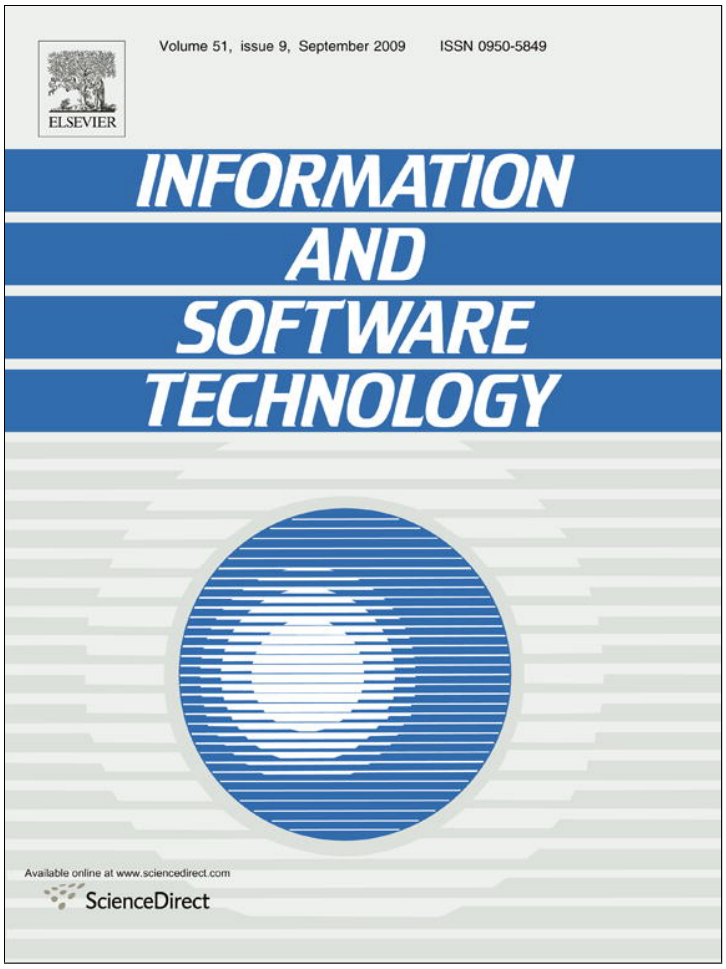
Volume 51, issue 9, September 2009
ISSN 0950-5849
ELSEVIER
INFORMATION AND SOFTWARE TECHNOLOGY
Available online at www.sciencedirect.com
ScienceDirect

Information and Software Technology 51 (2009) 1365–1376

Contents lists available at ScienceDirect

Information and Software Technology

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

A hybrid heuristic approach to optimize rule-based software quality estimation models

D. Azar, H. Harmanani *, R. Korkmaz

Department of Computer Science and Mathematics, Lebanese American University, Byblos 1401 2010, Lebanon

A R T I C L E   I N F O

Article history:
Received 1 March 2009
Received in revised form 18 May 2009
Accepted 18 May 2009
Available online 18 June 2009

Keywords:
Software quality
Search-based software engineering
Soft computing

A B S T R A C T

Software quality is defined as the degree to which a software component or system meets specified requirements and specifications. Assessing software quality in the early stages of design and development is crucial as it helps reduce effort, time and money. However, the task is difficult since most software quality characteristics (such as maintainability, reliability and reusability) cannot be directly and objectively measured before the software product is deployed and used for a certain period of time. Nonetheless, these software quality characteristics can be predicted from other measurable software quality attributes such as complexity and inheritance. Many metrics have been proposed for this purpose. In this context, we speak of estimating software quality characteristics from measurable attributes. For this purpose, software quality estimation models have been widely used. These take different forms: statistical models, rule-based models and decision trees. However, data used to build such models is scarce in the domain of software quality. As a result, the accuracy of the built estimation models deteriorates when they are used to predict the quality of new software components. In this paper, we propose a search-based software engineering approach to improve the prediction accuracy of software quality estimation models by adapting them to new unseen software products. The method has been implemented and favorable result comparisons are reported in this work.

1. Introduction

With the complexity of object-oriented (OO) software products on the rise, it is becoming crucial to evaluate the quality of the software products during the different stages of development in order to reduce time, effort and money. The quality of a software is evaluated in terms of characteristics such as maintainability, reusability, stability, etc. Though most of these characteristics cannot be measured before the software is used for a certain period of time, they can be deduced from several measurable software attributes such as cohesion, coupling and size. For this purpose, several metrics that capture such attributes have been proposed in the literature [7,17,15,10,27,26,33,32], and [35]. Examples of such metrics include number of children of a class in an object-oriented system (NOC), and number of methods (NOM). Software quality estimation models build a relationship between the desired software quality and the measurable attributes. These models are either statistical models (for example, regression models [21,30]) or logical models [29,19]. The latter have been extensively used because of their white-box nature as they provide practitioners with the prediction label as well guidelines to attain it. Logical models can take the form of decision trees or rule sets. In general, they suffer from degradation of their prediction when they are applied to new/unseen data. This is largely due to the lack of a representative sample that can be drawn from available data in the domain of software quality. Unlike other fields where public repositories abound with data, software quality data is usually scarce. The reason is that not many companies systematically collect information related to software quality. Furthermore, such information is normally considered confidential and in the case where a company is willing to make it public, usually only the resulting model is published. The latter is company-specific and is difficult to generalize, to cross-validate or to re-use it.

Search-based software engineering, first coined by Harman et al. [25], is an emerging field that applies metaheuristic search techniques to software engineering problems. The field has recently witnessed intense activity [16,37], and shown a lot of promise when applied to various software engineering problems including project management [2,4,14], prediction in software engineering management [20,31], project planning and quality assessment [1,13,31], and software testing [24,34,44]. Typically, search methods such as local search, simulated annealing, genetic algorithms, and genetic programming are used as sampling techniques [23]. However, to the best of our knowledge, not much work was reported using *hybrid metaheuristic approaches* nor using *tabu search* in search-based software engineering.

* Corresponding author.
E-mail address: haidar@acm.org (H. Harmanani).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.05.003

This paper presents a search-based software engineering approach that consists of building a better model from already-existing ones. This can be seen as combining the expertise of several expert models, built from common domain knowledge, and adapting the resulting models to context-specific data. To validate our technique, we use the specific problem of predicting the stability of object-oriented (OO) software components (classes, in this context). During its evolution, a software component undergoes various modifications due to changes in requirements, error detections, changes in the environment, etc. It is important for the software component to remain "usable" in the new changed environment. In [5], we presented a similar approach using genetic algorithms (GA). The results were promising. In this work, we present a hybrid approach to solve the above model. The approach is hybrid on two levels. It combines the strengths of different heuristics (genetic algorithms, tabu search and simulated annealing), and it works on two levels of optimization (rule set level and rule level). Results show that our hybrid heuristic outbeats C4.5 as well as the genetic algorithm presented in [5]. The remainder of this paper is organized as follows. In Section 2, we give an overview of the related work in the field. In Section 3, we state the problem and the objective of the work. In Section 4, we give an overview of the heuristics used in our approach. We also describe how we instantiate elements of these heuristics to our problem. In Section 6, we explain the experiments that we perform and the obtained results. Finally, in Section 7, we conclude with a brief recollection of the technique and future paths.

