Using the Case-Based Ranking Methodology for Test Case Prioritization

Paolo Tonella, Paolo Avesani, Angelo Susi
ITC-irst, Trento, Italy
{tonella, avesani, susi}@itc.it

Abstract

The test case execution order affects the time at which the objectives of testing are met. If the objective is fault detection, an inappropriate execution order might reveal most faults late, thus delaying the bug fixing activity and eventually the delivery of the software. Prioritizing the test cases so as to optimize the achievement of the testing goal has potentially a positive impact on the testing costs, especially when the test execution time is long.

Test engineers often possess relevant knowledge about the relative priority of the test cases. However, this knowledge can be hardly expressed in the form of a global ranking or scoring. In this paper, we propose a test case prioritization technique that takes advantage of user knowledge through a machine learning algorithm, Case-Based Ranking (CBR). CBR elicits just relative priority information from the user, in the form of pairwise test case comparisons. User input is integrated with multiple prioritization indexes, in an iterative process that successively refines the test case ordering. Preliminary results on a case study indicate that CBR overcomes previous approaches and, for moderate suite size, gets very close to the optimal solution.

1. Introduction

Testing amounts for a large proportion of the software development and evolution effort. This is especially true for the system level testing, that typically occurs before each major release of the software. During system testing the whole application is exercised in a realistic setting. Correspondingly, the opportunities for automation are often inferior with respect to the previous testing phases (unit and integration). In fact, it might be hard to run the whole application unattended and to simulate any asynchronous input (e.g., interactive inputs) the application may receive. In such cases, system testing can last days or weeks and can involve substantial human effort.

Test case prioritization aims at finding an execution order for the test cases which maximizes a given objective function. Among the others, the most important prioritization objective is probably discovering faults as early as possible, that is, maximizing the *rate of fault detection*. In fact, early feedback about faults allows anticipating the costly activities of debugging and corrective maintenance, with a related economical return. When the time necessary to execute all test cases is long, prioritizing them so as to discover most faults early might save substantial time, since bug fixing can start earlier.

Previous work on test case prioritization [6, 11, 13, 14, 15] is based on the computation of a prioritization index, which determines the ordering of the test cases (e.g., by decreasing values of the index). For example, the coverage level achieved by each test case was used as a prioritization index [13]. Another example is a fault proneness index computed from a set of software metrics for the functions exercised by each test case [6].

In this paper, we propose to incorporate user knowledge into the prioritization process and to integrate multiple prioritization indexes through the CBR (Case-Based Ranking) machine learning algorithm. CBR learns the target ranking from two inputs: a set of possibly partial indicators of priority and pairwise comparisons elicited from the user (cases). On one hand, all the information that can be gathered automatically about the test cases (coverage levels, fault proneness metrics, etc.) is used by CBR to approximate the target ranking. On the other hand, the user is involved in the prioritization process to resolve the cases where contradictory or insufficient data are available. The contribution required from the user consists of very local information and has the form of a pairwise comparison. Given two test cases, the user is requested to indicate the one that should be given higher priority. No quantification and no global evaluation is required. No consistency, such as transitivity, in the elicitation process is assumed. CBR operates iteratively and it produces a provisional ordering at each iteration. Thus, prioritization can be stopped at any time and CBR provides the user with the last ordering produced. Thus, the human effort dedicated to the prioritization process can be calibrated arbitrarily.

The main contributions of this paper over the state of the art are:

- A novel approach to test case prioritization, that incorporates knowledge elicited from the user into the process, thanks to machine learning.
- The capability to deal with multiple, diverse prioritization indexes, that are integrated automatically, without any a-priori specification of the relative weights.
- The capability to handle partial information (e.g., known only for a few test cases), inconsistent information and high-level information (e.g., estimates produced during test planning and test specification).
- Applicability in several phases/contexts, such as during initial test case specification, when information is mostly qualitative and the user has a central role; during regression testing, when information collected during previous test runs can be reused; at major releases, when historical information is only partially usable and qualitative information must be integrated for the new features being tested.

We obtained preliminary experimental results on the case study `space`, which has become a benchmark for test case prioritization techniques. CBR was compared with two existing techniques, widely referenced in the literature. Results indicate that CBR outperforms them at any suite size and, for moderate suite size, it gets very close to the optimal solution.

The paper is organized as follows: Section 2 summarizes the test case prioritization problem, describes our approach and gives some details about the machine learning algorithm used by CBR. Section 3 contains the experimental results. Related works are discussed in Section 4. Conclusions are drawn in Section 5, followed by an anticipation of the plan for our future work.

