, c_f$ exist in P such that $dis(c_1, P) = \cdots = dis(c_f, P) = dis(p^*, P)$, then all such candidates are said to have tied for that consensus position. This aspect is input-driven and depends directly on the weak majority graph (WMG) associated with the profile of rankings. As this problem is NP-hard, our consensus computations are approximate. Likewise, the publicly available C++ implementation of Kemeny–Young consensus also employs approximations but different from ours.

Given a consensus ranking, p^*, with respect to a set of participating m individual ranked lists $L = \{p_1, \ldots, p_m\}$, each having

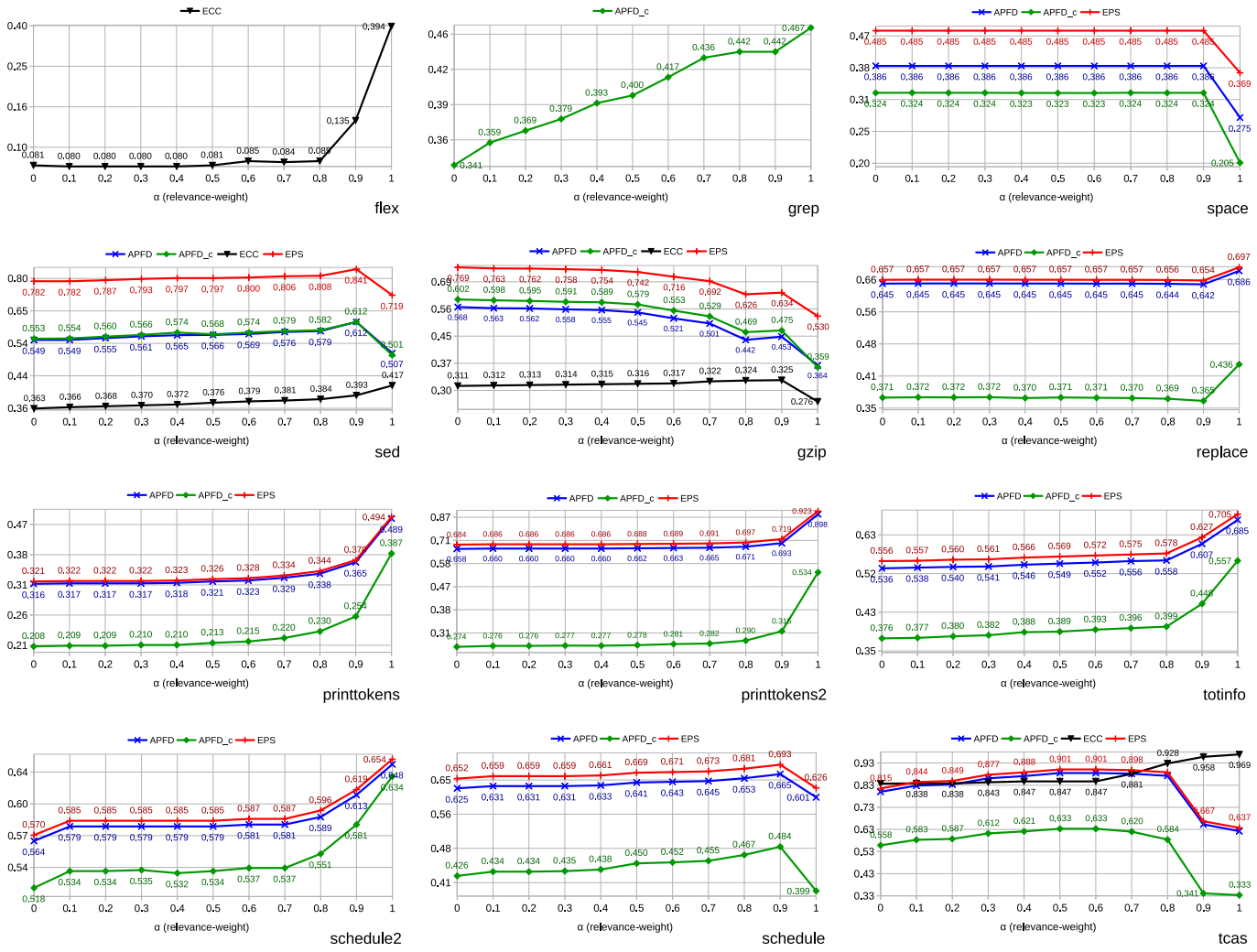

Fig. 11. Impact of relevance-weightage (α) on weighted-sum hybridized prioritization for Dataset #1. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

n candidates, the average consensus distance (ACD) of p^* with respect to L, is given by Eq. (15).

$$ACD(p^*, L) = \frac{1}{m} \times \sum_{i=1}^{m} dis(p^*, p_i) \tag{15}$$

$$NACD(p^*, L) = \overline{ACD}(p^*, L) \tag{16}$$

$$Q(p^*, L) = 1 - NACD(p^*, L) \tag{17}$$

Subsequently, Eq. (15) is normalized Eq. (16) based on the distance function used. For KT, the normalizer is $\binom{n}{2}$ (see Section 3.1). In case of HD, the normalizer is the Hansie distance between the label permutation and its reverse (see Section 3.2). Eq. (16) quantifies the overall normalized disagreement of p^* from L. The corresponding complement in [0, 1] Eq. (17) denotes the *quality* (magnitude of agreement) of p^* with respect to L. To adjudge the quality (Q) of consensus in our experiments, we introduce some evaluation parameters.[16]

- ***Q-cons-HD***: Eq. (17) with $dis()$ substituted by Hansie distance. p^* corresponds to the appropriate value of *cons*. The closer the value is to 1, the better is this quality.
- ***Q-ky-HD***: Eq. (17) with $dis()$ substituted by Hansie distance. p^* corresponds to Kemeny–Young consensus. The closer the value is to 1, the better is this quality.
- ***Q-cons-KT***: Eq. (17) with $dis()$ substituted by Kendall-tau distance. p^* corresponds to the appropriate value of *cons*. The closer the value is to 1, the better is this quality.
- ***Q-ky-KT***: Eq. (17) with $dis()$ substituted by Kendall-tau distance. p^* corresponds to Kemeny–Young consensus. The closer the value is to 1, the better is this quality.
- ***Approx***$_1$: Denotes the ratio by which the (p^*_{cons}) consensus approximates the Kemeny–Young (p^*_{kemeny}) consensus in terms of *sum-total of Hansie distances*. The lower the value, the better is this approximation. This is mathematically expressed as follows.

$$Approx_1 = \frac{\sum_{i=1}^{m} HD(p^*_{cons}, L_i)}{\sum_{i=1}^{m} HD(p^*_{kemeny}, L_i)} \tag{18}$$

- ***Approx***$_2$: Denotes the ratio by which the computed consensus (p^*_{cons}) approximates the Kemeny–Young (p^*_{kemeny}) consensus in terms of *sum-total of Kendall-tau distances*. The lower the value, the better is this approximation. This is

16 *cons* takes values in {ky, borda, am, gm, hm, med}.

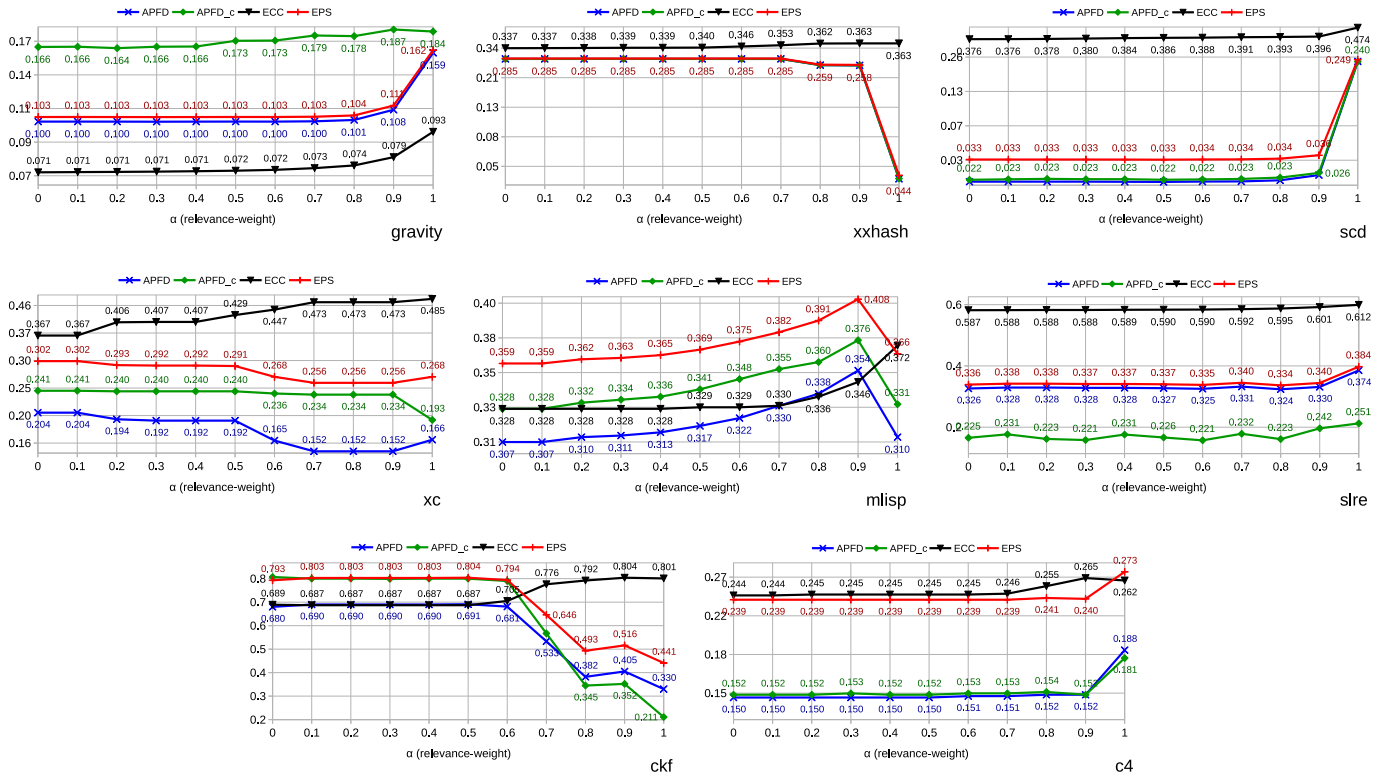

Fig. 12. Impact of relevance-weightage (α) on weighted-sum hybridized prioritization for Dataset #2. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

mathematically expressed as follows.

$$Approx_2 = \frac{\sum_{i=1}^{m} KT(p^*_{cons}, L_i)}{\sum_{i=1}^{m} KT(p^*_{kemeny}, L_i)} \tag{19}$$

4.2. RQ1: Impact of relevance-weight (α) on hybridization

Distribution of observed effectiveness values (APFD, $APFD_c$, EPS, ECC) was defined by the majority of relevance-weighted prioritizations. Interestingly, 11 out of 13 individual prioritizations in our study could be tuned by the relevance weightage α, and belonged to the same family. Figs. 11, and 12 show the impact of α on these 11 consensus participants, i.e., individual relevance-based prioritizations. In these plots, $\alpha = 0$ implies prioritization by confinedness (***conf.***), and $\alpha = 1$ implies prioritizations relevance (***rel.***). Intermediate values of α in increments of 0.1 encompassed the entire spectrum of hybridization by *relevance-confinedness*.

4.2.1. Discussion

In Figs. 11 and 12, we observe different effectiveness trends with increasing values of α. The respective trends arose due to not only the test-coverage distribution which formed the basis of our prioritizations but also the input values of test-cases themselves which our prioritization heuristics were not aware of. Note that two test-cases having the same priority may be change-coverage-wise identical but functionally (output-wise) different. A higher priority test-case may not fail whereas a lower priority test-case may fail only because the input–output relationship was not accounted for by the prioritization strategy. Besides these aspects, other role players were: number of changes in basic-blocks, test-selection ratios, and blocks (newly added) which could not be executed using the existing test-suite. For failure-aware metrics such as APFD, $APFD_c$, and EPS, a low value could have two interpretations. One was associated with a slow rate of failure observation, and the other involved non-observance of failures. The latter had two further variations. In the first case, a non-zero amount of test-selection was performed but the code changes were such that the intended system functionality was still intact. This implied that no selected regression test-cases failed. The second variation did not select any test-cases, in which case the `diff` had either occurred outside basic-blocks which we did not inspect in Hansie, or it truly reported that there were indeed no changes.

For ECC, the quantifications could be interpreted as described in Section 4.1.7. A low value indicated slower rate of affected code coverage besides having indicated a large amount of untested changed basic-blocks that arose in case of major updates. A higher value indicated faster convergence in covering all basic-blocks that were momentarily recorded in the test-coverage map. In general, the observed values of effectiveness measurements depended on the nature of changes that occurred in code revisions, criticality of the change, failure-and-fault mapping, and characteristics of the execution platform. Intuitively, an affected-block is critical if it is traversed by almost all test-cases, which leads to more testing effort (worst-case) than running all test-cases (RetestAll) (Yoo and Harman, 2012) alone.

A regression test-suite consisting of more number of selected tests may exhibit differences in end-to-end execution times depending on the caching effect. This depends on whether the prioritized permutation of test-cases has closer average distances in terms of basic-blocks so that the execution proceeds in a cache friendly manner (Stratis, 2017; Stratis and Rajan, 2018). In our case, this directly affected $APFD_c$ values as these took into consideration test-case execution time. Thus, a low $APFD_c$ score implied that early occurring non-failing test-cases were heavy, besides having indicated a fruitless prioritization.

Different test-cases may have similar coverage but with a change in stability of the sorting algorithm, insignificant differences in effectiveness may be observed. Note that ties in individual prioritizations arise due to the same priority score under the employed heuristic.

Answering RQ1 (hybridization effectiveness)

- ***Failure-observation rate***: For 12 out of 20 benchmarks, APFD, $APFD_c$, EPS increased with increasing relevance-weightage (α) up to $\alpha = 0.9$, and dipped thereafter. For 13 out of 20 benchmarks, APFD, $APFD_c$, EPS simultaneously showed strong correlation with increasing values of α.
- ***Effective hybridization***: For 17 out of 20 benchmarks, overall effectiveness of prioritization came with α lying in the range [0.5, 1].
- ***Change-coverage rate***: For 18 out of 20 benchmarks, ECC profiles were non-decreasing up to $\alpha = 0.9$, and dipped thereafter.

4.3. RQ2: Effectiveness of consensus prioritizations

This subsection reports and discusses the evaluation of consensus prioritization employed in Hansie. Additionally, the underlying data behind generation of the plots (in Figs. 11 and 12) from the previous subsection (RQ1), is elaborated here.

4.3.1. Layout of tabulated evaluation reports

We show the effectiveness values in our study in terms of various evaluation metrics (Section 4.1.7): (i) APFD, (ii) $APFD_c$, (iii) EPS, and (iv) ECC. Corresponding quantifications are presented in Tables 11, 12, 13, and 14, respectively. The horizontal partition comprises of three segments: partition 1 (column 1), partition 2 (columns 2–14), and partition 3 (columns 15–20). The first partition displays the benchmark name. The second partition comprises of quantifications for each prioritization strategy considered individually. The rightmost partition corresponds to consensus approaches and reports effectiveness measurements under appropriate columns labeling the names of consensus prioritizations.

There are 5 individual prioritization categories denoted as confinedness (***conf.***), relevance (***rel.***), cost-only (***cost***), greedy additional (***add.***), and weighted hybridization of ***(rel,con)*** denoted as (x, y) such that x% and y% weights are assigned to the underlying relevance and confinedness scores, respectively. The rightmost partition corresponds to 6 consensus approaches: social choice theoretic: {Borda-count (***Borda***), Kemeny–Young (***Kem.***)}, and initial approximate consensus by {arithmetic mean (***AM***), geometric mean (***GM***), harmonic mean (***HM***), median (***med.***)}. In total, 19 regression test-case prioritization strategies are tabulated as specified in the aforementioned order. The first 4 prioritizations are from the existing literature. The next 9 are weighted-sum hybridizations of the heuristics specified in the third and the second columns, in that order. Each of the 6 consensus approaches in the rightmost partition aggregates all 13 prioritized test-orderings determined in isolation by each participating heuristic from the second horizontal partition.