2. Related work

Logical estimation models have been widely used in the domain of software quality. Selby and Porter [42] have used machine learning to build such models in the form of decision trees as early as 1988. Later on, such models were very popular in the field. Mao et al. [36] used C4.5 to build models that predict reusability of a class in an object-oriented software system from metrics for inheritance, coupling and complexity. The authors proposed the use of estimation models as guidelines for future software development. Basili et al. [8] use C4.5 to build models that estimate the cost of rework in a library of reusable components. De Almeida et al. [3] use C4.5 rule sets to predict the average isolation effort and the average effort. Briand et al. [10] investigated the relationship between most of the existing coupling and cohesion measures defined at the level of a class in an object-oriented system on one hand, and fault-proneness on the other hand. In all the cases, the models were not very useful in predicting the quality characteristic of unseen software. The main reason is that the data used to build the models is scarce so the models become hard to generalize. Various search-based software engineering approaches have been also proposed in the domain of software quality. For example, Azar and Precup [5] and Bouktif et al. [9] presented two genetic algorithm-based approaches that optimize the accuracy of these models on new data. One approach relies on the recombination of several models into new ones. The other one relies on the adaptation of a single model to a new data set. Both approaches outbeat C4.5 when tested on the stability of classes in an object-oriented software system with the recombining approach showing the highest performance [6]. Pedrycz and Succic [39] represent classifiers as hyperboxes and uses genetic algorithms to modify these hyperboxes (and the underlying classifiers). Similarly to our approach, this technique preserves the interpretability of the classifiers and can easily be extended to a problem with multiple classification labels. However, the data that is used to validate this approach uses different metrics than those reported in this work, and is concerned with software maintenance rather than stability. Vivanco [43] uses a genetic algorithm to improve a classifier accuracy in identifying problematic components. The approach relies on the selection of metrics that are more likely to improve the performance of predictive models.

3. Problem statement and objective

The problem notation originates from the machine learning formalism. A *data set* is a set $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ of n *instances* or *cases* where $x_i = \langle a_1, a_2, \ldots, a_d \rangle \in \Re^d$ is an *attribute vector* of d *attributes*, and $y_i \in C$ is a *classification label*. In the particular problem that we are considering, a case represents a software component (a class in an OO software system). The attributes $(a_1, \ldots, a_d)$ are metrics (such as number of methods, number of children, etc.) that are considered to be relevant to the software quality factor being predicted (stability). The label y_i represents the software quality factor. In this problem, $y_i \in \{0(\text{stable}), 1(\text{unstable})\}$.

A *classifier* is a function $f : \Re^d \mapsto C$ that predicts the label y_i of any attribute vector x_i. In the framework of supervised learning, it is assumed that (vector, label) pairs are random variables (X, Y) drawn from a fixed but unknown probability distribution, and the objective is to find a classifier f with a low error (misclassification) rate. Since the data distribution is unknown, both the selection and the evaluation of f must be based on the data set D. For this purpose, D is partitioned into two parts, the *training set* D_{train} and the *testing set* D_{test}. Most learning algorithms take the training set as input and search the space of classifiers for one that minimizes the error on D_{train}. The output classifier is then evaluated on the testing sample D_{test}. Examples of learning algorithms that use this principle are the back-propagation algorithm for feed forward neural nets [41] and C4.5 [40]. In our experiments, we use *10-fold cross validation* that allows us to use the whole data set for training and to evaluate the error probability more accurately.

In this paper, we focus on rule-based classifiers i.e. classifiers that take the form of rule sets. A rule-based classifier is a disjunction of conjunctive rules and a default classification label. Fig. 1 illustrates a rule-based classifier that predicts the stability of a component (a class in an object-oriented software system) based on the metrics number of classes used by a member function (CUBF), number of parents (NOP) and number of used classes (CUB). The example model that has three rules and a default classification label. The first rule estimates that if the number of classes used by a member function (CUBF) in the designated class is greater than 7 then the class is unstable (1). The second rule classifies a class with number of parents (NOP) greater than 2 as unstable. The third rule classifies a class with the number of used classes (CUB) less than or equal to 2 and number of parents (NOP) less than or equal to 2 as stable (0). The classification is sequential. The first rule (toward the top) whose left hand side is satisfied by a case fires. If no such rule exists, the default classification label is used to classify the case (default class 1).

The model has an accuracy that can be measured using the correctness of the classifier (percentage of cases correctly classified) or its Youden's J_{index}, $J(R)$, (average correctness per class label) as follows Eq. (1):

$$J(R) = \frac{1}{k} \sum_{i=1}^{k} \frac{n_{ii}}{\sum_{j=1}^{k} n_{ij}}. \qquad (1)$$

Rule 1: $\text{CUBF} > 7 \rightarrow 1$
Rule 2: $\text{NOP} > 2 \rightarrow 1$
Rule 3: $\text{CUB} \le 2 \wedge \text{NOP} \le 2 \rightarrow 0$
Default class: 1

Fig. 1. Example rule-based classifier.

Table 1
The confusion matrix of a decision function f where n_{ij} is the number of cases with real label c_i classified as c_j.

		Predicted label			
		c_1	c_2	$\dots$	c_k
Real label	c_1	n_{11}	n_{12}	$\dots$	n_{1k}
	c_2	n_{21}	n_{22}	$\dots$	n_{2k}
	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	c_k	n_{k1}	n_{k2}	$\dots$	n_{kk}

The correctness of the rule set $R, C(R)$, computed on a data set D is equal to the percentage of cases in D that are correctly classified by R and is given in Eq. (2) as follows:

$$C(R) = \frac{\sum_{i=1}^{k} n_{ii}}{\sum_{i=1}^{k}\sum_{j=1}^{k} n_{ij}} \tag{2}$$

It should be noted that n_{ij} in Eqs. (1) and (2) is the number of cases in D that are classified by R as having class label c_j while they actually have class label c_i (Table 1).

Intuitively, if we have the same number of instances of each label, then $J(R) = C(R)$. However, if the data set is unbalanced, $J(R)$ gives more relative weight to cases with minority labels. Both a constant classifier and a guessing classifier (that assigns random uniformly distributed labels to input vectors) would have a $J(R)$ close to 0.5, while a perfect classifier would have a $J(R)$ equal to 1. On the other hand, in the case of an unbalanced data set, $C(R) \approx 0.5$ but $C(\text{const})$ can be close to 1. Our goal is to maximize the accuracy on the unseen data. In this work, we present a hybrid heuristic and compare it to the genetic algorithm presented in [5]. Results show that our hybrid heuristic outbeats the GA when tested on the stability of object-oriented classes.

4. Heuristic overview

In this section, we give a brief overview of the three heuristics that we used in our algorithm. These are genetic algorithms, simulated annealing and tabu search.

4.1. Genetic algorithms

Genetic algorithms (GAs) were first introduced by Holland [28] and were inspired by the Darwinian theory of evolution [18]. They provide a learning method motivated by an analogy to biological evolution. According to the theory, individuals in an environment compete for survival. Fitter individuals have a higher chance for

```
Genetic_Algorithm
{
    Let $P_0$=initial population of chromosomes
    Compute fitness and selection probability of all chromosomes in$P_0$
    Let $P_i = P_0$
    Repeat for $n$ iterations:
        Repeat until size of $P_{i+1} = sizeof P_i$
            Select $n$ chromosomes from $P_i$ and copy chromosomes to $P_{i+1}$   // elitism
            Crossover $(C_1, C_2) \rightarrow C_1'$ and $C_2'$
            Mutate $(C_1')$
            Mutate $(C_2')$
            Copy mutated offspring to $P_{i+1}$
            $P = P_{i+1}$
}
```

Fig. 2. Genetic algorithm pseudocode.

```
Simulated_Annealing
{
    Given: An initial solution $S_0$, an initial temperature $T_0$, a fraction $K$, $0 \leq K \leq 1$
    Set initial set of solutions $X = \{S_0\}$
    Repeat for a certain number of iterations:
        Set $T = T_0$;
        Repeat for a number of iterations for each temperature:
            Select randomly a solution from the set X
            Perturb the solution.
            Calculate :
            $\Delta E$ = evaluation (old-sol)  evaluation (new-sol)
            if $(\Delta E > 0)$
              accept new solution
            else:
              generate random number $r$, $0 \leq r \leq 1$
            if $(r < e^{\frac{-\Delta E}{T}})$
              accept new solution
            else
              reject new solution.
            $T = K * T$
    Return best solution in X
}
```

Fig. 3. Simulated annealing pseudocode.

```
Tabu_Search
{
  Let $S$ be a solution, $N(S)$, the neighborhood solution and $f(S)$ the evaluation function.
      Given: Initial solution $S_0$, a certain number of iterations x:
      Initialize current solution: $S = S_0$
      Initialize best solution to be the current solution: $S^* = S$
      Initialize the list of tabu to an empty list: $T = \phi$
      Repeat for x iterations
        $S = N(S)$
        if $f(S) > f(S*)$
          $S^* = S$.
        add $S$ to $T$
  Return $S^*$
}
```

Fig. 4. Tabu search pseudocode.

surviving and producing progeny. The pseudocode of a GA is shown in Fig. 2.