2. Case-based ranking

Test case prioritization aims at finding an ordering of the test cases such that the execution of the test cases in that order meets (or gets close to meeting) a given criterion. Among the objectives of test case prioritization, the most important one is probably maximizing the rate of fault detection, which consists of revealing faults as early as possible in the testing process. Other objectives include the ability to reveal high-risk faults early, or to reveal faults associated with specific code changes, or to achieve the target coverage or reliability level as early as possible.

In general, given a test suite TS consisting of the test cases $\{t_1, ..., t_n\}$, every ordering of the test cases, such as $< t_{i_1}, ..., t_{i_n} >$, is associated with a *reward value* through a *reward function* G. The goal of prioritization is finding the ordering which maximizes G. Assuming that the ordering is induced by the decreasing values of an *order function* H (i.e., $H(t_{i_1}) > ... > H(t_{i_n})$), the goal can be restated as estimating the order function H which maximizes G.

test	fault				
t_1	$\times$	$\times$	$\times$	$\times$	
t_2	$\times$			$\times$	$\times$
t_3	$\times$		$\times$		
t_4					$\times$

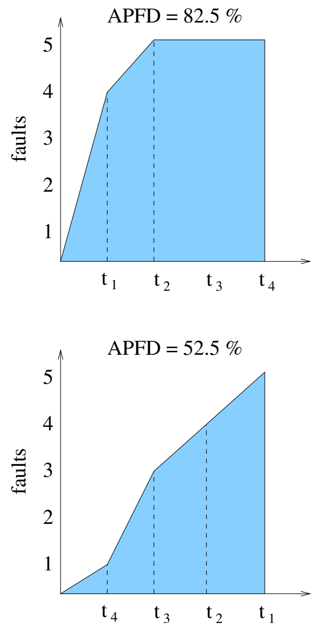

Figure 1. APFD is higher for test case orders that reveal most faults early.

Let us consider the goal of maximizing the rate of fault detection. In such a case, a possible choice for the reward function G is the weighted average of the percentage of faults detected (APFD) [13]. As shown in Figure 1, the APFD tends to have high values when the ordering of the test cases is such as to reveal most faults early. Intuitively, the APFD is the portion of area below the curve in Figure 1. Formally, the APFD can be computed according to the following equation:

$$APFD = 1 - \frac{\sum_{j=1}^{k} pos(e_j)}{nk} + \frac{1}{2n} \qquad (1)$$

where n is the number of test cases, k is the number of revealed faults and $pos(e_j)$ is the position (between 1 and n) of the first test case revealing the fault e_j, in the prioritized sequence.

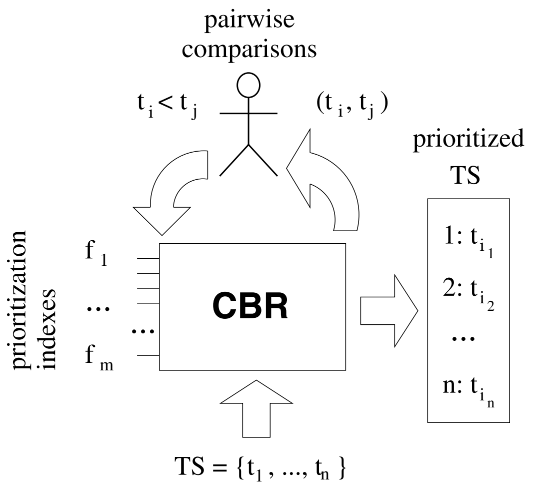

Figure 2. CBR learns the target ranking of the test cases in TS from the prioritization indexes and from user input, in the form of pairwise comparisons.

2.1. The prioritization process

We reformulate the test case prioritization problem as a machine learning problem. The goal is to approximate the order function H, by learning it from metrics, that are supposed to be related with the fault revealing ability of the test cases, and from knowledge about the relative priority of pairs of test cases, as elicited from the user. The Case Based Ranking (CBR) algorithm fits this goal, since it is specifically designed to integrate multiple ranking indexes with case based knowledge having the form of pairwise comparisons.

Figure 2 gives a high level view of the proposed process. CBR transforms a test suite TS, consisting of an unordered set of test cases $t_1, ..., t_n$, into the prioritized sequence $< t_{i_1}, ..., t_{i_n} >$. It takes two inputs: one or more prioritization indexes $f_1, ..., f_m$ and a sample of cases elicited from the user, consisting of test case pairs (t_i, t_j) that the user is requested to compare (e.g., deciding that $t_i < t_j$, or t_i has a higher value of the order function H than t_j).