In Tables 11, 12, 13, and 14, the reported numbers represent geometric mean values calculated across all version pairs of a subject. Subjects `flex` and `sed`, had 4 version-pairs (5 versions) each, and all other benchmarks had 5 version-pairs (6 versions) each. An entry (middle partition) is highlighted in blue if it performed worse than at least one strategy in the right partition. This also holds the other way, i.e., effectiveness for consensus technique is highlighted in blue if it was outperformed by at least one participating individual. This comparison occurs along each row that is bound to APFD, $APFD_c$, ECC, and EPS values, for each benchmark in all four tabulations. Additionally, the best performing strategy (within a partition) has its entry marked in bold font. If this entry was outperformed by a strategy from the other partition, then this entry is bold-highlighted in **blue** font. The most effective strategy among all techniques, appears bold-highlighted in **black** font.

Note that ECC values were not applicable for consensus prioritizations as their only inputs were test-case orderings with no associated priority score or test-coverage information. It is to be noted that with respect to precision of three decimal places, there may be multiple best performing strategies. For instance, in case of `flex`, *add.* with an APFD value of 0.503 outperformed all individual heuristics, and all consensus approaches performed equally best with APFD of 0.501, except *GM*. In case of `flex`, seven individual approaches were outperformed in terms of $APFD_c$ by at least one consensus approach, and all consensus were outperformed by at least one individual strategy in the middle horizontal partition. These outperformed entries are highlighted in blue. For example, with `flex`, *Kem.* exhibited the highest $APFD_c$ of 0.506 among consensus approaches. But at the same time, at least one participating individual from the middle partition turned out to be more effective. Therefore, this defeated entry is bold-highlighted in **blue**.

4.3.2. Effectiveness in terms of test-failure observation rate

The failure observation rate along the timeline of test-execution was quantified by the following three metrics:

- *APFD* (Table 11): This metric measured the average percentage of failures detected. All the test-cases were assumed to have unit cost and all the failures to have the same severity.
- $APFD_c$ (Table 12): This was cost-cognizant APFD. The 'c' accounted for the cost of a test-execution for the new version-under test. Like APFD, all the failures were assumed to have the same severity.
- *EPS* (Table 13): This metric measured the normalized similarity (in terms of the Manhattan distance) of the binary failure pattern vector (0: passed, 1: failed) with the fictitious optimal (*opt*) prioritization. If there were n failures, the vector for *opt* was assumed to be: $\langle 1, 1, 1, \ldots, 0, 0\rangle$ with first n positions set to 1, and 0 for the remaining.

The reported values for the above metrics for 13 isolated prioritizations, and their dependent values in case of consensus approaches (11 hybridizations: majority of 84.61%), synchronize well with the explanation presented in Section 4.2.1. Otherwise, deviations in the consensus effectiveness were rendered primarily by the generalized coverage-unaware consensus operators, and secondly due to the two non-dominating prioritizations: (i) cost-only (***cost***), and (ii) greedy additional (***add.***).

4.3.3. Effectiveness in terms of code-change coverage rate

The affected basic-block coverage rate was quantified by the following metric:

ECC (Table 14): Having recorded the basic-block coverage for each test-case from the previous (older) version, this metric measured how quickly the affected (added/modified/deleted) basic-blocks in the new version were cumulatively covered when the test-cases were executed respecting the prioritized sequence.

Table 11
APFD values of regression test-execution (sequential) due to individual prioritizations and different consensus approaches.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Dataset #1	***conf.***	***rel.***	***cost***	***add.***	***(10,90)***	***(20,80)***	***(30,70)***	***(40,60)***	***(50,50)***	***(60,40)***	***(70,30)***	***(80,20)***	***(90,10)***	***Borda***	***Kem.***	***AM***	***GM***	***HM***	***med.***
flex	0.501	0.502	0.501	**0.503**	0.501	0.501	0.501	0.502	0.501	0.501	0.501	0.502	0.502	**0.501**	**0.501**	**0.501**	0.500	**0.501**	**0.501**
grep	0.723	**0.727**	0.724	0.726	0.723	0.723	0.724	0.724	0.724	0.725	0.726	0.726	0.726	0.725	0.725	0.725	**0.726**	**0.726**	0.725
space	**0.386**	0.275	0.382	0.275	**0.386**	**0.386**	**0.386**	**0.386**	**0.386**	**0.386**	**0.386**	**0.386**	**0.386**	0.378	**0.386**	0.378	0.379	0.375	**0.386**
sed	0.549	0.507	0.549	0.481	0.549	0.555	0.561	0.565	0.566	0.569	0.576	0.579	**0.612**	0.399	0.400	0.399	0.395	**0.401**	0.400
gzip	0.568	0.364	0.540	**0.605**	0.563	0.562	0.558	0.555	0.545	0.521	0.501	0.442	0.453	0.550	0.550	0.549	0.550	**0.553**	0.546
replace	0.645	**0.686**	0.652	0.631	0.645	0.645	0.645	0.645	0.645	0.645	0.645	0.644	0.642	0.633	0.645	0.633	0.642	**0.661**	0.645
printtokens	0.316	**0.489**	0.283	0.361	0.317	0.317	0.317	0.318	0.321	0.323	0.329	0.338	0.365	0.326	0.322	0.326	0.343	**0.400**	0.320
printtokens2	0.658	**0.898**	0.641	0.775	0.660	0.660	0.660	0.660	0.662	0.663	0.665	0.671	0.693	0.680	0.662	0.680	0.719	**0.819**	0.662
totinfo	0.536	**0.685**	0.512	0.669	0.538	0.540	0.541	0.546	0.549	0.552	0.556	0.558	0.607	0.569	0.551	0.569	0.582	**0.608**	0.550
schedule2	0.564	**0.648**	0.550	0.255	0.579	0.579	0.579	0.579	0.579	0.581	0.581	0.589	0.613	0.561	0.579	0.561	0.591	**0.638**	0.579
schedule	0.625	0.601	**0.665**	0.571	0.631	0.631	0.631	0.633	0.641	0.643	0.645	0.653	**0.665**	0.628	**0.640**	0.628	0.635	0.639	0.636
tcas	0.800	0.622	0.758	0.610	0.828	0.833	0.861	0.871	**0.885**	**0.885**	0.882	0.872	0.653	0.874	**0.885**	0.874	0.869	0.847	0.859
Dataset #2																			
gravity	0.100	0.159	**0.162**	0.104	0.100	0.100	0.100	0.100	0.100	0.100	0.100	0.101	0.108	0.097	**0.100**	0.097	0.098	0.097	**0.100**
xxhash	**0.285**	0.044	0.136	0.046	**0.285**	**0.285**	**0.285**	**0.285**	**0.285**	**0.285**	**0.285**	0.259	0.258	0.268	**0.285**	0.268	0.276	0.261	**0.285**
scd	0.021	**0.240**	0.069	0.208	0.021	0.021	0.021	0.021	0.021	0.021	0.022	0.022	0.024	0.024	0.021	0.024	0.047	**0.101**	0.021
xc	**0.204**	0.166	0.203	0.166	**0.204**	0.194	0.192	0.192	0.192	0.165	0.152	0.152	0.152	0.180	**0.192**	0.180	0.172	0.166	**0.192**
mlisp	0.307	0.310	0.250	0.326	0.307	0.310	0.311	0.313	0.317	0.322	0.330	0.338	**0.354**	0.324	0.317	0.324	**0.328**	0.331	0.317
slre	0.326	**0.374**	0.246	0.205	0.328	0.328	0.328	0.328	0.327	0.325	0.331	0.324	0.330	0.289	0.325	0.287	0.279	0.305	**0.326**
ckf	0.680	0.330	0.677	0.304	0.690	0.690	0.690	0.690	**0.691**	0.681	0.533	0.382	0.405	0.612	0.690	0.612	0.640	0.638	**0.694**
c4	0.150	**0.188**	0.142	0.182	0.150	0.150	0.150	0.150	0.150	0.151	0.151	0.152	0.152	0.150	0.150	0.150	0.154	**0.169**	0.150

Table 12
$APFD_c$ values of regression test-execution (sequential) due to individual prioritizations and different consensus approaches.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Dataset #1	***conf.***	***rel.***	***cost***	***add.***	***(10,90)***	***(20,80)***	***(30,70)***	***(40,60)***	***(50,50)***	***(60,40)***	***(70,30)***	***(80,20)***	***(90,10)***	***Borda***	***Kem.***	***AM***	***GM***	***HM***	***med.***
flex	0.505	0.499	**0.512**	0.504	0.505	0.504	0.506	0.506	0.506	0.506	0.505	0.505	0.506	0.504	**0.506**	0.503	0.505	0.503	0.505
grep	0.341	**0.467**	0.460	0.448	0.359	0.369	0.379	0.393	0.400	0.417	0.436	0.442	0.442	0.427	0.407	0.428	0.431	**0.442**	0.433
space	**0.324**	0.205	0.322	0.206	**0.324**	**0.324**	**0.324**	0.323	0.323	0.323	**0.324**	**0.324**	**0.324**	0.314	0.321	0.314	0.315	0.310	**0.322**
sed	0.553	0.501	0.557	0.472	0.554	0.560	0.566	0.574	0.568	0.574	0.579	0.582	**0.612**	0.400	0.404	0.400	0.399	0.402	**0.407**
gzip	0.602	0.359	0.579	**0.604**	0.598	0.595	0.591	0.589	0.579	0.553	0.529	0.469	0.475	0.579	0.583	0.579	**0.585**	0.581	0.581
replace	0.371	**0.436**	0.385	0.356	0.372	0.372	0.372	0.370	0.371	0.371	0.370	0.369	0.365	0.353	0.371	0.352	0.372	**0.407**	0.370
printtokens	0.208	**0.387**	0.169	0.265	0.209	0.209	0.210	0.210	0.213	0.215	0.220	0.230	0.254	0.216	0.213	0.217	0.234	**0.288**	0.212
printtokens2	0.274	**0.534**	0.251	0.393	0.276	0.276	0.277	0.277	0.278	0.281	0.282	0.290	0.315	0.294	0.278	0.295	0.341	**0.453**	0.278
totinfo	0.376	**0.557**	0.382	0.541	0.377	0.380	0.382	0.388	0.389	0.393	0.396	0.399	0.448	0.414	0.394	0.416	0.433	**0.469**	0.391
schedule2	0.518	**0.634**	0.504	0.224	0.534	0.534	0.535	0.532	0.534	0.537	0.537	0.551	0.581	0.522	0.535	0.525	0.562	**0.618**	0.535
schedule	0.426	0.399	**0.488**	0.355	0.434	0.434	0.435	0.438	0.450	0.452	0.455	0.467	0.484	0.432	**0.448**	0.433	0.441	**0.448**	0.442
tcas	0.558	0.333	0.522	0.379	0.583	0.587	0.612	0.621	**0.633**	**0.633**	0.620	0.584	0.341	0.611	**0.630**	0.612	0.603	0.582	0.612
Dataset #2																			
gravity	0.166	0.184	**0.262**	0.047	0.166	0.164	0.166	0.166	0.173	0.173	0.179	0.178	0.187	**0.211**	0.170	**0.211**	0.208	0.188	0.165
xxhash	**0.285**	0.044	0.136	0.046	**0.285**	**0.285**	**0.285**	**0.285**	**0.285**	**0.285**	**0.285**	0.259	0.258	0.268	**0.285**	0.268	0.276	0.261	**0.285**
scd	0.022	**0.240**	0.069	0.208	0.023	0.023	0.023	0.023	0.022	0.022	0.023	0.023	0.026	0.025	0.022	0.025	0.047	**0.101**	0.022
xc	**0.241**	0.193	0.235	0.193	**0.241**	0.240	0.240	0.240	0.240	0.236	0.234	0.234	0.234	0.238	**0.240**	0.238	0.237	0.235	0.239
mlisp	0.328	0.331	0.305	0.352	0.328	0.332	0.334	0.336	0.341	0.348	0.355	0.360	**0.376**	0.355	0.341	0.357	0.358	**0.360**	0.340
slre	0.225	**0.251**	0.106	0.042	0.231	0.223	0.221	0.231	0.226	0.221	0.232	0.223	0.242	0.177	0.222	0.176	0.172	0.206	**0.225**
ckf	0.807	0.211	**0.854**	0.242	0.800	0.800	0.799	0.800	0.800	0.790	0.567	0.345	0.352	0.730	0.800	0.730	0.764	0.768	**0.802**
c4	0.152	**0.181**	0.159	0.175	0.152	0.152	0.153	0.152	0.152	0.153	0.153	0.154	0.152	0.151	0.152	0.151	0.153	**0.167**	0.153

For this metric, we observed that the prioritization by relevance (***rel.***) showed the fastest rate of affected basic-block coverage for 15 out of 20 benchmarks. This outperformed the second-best strategy, greedy additional, which showed the best values for only 8 benchmarks. The best-performing entries are highlighted in **black**. Consensus approaches have their entries marked as dashes (—) because the aggregation of test-prioritization was inherently a black-box mechanism that was unaware of the basic-block coverages.

In case of space, Hansie achieved the maximum constant performance (ECC=1.0), irrespective of any prioritization technique. This was due to a test-selection of 58.98% from a pool of 13585 tests, and a low change of 0.07% for all its version pairs. Interestingly, all of them were covered by the very first few test-cases in the prioritized sequence. The invariance to prioritization was attributed by test-selection that was coverage redundant. Hence in this case, the change-coverage rate did not vary even if the test-permutation changed a little.