In a GA, a population of chromosomes or individuals is created. Typically, each chromosome represents a solution to the optimization problem and has a fitness function that reflects the quality of the underlying solution. The GA iterates to create new populations (of fitter chromosomes ideally) through the application of genetic operators. The most commonly used ones are crossover and mutation. Crossover involves two chromosomes. Under this operator, the chromosomes exchange chunks of their genetic material and two other chromosomes are created. Mutation, on the other hand, is applied to one chromosome only. Under this operator, one or more gene in the chromosome are randomly perturbed. Both operators occur with a certain probability.[1] The resulting chromosomes constitute the new population (or a major part of it). The best chromosomes found in a population are preserved by the elitism operator (by copying them as is to the next population). Usually, all populations have equal size. The whole process of creating new populations is repeated until a certain stopping criteria is met (a predetermined number of iterations or a desired fitness value is achieved). It is important to point out that selection of the chromosomes happens with a probability that is proportional to their fitness.

4.2. Simulated annealing

Simulated annealing (SA) was first introduced in [38]. It is a generalization of a Monte Carlo method for examining the equations of state and frozen states of *n*-body systems [38]. The technique is inspired from the procedure used to create perfect crystallizations. This consists of heating the glass to a very high temperature at which the glass becomes liquid and the atoms move relatively freely. Subsequently, the temperature of the glass is slowly lowered so that, at each temperature, the atoms can move enough to begin adopting the most stable orientation. This slow cooling is known as annealing. In simulated annealing, the goal is to improve a set of solutions to a problem. Through each iteration, a solution is *perturbed*. The new solution is then compared to the old one. An evaluation function allows the algorithm to either accept the new solution or reject it. It is important to point out that this heuristic main goal is to escape local optima by allowing moves to worse solutions. The pseudo code for SA is shown in Fig. 3.

4.3. Tabu search

Tabu search is a heuristic method proposed by Glover [22]. The main goal of this heuristic is to overcome the problem that most heuristics face: getting stuck at local optima. For this, visited solutions are recorded on a tabu list preventing the search from cycling back to them (and thus getting stuck at local optima). The solutions remain on the tabu list for a certain period of time (a certain number of iterations). This allows the algorithm to explore portions of the search space that have not been explored yet. A transformation is used to move from one solution in the search space to another. Fig. 4 shows the pseudocode for Tabu search.

[1] Normally, crossover has a much higher probability than mutation.

5. Problem encoding

In our solution, we have designed three different hybrid heuristics which differ in the heuristics used. In the first one *(SA–GA)*, we combine simulated annealing with genetic algorithms. In the second one (*TS–GA*), we combine tabu search and genetic algorithms. In the third heuristic (*SA–TS–GA*), we combine simulated annealing, tabu search and genetic algorithms. Each hybrid heuristic works in a three phase optimization scheme as shown in Fig. 5. Rule sets generated by C4.5 are first individually optimized using either simulated annealing or tabu search. Then, the rule sets thus obtained are fed to the GA that re-combines and modifies them to produce new rule sets. The best rule set obtained by the GA is then passed to tabu search or simulated annealing to further improve it. Next, we explain how we instantiate each element of our heuristics to our problem.

5.1. Genetic algorithms

In our GA, each chromosome represents a rule set. We represent a population of chromosomes as a tripartite graph. Fig. 6 illustrates the graph representation of a rule set. The graph includes the following types of nodes:

- *Classification nodes* – nodes that represent classification labels of rules.
- *Attribute nodes* – nodes that represent attributes in conditions forming rules.
- *Value nodes* – nodes that represent values in conditions.

Nodes are connected with two types of edges. The first type, *regular edges*, represent conditions in rules and connect attribute nodes to value nodes. The second type, *special edges*, link classification nodes to attribute nodes. Regular edges are assigned a 0 or 1 weight and special edges are assigned a -1 weight. Rule sets are identified by a unique color while rules in the same rule set by a unique line pattern. We do not represent the default classification label of a rule set in the graph since it is not involved in any of the GA operations. When a rule set is extracted from a graph, we assign to it the majority classification label as a default classification label.

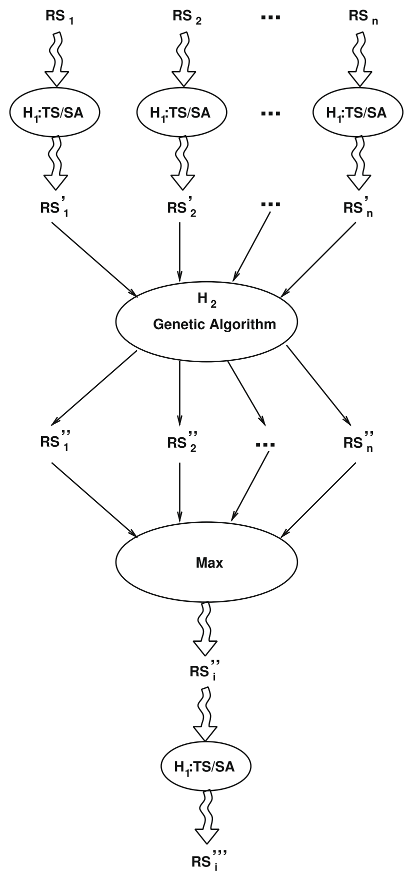

Fig. 5. General functional model describing the hybrid heuristics. The initial rule sets are improved individually using tabu search or simulated annealing. The obtained rule sets are recombined by a GA. The best obtained rule set is then optimized using tabu search or simulated annealing.

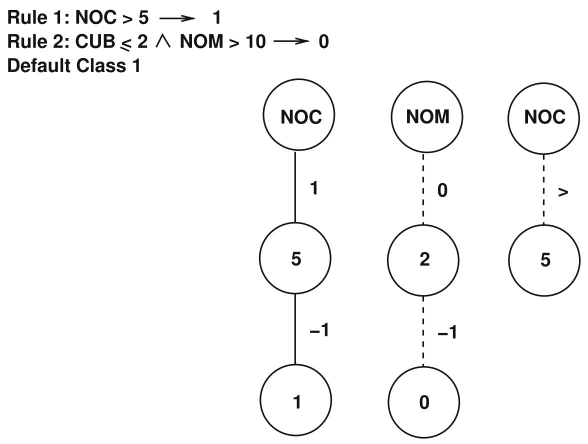

Fig. 6. A rule set and its graph representation. The rule set is formed of two rules and a default classification label. Each rule is identified with a different line pattern. Regular edges connect values to metrics and special edges (dotted) connect classification labels to values. Weights on regular edges represent relational operators (1 for > and 0 for ⩽). Special edges are weighted −1.