The prioritization indexes are supposed to be tentative approximations of the order function H. Thus, for each test case $t_1, ..., t_n$, the i-th index values $f_i(t_1), ..., f_i(t_n)$ define a ranking approximating the one produced by H. Examples of prioritization indexes are: (1) the level of coverage achieved by each test case–e.g., number of statements or branches traversed during test case execution; (2) complexity metrics, such as the cyclomatic complexity, computed for the procedures executed by each test case; (3) the priority or criticality of the requirements associated with each test case; (4) historical information, such as fault prone modules known from the past that are executed by the given test case; etc. The set of indexes used for prioritization are not required to be coherent with each other. This means that individual indexes might produce different orderings. It is the machine learning algorithm CBR that takes care of reconciling such differences.

CBR resorts to user knowledge each time it is hard to decide on the relative ordering of two test cases based solely on the prioritization indexes. The information elicited from the user is pretty local, consisting just of the comparison between two test cases to decide their relative importance. The user is not asked to score all of the test cases (as happens with the prioritization indexes), neither is she/he required to quantify the difference between the two compared test cases. Thus, the user input is compatible with the cognitive abilities of humans, who can manage pretty well local and limited information (such as a pairwise comparison), while they have a hard time with large amounts of data to consider simultaneously.

During training, CBR iteratively involves the user and exploits the elicited information to refine the test case ranking produced so far. The learning process can be interrupted at any time to produce the resulting prioritized test sequence. Clearly, higher number of iterations are expected to produce higher quality orderings. So, there is a trade-off between manual effort involved (i.e., number of elicited pairs) and quality of the prioritization (e.g., APFD value).

The proposed prioritization method enjoys the following properties:

Multiple indexes: CBR can handle multiple prioritization indexes, which are automatically integrated during learning. No relative weights have to be specified externally for the various indexes. Weights are automatically learned by the algorithm.

User centered: CBR gives the user some control about the prioritization process. Her/his knowledge, possibly obtained during previous releases of the software, is smoothly integrated with the prioritization indexes, in order to steer the prioritization process.

Partial information: CBR manages the case where the available information is not complete. For example, prioritization indexes may be defined only for subsets of test cases. Moreover, as explained above, the elicited pairs are just a sample of all possible pairs.

Incoherent data: CBR is robust with respect to contradictory data, such as pairwise comparisons that do not respect transitivity and indexes that induce incompatible rankings.

Wide applicability: CBR makes minimal assumptions about the input it requires, which makes it applicable in several contexts. For example, CBR can be used during the initial specification of the test cases, when no execution information is yet available, since it can exploit indexes obtained directly from the specifications. Of course, it can be used during regression testing, when it can reuse information from previous test case executions (e.g., coverage information) as well as previously elicited pairwise comparisons. But it can be also used in a regression testing context where only a (small) fraction of available information can be reused. This happens when the code changes make most test cases no longer usable. In fact, the partial information available from previous runs can be complemented with information determined for the new test cases (e.g., metrics). Thus, CBR can prioritize existing test cases as well as new ones, or a mixture of both, and it is applicable both at the early stages of the testing phase and during regression testing.

2.2. The machine learning algorithm

The kernel of the case-based iterative process for test case ranking is a learning algorithm based on the boosting approach [9]. Boosting is a framework to combine simple learners into one more general and effective learner. In the following we will refer to Rankboost [8], a boosting algorithm designed for ranking learning.

The learning algorithm input is threefold: a finite set of test cases $TS = \{t_1, \ldots, t_n\}$, the prioritization indexes $F = \{f_1 \ldots f_m\}$ and a sample of pairwise priority relations on test cases $\Phi = \{(t_i, t_j) \in TS \times TS | \phi(t_i, t_j) \neq 0\}$.

The function ϕ describes the priority index elicited from the users in terms of pair relations: $\phi : TS \times TS \rightarrow \{-1, 0, 1\}$. In particular, $\phi(t_i, t_j) = 1$ means that $t_j \prec t_i$, $\phi(t_i, t_j) = -1$ means that $t_i \prec t_j$, and $\phi(t_i, t_j) = 0$ indicates that no preference related to the pair t_i and t_j has been elicited from the user[1].

The output of the learning algorithm is a ranking function $H : TS \rightarrow \mathbb{R}$ such that $t_i \prec t_j$ if $H(t_i) > H(t_j)$. The goal of the computation of $H(t)$ is to obtain a ranking function according to the sample Φ while producing a good approximation for those relations not elicited from the users. The working assumption here is that the sample Φ is a partial representation of the optimal ranking. Of course, in practice, the feedback provided by the user is not necessarily an accurate representation of the optimal ranking, since the elicitation process is usually noisy and non monotonic. For simplicity, in the following we assume an omniscient user, therefore Φ can be conceived as a sample of the optimal ranking.