The most interesting scenario occurred for gravity where the ECC values always lied below 0.1. This was not because of the rate of change coverage but due to another aspect where the

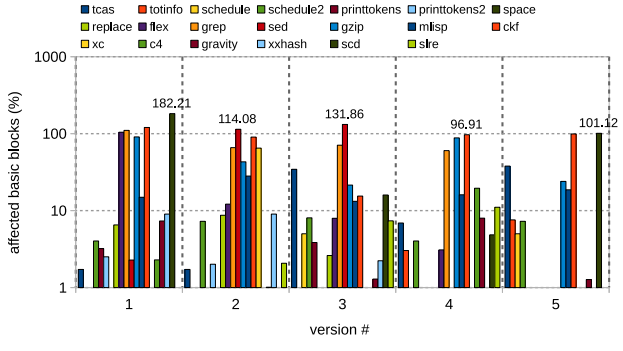

Fig. 13. Number of affected basic-blocks for each version in our study. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 13
EPS values of regression test-execution (sequential) due to individual prioritizations and different consensus approaches.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Dataset #1	***conf.***	***rel.***	***cost***	***add.***	***(10,90)***	***(20,80)***	***(30,70)***	***(40,60)***	***(50,50)***	***(60,40)***	***(70,30)***	***(80,20)***	***(90,10)***	***Borda***	***Kem.***	***AM***	***GM***	***HM***	***med.***
flex	0.995	0.998	**1.000**	0.996	0.996	0.997	0.997	0.997	0.997	0.997	0.998	0.998	0.997	**0.997**	**0.997**	**0.997**	**0.997**	**0.997**	**0.997**
grep	0.730	**0.734**	0.731	**0.734**	0.730	0.731	0.731	0.732	0.732	0.733	**0.734**	**0.734**	**0.734**	0.732	0.732	0.733	0.733	**0.734**	0.733
space	**0.485**	0.369	0.481	0.369	**0.485**	**0.485**	**0.485**	**0.485**	**0.485**	**0.485**	**0.485**	**0.485**	**0.485**	0.477	**0.485**	0.477	0.479	0.474	**0.485**
sed	0.782	0.719	0.784	0.692	0.782	0.787	0.793	0.797	0.797	0.800	0.806	0.808	**0.841**	0.551	**0.553**	0.551	0.548	**0.553**	0.552
gzip	0.769	0.530	0.736	**0.799**	0.763	0.762	0.758	0.754	0.742	0.716	0.692	0.626	0.634	0.747	0.748	0.745	0.748	**0.751**	0.743
replace	0.657	**0.697**	0.664	0.643	0.657	0.657	0.657	0.657	0.657	0.657	0.657	0.656	0.654	0.645	0.657	0.645	0.654	**0.674**	0.657
printtokens	0.321	**0.494**	0.288	0.366	0.322	0.322	0.322	0.323	0.326	0.328	0.334	0.344	0.370	0.331	0.327	0.331	0.348	**0.406**	0.325
printtokens2	0.684	**0.923**	0.667	0.800	0.686	0.686	0.686	0.686	0.688	0.689	0.691	0.697	0.719	0.706	0.688	0.706	0.745	**0.844**	0.688
totinfo	0.556	**0.705**	0.531	0.689	0.557	0.560	0.561	0.566	0.569	0.572	0.575	0.578	0.627	0.589	0.571	0.589	0.601	**0.628**	0.570
schedule2	0.570	**0.654**	0.556	0.261	0.585	0.585	0.585	0.585	0.585	0.587	0.587	0.596	0.619	0.567	0.586	0.567	0.598	**0.644**	0.586
schedule	0.652	0.626	**0.693**	0.596	0.659	0.659	0.659	0.661	0.669	0.671	0.673	0.681	**0.693**	0.656	**0.668**	0.656	0.662	0.667	0.664
tcas	0.815	0.637	0.774	0.625	0.844	0.849	0.877	0.888	**0.901**	**0.901**	0.898	0.888	0.667	0.890	**0.901**	0.890	0.885	0.863	0.875
Dataset #2																			
gravity	0.103	0.162	**0.165**	0.107	0.103	0.103	0.103	0.103	0.103	0.103	0.103	0.104	0.111	0.101	**0.103**	0.101	0.101	0.100	**0.103**
xxhash	**0.287**	0.046	0.138	0.049	**0.287**	**0.287**	**0.287**	**0.287**	**0.287**	**0.287**	**0.287**	0.261	0.260	0.270	**0.287**	0.270	0.278	0.262	**0.287**
scd	0.033	**0.249**	0.080	0.218	0.033	0.033	0.033	0.033	0.033	0.034	0.034	0.034	0.036	0.036	0.033	0.036	0.058	**0.111**	0.033
xc	**0.302**	0.268	**0.302**	0.268	**0.302**	0.293	0.292	0.292	0.291	0.268	0.256	0.256	0.256	0.280	**0.291**	0.280	0.273	0.268	**0.291**
mlisp	0.359	0.366	0.301	0.379	0.359	0.362	0.363	0.365	0.369	0.375	0.382	0.391	**0.408**	0.377	0.369	0.377	0.381	**0.383**	0.369
slre	0.336	**0.384**	0.256	0.215	0.338	0.338	0.337	0.337	0.337	0.335	0.340	0.334	0.340	0.299	0.335	0.297	0.289	0.316	**0.336**
ckf	0.793	0.441	0.790	0.415	0.803	0.803	0.803	0.803	**0.804**	0.794	0.646	0.493	0.516	0.724	0.803	0.724	0.753	0.751	**0.807**
c4	0.239	**0.273**	0.232	0.267	0.239	0.239	0.239	0.239	0.239	0.239	0.239	0.241	0.240	0.239	0.239	0.239	0.242	**0.257**	0.239

Table 14
ECC values of regression test-execution (sequential) due to individual prioritizations.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Dataset #1	***conf.***	***rel.***	***cost***	***add.***	***(10,90)***	***(20,80)***	***(30,70)***	***(40,60)***	***(50,50)***	***(60,40)***	***(70,30)***	***(80,20)***	***(90,10)***	***Borda***	***Kem.***	***AM***	***GM***	***HM***	***med.***
flex	0.081	**0.394**	0.250	0.383	0.080	0.080	0.080	0.080	0.081	0.085	0.084	0.085	0.135	–	–	–	–	–	–
grep	0.119	**0.122**	**0.122**	**0.122**	0.119	0.120	0.120	0.121	0.121	0.121	**0.122**	**0.122**	**0.122**	–	–	–	–	–	–
space	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**	–	–	–	–	–	–
sed	0.363	**0.417**	0.359	0.409	0.366	0.368	0.370	0.372	0.376	0.379	0.381	0.384	0.393	–	–	–	–	–	–
gzip	0.311	0.276	0.313	0.300	0.312	0.313	0.314	0.315	0.316	0.317	0.322	0.324	**0.325**	–	–	–	–	–	–
replace	0.989	**0.990**	0.989	0.989	0.989	0.989	0.989	0.989	0.989	0.989	0.989	0.989	0.989	–	–	–	–	–	–
printtokens	0.691	**0.705**	0.691	0.700	0.692	0.692	0.692	0.692	0.692	0.693	0.694	0.696	0.698	–	–	–	–	–	–
printtokens2	0.881	**0.907**	0.873	0.884	0.883	0.883	0.883	0.883	0.887	0.887	0.888	0.895	0.902	–	–	–	–	–	–
totinfo	0.970	0.978	0.972	**0.979**	0.970	0.970	0.970	0.970	0.970	0.970	0.970	0.970	0.970	–	–	–	–	–	–
schedule2	0.944	**0.953**	0.943	0.874	0.945	0.945	0.945	0.945	0.945	0.945	0.945	0.945	0.946	–	–	–	–	–	–
schedule	0.999	**1.000**	**1.000**	**1.000**	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	0.999	–	–	–	–	–	–
tcas	0.837	0.969	0.968	**0.970**	0.838	0.838	0.843	0.847	0.847	0.847	0.881	0.928	0.958	–	–	–	–	–	–
Dataset #2																			
gravity	0.071	**0.093**	0.083	0.087	0.071	0.071	0.071	0.071	0.072	0.072	0.073	0.074	0.079	–	–	–	–	–	–
xxhash	0.337	**0.363**	0.340	0.344	0.337	0.338	0.339	0.339	0.340	0.346	0.353	0.362	**0.363**	–	–	–	–	–	–
scd	0.376	**0.474**	0.369	0.443	0.376	0.378	0.380	0.384	0.386	0.388	0.391	0.393	0.396	–	–	–	–	–	–
xc	0.367	**0.485**	0.367	**0.485**	0.367	0.406	0.407	0.407	0.429	0.447	0.473	0.473	0.473	–	–	–	–	–	–
mlisp	0.328	**0.372**	0.336	0.357	0.328	0.328	0.328	0.328	0.329	0.329	0.330	0.336	0.346	–	–	–	–	–	–
slre	0.587	**0.612**	0.571	0.605	0.588	0.588	0.588	0.589	0.590	0.590	0.592	0.595	0.601	–	–	–	–	–	–
ckf	0.689	0.801	0.686	**0.808**	0.687	0.687	0.687	0.687	0.687	0.705	0.776	0.792	0.804	–	–	–	–	–	–
c4	0.244	0.262	0.251	**0.268**	0.244	0.245	0.245	0.245	0.245	0.245	0.246	0.255	0.265	–	–	–	–	–	–

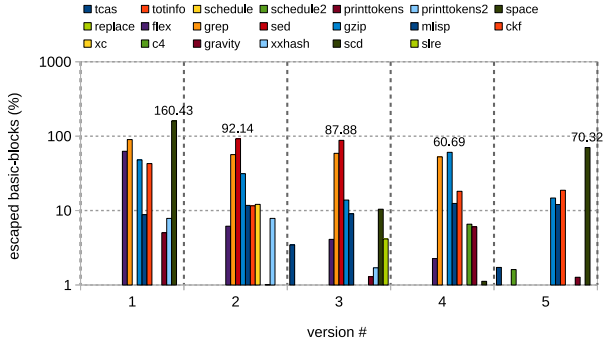

Fig. 14. Number of escaped (untested) basic-blocks for Fig. 13. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

prioritized regression test-suite was not sufficient enough to exercise the changes. We inspected that a very large portion of these changes formed new paths in the inter-procedural control-flow graph.

Additional information hidden in the ECC values (Table 14) was revealed by an explicit analysis that inspected the amount of code-change exercised/missed by the test-executions. We present the outcome of this inspection below.

Affected and escaped (untested) basic-blocks: The number of affected and escaped basic-blocks for all the subjects in our study is shown in Fig. 13 and Fig. 14, respectively. The *x*-axes correspond to the version number and *y*-axes correspond to the percentage (log-scale) of affected and escaped code elements. In both the figures, the vertical dashed lines partitions the version-space (*x*-axes). Thus, for a version *i*, the associated segment is bounded from left and right by preceding and succeeding versions, while within a segment, vertical bars represent corresponding data.

We observe that Fig. 13 is denser than Fig. 14, due to a larger number of changed basic-blocks for almost all the versions. However, the exceptions with 0% affected-blocks are: `slre`: $v_0 \rightarrow v_1$, `c4`: $v_1 \rightarrow v_2$, `xc`: $v_2 \rightarrow v_3$, `printtokens`: $v_3 \rightarrow v_4$, `sed`:

$v_3 \rightarrow v_4$, and c4: $v_4 \rightarrow v_5$. There are no corresponding plots (Figs. 13 and 14) for $v_4 \rightarrow v_5$ for `flex` and `sed` as they had four version pairs in our study, as shown in Table 7.

The number of affected-blocks reached as high as 182.21% for version pair $v_0 \rightarrow v_1$ in `scd`. This indicated a significant amount of newly added code in addition to changes that occurred in the existing basic-blocks. A high escape value of 160.43% was observed for $v_0 \rightarrow v_1$ in `scd`, whose corresponding interpretation stated that the older test-coverage map (v_0) was outdated and needed to be refreshed for v_1 by taking into account coverage of newly added code elements. This was also an indication that the existing regression test-suite did not suffice, and new tests needed to be incorporated. In other words, the test-coverage graph also needed to evolve.

In Figs. 13 and 14, the highest percentage value is shown over the top of its corresponding bar in each segment. For instance, `sed` had 114.08% of affected-blocks for $v_1 \rightarrow v_2$, and `gzip` had 60.69% of basic-blocks in v_4 that could not be covered by the existing test-coverage graph that corresponded to v_3.

4.3.4. Implications and lessons learned

As a general implication, a basic-block lies in the scope of a test-suite if and only if that basic-block is reachable by at least one test-case. Blocks lie out-of scope when the test-suite is not sufficient enough to exercise that change. This brings down the effectiveness of prioritization mainly in terms of change coverage, i.e., ECC. A low value of ECC can be interpreted in two ways. Firstly, it can occur if the changed blocks are spread distantly and requires one to run say 40–50% of the prioritized test-suite before all (100%) in-scope changes are exercised. This implies a low rate of change coverage (ECC) and provides an indication of the regression test-suite quality in the context of change traversal. Secondly, even after running all the test-cases, out-of-scope changes cannot be exercised; this penalizes the test-suite effectiveness (ECC) for being inadequate.

The associated effectiveness trends (dependent on α, and as discussed for RQ1 in Section 4.2) were *primarily responsible* for shaping the effectiveness of our six pursued consensus prioritization strategies. This was because the majority voting in terms of test prioritization directly agreed with the test-priority opinions of the majority of individuals. This directly placed a test-case t_i before t_j in the consensus if and only if the majority agreed, i.e., no ties. Otherwise the pair was inverted in the consensus prioritization.

The secondary factor affecting the consensus ordering, hence, its effectiveness of prioritization, was the nature of consensus functions. For 13 out of 20 benchmarks, APFD, $APFD_c$, EPS simultaneously showed strong correlation with increasing values of α. For the rest 7, APFD and EPS did correlate but $APFD_c$ did not. The reason for this was the cost-cognizance of a test-case which $APFD_c$ was aware of but the other two metrics (APFD, EPS) did not consider for computing the effectiveness of test-prioritization.

Different test-cases may have similar coverage but with a change in stability of the sorting algorithm, insignificant differences in effectiveness may be observed. Note that ties in individual prioritizations arose due to the same priority score, whereas ties in consensus ordering depended on whether cycles were present in the associated majority preference graphs or whether there existed multiple (Section 3.3: Observation 7) consensuses (see Table 15).

Table 15
Occasions when consensus outperformed individual prioritizations.

Metric	*Borda*	*Kem.*	*AM*	*GM*	*HM*	*med.*
APFD	37.31%	35.00%	37.31%	47.69%	**53.46%**	35.38%
$APFD_c$	41.15%	41.54%	41.54%	55.00%	**58.08%**	41.54%
EPS	38.08%	37.69%	38.85%	48.85%	**54.23%**	38.85%
Geo. mean	*38.81%*	*37.98%*	*39.19%*	*50.41%*	***55.22%***	*38.51%*

Answering RQ2 (consensus effectiveness)

Consensus prioritization by harmonic mean (***HM***) of ranks was the most effective one (55.22%). The second best consensus was ***GM*** (50.41%). Table 15 reports the percentage of occasions when consensus prioritization outperformed participating traditional ones. Consensus effectiveness is ranked ($\succ$) as follows.

$$\textbf{\textit{HM}} \succ \textbf{\textit{GM}} \succ \textbf{\textit{AM}} \succ \textbf{\textit{Borda}} \succ \textbf{\textit{med.}} \succ \textbf{\textit{Kem.}}$$

4.4. RQ3: Comparison with state-of-the-art prioritizations

In this subsection we compare our approach against selected baselines, both qualitatively, and quantitatively.

4.4.1. Comparison with optimal and greedy additional

A visual interpretation of the reported EPS values for various prioritizations is depicted in Figs. 15 and 16, where *x*-axis specifies timeline positions and *y*-axis denotes the cumulative number of test-failures observed at a given position. In the visualization plots we have two baselines: (i) greedy additional (shown in **red**), and (ii) optimal case (shown in **navy blue**). In the fictitious optimal case, denoted as opt, failures are highly coalesced toward the beginning of the timeline. All others are compared against opt in terms of how close (Torres et al., 2019) their failure distributions are.

`tcas`: The first two rows of Fig. 15 show failure distribution of regression testing `tcas` for version pairs: $\{v_0 \rightarrow v_1, v_0 \rightarrow v_2, v_0 \rightarrow v_3, v_0 \rightarrow v_4\}$. The test-execution schedule was as per the permutation specified by greedy additional and all six consensus approaches in our study. We observed that for `tcas` ($v_0 \rightarrow v_1$), failures due to additional prioritization occurred the earliest, starting from position 1, with occasional failures around the region bounded by: $x = 29..43$, $x = 85..99$, and $x = 113..127$. The first failure for GM consensus occurred immediately before position 113, thereafter we observed that all other prioritizations revealed test-failures. For `tcas` ($v_0 \rightarrow v_1$), corresponding greedy additional EPS value was 0.914622 as compared to GM's EPS of 0.88904. Respective APFD values were 0.873577 and 0.847995. However, kemeny exhibited EPS value 0.894853, and APFD of 0.853809. This implied that the best-case failure distribution did not necessarily follow a schedule where the first failure occurred too early. Rather, the entire compactness of the distribution starting from the first failure contributed to effectiveness in terms of failure-aware metrics.