5.1.1. Genetic operators and the fitness function

The GA starts by copying a percentage of the best chromosomes to the next population.[2] The population is then completed by crossover and mutation. Chromosomes that undergo crossover are selected according to their fitness – fitter chromosomes get a higher chance of being selected. Since the fitness of a chromosome reflects the quality of the underlying solution, we the following three different fitness functions Eq. (3), (4), and (5) depending on how much weight we want to give to the correctness and the $J(R)$:

$$f_1 = C(R), \tag{3}$$
$$f_2 = J(R), \tag{4}$$
$$f_3 = C(R) \times 0.5 \times J(R). \tag{5}$$

We implemented two different crossover operators and four different mutation operators.[3] Since crossover happens with a certain probability, it is possible for two selected chromosomes not to undergo crossover in which case, they get copied as is to the new population. Before copying the new chromosomes to the new population, the genetic algorithm mutates them with a certain probability.

- *Rule crossover* consists of selecting patterns randomly and changing their colors. On the rule set level, this translates to exchanging a random number of rules between rule sets. Fig. 7 illustrates this operator.
- *Condition crossover* consists of swapping colors between two regular edges. The edges should initially be of different colors. This results in swapping two conditions between two different rule sets. Fig. 8 illustrates this operator.
- *Operator mutation* consists of changing the weight of a normal edge from 1 to 0 or from 0 to 1. This is equivalent to changing the operator in a condition inside the rule set. Fig. 9 illustrates the operator.
- *Class mutation* consists of changing the value of a class label node chosen randomly in the graph. This is equivalent to changing the classification label of a randomly chosen rule inside the rule set. Fig. 11 illustrates the operator.
- *Value mutation*: consists of changing randomly a value in a value node. This is equivalent to choosing a condition inside a rule set and changing the value of the numerical operand. Fig. 10 shows a chromosome before and after value mutation.
- *Condition addition mutation* consists of inserting two connected nodes in the graph. One node encodes a random attribute and the other one a random value. The value is chosen from the set of cut-points of this attribute.[4] The pattern of the inserted edge is chosen from the set of line patterns already present in the graph. A random weight is attributed to the edge. This corresponds to adding a new condition to the underlying rule in the rule set. The rule is identified by the pattern of the newly added edge. Fig. 12 shows a chromosome before and after *condition addition mutation* is applied.

5.1.2. Post-processing

The genetic operators introduced above might result in redundant or inconsistent rule sets. Redundancy occurs when two conditions exist in the same rule and one implies the other, when two rules have the same conditions or when two rule sets have the same rules. Inconsistency occurs when two conditions inside a rule are contradictory (e.g. DIT > 3 and DIT < 1). When the GA finishes, all the rule sets are post processed to eliminate redundancy and inconsistency by deleting from the rule sets all rules that show inconsistency. It might be worth pointing that such rules naturally die during the evolution process since they have a fitness of 0.

[2] This percentage is a parameter with a value assigned at the onset of the experiments.

[3] Crossover between two identical chromosomes is not allowed since no real information exchange would occur in this case.

[4] A cut-point of an attribute is the median of the two values of the attribute in the training set where the classification label changes [40].

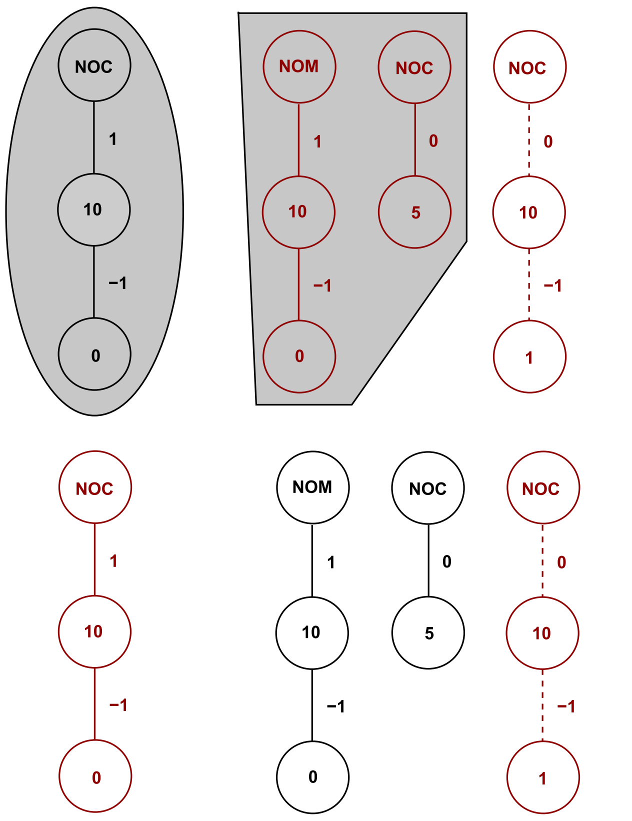

Fig. 7. Rule crossover: each color identifies a rule set. The parent rule sets are *Parent 1:* NOC $>$ 10 $\to$ CL0; and *Parent 2:* NOM $>$ 10 $\wedge$ NOC $\leqslant$ 5 $\to$ CL0; NOC $\leqslant$ 10 $\to$ CL1. The highlighted regions correspond to parts of the graph that have been re-colored. The resulting rulesets are: *Child 1:* NOC $>$ 10 $\to$ CL0 and *Child 2:* NOM $>$ 10$\wedge$ NOC $\leqslant$ 5 $\to$ CL0; NOC $\leqslant$ 10 $\to$ CL1.

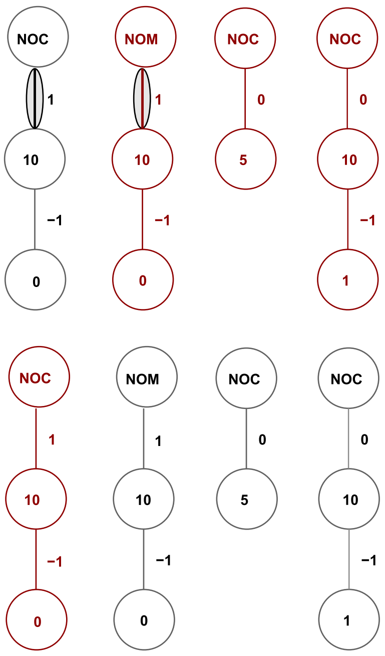

Fig. 8. Condition crossover: the highlighted edges are the ones whose colors are swapped. Parent rule sets are: *Parent 1:* NOC $>$ 10 $\to$ CL0; *Parent 2:* NOM $>$ 10 $\wedge$ NOC $\leqslant$ 5 $\to$ CL0; NOC $\leqslant$ 10 $\to$ CL1. The resulting child rule sets are: *Child 1:* NOC $>$ 10 $\to$ CL0; *Child 2:* NOM $>$ 10 $\wedge$ NOC $\leqslant$ 5 $\to$ CL0; NOC $\leqslant$ 10 $\to$ CL1.

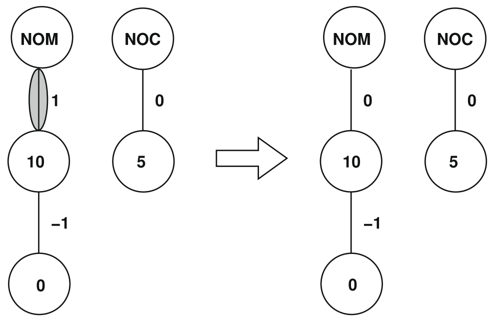

Fig. 9. Operator mutation: the weight of the highlighted edge is changed from 1 to 0. The rule set before mutation was NOM $>$ 10 $\wedge$ NOC $\leqslant$ 5 $\to$ CL0. The rule set after mutation becomes NOM $\leqslant$ 10 $\wedge$ NOC $\leqslant$ 5 $\to$ CL0.