As mentioned above, a boosting algorithm is a composition schema of simpler learners. The basic idea is that it is possible to obtain highly accurate ranking prediction rules combining many weak ranking prediction rules which may be moderately accurate. The structure of the algorithm is based on an iterative process, namely the boosting cycle, to produce the simple learners and a final step where the simple learners are combined together to obtain the final hypothesis for the ranking function.

The boosting cycle relies on the notion of density function defined over the sample Φ: $D : TS \times TS \rightarrow \mathbb{R}$ such that $D(t_i, t_j) = \gamma \cdot max(0, \phi(t_i, t_j))$ where, while setting to 0 all negative entries of ϕ, γ is a positive constant chosen in such a way that D is a distribution, satisfying the following normalization property: $\sum_{t_i, t_j} D(t_i, t_j) = 1$. The intuitive idea is that a boosting cycle doesn't address homogeneously the learning task over the sample Φ. A weak learner is trained according to the distribution D.

The pseudocode of the learning algorithm is summarized in Figure 3. The fixed threshold of the boosting loop B can be replaced by a conditional stop criterion, for example when a stable ranking hypothesis is achieved. Let us focus our attention on the three main steps of the boosting cycle.

Step 1. A weak ranking hypothesis is computed for the test cases taking into account the distribution D_b ($h_c : TS \rightarrow \mathbb{R}$). The prioritization indexes F are exploited to formulate a ranking function that maximizes the known pair relations Φ according to the weight of distribution D_b. In the experiments, we exploited the technique described in [8] as WeakLearner. It is a binary classifier that at every iteration b produces a dichotomy on the set of test cases. The result is a ranking hypothesis defined as a bipartition of the test cases.

Step 2. A value for the parameter α_b is computed. This pa-

[1] $\phi(t_i, t_i) = 0$ and $\phi(t_i, t_j) = -\phi(t_j, t_i)$ for all $t_i, t_j \in TS$

```
Algorithm RankBoost
    Input:
        $TS$     /* the set of test cases */
        $F$      /* the set of prioritization indexes */
        $\Phi$   /* the set of pairs relations */
    Output:
        $H(t)$   /* the ranking function */
begin
    $D_1$ = initialize($\Phi$);
        /* where $D(t_i, t_j) = \gamma \cdot max(0, \phi(t_i, t_j))$*/
    for $b = 1$ to $B$
            /* boosting cycle */
(1)     $h_b$ = WeakLearner($TS$, $F$, $D_b$);
            /* where $h_b : TS \rightarrow \mathbb{R}$ */
(2)     $\alpha_b$ = ChooseAlpha($D_b$, $h_b$);
            /* where $\alpha_b \in \mathbb{R}$ */
(3)     $D_{b+1}(t_i, t_j) = \frac{D_b(t_i,t_j)}{Z_b} e^{\alpha_t(h_b(t_i) - h_b(j))}$
            /* where $D_{b+1} : TS \times TS \rightarrow \mathbb{R}$ */
    end for
    return $H(t) = \sum_{b=1}^{B} \alpha_b h_b(t)$;
end.
```

Figure 3. Pseudocode of the learning algorithm.

rameter defines the weight of the current weak ranking hypothesis for the linear combination of the final ranking function. The value is a measure of the accuracy of the partial order h_b with respect to the final order H.

Step 3. A new distribution D_b is computed that will be given as input to the next cycle of boosting. The basic idea is to revise the distribution on known pair relations in such a way that at the next cycle the learning effort will be directed towards those pairs that the current weak hypothesis h_b doesn't rank accurately.

It is straightforward to notice that the choice of the parameters α_b is a key factor in the learning algorithm. The objective of the computation of the ranking function $H(t)$ is to minimize the misordered pairs. We can introduce the notion of ranking loss ($rloss$) as the portion of misorderings weighted by their criticality given by the distribution D_b:

$$\begin{aligned} rloss_D(H) &= \sum_{t_i,t_j} D(t_i,t_j)[\![H(t_j) \le H(t_i)]\!] \\ &= Pr_{(t_i,t_j)\sim D}[H(t_j) \le H(t_i)] \end{aligned}$$

where $[\![H(t_j) \le H(t_i)]\!] = 1$ if $H(t_j) \le H(t_i)$ is true, 0 otherwise. We forward to [8] for the methods to compute the parameters α_b that minimize the ranking loss.