Now consider a scenario, `tcas` ($v_0 \rightarrow v_3$), where every other prioritization strategy outperformed the failure distribution obtained due to state-of-the-art greedy additional with APFD of 0.40554, and EPS of 0.412692, all consensus approaches had values greater than 0.8 which turned out to be $\approx 2\times$ more effective. We observed dense failures due to greedy additional around the region bounded by $x = 886..1090$. Otherwise, all other approaches exhibited their last failures on or before the timeline segment that stretched up to $x = 546$. With `tcas` ($v_0 \rightarrow$

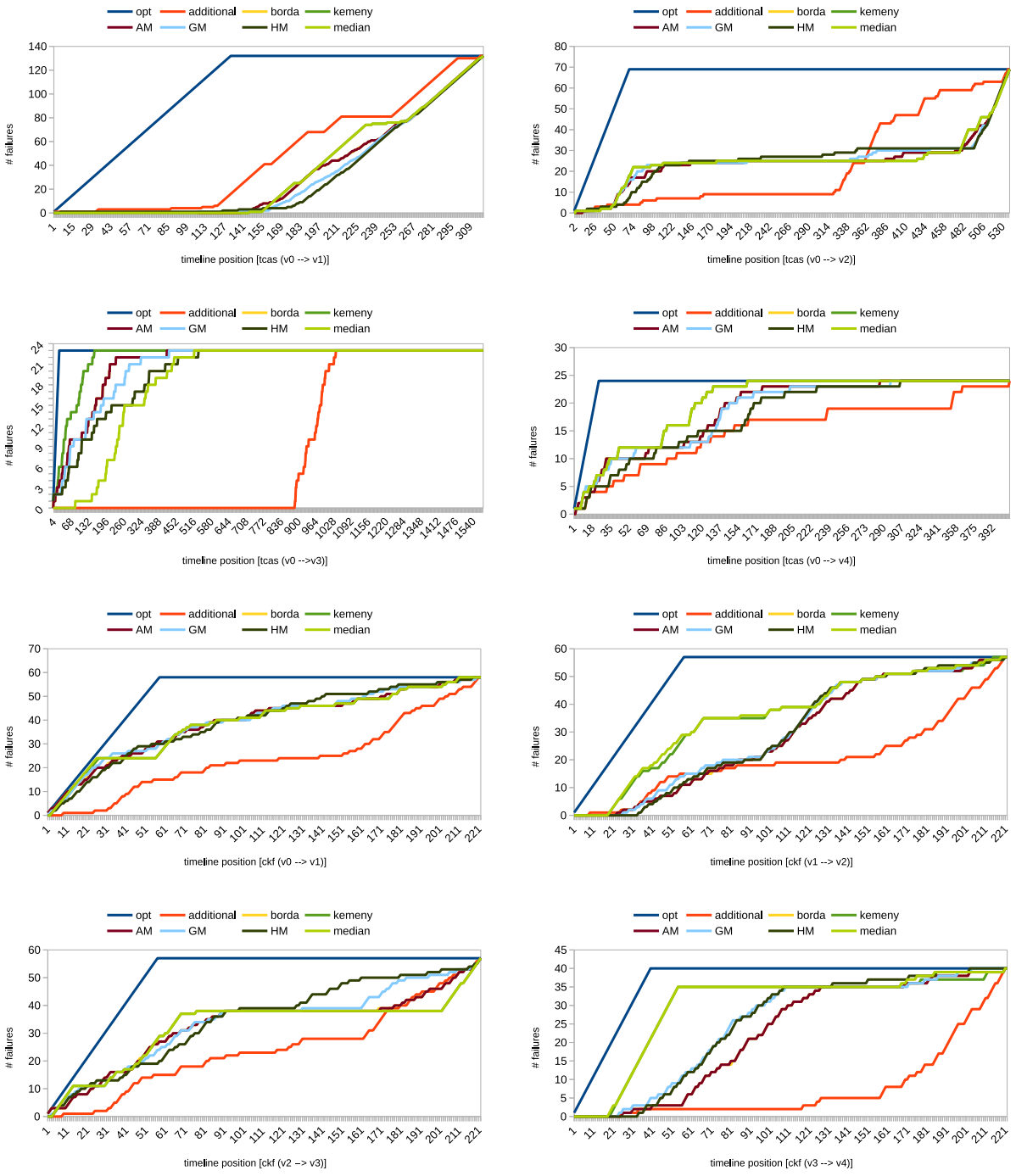

Fig. 15. Visualization of test-failures along sequential execution timeline for `tcas` (top four plots) and `ckf` (bottom four plots). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

v_3), non-cumulative failure profile followed a symmetrical failure distribution centered around $x = 265..289$. In the case of `tcas` ($v_0 \rightarrow v_4$), the non-cumulative failure profile followed a decaying failure distribution with almost a right-tail that occurred after $x = 181$.

`ckf`: Failure distributions for `ckf` (rows 3 and 4 in Fig. 15) exhibited different shapes with three of the version pairs: ($v_0 \rightarrow v_1$), ($v_1 \rightarrow v_2$), and ($v_3 \rightarrow v_4$) that had red-patches (greedy additional) of non-cumulative failures even after first ≈75% of the timeline.

`mlisp, c4, totinfo, printtokens`: In Fig. 16, `mlisp` ($v_3 \rightarrow v_4$), the distribution was consistently dense and implied that all strategies were at par with one another for this case. For `c4` ($v_0 \rightarrow v_1$), we saw strong non-cumulative failure agreements around $x = 205..358$, $460..613$, and $973..1170$. We also observed long patches of non-cumulative failures similar to `tcas` ($v_0 \rightarrow v_1$) (Fig. 15). With `totinfo` ($v_0 \rightarrow v_2$), failure distribution was sparse throughout the timeline, with an isolated failure which occurred solely due to the initial consensus by HM. In case of `printtokens` ($v_0 \rightarrow v_5$), non-cumulative failure occurrences shaped a distribution which had gradually increasing agreements up to $x = 1081$, and increased sharply thereafter. This turned out to be nearly opposite to what was observed during regression testing of `tcas` ($v_0 \rightarrow v_4$).

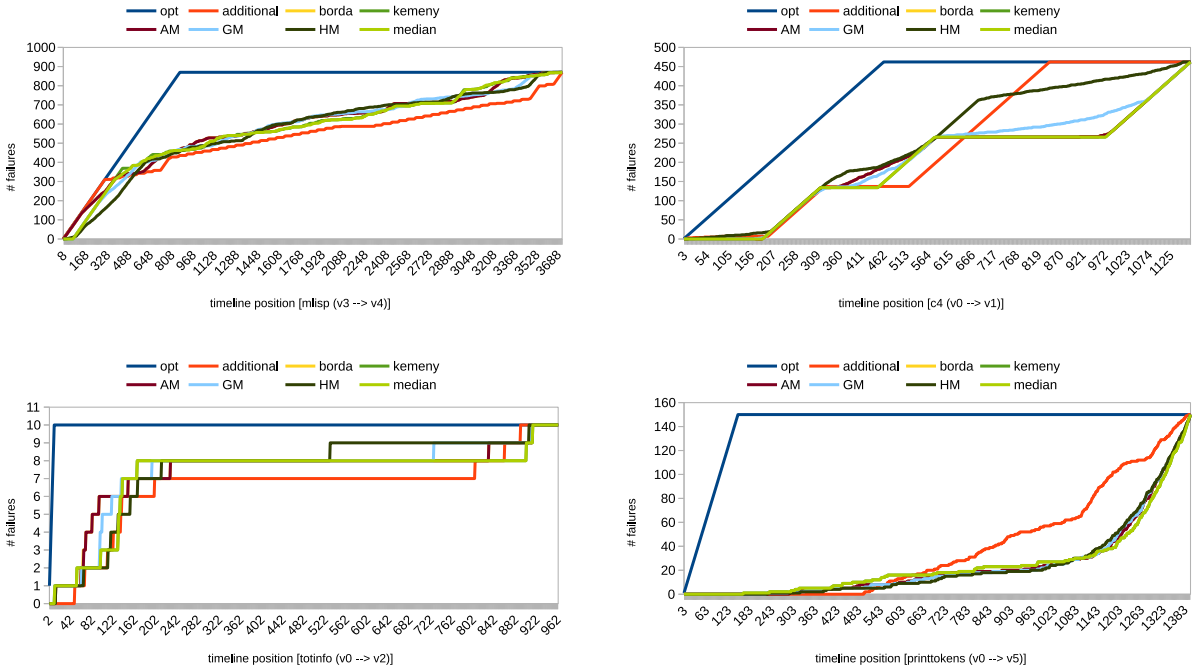

Fig. 16. Visualization of test-failures along sequential execution timeline for {`mlisp`, `c4`, `totinfo`, `printtokens`}. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

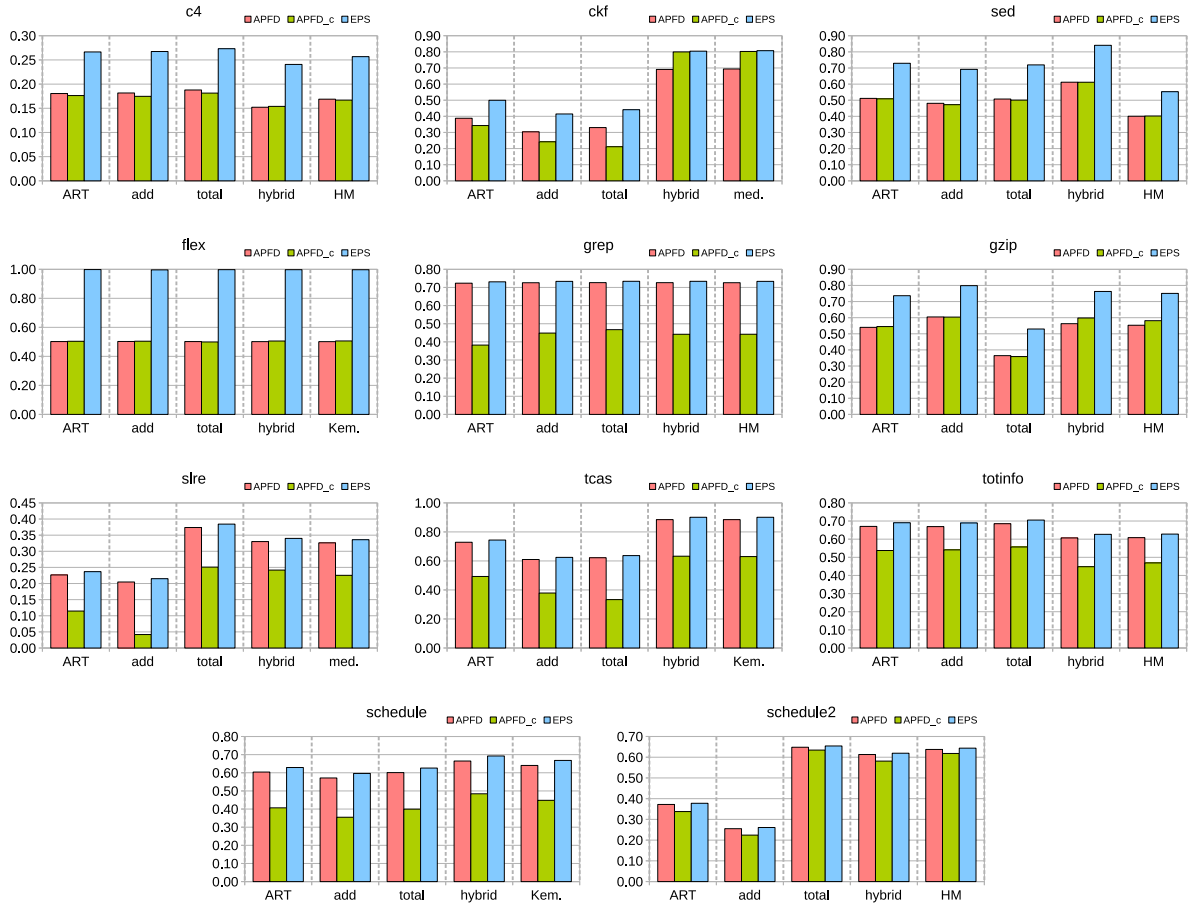

Fig. 17. Comparison of effectiveness against: (i) ART, (ii) greedy additional, and (iii) greedy total with best performing *hybrid*, and *consensus* prioritization. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

The take-away from this visualization was a more informed distribution of failures after executing a prioritized test-schedule. The nature of distribution was dependent on several attributes of the underlying regression test-suite. Important attributes were test-coverage distribution, nature and position of changes in the updated program execution path corresponding to the new version.

4.4.2. Comparison with Adaptive Random Prioritization (ART)

Fig. 17 shows {APFD, $APFD_c$, EPS} comparison of the best performing *hybrid*, and *consensus* prioritization against three state-of-the-art prioritizations: (i) adaptive random prioritization (ART), (ii) greedy additional, and (iii) greedy total.

On evaluating ART on 11 out of 20 benchmarks in our study, we observed that for 6 out of these 11 benchmarks, both hybridization and consensus outperformed all three state-of-the-art approaches. Among these 6, one case was observed where the best-performing consensus failed to outperform the state-of-the-art approaches but the best performing relevance-confinedness hybridization outperformed all of them including the consensus prioritization.

4.5. RQ4: Impact of parallelization windows

We now report and discuss experimental results on the impact of parallelization windows on consensus prioritizations.

4.5.1. Workload and parallelization

Fig. 18 shows the speedup obtained by the application of parallelization test-execution windows on consensus test-orderings. For each benchmark, the speedup plots correspond to geometric mean values across all six consensus approaches. The geometric mean across all benchmarks is shown as the rightmost point (geomean) of the *x*-axis in Fig. 18.

Discussion

We observed that Hansie achieved considerable overall speedup across the benchmarks. We also observed that the speedup was not always directly proportional to the number of threads employed. The geometric mean speedup trend across all consensus approaches across all subject programs turned out to be: (7.50×, 8.56×, 8.51×) for (10, 20, 30) threads, respectively. The reasons for this trend were: (i) unbalanced loads inside windows which arose out of test monopolization due to the longest running test-case in a parallel test-execution window, (ii) hyper-threading in the execution platform, and (iii) scheduling and context-switching in process-level parallelism.

Yet another reason for low-speedup was due to the changes that occurred in non-code portions for some versions which led to no test-selection, which effectively brought down the rectified geometric mean in the presence of zeros. As a result of no test selection for associated versions, zero speedup was also obtained for those cases. All benchmarks with a consistent non-zero test-selection had an ample selection of regression test-cases due to code-changes that occurred in every new version. The more the unit test-cases, more is the chance of obtaining higher speedup values. It should also be noted that the heaviness of test-cases in terms of disk-space consumed (elaborated in Section 4.1.5) also opposed parallelism by accessing disk files on the same file system. Although each test-case accessed different disk file, all files still resided on the same file system which when accessed in parallel might have caused significant resistance toward parallelism. These were possible reasons for speedups not reaching up to 30× when 30 threads are employed. The workloads were clearly not embarrassingly parallel.

An anomalous profile was observed for a few benchmarks where the speedup increased super-linearly up to 14.376× for 10 threads. We inspected and verified that this was due to the onset of thread-pinning in the parallelized code, and a separate purely sequential (baseline) implementation with the default kernel managed CPU-affinity.

4.5.2. On EPL_ω versus parallelization window workloads

In this subsection, we show that with nature of workload in parallel execution of independent unit test-cases, process-level parallelism coupled with consensus orderings gave rise to different workloads. For a given window size, and a consensus ordering, prioritization effectiveness is given by EPS_ω values, and workload in terms of latency of the overall execution timeline, is specified by EPL_ω values.

We partitioned the set of subjects in our testbed into two subsets based on whether changes had occurred in code components or non-code artifacts. This way we respectively report results in Figs 19, and 20.

Discussion

For Fig. 19, we found that EPS values remained consistently close to 1 when at least 10 threads were employed. This means better load-balancing came at around 10 threads for these benchmarks. However, exceptions are explained as follows. `mlisp`, and `space` showed that EPS values always remained close to 0.5 even when we increased the degree of parallelism from 10 to 30 in steps of 10. This was because the first two version pairs had no associated test-failures even if 100% (3721) test-selection happened consistently throughout all five versions of `mlisp`. In the latter case, the first two versions had no observed failures for test-selections 91.62% (12,447) and 30.33% (4121), respectively. For `scd` (EPS < 0.4), the last three version pairs had 100% (9261) tests selected which exhibited failures when executed. Surprisingly, different from the aforementioned exceptions, `xxhash` (EPS $<$ 0.4) had a 9.78% (1528) test-selection for the last three of its successive version pairs, again without test-failures.