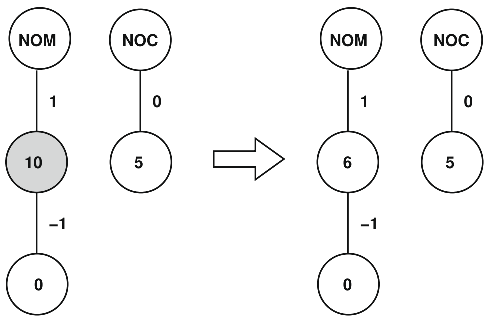

Fig. 10. Value mutation: the value of the highlighted node is changed. The rule set before mutation was NOM $>$ 10 $\wedge$ NOC $\leqslant$ 5 $\to$ CL0. The rule set after mutation becomes NOM $>$ 6 $\wedge$ NOC $\leqslant$ 5 $\to$ CL0.

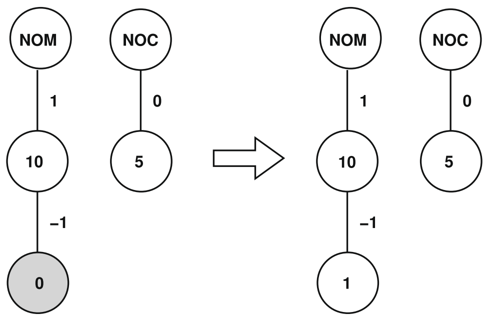

Fig. 11. Class mutation: the value of the highlighted node is changed. The rule set before mutation was NOM $>$ 10 $\wedge$ NOC $\leqslant$ 5 $\to$ CL0. The rule set after mutation becomes NOM $>$ 10 $\wedge$ NOC $\leqslant$ 5 $\to$ CL1.

However, since it might still be the case that such weak rules may survive, we eliminate them anyways.

5.1.3. Termination condition

The GA iterates over 500 generations. The number was chosen after several runs showed that no significant improvement could be obtained beyond this.

5.2. Simulated annealing

The first phase in our hybrid algorithm consists of optimizing the individual rule sets obtained by C4.5 prior to feeding them to the GA. The last phase of our hybrid algorithm consists of optimizing the rulesets generated by the GA. At both stages,

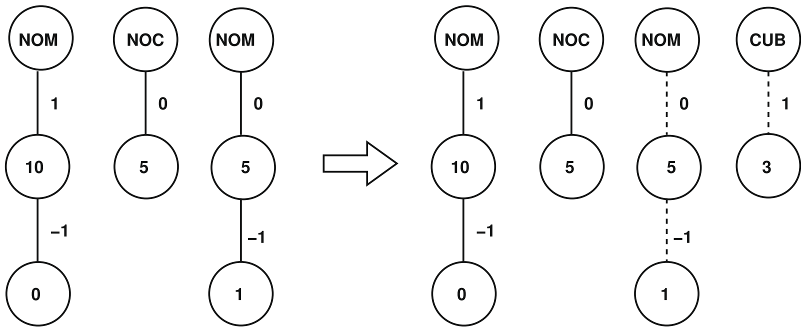

Fig. 12. Condition addition mutation: the operator adds two connected nodes to the graph. The new edge receives the dotted-pattern and a weight equal to 1. The rule set before the application of the operator is $NOM > 10 \wedge NOC \leqslant 5 \rightarrow CL0; NOM \leqslant 5 \rightarrow CL1$. The rule set after the mutation becomes: $NOM > 10 \wedge NOC \leqslant 5 \rightarrow CL0; NOM \leqslant 5 \wedge CUB > 3 \rightarrow CL1$.

simulated annealing (or tabu search) are used to individually optimize these rule sets. The basic elements of simulated annealing are the evaluation function and the perturbation function.

5.2.1. Evaluation function

We use the same evaluation function that we used in the GA namely.

5.2.2. The perturbation function

We implemented six types of perturbation functions each applied with equal probability.

- *Rule deletion*: consists of removing a random rule from the rule set.
- *Condition deletion*: consists of deleting a random condition from a randomly selected rule in the rule set.
- *Rule addition*: consists of creating a new random rule and inserting it in the rule set at a random position.
- *Condition addition*: consists of creating a random condition and inserting it in a randomly chosen rule in the rule set.
- *Rule change*: consists of replacing a randomly selected rule in the rule set with a randomly created one.
- *Condition change*: consists of replacing a randomly selected condition in the rule set with a randomly created one.

Fig. 13 shows an illustration of two of the three operators.

5.3. Tabu search

Each solution encodes a rule set and we use the same evaluation function that we use in our previous two heuristics Eq. (3). The major elements of tabu search are the neighborhood function and the tabu list replacement policy.

5.3.1. Neighborhood function

A neighbor of a solution *S* is defined as a rule set that is not on the tabu list, and that differs from *S* either by a rule or by a condition. We designed six different transformations that create neighbor solutions. They all occur with equal probability. They are

- *Rule deletion*: consists of removing a random rule from the rule set.
- *Condition deletion*: consists of removing a random condition from the rule set.
- *Rule addition*: consists of creating a new random rule and inserting it in the rule set at a random position.
- *Condition addition*: consists of creating a random condition and inserting it in a random rule in the rule set.
- *Rule change*: consists of choosing a random rule from the rule set and replacing it with a new randomly created one.
- *Condition change*: consists of choosing a random condition in the rule set and replacing it with a new randomly created one.

Fig. 14 shows an illustration of two of the three operators.

5.3.2. The tabu list replacement policy

The relatively big number of transformations that we have designed makes it unlikely for the search to cycle back to a previously visited solution within a small number of iterations. Empirical experiments have shown that we could keep a solution on the tabu list for 10 iterations only and then copy it back to the search pool without increasing the risk of the algorithm getting stuck at a local optimum.

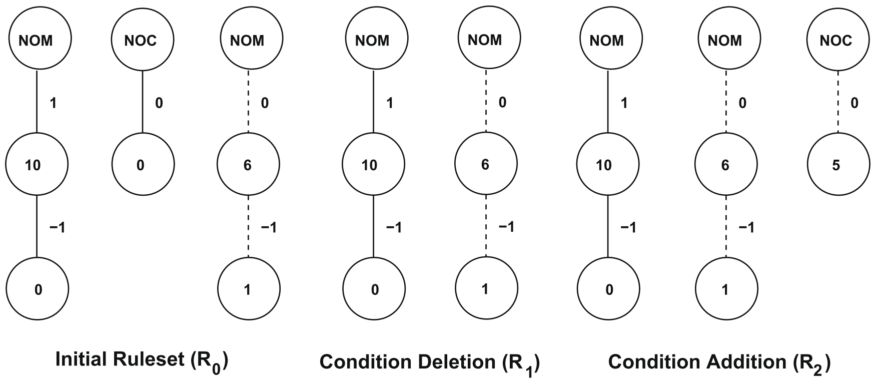

Fig. 13. An inititial rule set (R_0), and the two rulesets obtained after applying two annealing perturbation functions: condition deletion (R_1) and condition addition (R_2).