3. Experimental results

We obtained preliminary results on the applicability of the proposed approach by prioritizing the test cases for the case study program `space`. The program `space` was obtained from the Subject Infrastructure Repository (SIR) [2], a repository of Java and C programs available for experimentation of source code analysis and testing techniques. These programs are distributed with material (such as coverage information or fault matrix) that facilitates their use for research purposes. In particular, `space` has been employed extensively as a benchmark to compare alternative test case prioritization techniques [4, 5, 6, 10, 11, 13].

`Space` is an interpreter that accepts input written in an array definition language. When no parse error occurs, the output of `space` consists of a list of array elements, with positions and excitations. `Space` was developed by ESA (European Space Agency). Its size is non trivial (9564 lines of code or 6218 executable lines of code; 136 functions). It is written in C and it is distributed at SIR with 1000 test suites having an average size of 155 (min: 141; max: 169). Moreover, it comes with faults and coverage information. Each of the 38 faults that are known for this program are related to the test cases that are able to reveal them. Such relationship is encoded as a fault matrix. The statements exercised by each test case are also provided.

3.1. Setting

Since the number of pairwise comparisons that can be reasonably elicited from the user is limited, we expect that the performance of the proposed technique depends critically on the test suite size. In fact, assuming anti-symmetry, but not transitivity, complete information is obtained by eliciting $N(N-1)/2$ pairs for a test suite of size N. As N increases, only a fraction of the total number of pairs can be sampled, with a corresponding performance degradation.

In order to investigate the behavior of CBR at variable test suite size, we sub-sampled the available suites and we produced test suites with a size ranging from 10 to 100, at increments of 10. Then, user information was elicited iteratively. For this study we made the hypothesis of an *ideal* user, that is, a user who never makes errors when requested to perform pairwise comparisons. We expect CBR to be quite robust with respect to the presence of small amounts of noise (errors), but the precise investigation of this aspect of the algorithm requires the definition of a model for the user errors, which goes beyond the scope of this preliminary study.

The second input required by CBR is a set of prioritization indexes, computed for the given test cases. In this study, we used two indexes: statement coverage and cyclomatic complexity. The *statement coverage index* measures the number of statements executed by each test case. It has been widely used in the test case prioritization literature. The *cyclomatic complexity index* gives the sum of the cyclomatic complexity metrics computed for the functions executed by each test case. Given the control flow graph of a function, its cyclomatic complexity metric can be computed as $e-n+1$, with e the number of edges and n the number of nodes. The motivation for choosing this index is that more complex functions are likely to be more fault prone (there is an ongoing debate on predictors of fault proneness, but this is out of scope here). Of course, other complexity metrics, such as Halstead's, could be added to the indexes used by CBR.

The prioritized test suites have been saved regularly during the successive refinement iterations involving the elicitation of pairwise comparisons. Correspondingly, the value of the APFD (the prioritization criterion we adopted is maximum rate of fault detection) was computed over time.

The results produced by CBR have been compared against optimal and random prioritizations, to verify whether our technique performs better than random and whether it gets close enough to the optimum. As further comparisons, we considered two prioritization techniques that have been widely investigated in the literature, statement coverage and additional statement coverage [4, 5, 6, 13]. Below is a short description of the prioritization techniques compared to CBR:

Optimal: The ordering which maximizes APFD. Since the number of permutations grows exponentially with the test suite size, the "true" optimal ordering cannot be determined exactly. It can only be approximated, for example by means of a greedy algorithm, that iteratively adds the test case with the highest number of revealed faults, among those not yet discovered, to the prioritized suite.

Random: The test cases in each suite are ordered randomly.

Statement coverage (Stmt cov): Test cases are ordered by decreasing number of covered statements. The rationale behind this technique is that test cases exercising a high number of statements are more likely to reveal faults than test cases exercising a lower number of statements.

Additional statement coverage (Add stmt cov): Test cases are ordered based on the number of covered statements among those not yet covered by test cases added to the prioritized suite previously. When all statements exercised by some test case are covered, the coverage achieved by the previous test cases is reset to zero and prioritization restarts as at the beginning.

3.2. Data

Figure 4 shows the performance of CBR at different test suite size, compared with Optimal, Statement coverage, Additional statement coverage, and Random. The value of APFD reported for CBR is the maximum obtained in the pair elicitation process. The total number of elicited pairs was kept linear with the suite size (βN, with β between 2 and 10, and N the suite size).