In the absence of failures, the valuation of EPS happened to be zero, which brought down the overall adjusted geometric mean in the presence of zeros. This implies that the existing regression test-cases were not strong enough to uncover regression faults signaling the testing team to incorporate stronger regression verifying test-cases. It is to be noted that these exceptional benchmarks did not exhibit failures but on an average (in our case geometric mean), the corresponding EPL values were higher than EPS values. This is expected as the end goals of EPS and EPL are different. The interpretation from the associated plots is that the rate of failure detection was less due to consensus prioritizations but the load-balancing was effective in the parallel setting.

EPL values exhibited a decaying trend of effectiveness in terms of workload quantifications. Values kept on decreasing with increasing amount of parallelism. This shows that the collective preference in the consensus ordering was invariant to test-case loads. This is expected as only 1 out of 13 individual heuristics performed cost-based prioritization. All others were based on affected basic-block coverage. Thus, the number of distinct basic blocks did not correlate with the amount of time taken for the test-case to execute. However, we do observed exceptions: (i) ***AM***, ***HM***, and ***med.*** for `ckf`, (ii) ***borda*** for `totinfo`, (iii) ***med.*** for `mlisp`, and `flex`, (iv) ***HM*** for `scd`, where peak EPL was observed for 20 threads.

As depicted in Fig. 20, consensus approaches consistently showed poor performance for `c4`, `gravity`, and `xc`, in terms of EPS and EPL measurements, both values lying close to 0.3 with bar plots of similar heights. Low EPS values were because of weaker test-cases where the selection sizes were 5.335%, 5.335%, and

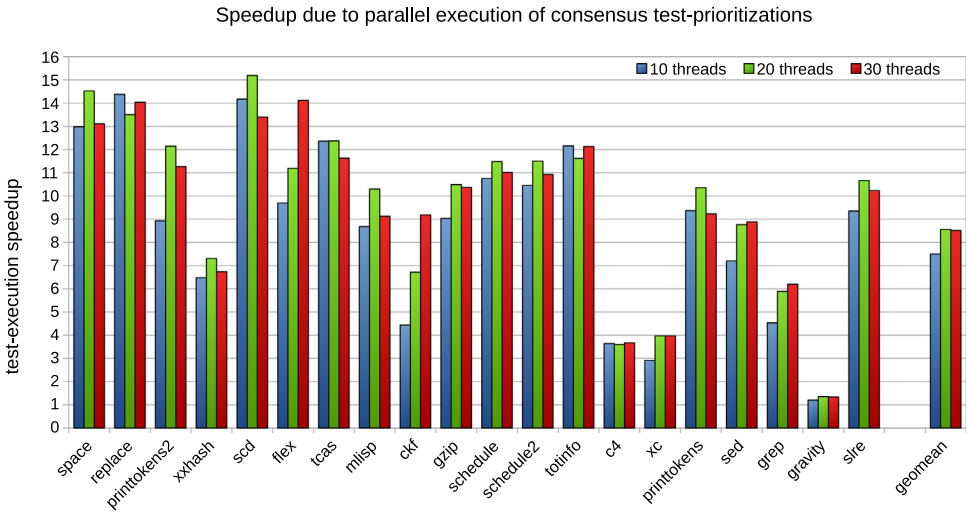

Fig. 18. Test-execution speedup due to parallelization execution windows of size 10, 20, and 30 threads, on consensus prioritizations. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

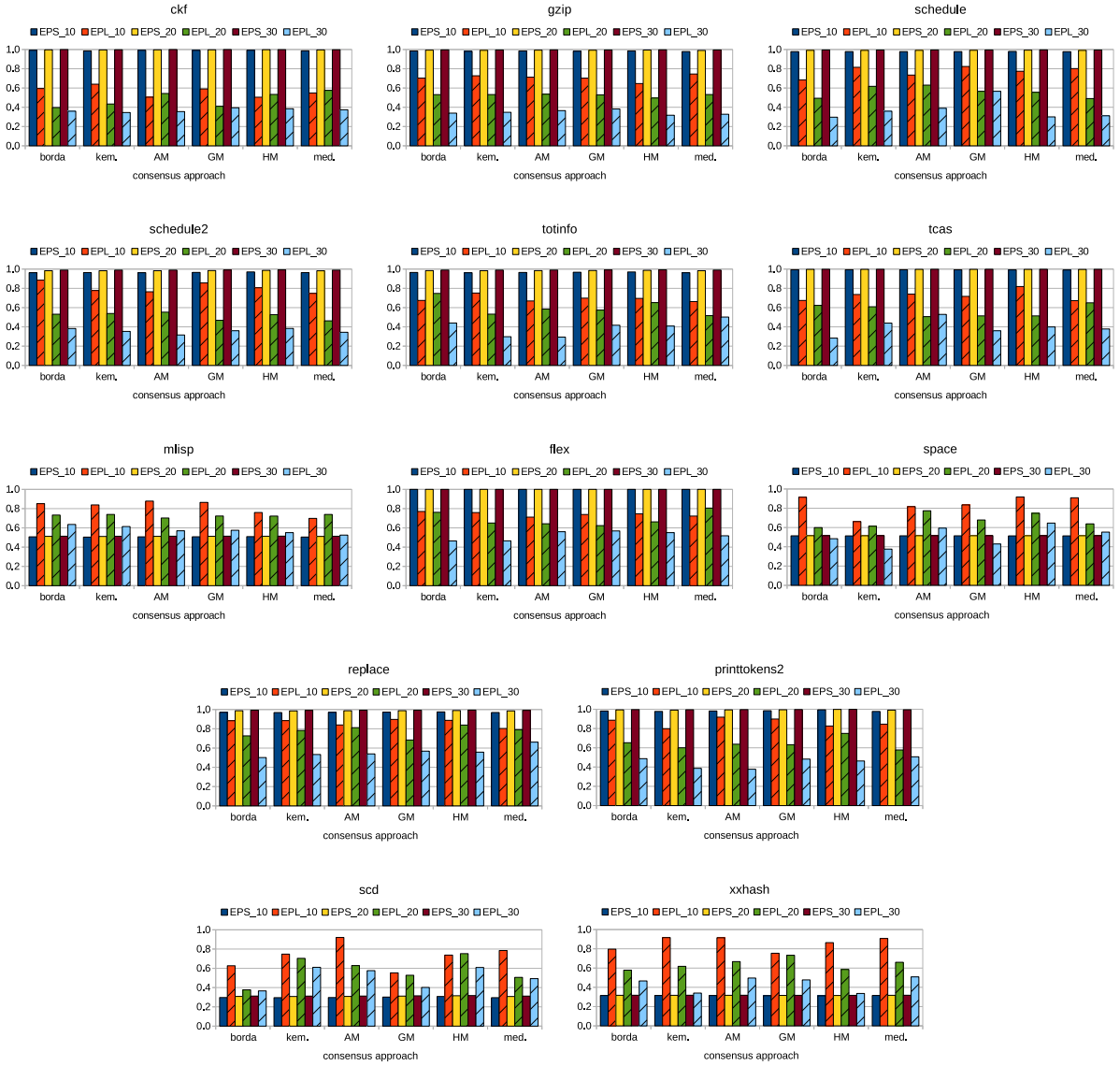

Fig. 19. Variation of EPS_ω and EPL_ω values observed by weakening consensus permutations for benchmarks having non-zero test-selection for each new version due to the presence of changes in code elements (basic-blocks). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

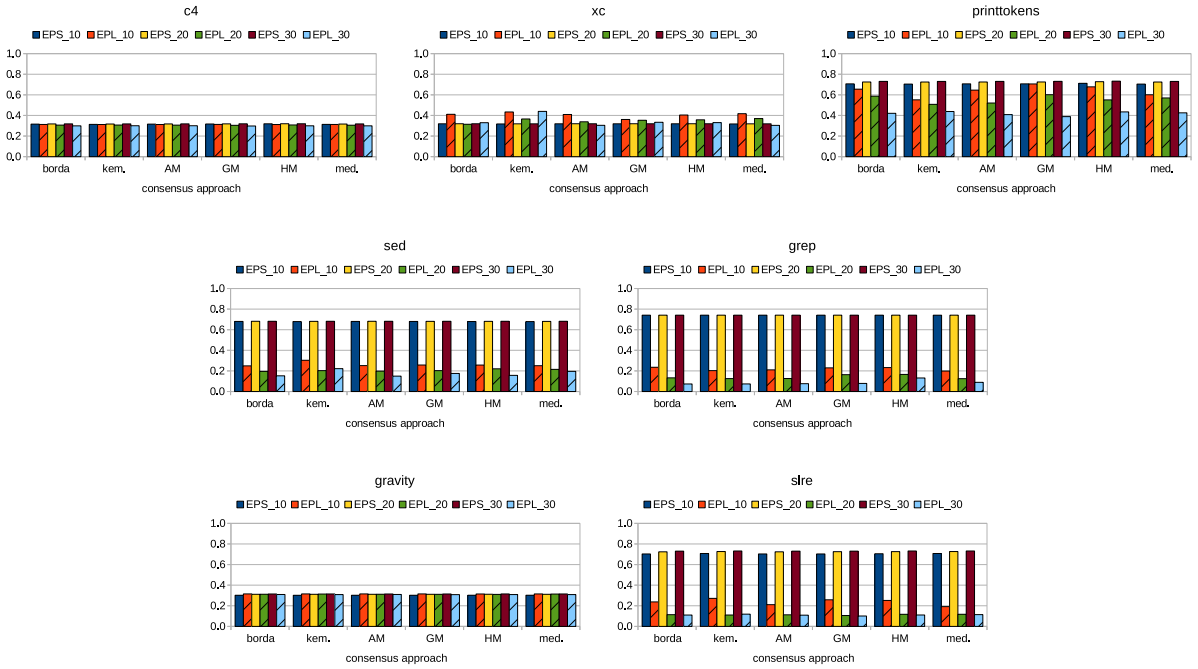

Fig. 20. Variation of EPS_ω and EPL_ω values observed by weakening consensus permutations for benchmarks have zero test-selection for some of their new versions due to the presence of non-code changes. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

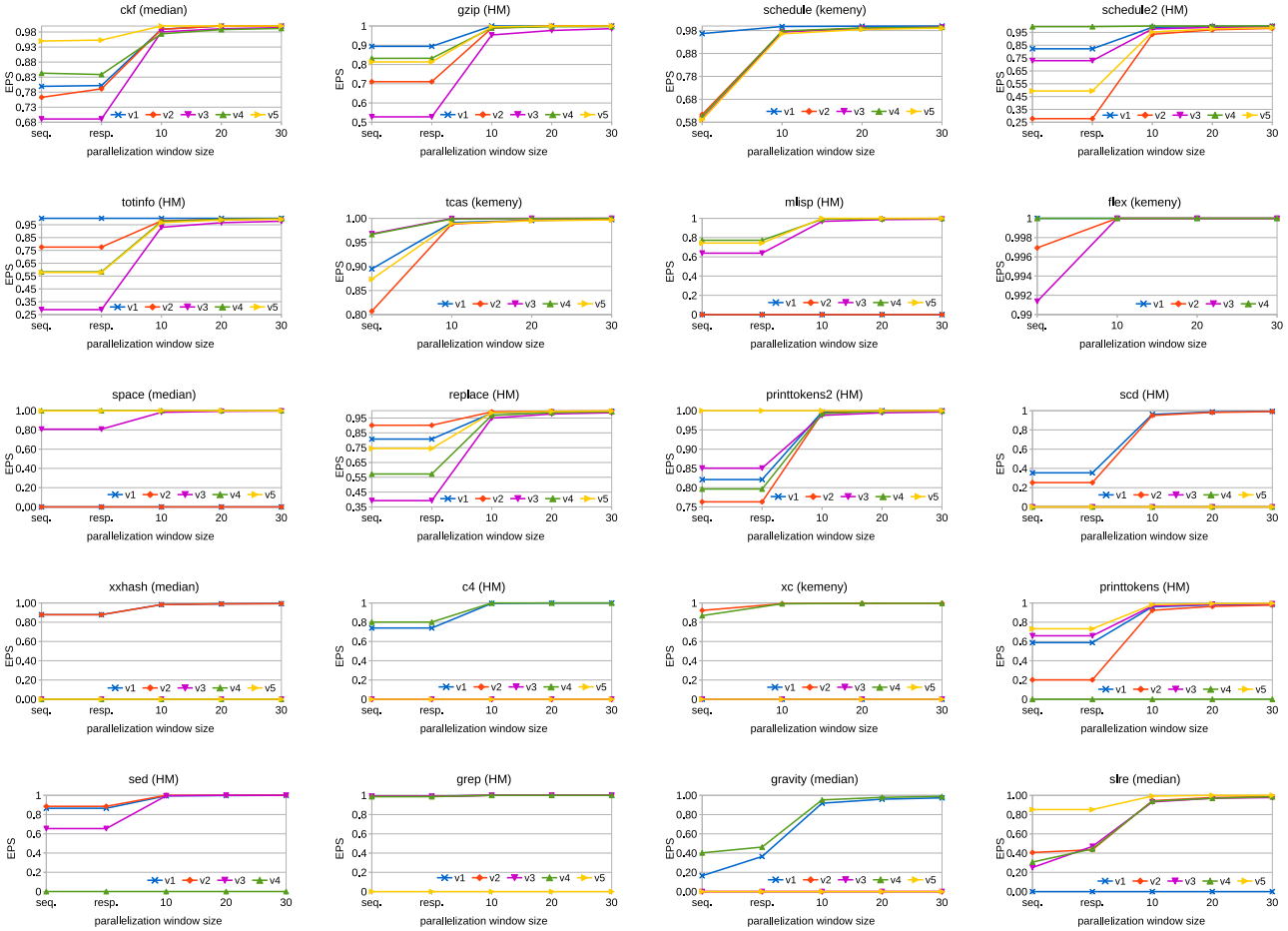

Fig. 21. Boosting EPS values using parallelization windows, version-wise for each benchmark in our study. Plots are shown for the best performing consensus approach. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

11.177%, respectively (see Table 9). For `sed`, `printtokens`, `slre`, and `grep`, EPS values close to 0.7 were consistently observed across all consensus prioritization, even with different window sizes. EPS and EPL profiles turned out to be almost similar for `sed`, `slre`, and `grep`, where EPL values were significantly low as compared to corresponding EPS values. This did not occur for other cases where EPL dipped to a certain extent. The reason for this was highly unbalanced parallelization windows which appeared early in the timeline of batch-wise test execution. This was in addition to collective disk access costs for individual test-cases in parallel and their associated synchronization overheads.

4.5.3. Enhancing EPS values using parallelization windows

In this subsection, we report and discuss how the rate of failure observation, EPS (EPS_ω), was boosted using parallelization windows (ω) for consensus prioritizations. In all cases we observed that EPS_ω values increased with the threading factor $|\omega|$, the parameter defining the window size. Fig. 21 shows the subject-wise improvement in EPS values with increasing parallelism. Note that ***resp.*** which was the maximum tie-size remained below 5 in all cases. This called for further weakening the consensus using bigger windows of size 10 and beyond. In our case, this translated to increasing the parallelization factor. The plots conveying the effectiveness values of consensus prioritization indeed show a non-decreasing profile with a weaker form of consensus.

In Fig. 21, the Kemeny consensus (***Kem.***) has no associated value for respecting ties (***resp.***) as ties never occur in a Kemeny-optimal consensus prioritization. The plots show a detailed view of the EPS measurements reported in Fig. 19 and 20 besides indicating the values of ties. It also indicates with zero EPS values, that on some occasions test-cases were either not strong enough to cover changed-code or unable to detect faults. The former case may be rectified by introducing test-cases for the changed versions alongside the regression tests, and then by performing prioritization. In such scenarios, preference should be given to the newly added tests. In the latter case, automated test-case updates may be pursued to further boost prioritization.