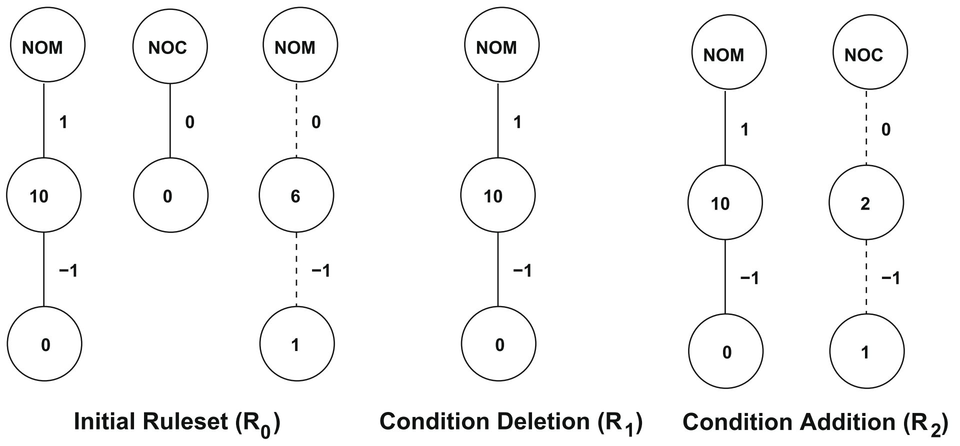

Fig. 14. An initital ruleset (R_0), and the consecutive neighborhood solutions obtained after applying two *tabu* neighborhood functions: rule deletion (R_1) and rule addition (R_2).

6. Experiments and results

In order to assess the performance of the heuristics, we used two data sets that describe the stability of object-oriented software components. Briefly, a class in an object-oriented system is said to be *stable* if its public interface remains valid between different versions of the system; otherwise, it is said to be *unstable*. The two data sets differ in the data distribution. The first one, namely STAB1, is an unbalanced data set. The second one, namely STAB2, is a balanced one. We describe more these data sets in Section 6.1. In our assessment, we use 10-fold cross validation (10 is a commonly used number in the area of machine learning) in which a data set is divided into ten folds of roughly equal size. The heuristic is trained on the union of nine of the ten folds and tested on the remaining one. This process is repeated 10 times, each time using a different fold as the testing set. Moreover, to account for the randomness factor inherent to the heuristics, we repeat each experiment 30 times and we report the average and standard deviation over the 30 runs.

Table 2
STAB1-software quality metrics used as attributes in the classifiers.

Name	Description
Cohesion metrics	
LCOM	lack of cohesion methods
COH	cohesion
COM	cohesion metric
COMI	cohesion metric inverse
Coupling metrics	
OCMAIC	other class method attribute import coupling
OCMAEC	other class method attribute export coupling
CUB	number of classes used by a class
Inheritance metrics	
NOC	number of children
NOP	number of parents
DIT	depth of inheritance
MDS	message domain size
CHM	class hierarchy metric
Size complexity metrics	
NOM	number of methods
WMC	weighted methods per class
WMCLOC	LOC weighted methods per class
MCC	McCabe's complexity weighted methods per class
NPPM	number of public and protected methods in a class
NPA	number of public attributes
The stress metric	
STRESS	stress applied to the class

Table 3
STAB1-software systems used to build classifiers with C4.5.

Software system	Number of versions (major)	Number of classes
Bean browser	6(4)	388–392
Ejbvoyager	8(3)	71–78
Free	9(6)	46–93
Javamapper	2(2)	18–19
Jchempaint	2(2)	84
Jedit	2(2)	464–468
Jetty	6(3)	229–285
Jigsaw	4(3)	846–958
Jlex	4(2)	20–23
Lmjs	2(2)	106
Voji	4(4)	16–39

6.1. Data collection

STAB1 [6]: Nineteen structural metrics (Table 2) were extracted from the eleven software systems shown in Table 3 using the ACCESS tool and the Discover environment©.[5] Detailed description of these metrics can be found in [10,12,11,15], and [45]. Fifteen subsets (of size 2, 3, and 4) were created by combining these metrics in all possible ways. The combination was based on the relationship desired among the different quality characteristics.[6] The 15 subsets of metrics were used with the 11 software systems to create $15 \times 11 = 165$ data sets. C4.5 was used to create 165 decision tree classifiers with these data sets. Constant classifiers (classifiers that have a single classification label) and classifiers with a training error more than 10% were eliminated and the remaining 40 were retained. These decision trees were then converted into rule sets by C4.5. Our heuristics uses these rule sets as experts built from common domain knowledge and combines and adapts them to the four software systems shown in Table 4. These form a data set which consists of 2920 instances and simulate the in-house data.

STAB2 [9]: the previous data set is highly imbalanced. As a matter of fact, 2481 cases are stable and 439 unstable. In order to test our algorithm on a balanced data set, we used STAB2 which also involves stability. Table 7 shows the 22 software metrics used. Nine of the 11 software systems shown in Table 3 were used to build experts with C4.5 (Table 5). Fifteen subsets of metrics were created by combining 2, 3, or 4 groups in all possible ways. These subsets were used with the 9 chosen software systems to create

[5] Available at: http://www.mks.com/products/discover/developer.shtml.
[6] For example, the relationship between cohesion and coupling on one hand and stability on the other.

Table 4
STAB1- Software systems used to train and test the GA.

JDK version	Number of classes
jdk1.0.2	187
jdk1.1.6	583
jdk1.2.004	2337
jdk1.3.0	2737

Table 5
STAB2-software systems used to build classifiers with C4.5.

Software system	Number of versions (major)	Number of classes
Bean browser	6(4)	388–392
Ejbvoyager	8(3)	71–78
Free	9(6)	46–93
Javamapper	2(2)	18–19
Jchempaint	2(2)	84
Jigsaw	4(3)	846–958
Jlex	4(2)	20–23
Lmjs	2(2)	106
Voji	4(4)	16–39

Table 6
STAB2-software systems used to train and test the GA.

Software system	Number of versions (major)	Number of classes
Jedit	2(2)	464–468
Jetty	6(3)	229–285

135 data sets. C4.5 was used to construct a decision tree from each data set. Constant classifiers and classifiers with an error rate higher than 10% were eliminated and 23 retained. The decision trees were then converted to rule sets by C4.5. Our heuristic uses the systems shown in Table 6 for training and testing (in-house data simulation).

Table 7
STAB2-software quality metrics used as attributes in the classifiers.

Name	Description
Cohesion metrics	
LCOM	lack of cohesion methods
COH	cohesion
COM	cohesion metric
COMI	cohesion metric inverse
Coupling metrics	
OCMAIC	other class method attribute import coupling
OCMAEC	other class method attribute export coupling
CUB	number of classes used by a class
CUBF	number of classes used by a member function
Inheritance metrics	
NOC	number of children
NOP	number of parents
NON	number of nested classes
NOCONT	number of containing classes
DIT	depth of inheritance
MDS	message domain size
CHM	class hierarchy metric
Size complexity metrics	
NOM	number of methods
WMC	weighted methods per class
WMCLOC	LOC weighted methods per class
MCC	McCabe's complexity weighted methods per class
DEPCC	operation access metric
NPPM	number of public and protected methods in a class
NPA	number of public attributes

Table 8
Parameters of all three heuristics. These values gave the highest experimental results.

Parameter	Value
GA	
Generations	500
Rule crossover	80%
Condition crossover	10%
Operator mutation	2.5%
Value mutation	2.5%
Class mutation	2.5%
Condition addition mutation	2.5%
Elitism	1
SA	
Iterations for initial rule set	10
Iterations per temperature for initial rule set	10
Iterations for final rule set	100
Iterations per temperature for final rule set	100
Initial temperature	0.7
Boltzmann constant (K)	0.5
TS	
Iterations for each initial rule set	100
Iterations for final rule set	100
Tabu list size	10

6.2. Experiment 1

The first experiment was done on STAB2 using the correctness of the rule sets as the evaluation function Eq. (3). We report the parameters that gave the highest results in Table 8. Table 9 shows the results of the three hybrid heuristics compared to C4.5 with standard deviation shown in parentheses. We can see from the table that all three heuristics were able to outperform C4.5 on both the training and testing sets. However, *SA–GA* showed the highest improvement of all with an increase in correctness of 7.5% on the training set and 3.6% on the testing set compared to C4.5. We find it important to point out that all three heuristics improved the $J(R)$ of the rule sets although this was not included in the evaluation functions. The increase in $J(R)$ means that our heuristics have a better accuracy than C4.5 on cases with a minority classification. The *SA–GA* is also the best performer with an average $J(R)$ improvement of 11.8% on training sets and 7.5% on testing sets.