When the suite size is below 50, CBR max gets very close to the optimum and outperforms the other methods. At suite size 10, CBR produces a prioritized suite with APFD very close to the optimum, while both Stmt cov and Add stmt cov are around 10% far from the optimum. Above size 50, CBR max continues to outperform the other two methods (as well as Random), but there is some residual margin for further improvement (around 5%). The explanation for this may be that at this size the exploration of the possible pairs gets more and more limited, since their number grows quadratically with the size. Notwithstanding the limited exploration of the possible pairs, CBR can take advantage of such information to make some improvement of the APFD values, over the other methods. Overall, the best performances of CBR, in terms of getting close to the optimum and surpassing the other techniques, are for a suite size of 60 or less. It should be noticed that the high variability of the histograms plotted in Figure 4 depends on the specific characteristics of the considered test suites. Replication with more suites would be required to absorb the individual differences and produce results that are less dependent on the specific suites selected.

Figure 5 shows the number of pairs that have been elicited in order to get the maximum APFD (histogram CBR max) or to surpass both Stmt cov and Add stmt cov (histogram below CBR max, depicted in light grey). Once more, the irregular plot of these histograms depends on the specific features of the considered test suites and demands for further replication. However, there seems to be a cut point around size 60 also in these histograms. In fact, for a size of 60 or less, CBR succeeds surpassing Stmt cov and Add stmt cov with a limited manual effort. In fact, the number of pairs to elicit so as to outperform the other methods is the order of the suite size, with a multiplier (β, see above) between 0.5 and 3. Above 60 more pairs need to be elicited in order to get good results, due to the larger space of all possible pairs. Often the number of elicited pairs to surpass Stmt cov and Add stmt cov is close to the number of pairs necessary to reach CBR max, with a few exceptions (e.g., at size 40, 50, 60).

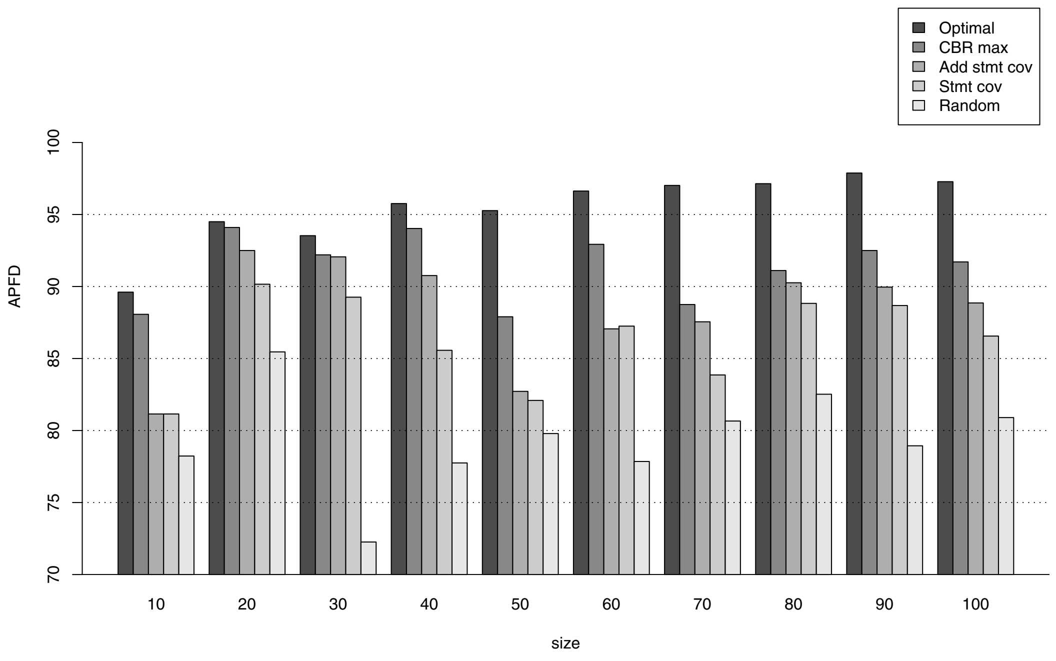

Figure 4. APFD values for CBR (max), statement and additional statement coverage, compared with optimal and random values, at increasing test suite size.

3.3. Discussion

The experimental results confirm the capability of the proposed method to improve the existing techniques, by integrating knowledge elicited from the user. Such a knowledge is extremely valuable and allowed outperforming all considered competitors.

The effort required to elicit the user input is linear with the suite size. Each atomic operation that the user must perform is a pairwise comparison. This means that the user has to decide which of two test cases should be given higher priority (no quantification is requested). Such an operation needs typically a few seconds to be completed. Thus, the overall time for the elicitation process can be estimated between seconds and minutes, depending on the suite size. This makes it cost-effective, especially when the execution time of the test suite is high (e.g., for interactive test cases that cannot be run unattended).