Answering RQ4 (parallelization)

- The workload was not embarrassingly parallel. Hence, the load-imbalance increased with an increase in window size (10→20→30). Peak geometric mean speedup was 7.56× for 20 threads.
- EPS values showed a non-decreasing trend with an increase in the parallelization factor, i.e., (***seq.***→***resp.***→10→20→30). The variable ***resp.*** attained a maximum value of 4.
- The maximum tie-size of 4 indicates that 13 individual prioritizations were strong enough to combat paradoxical consensuses, even with loosely formed consensus prioritizations.
- Two major challenges to effective EPL measurements were: (i) overhead of isolated parallel processes, and (ii) load-imbalanced workloads.

4.6. RQ5: Qualities of computed consensuses

To determine the *quality* of consensus permutations in terms of *Hansie*, and *Kendall-tau* distances, we evaluated Hansie on 13 out of 20 benchmarks. Table 16 reports the approximations (Approx) and consensus qualities (Q) for these benchmarks where computation of the quality metrics was reasonably less time-consuming, given the total number of distinct test-cases n and the time complexity, $O(n^3)$, of permutation-distance (*KT*, *HD*) computation.

4.6.1. Discussion

In Table 16, most of the benchmarks have Q values of 0.979 (Q-hm-HD for `ckf`) or more. As an exception, `grep` with a geometric mean 3.13% test-selection ratio, the least among all 20 benchmarks, consistently underperformed in terms of quality of the obtained consensuses. It exhibited lowest Q values in the range 0.861–0.925. This is shown in the entire third column (entries highlighted in **blue**) of Table 16, across all consensus strategies including our baseline, Kemeny–Young consensus. The primary reason for this was the low population of individual ranked lists. We noticed that $\approxeq$ 26 (3.13% of 809) test-cases (candidates) turned out to be inadequate for computing a good quality consensus.

This indicates that along one dimension, a fairly larger number of test-cases backed up by individual prioritizations, increases the chance of a near optimal consensus. The second dimension is the number of voters, i.e., individual participating prioritizations.

Another aspect (the third dimension) driving the consensus quality measurements is the number of preferences versus the number of aversions between any two candidates (test-cases). The differences in their counts position the quality of a consensus ordering in the spectrum lying in the interval [0, 1]. An undesirable situation is one where the counts are equal for a significantly larger number of pairs of candidates, in which case paradox ensues. For these cases, consensus quality is computed in terms of Hansie distance in the presence of ties. In all other cases when there is a significant difference in the count values (preferences versus aversions), the consensus quality may shift toward either 0 (no agreement), or 1 (full agreement). This depends upon the overall diversity of individual prioritizations when viewed collectively. In practice, as individual prioritizations differ in their heuristics, occurrence of the two extremes are highly unlikely.

The reported values do not reflect the quality of prioritization in terms of effectiveness metrics, rather the quantification of quality parameters is supplementary, and highlights the social choice perspective. The intended implication is the amount of agreement the consensus prioritization has with respect to the collection of participating prioritizations.

Answering RQ5 (consensus quality)

- To obtain a good quality consensus in terms of Hansie and Kendall-tau distances, it is desirable that the number of test-cases (candidates) be sufficiently larger than the number of individual prioritizations (voters).
- Hansie and Kendall-tau distances had a positive correlation even though the former provided an underestimated Q value. However, the difference in reported values diminished to void when a consensus was of very high quality.
- Overall, *Median* (*med.*) and *Harmonic-Mean (HM)* were the best and the worst consensus approaches, respectively in terms of both approximation factors, and consensus qualities.
- Approximation factors ($Approx_1$, $Approx_2$) and Q values did not correlate across subject programs or different consensus approaches.

4.7. Threats to validity

This section mentions Hansie's limitations and threats to validity of our experimental results, and their interpretations.

Table 16
Approximations and consensus qualities for some benchmarks in our study. (worst-value in: **blue**, best-value in: **red**).

1	2	3	4	5	6	7	8	9	10	11	12	13	14
Kemeny-Young (Kem.)	`flex`	`grep`	`sed`	`gzip`	`printtokens`	`totinfo`	`schedule2`	`schedule`	`tcas`	`xc`	`mlisp`	`ckf`	`c4`
Q-ky-HD	0.996	**0.881**	0.992	0.990	**0.999**	0.997	**0.999**	**0.999**	0.995	0.997	**0.999**	0.985	0.998
Q-ky-KT	0.997	**0.925**	0.994	0.992	**1.000**	0.998	0.999	0.999	0.997	0.998	**1.000**	0.991	0.999
Borda-count (Borda)													
$Approx_1$	1.227	**1.045**	1.211	1.273	1.372	1.413	1.408	1.354	1.249	**1.618**	1.226	1.251	1.372
$Approx_2$	1.187	**1.057**	1.176	1.202	1.305	1.361	1.324	1.321	1.255	**1.379**	1.156	1.215	1.189
Q-borda-HD	0.995	**0.877**	0.991	0.987	**0.999**	0.995	**0.999**	**0.999**	0.994	0.996	**0.999**	0.981	0.998
Q-borda-KT	0.997	**0.921**	0.993	0.991	0.999	0.997	0.999	0.999	0.996	0.998	**1.000**	0.989	0.999
Arithmetic-Mean (AM)													
$Approx_1$	1.196	**1.028**	1.193	1.241	1.333	1.385	1.378	1.323	1.227	**1.427**	1.184	1.215	1.242
$Approx_2$	1.187	**1.057**	1.176	1.202	1.305	1.362	1.324	1.321	1.255	**1.380**	1.156	1.215	1.189
Q-am-HD	0.996	**0.879**	0.991	0.987	**0.999**	0.995	**0.999**	**0.999**	0.994	0.996	**0.999**	0.982	0.998
Q-am-KT	0.997	**0.921**	0.993	0.991	0.999	0.997	0.999	0.999	0.996	0.998	**1.000**	0.989	0.999
Geometric-Mean (GM)													
$Approx_1$	1.371	**1.063**	1.338	1.277	1.425	**1.628**	1.533	1.394	1.320	1.494	1.333	1.245	1.312
$Approx_2$	1.334	**1.091**	1.302	1.236	1.391	1.548	1.471	1.385	1.337	**1.557**	1.285	1.238	1.279
Q-gm-HD	0.995	**0.875**	0.990	0.987	**0.999**	0.995	**0.999**	0.998	0.994	0.996	**0.999**	0.982	0.998
Q-gm-KT	0.996	**0.919**	0.993	0.991	**0.999**	0.997	**0.999**	**0.999**	0.996	0.998	**0.999**	0.989	**0.999**
Harmonic-Mean (HM)													
$Approx_1$	2.012	**1.203**	1.874	1.644	1.794	**2.158**	2.120	1.673	1.581	1.788	1.998	1.464	1.894
$Approx_2$	1.868	**1.260**	1.776	1.564	1.734	**1.959**	1.956	1.644	1.609	1.878	1.850	1.432	1.778
Q-hm-HD	0.992	**0.861**	0.986	0.983	**0.999**	0.993	**0.999**	0.998	0.993	0.995	**0.999**	0.979	0.997
Q-hm-KT	0.995	**0.908**	0.990	0.988	**0.999**	0.996	**0.999**	**0.999**	0.995	0.997	**0.999**	0.987	0.998
Median (med.)													
$Approx_1$	1.392	1.228	1.476	1.253	1.255	1.076	1.139	1.189	**1.042**	1.078	**1.765**	1.054	1.124
$Approx_2$	1.008	1.052	1.011	1.006	1.003	1.006	**1.001**	1.008	1.013	**1.054**	1.004	1.023	**1.001**
Q-med-HD	0.995	**0.856**	0.989	0.987	**0.999**	0.996	**0.999**	**0.999**	0.995	0.996	**0.999**	0.984	0.998
Q-med-KT	0.997	**0.920**	0.994	0.992	**1.000**	0.998	0.999	0.999	0.997	0.998	**1.000**	0.990	0.999

4.7.1. External validity

Our study was performed on twelve C programs from the SIR repository and eight C projects from GitHub, all of which consisted of the minimalist build system, i.e., a straightforward compilation using `gcc` with appropriate flags. Benchmark programs are available in more versions than those used in our experiments and might provide a threat to the validity of our experimental results in the sense that they may not generalize to other subject programs with generalized build approaches. To mitigate this threat, our choice of subjects (especially SIR programs) was as per extensive empirical usage in software testing research in the academic setting. We plan to reduce this threat by adapting Hansie to wider range of projects with different build systems.

Based on the size (totaling 287,530 SLOC), the subject programs can be categorized as follows: (i) small (Siemens suite, `slre`, `ckf`, `c4`), (ii) medium (`xc`, `mlisp`), (iii) medium–large (`scd`, `xxhash`), and (iv) large (Unix utilities, `space`, `gravity`). However, this was certainly not exhaustive and larger multilingual open-source projects may provide threats to our empirical findings. In the future, we plan to reduce this threat by first using synthetic large codes, i.e., in millions scale, exponential number of paths, and larger cyclomatic complexity), as Mondal and Nasre (2019) used in their scalability study. Subsequently, we would select larger real world projects from open-source repositories. Regarding the scale of our test-set (totaling 69,305 test-cases), we believe it to be large enough to mitigate any potential external threats. However, our reported parallelization model may not scale to larger multi-core systems, distributed systems, or many-core systems like GPUs.

4.7.2. Internal validity

Current implementation of Hansie may have missed corner situations of dealing with change detection, priority valuations, and sorting. The threat to change detection was reduced by leveraging Linux `diff` utility which is open-source and widely maintained and used. C/C++ implementations of the participating individual prioritizations, the non-participating ART prioritization, and consensus approaches including the publicly available Kemeny consensus C++ program (Press, 2012) may have associated bugs in implementation of individual and consensus approaches. To mitigate this threat, we manually performed a code walk-through, and validated all results for `tcas` during initial stages of construction.

In the future, we would also like to increase the efficiency of our implementation of the adaptive random prioritization for a more comprehensive comparison against this state-of-the-art non-consensus based prioritization.

4.7.3. Construct validity

Measurement strategies used in Hansiewere a possible way of experimental evaluations. Our choice was to disregard fault knowledge although works having controlled experimentation using SIR subjects have extensively utilized this information. We observed that treating each failure as a different fault has already been leveraged in some existing works (Yang et al., 2011; Lidbury et al., 2015; Chen et al., 2018; Miranda et al., 2018; Yu et al., 2019; Peng et al., 2020; Lam et al., 2020). We followed this approach due to a wider scope as software testing can be performed even when fault information was not available. Even in case of multiple-faults and their prior knowledge, prima facie we could not be sure which one of the faults or a combination thereof, caused a failure. To the best of our knowledge, none of the existing works computes APFD or $APFD_c$ when this issue arises, or when composite faults or fault-masking is present. Such an information enforces fault information to be available before testing but our goal in this study was to be realistic in the context of fault-knowledge.

Validity of observations was subject to safety, and change-identifying limitations of the publicly available tool, Mahtab

(Mondal and Nasre, 2019), on top of which our Hansie is implemented. Timing measurements in our evaluation were a mixture of CPU time and wall-clock time. The latter occurred during calculations of parallelization window costs (latencies). Our test-suite may have flaky test-case with non-determinism which may cause deviations in observed values of effectiveness metrics, when evaluated on different execution environments. To the best of our knowledge, the chosen subjects were found to be free from test-flakiness and non-determinism, which we manually inspected during selection of experimental datasets. This reduced the threat to some extent but non-determinism due to parallelism and hidden dependencies may still exist among test-cases.

However, we believe that test-suite independence assumption still holds in our case as found out by prior extensive empirical research on SIR programs. We have made Hansie open-source[17] and would like to mitigate hidden threats to validity in the near future.

Our experimental findings suggest that Hansie should not be used when individual prioritizations are very different or are conflicting, in which case, the overall consensus may not be favorable. This ultimately leads to a prioritization which essentially cancels out benefits of each individual prioritization, and the resulting consensus may be even worse than participating individuals. To mitigate this threat, we plan to augment Hansie using a decision-making ensemble selection module, the role of which is to determine whether a consensus is actually needed. In case a consensus is feasible, the ensemble selector should be able to judiciously select a few rankings that should only improve the quality of consensus prioritization.

5. Related work

There exists a multitude of test-case prioritization approaches in regression testing. In this section, we compare and contrast our approach with the relevant related work, and discuss some of the most widely known approaches in the literature.

5.1. Weighted-sum hybridization and weight-based strategies

The treatment of hybridization, i.e., weighted-sum of priorities due to two heuristics is not new and appears in Shin et al. (2019). The survey paper by Yoo and Harman (2012) also states that prior work (Yoo and Harman, 2007) used weighting factor to combine multiple objectives. Hsu and Orso (2009) used this technique in a tool, namely MINTS, for test-suite minimization. Walcott et al. (2006) employed the same for time-aware test-suite prioritization. Earlier, Sampath et al. (2013) had proposed generalized hybridization, other than the weighted-sum approach.

Carlson et al. (2011) propose a two level cluster-based prioritization heuristic that combines scores of code-complexity and fault-detection ratio per test-case with equal weightage.

Arafeen and Do (2013) perform prioritization of requirement-based test-clusters using weighted-mean of constituent software requirements. This final value denotes priority at the granularity of test-clusters. Further, the authors also use a weighted-sum hybridization of several normalized code metrics: (i) lines of code, (ii) nested block depth, and (iii) Cyclomatic complexity.

Paterson et al. (2019) proposes G-clef that utilizes a bug prediction model to prioritize Java classes. The underlying model is built upon a publicly available tool, namely Schwa (Freitas, 2015). The likelihood measure is calculated by a weighted sum of three parameters: (i) #revisions, (ii) #fixes, and (iii) #authors. The authors empirically recommend weights of (0.6, 0.1, 0.3) for the configuration of Schwa. The higher the likelihood, the higher is the priority of the associated test-case.

The most extensive utilization of the weighted-sum approach occurs in test-execution history-based prioritization algorithms (Kim and Porter, 2002; Wei-Tek Tsai et al., 2005; Fazlalizadeh et al., 2009; Engström et al., 2011; Marijan et al., 2013; Elbaum et al., 2014; Hemmati et al., 2015; Aman et al., 2016; Kwon and Ko, 2017; Yu et al., 2019).

As a general template for history-based approaches, the priority of a test-case for a version v_i is a function of its priorities in the previous revisions. In the existing literature, this function generally comprises of weighted sum as the core component, additionally augmented with approach-specific customizations.

A slightly different weight-based approach appears in Qu et al. (2007) where interaction weights are used to prioritize combinatorial interaction testing (CIT) (Cohen et al., 1997) test-suites. The proposed methodology is an extension of the hybrid regenerate-prioritize algorithm proposed in Bryce and Colbourn (2006), and explores coverage-based and TSL (Ostrand and Balcer, 1988) specification-based weightings.

Although the aforementioned works leverage a similar treatment (weighted-sum), individual components (heuristics) employed are different from that of Hansie. Moreover, Hansie does not consider test-execution history from previous code revisions, bug prediction in affected basic-blocks, combinatorial interactions, and test-suite reduction.

5.2. Prioritization by learning to rank test-cases

Busjaeger and Xie (2016) propose a test-prioritization approach designed on top of a technique named "learning-to-rank" from information retrieval and machine learning domain. The approach systematically integrates five existing prioritization features: (i) code-coverage, (ii) text-path similarity, (iii) text-content similarity, (iv) failure history, and (v) test-age. The linear model is trained using support vector machine (SVM).

For a given set of changes $\Delta = \{\delta_1, \delta_2, \ldots, \delta_m\}$, and universe of test-cases $T = \{t_1, t_2, \ldots, t_n\}$, the ranking model takes the form of a function which produces permutations and optimizes some prioritization-based objective for newly introduced test-cases. In this setting, ranking T translates to sorting T based on scores which quantify the probability that a certain test-case $t_i \in T$ would fail for a particular change $\delta_j \in \Delta$.

The technique is specifically targeted toward industrial settings where heterogeneity plays a major role by taking the form of multilingual projects, non-code changes (Nanda et al., 2011), and larger projects with integration tests instead of units tests.