6.3. Experiment 2

This experiment was done on *STAB1*. The evaluation function is a linear combination of both the correctness and $J(R)$ Eq. (3). Since STAB1 is an unbalanced data set, we decided to include the $J(R)$ in the evaluation function. We also included the correctness and gave it a higher weight since it remains the main criteria for evaluating software quality estimation models. The parameters that gave us the highest results are the same as those used in Experiment 1 (Table 8)

Table 10 shows the results of the three hybrid heuristics compared to C4.5 with standard deviation shown in parenthesis. All three hybrid heuristics were able to outbeat C4.5 in both the correctness and the $J(R)$. Since *STAB1* is an imbalanced data set, we consider $J(R)$ as a better quality criteria. *SA–GA–TS* showed the highest improvement.

6.4. Experiment 3

In this experiment, we used the unbalanced data set STAB1 and the $J(R)$ of the rule sets as the evaluation function. The parameters that gave us the highest results are also found in Table 8. Table 11 shows the results of the three hybrid heuristics compared to C4.5 and the GA in [5] with standard deviation shown in parentheses. All three heuristics outperformed C4.5 on the $J(R)$. *SA–GA* scores

Table 9
Results of Experiment 1. Correctness and J-Index (standard deviation) on testing and training sets.

	Correctness training (%)	J-Index training (%)	Correctness testing (%)	J-Index testing (%)
C4.5	68.55 (0.7)	57.43 (0.5)	67.97 (5.57)	57.47 (3.50)
[5]	74.5(1)	65(3)	70(6)	60.5(5)
SA–GA	76.01 (0.54)	69.20 (1.08)	71.58 (5.04)	64.94 (4.67)
TS–GA	74.08 (0.56)	65.72 (0.95)	70.52 (5.53)	62.05 (3.93)
SA–TS–GA	75.28 (0.41)	68.17 (1.13)	70.59 (5.38)	63.33 (4.50)

Table 10
Results of Experiment 2. Correctness and J-Index (standard deviation) on testing and training sets.

	Correctness training (%)	J-Index training (%)	Correctness testing (%)	J-Index testing (%)
C4.5	68.44 (0.7)	58.22 (1.12)	66.52 (7.56)	55.70 (3.34)
[5]	86(1)	60.50(1)	85.5(1)	59(4)
SA–GA	75.65 (0.53)	70.38 (0.97)	70.84 (5.3)	65.17 (5.27)
TS–GA	73.74 (0.66)	67.31 (1.24)	69.13 (4.92)	62.52 (3.82)
SA–TS–GA	74.95 (0.55)	69.71 (0.74)	71.45 (4.47)	65.88 (4.78)

Table 11
Results of Experiment 3. Correctness and J-Index (standard deviation) on testing and training sets.

	Correctness training (%)	J-Index training (%)	Correctness testing (%)	J-Index testing (%)
C4.5	78.52 (0.63)	66.50 (0.5)	78.32 (1.98)	65.98 (4.26)
[5]	86(1)	60.50(1)	85.5(1)	59(4)
SA–GA	79.89 (1.84)	79.17 (0.89)	79.06 (2.16)	77.45 (2.17)
TS–GA	77.81 (1.91)	72.97 (1.72)	77.34 (1.79)	71.86 (5.03)
SA–TS–GA	77.86 (2.71)	76.46 (1.82)	76.90 (2.38)	74.86 (3.23)

the highest improvement on $J(R)$ (12.7% on the training sets and 11.5% on the testing set). Is also important to point out the improvement that *SA–GA* scored with respect to the GA in [5]. As a matter of fact, *SA–GA* outperformed the GA by 18.45% on the testing set. Also, the *SA–GA* was able to slightly outperform the correctness of the rule sets built by C4.5. However, since STAB1 is an unblanced data set, we care more about the $J(R)$ than the correctness.

6.5. Insight on the tri-phased optimization process of the hybrid heuristics

In Figs. 15 and 16, we show the intermediate results obtained at the end of each optimization phase in Experiment 1 and Experiment 3. We chose Experiment 1 because it showed the best results (correctness) on STAB2 and Experiment 3 because it showed the best results ($J(R)$) on STAB1. We can see that for all the heuristics, it is during the second optimization phase (the genetic algorithm phase) that the most remarkable improvement is attained. This

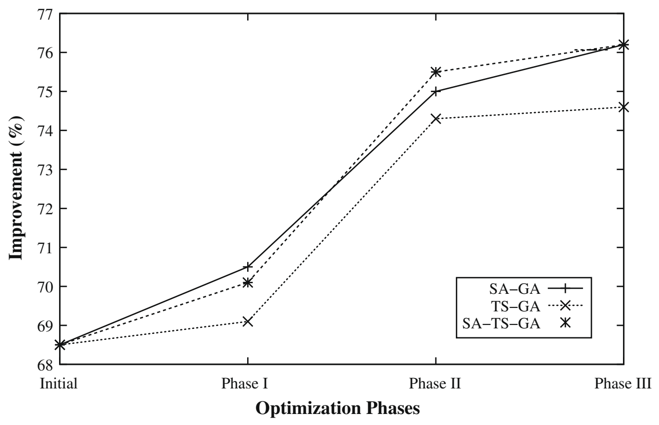

Fig. 15. Plot showing the improvement in correctness in Experiment 1 on STAB2 at the three phases of optimization for each hybrid heuristic.

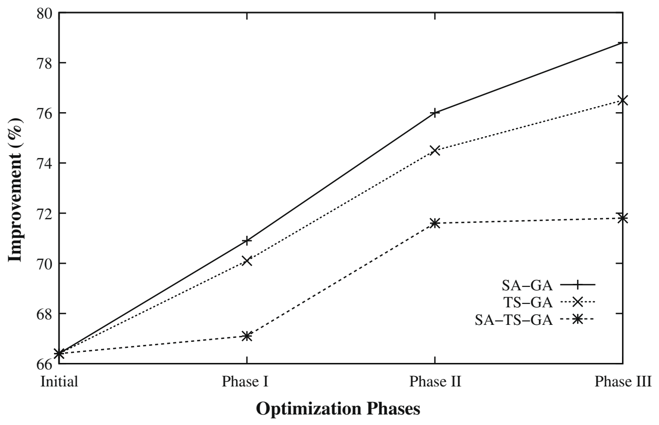

Fig. 16. Plot showing the improvement in J-Index in Experiment 3 on STAB1 at the three phases of optimization for each hybrid heuristic.

Table 12
Size of rule sets produced by the heuristics (C4.5).

	Rules per rule set	Conditions per rule
Experiment 1	14.9(3.9)	2.7(2)
Experiment 2	16(3.7)	2.9(2.1)
Experiment 3	17.3(3.7)	17.3(2.1)

can be explained by the strong combinational power of the GA as it is the only heuristic that optimizes a set of rule sets by combining them and adapting them to the dataset while the other two only adapt a single rule set at a time to the dataset.