From the economical point of view, it is hard to estimate the cost-benefit trade off of the proposed technique. In fact, a given increase of the APFD value for the prioritized test suite might have a high, medium or low impact on the testing costs, depending on the specific cost model that holds in the given context [6]. However, at suite size of 60 or less, the APFD values produced by CBR are close to the optimum or substantially higher than Stmt cov and Add stmt cov. Moreover, the effort required is quite limited. Thus, we expect that at least in this range the benefits are very likely to overcome the extra costs.

4. Related works

Most existing works on test case prioritization are based on a single prioritization index. Often this index is determined from execution information gathered from previous test runs. It is possible to classify the existing techniques into the following groups:

1. **Execution-based techniques** [1, 6, 10, 13, 15, 16]. These techniques prioritize the test cases based on the level of coverage reached during previous test runs, where coverage can be computed at various levels, such as statement, basic block, branch, condition/decision, function. Additional coverage prioritization is a variant of these techniques that takes into account the increment of coverage provided by each test case with respect to the coverage achieved by the previously selected test cases. Other variants take into

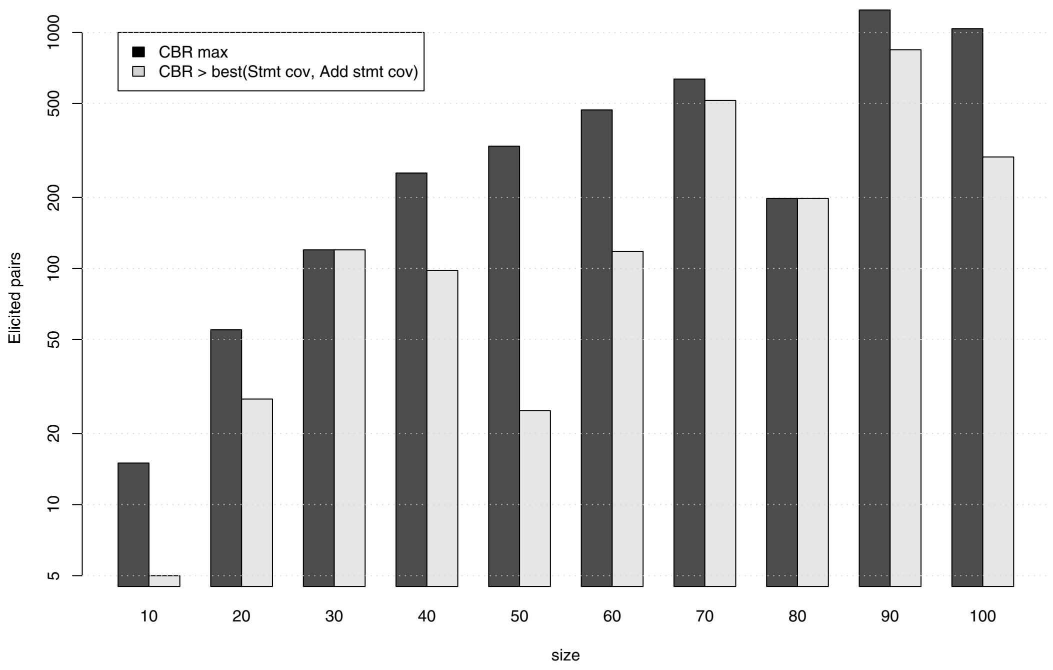

Figure 5. Number of pairwise comparisons elicited from the user at increasing test suite size, necessary to achieve maximum CBR or to surpass the best performing technique between statement and additional statement coverage.

account information on the modified code portions and prioritize the test cases based on the (additional) coverage of such portions only.

2. **History-based techniques** [11]. Historical execution data are exploited to prioritize the test cases. Test cases are selected which either have not been executed recently, or which revealed faults recently, or which cover functions covered infrequently in recent test runs.

3. **Requirement-based techniques** [14]. Test cases are prioritized based on properties of the requirements, such as volatility, customer priority, implementation complexity and fault proneness, whose quantification is estimated by the user. The user is also required to specify the relative weights of these factors.

4. **Metrics-based techniques** [6]. A fault proneness index is computed from a set of measurable software attributes (e.g., complexity metrics) and is used to prioritize the test cases. Additional fault index prioritization is similar to additional coverage for the execution-based techniques: The next test case added to the prioritized list is the one which maximizes the sum of the fault indexes over the yet to be covered functions.

Only a few works [5, 14] attempted to combine multiple prioritization indexes. Elbaum et al. [5] proposed a variant of the APFD metric (called APFD$_C$), which accounts for the cost of executing each test case and for the severity of the revealed faults. They accordingly proposed to modify the existing execution-base techniques to incorporate test cost and fault severity by weighting each covered element by the ratio: *criticality / cost*, i.e., estimated severity of a fault occurring at the covered element, over the cost of the test case. Srikanth et al. [14] proposed a combination method based on fixed weights specified by the user. They applied this method to requirement-based prioritization.