Focus of traditional academic research on regression test-case prioritization is mainly on scenarios that arise beyond experimental setting. Reported experimental results exhibit a more effective test-case prioritization when compared to individual approaches. However, the approach described by Busjaeger and Xie (2016) contrasts with Hansie as follows. The major difference lies in building a ranking model for machine learning. Although there is some similarity only in the context of on-the-fly hybridization (priority-aware), heuristics employed are different, compared to Hansie. The methodology does not use methods form social choice theory such as Kemeny–Young (Kemeny, 1959; Young and Levenglick, 1978), Borda count (de Borda, 1781), and other classical methods (Dwork et al., 2001; Lv, 2014) of rank aggregation. The second major difference is that Hansie uses parallelization windows for strengthening, weakening, or respecting tied orderings in the final consensus permutation. We also quantify consensus quality (with and without ties) and introduce Hansie distance to overcome a few limitations (Press, 2012) of the classical Kendall-tau distance (Kendall, 1938, 1948).

[17] Hansie is available at https://doi.org/10.5281/zenodo.3988245.

There exist several variants (Fagin et al., 2003, 2004; Betzler et al., 2008; Kumar and Vassilvitskii, 2010) of the classical Kendall-tau distance. However, to the best of our knowledge, all such formulations are different from that of Hansie distance.

5.3. Tie-breaking in test-case prioritization

Most coverage-based test-case prioritizations resolve ties randomly (Rothermel et al., 2001; Elbaum et al., 2002; Yoo and Harman, 2012; Khatibsyarbini et al., 2018). However, a few exceptions are briefly mentioned below.

As a minimalist tie-breaker, the coverage-based technique by Di Nardo et al. (2013) leverages the original relative ordering of test-cases within the unprioritized test-suite, should a tie-breaking is required. Eghbali and Tahvildari (2016), Eghbali et al. (2019) propose defect prediction models for supervised tie-breaking using classifier ensembles.

Shin et al. (2019) present an empirical study on mutation-based test-prioritization using both single and multi-objective optimizations. The two objectives pursued are maximizations of (i) number of killed mutants, and (ii) diversity of mutants. Multiple pareto optimal orderings may be tied for a particular candidature. Ties are broken in either picking the one that speeds-up the kill rate, or maximizes the diversity in terms of mutant distinguishment.

Ali et al. (2019) propose CTFF, a test-prioritization-selection technique for Continuous Integration (CI) systems. Considering test-execution history, primary criteria for prioritization happens to be the frequency of test-failures. Frequently failing test-cases are given more priority. In this scenario, ties are resolved by preferring the test-case with greater code-coverage.

Huang et al. (2014) considers interaction-based test-suite prioritization driven by interaction strength parameter. The authors study five tie-breaking strategies, two of which are deterministic while the rest involve randomization. From among a set of tied test-cases, deterministic breakers select either the first or the last one in the segment. The latter one empirically shows the best performance. Except the random strategy, ties are broken such that a test-case with a higher level of interaction strength gets priority. In this category, random breaker exhibits performance comparable to the best deterministic approach.

Differing from all the existing works, Hansiedoes not break ties, instead it respects tied rankings in test-case orderings by assigning parallel execution windows. This is because breaking ties in a consensus permutation would introduce unnecessary bias toward some individual's ranking, ultimately defeating the purpose of social welfare which is agreement seeking.

5.4. Predictive test-prioritization strategy

Chen et al. (2018) propose *predictive test prioritization* (PTP), a framework that predicts the best-performing strategy chosen from a library of six prioritization approaches: (i) greedy additional, (ii) greedy total, (iii) search-based strategy using genetic algorithm, (iv) random strategy with feedback for readjustment, (v) cost-based additional strategy, and (vi) minimalist cost-only prioritization that reorders tests based only on execution time.

The predictive model works by analyzing three types of test features (distributions) in preceding code versions: (i) coverage, (ii) execution, and (iii) coverage per unit time.

The emphasis is centered around prioritization effectiveness measured along overall execution time as compared to conventional approaches that focus only on number of executed tests. Experimental evaluation of PTP reports an accuracy of 92% on open-source projects from GitHub.

The major difference between Hansie and PTP is that the latter's end result is a single predicted prioritization that uses machine learning to select a single prioritization strategy from multiple alternatives, whereas Hansie computes a consensus reordering of test-cases that respects multiple individual prioritizations, to the extent possible.

5.5. Consensus-based approaches for fault localization

Debroy and Wong (2013) propose the first consensus oriented rank-aggregation in the context of software fault localization. To the best of our knowledge, this is the first work that applies social choice theory to regression testing. The consensus (of three techniques) is performed by the well-established Borda-count method (de Borda, 1781).

Hansie differs from this work by focusing entirely on test-case prioritization which occurs before execution of test-cases, whereas the method of Debroy and Wong (2013) is orthogonal and comes into play when test outcomes are revealed after the execution phase. This forms an important starting point of program debugging and repair.

Ties in the ranked list produced are broken by statement numbers in Debroy and Wong (2013). However, Hansie respects ties without breaking them, and instead executes them using parallel windows.

Additionally, Hansie also performs a weighted-sum hybridization of two heuristics presented in Mondal and Nasre (2019), proposes a distance metric between test-permutations which is defined in the presence of ties. On the other hand, the approach proposed in Debroy and Wong (2013) forces a total order on suspicious ranking of statements and, unlike Hansie, does not provide a tuning parameter (e.g., window size) to weaken or strengthen a consensus ordering.

5.6. Strategies between total and additional heuristics

Zhang et al. (2013) introduce a unification of two widely celebrated prioritization heuristics: *additional* and *total*, using a strategy control parameter p. The value of p helps in obtaining intermediate strategies lying in-between two extremes, and some of them even outperform both greedy additional and total.

This work is orthogonal to our hybridization component which performs the weighted sum after computing priority scores due to two heuristics (Mondal and Nasre, 2019): *relevance* and *confinedness*.

Apart from different heuristics employed in both these works, the approach by Zhang et al. (2013) involves the notion of probability, which systematically assigns decaying weights to code elements each time they are covered by a test-case. This weight is proportional to the probability of faults being present in the code-element. One variant uses a fixed value for parameter p whereas another uses non-uniform values such that different code-elements are treated to have different amounts of fault proneness, based on given code-changes. However, this is not followed in the hybridization component of Hansie where the only control parameter appears in the form of α which weights the relevance score. The spectrum in this case is obtained by adjusting the residual weight $(1 - \alpha)$ to confinedness, based on the code-change being considered in the new version.

Additionally, in Hansie, the hybrid strategies obtained due to a set of empirically chosen α values, take part in the consensus in the form of individual prioritizations. The end-goal thereafter in Hansie is to compute a consensus ordering which takes into account views of all such participating individuals. However, this was not pursued by Zhang et al. (2013) where individual prioritizations are independent and non-participatory.

5.7. Randomized approaches to test-case prioritization

Inspired by the performance of randomized (RT-based) approaches (Chan et al., 2002; Chen et al., 2005; Chen and Merkel, 2007; Chen et al., 2009) to software testing, Jiang et al. (2009) introduced adaptive random (ART) test-case prioritization, which is currently a state-of-the-art technique.

The proposed white-box ART-based prioritization resolves the issue of uneven test-case distribution occurring in traditional RT-based methods. This is achieved by leveraging the white-box test-coverage information to adaptively guide randomization. This informedness comes from feedback measured in terms of test-set distances from already prioritized test-cases.

Currently, Hansie is free from components involving randomized testing. However, the power of Hansie's consensus operators could be tested at a large scale when each individual (participating) test-case permutation corresponds to an output of the ART prioritization algorithm. That way, ART may be further enhanced by a consensus of multiple adaptive randomized prioritizations. For instance, instead of the repetition factor of 30 (Arcuri and Briand, 2014) for ART, the consensus of 30 independent ART permutations may itself yield a different prioritization that may be more effective.

5.8. Clustering-based strategies for test-prioritization

White-box test-case coverage information lies at the heart of any coverage-based prioritization. There exists several approaches in the literature that effectively utilize this information for creation of test-clusters.

Yoo et al. (2009) propose interleaved cluster (ICP) prioritization, wherein agglomerative hierarchical clustering is employed as a preprocessing step to create clusters. The cut level of associated dendrogram acts as a tuning parameter to control the number of clusters. After this process, intra-cluster prioritization sets a representative per cluster, and an inter-cluster prioritization is employed to prioritize the clusters themselves.

The main algorithm (ICP) begins by selecting a representative from each cluster considered in a round-robin fashion, and are subsequently removed. Iteratively next best representatives are selected by interleaving as long as the collection of clusters is non-empty. This way of selecting test-cases ensures coverage diversity of subsequent test-cases in the prioritization order.

Carlson et al. (2011) propose a technique along similar lines but uses Euclidean distance instead of Hamming distance for clustering test-coverages. However, they specifically use (i) code-coverage, (ii) complexity, (iii) fault-history information, and a hybrid-combination (arithmetic mean) of (ii) and (iii), to perform prioritization within and across clusters. This latter combination can be interpreted in terms of Hansie as weighted-sum-hybridization of (50%) code-complexity, and (50%) fault-detection ratio. Further, prioritization is also evaluated using time budgets: 75%, 50%, and 25%, and the empirical evidence shows that clustering is indeed effective under time constraints.

However, this decision does not necessarily guarantee 100% safety of regression testing as some failures may be missed whose underlying cause was an escaped (undetected) fault.

Arafeen and Do (2013) introduce a variation of the formerly mentioned approaches. Instead of hierarchical clustering, K-means is utilized to create requirements-based clusters. Note that this variation requires text-mining of requirements from associated software-requirements-specification documents.

Intra-cluster prioritization is achieved by a ranking based on the importance of constituent requirements. Inter-cluster prioritization is achieved by a tunable weighted-sum of code metrics: (i) LOC, (ii) nesting depth, and (iii) cyclomatic number. The latter approach has three variations: (i) order by cluster formation, (ii) order by cluster rank, and (iii) randomly ordered.

Hansie and its constituent building blocks do not involve clustering of test-cases explicitly as pursed by the aforementioned approaches. However, the individual prioritizations employed in Hansie may potentially form coverage-based clusters (although with respect to different criteria) in the event of coverage redundancy in the regression test-suite.

6. Conclusions and future work

In this paper, we performed a study on hybrid and consensus regression test-case prioritization by developing a framework named Hansie, and introduced its permutation distance metric. We utilized four different heuristics from existing test-case prioritization literature and designed nine more heuristics by a weighted-sum priority-score hybridization of two particular approaches. In total, we employed thirteen individual prioritization strategies (distinguishing weighted variants) and six different consensus approaches.

We showed that consensus prioritizations generally involve ties and stated necessary and sufficient conditions for the same. We further introduce a new latency-cognizant metric for quantifying effectiveness of parallel windows with non-uniform load distribution in the parallel setting.

We evaluated Hansie on twelve subject programs from SIR and eight projects from GitHub. Experimental results showed that two consensus approaches are the most effective, and showed the quality of computed consensus in terms of the proposed Hansie distance and the existing Kendall-tau distance. Finally, we illustrated benefits of test-execution at the level of isolated parallel processes on a multi-core processor.

In this work, we have considered only independent unit test-cases. Therefore, one direction of future work is to extend Hansie for test-case prioritization with test dependencies. One way to achieve this is to specify the dependency relation as a collective preference graph of multiple dependency-driven prioritized permutations. Individual permutations may be obtained due to existing prioritizations that schedule test-cases for safe concurrent execution after finding dependencies (if any). Thereafter, ties in the majority preference graph and the associated consensus can be resolved by respecting the dependency ordering which will most likely be majority votes disregarding the position as in Hansie distance. The final consensus prioritization can then be safely executed using non-uniform parallel windows such that residents of each window are mutually non-conflicting with respect to test dependencies, and windows themselves being prioritized according to the precedence graph of dependencies to avoid conflicts.

Another potential direction of extending our work is to construct a scheduler which allocates test-cases individually or in windows on isolated execution environments, whereby dependencies need not be broken. In this case conflicting test-cases can still be concurrently executed in isolated containers by replicating dependents and their dependees, wherever applicable. The major challenge will be in load-balancing across different environments with non-uniform parallelization windows. This may also lead to extensions of existing effectiveness measurements such as the existing failure-aware EPS, and our load-cognizant EPL.

CRediT authorship contribution statement

Shouvick Mondal: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing - original draft, Writing - review & editing. **Rupesh Nasre:** Writing - review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The second author is partially supported by NSM, India Project CS/19-20/1123/MEIT/008606.

References

Ali, S., Hafeez, Y., Hussain, S., Yang, S., 2019. Enhanced regression testing technique for agile software development and continuous integration strategies. Softw. Qual. J. http://dx.doi.org/10.1007/s11219-019-09463-4.

Aman, H., Tanaka, Y., Nakano, T., Ogasawara, H., Kawahara, M., Application of Mahalanobis-Taguchi method and 0-1 programming method to cost-effective regression testing, in: 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2016, pp. 240–244.

Arafeen, M.J., Do, H., Test case prioritization using requirements-based clustering, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 2013, pp. 312–321.

Arcuri, A., Briand, L., 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. 24 (3), 219–250. http://dx.doi.org/10.1002/stvr.1486, https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1486.

Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C., 2004. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Secure Comput. 1 (1), 11–33. http://dx.doi.org/10.1109/TDSC.2004.2.

Betzler, N., Fellows, M.R., Guo, J., Niedermeier, R., Rosamond, F.A., 2008. Fixed-parameter algorithms for kemeny scores. In: Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management. AAIM '08, Springer-Verlag, Berlin, Heidelberg, pp. 60–71. http://dx.doi.org/10.1007/978-3-540-68880-8_8.

de Borda, J.C., 1781. Memoire sur les elections au scrutin (Mr. de BORDA, 1772). http://asklepios.chez.com/XIX/borda.htm. (Accessed on 14 August 2019).

Bryce, R.C., Colbourn, C.J., 2006. Prioritized interaction testing for pair-wise coverage with seeding and constraints. Inf. Softw. Technol. 48 (10), 960–970. http://dx.doi.org/10.1016/j.infsof.2006.03.004, Advances in Model-based Testing.

Busjaeger, B., Xie, T., 2016. Learning for test prioritization: An industrial case study. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2016, ACM, New York, NY, USA, pp. 975–980. http://dx.doi.org/10.1145/2950290.2983954, http://doi.acm.org/10.1145/2950290.2983954.

Carlson, R., Do, H., Denton, A., A clustering approach to improving test case prioritization: An industrial case study, in: 2011 27th IEEE International Conference on Software Maintenance, ICSM, 2011, pp. 382–391.

Chan, K.P., Chen, T.Y., Towey, D., 2002. Restricted random testing. In: Software Quality — ECSQ 2002. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 321–330.

Chen, T.Y., Kuo, F.-C., Merkel, R.G., Tse, T., 2009. Adaptive random testing: The ART of test case diversity. J. Syst. Softw. 83 (1), 60–66. http://dx.doi.org/10.1016/j.jss.2009.02.022, SI: Top Scholars.

Chen, T.Y., Leung, H., Mak, I.K., 2005. Adaptive random testing. In: Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making. Springer Berlin Heidelberg, pp. 320–329.

Chen, J., Lou, Y., Zhang, L., Zhou, J., Wang, X., Hao, D., Zhang, L., 2018. Optimizing test prioritization via test distribution analysis. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2018, ACM, New York, NY, USA, pp. 656–667. http://dx.doi.org/10.1145/3236024.3236053, http://doi.acm.org/10.1145/3236024.3236053.

Chen, T.Y., Merkel, R., 2007. Quasi-random testing. IEEE Trans. Reliab. 56 (3), 562–568.

Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C., 1997. The AETG system: An approach to testing based on combinatorial design. IEEE Trans. Softw. Eng. 23 (7), 437–444. http://dx.doi.org/10.1109/32.605761.

Davenport, A., Kalagnanam, J., 2004. A computational study of the kemeny rule for preference aggregation. In: Proceedings of the 19th National Conference on Artifical Intelligence. AAAI'04, AAAI Press, pp. 697–702, http://dl.acm.org/citation.cfm?id=1597148.1597260.