Several empirical studies [3, 6, 12, 13] have been conducted, especially with execution-based techniques, in the context of regression testing. The results of these studies show that prioritization techniques can significantly improve the rate of fault detection and that relatively simple techniques (e.g., coverage or additional coverage) have comparable performances with respect to the most sophisticated ones (e.g., fault exposing potential, FEP, or additional

FEP). Overall, the results indicate that there is a gap between available techniques and optimal prioritization, such that there is room for novel, improved techniques.

Other studies [4, 7] investigated the factors which affect the effectiveness of a test prioritization technique and proposed the usage of selection strategies (such as classification trees) that support the user choosing the most appropriate technique for the given test scenario (characterized through program, modification and test suite metrics). The economic trade-offs associated with the selection of a given prioritization technique are investigated in the study by Elbaum et al. [6].

Our work advances the state of the art in several respects. It is applicable even when historical data about previous test executions are not available or only partially available. It handles multiple prioritization indexes transparently, without requiring the user to specify any arbitrary combination weights. It is the only technique that can incorporate user knowledge about the relative fault exposing capabilities of the test cases. It works even in the presence of partial and incomplete information. It is applicable at the early stages of the testing phase, when test cases are specified but not yet implemented. It can prioritize newly defined (in addition to existing) test cases.

5. Conclusions and future work

We have investigated the relevance of the information gathered from the user to address the test case prioritization problem. We proposed a machine learning approach that integrates user knowledge with multiple prioritization indexes. From a theoretical point of view, the approach is appealing because it overcomes several difficulties exhibited by previous works. It can manage partial, diverse, multiple and incoherent data, which are integrated with user input, in a low-cost knowledge elicitation process. It is widely applicable, in several testing contexts.

Although still preliminary, the experimental results show that the proposed technique represents an improvement over existing methods, by adding information elicited from the user, and indicate that such an improvement is a substantial one. The effort required to achieve it was estimated as a reasonable one. The highest advantages are obtained for moderate suite size (at or below 60 test cases), allowing for an extensive exploration of the space of the possible test case pairs at limited costs.

We plan to conduct a thorough investigation of the applicability and of the cost-benefit trade off of the proposed technique. This involves:

- Replication of the study presented in this paper with multiple test suites and programs under test, so as to obtain results with higher statistical significance.

- Investigation of the effects of noise during the elicitation of the pairwise comparisons, in order to measure the robustness of the machine learning algorithm.

- Application of the technique in real-world scenarios, with the involvement of the actual test engineers and the estimation of the actual costs and benefits.

- Usage of the technique in testing phases/contexts where traditional test case prioritization techniques have not been tried, due to their intrinsic limitations. Examples are initial test specification, with the test cases not yet implemented, and testing of major releases, which make previous information only partially usable or unusable.

References

[1] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. A study of effective regression testing in practice. In *Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE)*, pages 264–274. IEEE Computer Society, November 1997.

[2] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[3] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE)*, pages 113–124. IEEE Computer Society, November 2004.

[4] S. Elbaum, D. Gable, and G. Rothermel. Understanding and measuring the sources of variation in the prioritization of regression test suites. In *Proceedings of the 7th International Software Metrics Symposium*, pages 169–179. IEEE Computer Society, April 2001.

[5] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, pages 329–338. IEEE Computer Society, May 2001.

[6] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, February 2002.

[7] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, September 2004.

[8] Y. Freund, R. Iyer, R. Schapire, and Y. Singer. An Efficient Boosting Algorithm for Combining Preferences. In *Proceedings 15th International Conference on Machine Learning*, 1998.

[9] Y. Freund and R. Schapire. A Short Introduction to Boosting. Journal of Japan. Soc. for Artificial Intelligence, 14(5) (1999), 771-780. 11, 1999.

[10] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on Software Engineering*, 29(3):195–209, March 2003.

[11] J.-M. Kim and A. A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 119–129. ACM Press, May 2002.

[12] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 179–188. IEEE Computer Society, August-September 1999.

[13] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[14] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *Proceedings of the 4th International Symposium on Empirical Software Engineering (ISESE)*, pages 62–71. IEEE Computer Society, November 2005.

[15] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–106. ACM Press, July 2002.

[16] J. Zheng, B. Robinson, L. Williams, and K. Smiley. An initial study of a lightweight process for change identification and regression test selection when source code is not available. In *Proceedings of the 16th International Symposium on Software Reliability Engineering (ISSRE)*, pages 225–234. IEEE Computer Society, November 2005.