Debroy, V., Wong, W.E., 2013. A consensus-based strategy to improve the quality of fault localization. Softw. - Pract. Exp. 43 (8), 989–1011. http://dx.doi.org/10.1002/spe.1146, https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.1146.

Di Nardo, D., Alshahwan, N., Briand, L., Labiche, Y., 2013. Coverage-based test case prioritisation: An industrial case study. In: Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. ICST '13, IEEE Computer Society, USA, pp. 302–311. http://dx.doi.org/10.1109/ICST.2013.27.

Do, H., Elbaum, S.G., Rothermel, G., 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empir. Softw. Eng. Int. J. 10 (4), 405–435.

Dwork, C., Kumar, R., Naor, M., Sivakumar, D., 2001. Rank aggregation methods for the web. In: Proceedings of the 10th International Conference on World Wide Web. WWW '01, ACM, New York, NY, USA, pp. 613–622. http://dx.doi.org/10.1145/371920.372165, http://doi.acm.org/10.1145/371920.372165.

Eghbali, S., Kudva, V., Rothermel, G., Tahvildari, L., 2019. Supervised tie breaking in test case prioritization. In: Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings. ICSE '19, IEEE Press, Piscataway, NJ, USA, pp. 242–243. http://dx.doi.org/10.1109/ICSE-Companion.2019.00095.

Eghbali, S., Tahvildari, L., 2016. Test case prioritization using lexicographical ordering. IEEE Trans. Softw. Eng. 42 (12), 1178–1195. http://dx.doi.org/10.1109/TSE.2016.2550441.

Elbaum, S., Malishevsky, A., Rothermel, G., 2001. Incorporating varying test costs and fault severities into test case prioritization. In: Proceedings of the 23rd International Conference on Software Engineering. ICSE '01, IEEE Computer Society, Washington, DC, USA, pp. 329–338, http://dl.acm.org/citation.cfm?id=381473.381508.

Elbaum, S., Malishevsky, A.G., Rothermel, G., 2002. Test case prioritization: A family of empirical studies. IEEE Trans. Softw. Eng. 28 (2), 159–182. http://dx.doi.org/10.1109/32.988497.

Elbaum, S., Rothermel, G., Penix, J., 2014. Techniques for improving regression testing in continuous integration development environments. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. FSE 2014, ACM, New York, NY, USA, pp. 235–245. http://dx.doi.org/10.1145/2635868.2635910, http://doi.acm.org/10.1145/2635868.2635910.

Engström, E., Runeson, P., Ljung, A., 2011. Improving regression testing transparency and efficiency with history-based prioritization – an industrial case study. In: Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation. ICST '11, IEEE Computer Society, USA, pp. 367–376. http://dx.doi.org/10.1109/ICST.2011.27.

Epitropakis, M.G., Yoo, S., Harman, M., Burke, E.K., 2015. Empirical evaluation of Pareto efficient multi-objective regression test case prioritisation. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. ISSTA 2015, ACM, New York, NY, USA, pp. 234–245. http://dx.doi.org/10.1145/2771783.2771788, http://doi.acm.org/10.1145/2771783.2771788.

Fagin, R., Kumar, R., Mahdian, M., Sivakumar, D., Vee, E., 2004. Comparing and aggregating rankings with ties. In: Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. PODS '04, ACM, New York, NY, USA, pp. 47–58. http://dx.doi.org/10.1145/1055558.1055568, http://doi.acm.org/10.1145/1055558.1055568.

Fagin, R., Kumar, R., Sivakumar, D., 2003. Comparing top k lists. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. SODA '03, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, pp. 28–36, http://dl.acm.org/citation.cfm?id=644108.644113.

Fazlalizadeh, Y., Khalilian, A., Azgomi, M.A., Parsa, S., Prioritizing test cases for resource constraint environments using historical test case performance data, in: 2009 2nd IEEE International Conference on Computer Science and Information Technology, 2009, pp. 190–195.

Freitas, A., 2015. Schwa: A tool that analyzes GIT repositories and estimates the defect probability of software components to help developers focusing their resources to fix bugs where they really are. https://github.com/andrefreitas/schwa. (Accessed on 25 April 2020).

Hemmati, H., Fang, Z., Mantyla, M.V., Prioritizing manual test cases in traditional and rapid release environments, in: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST, 2015, pp. 1–10.

Hsu, H.-Y., Orso, A., 2009. MINTS: A general framework and tool for supporting test-suite minimization. In: Proceedings of the 31st International Conference on Software Engineering. ICSE '09, IEEE Computer Society, Washington, DC, USA, pp. 419–429. http://dx.doi.org/10.1109/ICSE.2009.5070541.

Huang, R., Chen, J., Wang, R., Chen, D., 2014. How to do tie-breaking in prioritization of interaction test suites?. In: Reformat, M. (Ed.), The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1–3, 2013. Knowledge Systems Institute Graduate School, pp. 121–125.

Ionescu, V., 2011. Arrays - calculating the number of "inversions" in a permutation - stack overflow. https://stackoverflow.com/questions/6523712/calculating-the-number-of-inversions-in-a-permutation/6523781#6523781. (Accessed on 25 May 2020).

Jiang, B., Zhang, Z., Chan, W.K., Tse, T.H., 2009. Adaptive random test case prioritization. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. ASE '09, IEEE Computer Society, Washington, DC, USA, pp. 233–244. http://dx.doi.org/10.1109/ASE.2009.77.

Kemeny, J.G., 1959. Mathematics without numbers. Daedalus 88 (4), 577–591, http://www.jstor.org/stable/20026529.

Kendall, M.G., 1938. A new measure of rank correlation. Biometrika 30 (1–2), 81–93. http://dx.doi.org/10.1093/biomet/30.1-2.81.

Kendall, M.G., 1948. Rank correlation methods. https://psycnet.apa.org/record/1948-15040-000. (Accessed on 14 August 2019).

Khatibsyarbini, M., Isa, M.A., Jawawi, D.N., Tumeng, R., 2018. Test case prioritization approaches in regression testing: A systematic literature review. Inf. Softw. Technol. 93, 74–93. http://dx.doi.org/10.1016/j.infsof.2017.08.014, http://www.sciencedirect.com/science/article/pii/S0950584916304888.

Kim, J.-M., Porter, A., 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In: Proceedings of the 24th International Conference on Software Engineering. ICSE '02, ACM, New York, NY, USA, pp. 119–129. http://dx.doi.org/10.1145/581339.581357, http://doi.acm.org/10.1145/581339.581357.

Knight, W.R., 1966. A computer method for calculating Kendall's tau with ungrouped data. J. Amer. Statist. Assoc. 61 (314), 436–439. http://dx.doi.org/10.1080/01621459.1966.10480879.

Kumar, R., Vassilvitskii, S., 2010. Generalized distances between rankings. In: Proceedings of the 19th International Conference on World Wide Web. WWW '10, ACM, New York, NY, USA, pp. 571–580. http://dx.doi.org/10.1145/1772690.1772749, http://doi.acm.org/10.1145/1772690.1772749.

Kwon, J., Ko, I., Cost-effective regression testing using bloom filters in continuous integration development environments, in: 2017 24th Asia-Pacific Software Engineering Conference (APSEC), 2017, pp. 160–168.

Lam, W., Shi, A., Oei, R., Zhang, S., Ernst, M.D., Xie, T., 2020. Dependent-test-aware regression testing techniques. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2020, Association for Computing Machinery, New York, NY, USA, pp. 298–311. http://dx.doi.org/10.1145/3395363.3397364.

Lidbury, C., Lascu, A., Chong, N., Donaldson, A.F., 2015. Many-core compiler fuzzing. SIGPLAN Not. 50 (6), 65–76. http://dx.doi.org/10.1145/2813885.2737986, https://doi.org/10.1145/2813885.2737986.

Lv, G., 2014. An analysis of rank aggregation algorithms. https://arxiv.org/pdf/1402.5259.pdf. (Accessed on 14 August 2019).

Marijan, D., Gotlieb, A., Sen, S., Test case prioritization for continuous regression testing: An industrial case study, in: 2013 IEEE International Conference on Software Maintenance, 2013, pp. 540–543.

Marijan, D., Liaaen, M., 2018. Practical selective regression testing with effective redundancy in interleaved tests. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice. ICSE-SEIP '18, ACM, New York, NY, USA, pp. 153–162. http://dx.doi.org/10.1145/3183519.3183532, http://doi.acm.org/10.1145/3183519.3183532.

McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. 2 (4), 308–320. http://dx.doi.org/10.1109/TSE.1976.233837.

Miranda, B., Cruciani, E., Verdecchia, R., Bertolino, A., 2018. FAST Approaches to scalable similarity-based test case prioritization. In: Proceedings of the 40th International Conference on Software Engineering. ICSE '18, ACM, New York, NY, USA, pp. 222–232. http://dx.doi.org/10.1145/3180155.3180210, http://doi.acm.org/10.1145/3180155.3180210.

Mondal, S., Nasre, R., 2019. Mahtab: Phase-wise acceleration of regression testing for C. J. Syst. Softw. 158, 110403. http://dx.doi.org/10.1016/j.jss.2019.110403, http://www.sciencedirect.com/science/article/pii/S0164121219301773.

Nanda, A., Mani, S., Sinha, S., Harrold, M.J., Orso, A., 2011. Regression testing in the presence of non-code changes. In: Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation. ICST '11, IEEE Computer Society, Washington, DC, USA, pp. 21–30. http://dx.doi.org/10.1109/ICST.2011.60, http://dx.doi.org/10.1109/ICST.2011.60.

Ostrand, T.J., Balcer, M.J., 1988. The category-partition method for specifying and generating fuctional tests. Commun. ACM 31 (6), 676–686. http://dx.doi.org/10.1145/62959.62964.

Paterson, D., Campos, J., Abreu, R., Kapfhammer, G.M., Fraser, G., McMinn, P., An empirical study on the use of defect prediction for test case prioritization, in: 2019 12th IEEE Conference on Software Testing, Validation and Verification, ICST, 2019, pp. 346–357.

Peng, Q., Shi, A., Zhang, L., 2020. Empirically revisiting and enhancing IR-based test-case prioritization. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2020, Association for Computing Machinery, New York, NY, USA, pp. 324–336. http://dx.doi.org/10.1145/3395363.3397383.

Perez, A., Abreu, R., D'Amorim, M., 2017. Prevalence of single-fault fixes and its impact on fault localization. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE, New York, NY, USA, pp. 12–22. http://dx.doi.org/10.1109/ICST.2017.9.

Press, W.H., 2012. C++ program for Kemeny-Young preference aggregation. http://numerical.recipes/whp/ky/kemenyyoung.html. (Accessed 13 August 2019).

Qu, X., Cohen, M.B., Woolf, K.M., Combinatorial interaction regression testing: A study of test case generation and prioritization, in: 2007 IEEE International Conference on Software Maintenance, 2007, pp. 255–264.

Rothermel, G., Untch, R.J., Chu, C., 2001. Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. 27 (10), 929–948. http://dx.doi.org/10.1109/32.962562.

Sampath, S., Bryce, R., Memon, A., 2013. A uniform representation of hybrid criteria for regression testing. IEEE Trans. Softw. Eng. 39 (10), 1326–1344. http://dx.doi.org/10.1109/TSE.2013.16.

Sánchez, A.B., Segura, S., Ruiz-Cortés, A., A comparison of test case prioritization criteria for software product lines, in: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, 2014, pp. 41–50.

Shin, D., Yoo, S., Papadakis, M., Bae, D.-H., 2019. Empirical evaluation of mutation-based test case prioritization techniques. Softw. Test. Verif. Reliab. 29 (1–2), e1695. http://dx.doi.org/10.1002/stvr.1695, https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1695, e1695 stvr.1695.

Stratis, P., 2017. Improving test execution time with improved cache locality. In: Proceedings of the 39th International Conference on Software Engineering Companion. ICSE-C '17, IEEE Press, Piscataway, NJ, USA, pp. 82–84. http://dx.doi.org/10.1109/ICSE-C.2017.153.

Stratis, P., Rajan, A., 2018. Speeding up test execution with increased cache locality. http://homepages.inf.ed.ac.uk/arajan/My-Pubs/STVR2018.pdf. (Accessed on 12 February 2019).

Terzi, E., 2015. Presentation: Comparing and aggregating rankings. http://cs-people.bu.edu/evimaria/Italy-2015/voting.pdf. (Accessed on 13 August 2019).

Torres, W.N.M., Alves, E.L.G., Machado, P.D.L., An empirical study on the spreading of fault revealing test cases in prioritized suites, in: 2019 IEEE 43rd Annual Computer Software and Applications Conference, COMPSAC, vol. 1, 2019, pp. 129–138.

Truchon, M., 1998. Figure skating and the theory of social choice. Cahiers de recherche 9814, Université Laval - Département d'économique, https://ideas.repec.org/p/lvl/laeccr/9814.html.

Valgrind Developers, 2019. Valgrind. http://valgrind.org/docs/manual/cg-manual.html#cg-manual.running-cachegrind. (Accessed on 13 October 2019).

Walcott, K.R., Soffa, M.L., Kapfhammer, G.M., Roos, R.S., 2006. Time-aware test suite prioritization. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis. ISSTA '06, ACM, New York, NY, USA, pp. 1–12. http://dx.doi.org/10.1145/1146238.1146240, http://doi.acm.org/10.1145/1146238.1146240.

Wei-Tek Tsai, Xiaoying Bai, Yinong Chen, Xinyu Zhou, Web service group testing with windowing mechanisms, in: IEEE International Workshop on Service-Oriented System Engineering (SOSE'05), 2005, pp. 213–218.

Yang, X., Chen, Y., Eide, E., Regehr, J., 2011. Finding and understanding bugs in c compilers. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '11, Association for Computing Machinery, New York, NY, USA, pp. 283–294. http://dx.doi.org/10.1145/1993498.1993532.

Yoo, S., Harman, M., 2007. Pareto Efficient multi-objective test case selection. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis. ISSTA '07, ACM, New York, NY, USA, pp. 140–150. http://dx.doi.org/10.1145/1273463.1273483, http://doi.acm.org/10.1145/1273463.1273483.

Yoo, S., Harman, M., 2012. Regression testing minimization, selection and prioritization: A survey. Softw. Test. Verif. Reliab. 22 (2), 67–120. http://dx.doi.org/10.1002/stv.430.

Yoo, S., Harman, M., Tonella, P., Susi, A., 2009. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis. ISSTA '09, Association for Computing Machinery, New York, NY, USA, pp. 201–212. http://dx.doi.org/10.1145/1572272.1572296.

Young, H., Levenglick, A., 1978. A consistent extension of condorcet's election principle. SIAM J. Appl. Math. 35 (2), 285–300, http://www.jstor.org/stable/2100667.

Yu, Z., Fahid, F., Menzies, T., Rothermel, G., Patrick, K., Cherian, S., 2019. Terminator: Better automated UI test case prioritization. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2019, ACM, New York, NY, USA, pp. 883–894. http://dx.doi.org/10.1145/3338906.3340448, http://doi.acm.org/10.1145/3338906.3340448.

Zhang, L., Hao, D., Zhang, L., Rothermel, G., Mei, H., 2013. Bridging the gap between the total and additional test-case prioritization strategies. In: Proceedings of the 2013 International Conference on Software Engineering. ICSE '13, IEEE Press, Piscataway, NJ, USA, pp. 192–201, http://dl.acm.org/citation.cfm?id=2486788.2486814.

Shouvick Mondal is a Ph.D. candidate in the Department of Computer Science and Engineering, Indian Institute of Technology Madras, Chennai, India. His research interests are Program Analysis and Testing, Digital Geometry, and Data Mining.

Rupesh Nasre received the Ph.D. degree from the Indian Institute of Science, Bengaluru, India. He is an Assistant Professor with the CSE Department, Indian Institute of Technology Madras, Chennai, India. He was a Post-Doctoral Fellow with the University of Texas at Austin, Austin, TX, USA, from 2011 to 2013. His current research interests include compilers and parallelization.