7. Conclusion

In this paper, we proposed a hybrid heuristic to optimize existing software quality estimation models. The approach is adaptive as it adapts already built models to a new data set. The heuristic is hybrid on two levels: First, it optimizes the rules within the rule sets and the rule sets by recombining them. Second, it involves different heuristics (simulated annealing and/or tabu search and genetic algorithms). We have conducted experiments with two different data sets involving the stability of classes in an object-oriented system. One of the two stability data sets is unbalanced. All three heuristics outperform C4.5 as well as the GA that is presented in [5]. Due to the elitism operator in the GA, the technique guarantees no damage to the results and the end result is another rule-based estimation model hence, new guidelines for software engineers to follow in order to design stable classes in an object-oriented system. Our technique has two minor shortcomings. The first one is the execution time of the algorithm which is linear in the number of iterations of the GA, the number of rule sets in the population and the size of the rule sets. However, we believe that this is not a big disadvantage as we need to run the code once on a particular data set and the resulting rule sets can be used as often as needed. The second shortcoming of the technique is a slight increase in the complexity of the rule sets produced by the heuristic. This is summarized in Table 12. The rule sets produced by the heuristics have a slightly higher number of conditions per rule than those produced by C4.5. This is due to the condition addition crossover operator in the GA. Also, the number of rules per rule set is significantly higher. This is not a drawback as each rule provides a guideline for the experts to reach a certain classification label. Finally, it is worth pointing out that our approach is independent of the software quality characteristic that is being estimated and hence it is interesting to see how well it performs on other characteristics (such as maintainability, reusability, etc.).

Acknowledgements

This work was supported in part by a grant from the Lebanese National Council for Scientific Research (CNRS), and from the Research Council at the Lebanese American University.

References

[1] J. Aguilar-Ruiz, I. Ramos, J.C. Riquelme, M. Toro, An evolutionary approach to estimating software development projects, Information and Software Technology 43 (14) (2001) 875–882.
[2] E. Alba, J.F. Chicano, Software Project Management with GAs, Information Sciences 177 (11) (2007) 2380–2401.
[3] M.A. De Almeida, H. Lounis, W. Melo, An investigation on the use of machine learned models for estimating software correctability, in: International Journal of Software Engineering and Knowledge Engineering, Spcial Issue on Knowledge Discovery from Empirical Software Engineering Data, 1999.
[4] G. Antoniol, M. Di Penta, M. Harman, Search-based techniques for optimizing software project resource allocation, in: Proceedings of the 2004 Conference on Genetic and Evolutionary Computation (GECCO'04), vol. 3103, 2004, pp. 1425–1426.
[5] D. Azar, D. Precup, New Technologies, Mobility and Security, An Adaptive Approach to Optimize Software Component Quality Predictive Models: Case of Stability, Springer, Netherlands, 2007.
[6] D. Azar, D. Precup, S. Bouktif, B. Kégl, H. Sahraoui, Combining and adapting software quality predictive models by genetic algorithms, in: Proceedings of the 17th IEEE International Conference on Automated Software Engineering, IEEE Computer Society, 2002, p. 285.
[7] G.M. Barnes, R.B. Swim, Inheriting software metrics, Journal of Object Oriented Programming 6 (7) (1993) 27–34.
[8] V. Basili, K. Condon, K. El Emam, R.B. Hendrick, W.L. Melo, Characterizing and modeling the cost of rework in a library of reusable software components, in: Proceedings of the 19th International Conference on Software Engineering, Boston, MA, 1997, pp. 282–291.
[9] S. Bouktif, D. Azar, H. Sahraoui, B. Kégl, D. Precup, Improving rule set-based software quality prediction: a genetic algorithm-based approach, Journal of Object Technology 3 (4) (2004).
[10] L. Briand, P. Devanbu, W. Melo, An investigation into coupling measures for C++, in: Proceedings of the 19th International Conference on Software Engineering, 1997.
[11] L. Briand, J. Wüst, Empirical studies of quality models in object-oriented systems, Advances in Computers 20 (2000).
[12] L. Briand, J. Wüst, J. Daly, V. Porter, Exploring the relationships between design measures and software quality in object-oriented systems, Journal of Systems and Software 51 (2000).
[13] C.J. Burgess, M. Lefley, Can genetic programming improve software effort estimation? A comparative evaluation, Information and Software Technology 43 (14) (2001) 863–873.
[14] C.K. Chang, C. Chao, T.T. Nguyen, M. Christensen, Software project management net: a new methodology on software management, in: Proceedings of the 22nd Annual International Computer Software and Applications Conference (COMPSAC'98), 1998, pp. 534–539.
[15] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.
[16] J. Clark, J.J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, M. Shepperd, Reformulating software engineering as a search problem, IEE Proceedings – Software 150 (3) (2003) 161–175.
[17] J.C. Coppick, T.J. Cheatham, Software metrics for object-oriented systems, in: Proceedings of the ACM Computer Science Conference, 1992, pp. 317–322.
[18] C. Darwin, The Origin of Species by Means of Natural Selection: or the Preservation of Favored Races, John Murray, 1859.
[19] M.A. de Almeida, S. Matwin, Machine learning method for software quality model building, in: Proceedings of the 11th International Symposium on Foundations of Intelligent Systems, 1999, pp. 565–573.
[20] J.J. Dolado, On the problem of the software cost function, Information and Software Technology 43 (1) (2001) 61–72.
[21] F.B.e. Abreu, W.L. Melo, Evaluating the impact of object-oriented design on software quality, in: Proceedings of the 3rd International Symposium on Software Metrics, IEEE, 1996, p. 90.
[22] F. Glover, Future paths for integer programming and links to artificial intelligence, Computers and Operations Research 13 (5) (1986) 533–549.
[23] M. Harman, The current state and future of search based software engineering, in: 2007 Future of Software Engineering (FOSE'07), 2007, pp. 342–357.
[24] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, M. Roper, Testability transformation, IEEE Transactions on Software Engineering 30 (1) (2004) 3–16.
[25] M. Harman, B. Jones, Search based software engineering, Journal of Information and Software Technology 43 (14) (2001) 833–839.
[26] B. Hendersen-Sellers, Some metrics for object-oriented software engineering, in: Proceedings of the International Conference on Technology of Object-Oriented Languages and System, 1991, pp. 131–139.
[27] B. Henderson-Sellers, Object-Oriented Metrics: Measures of Complexity, Prentice Hall, 1996.
[28] J.H. Holland, Adaptation in Natural and Artificial Systems an Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, MIT Press, 1992.
[29] W.D. Jones, T.M. Khoshgoftaar, Using classification trees for software quality models: lessons learned, in: The 3rd IEEE International Symposium on High-Assurance Systems Engineering, 1998, pp 82–89.
[30] Taghi M. Khoshgoftaar, Edward B. Allen, Robert Halstead, Gary P. Trio, Ronald M. Flass, Using process history to predict software quality, Computer 31 (4) (1998) 66–72.
[31] C. Kirsopp, M. Shepperd, J. Hart, Search heuristics, case-based reasoning and software project effort prediction, in: Proceedings of the 2002 Conference on Genetic and Evolutionary Computation (GECCO'02), 2002, pp. 367–1374.
[32] W. Li, S. Henry, Maintenance metrics for the object-oriented paradigm, in: Proceedings of the First International Software Metrics Symposium, 1993, pp. 52–60.
[33] W. Li, S. Henry, Object-oriented metrics that predict maintainability, Journal of Systems and Software 23 (1993) 111–122.
[34] Z. Li, M. Harman, R.M. Hierons, Meta-heuristic search algorithms for regression test case prioritization, IEEE Transactions on Software Engineering 33 (4) (2007) 225–237.
[35] M. Lorenz, J. Kidd, Object-oriented Software Metrics: a Practical Approach, Prentice Hall, 1994.
[36] Y. Mao, H.A. Sahraoui, H. Lounis, Reusability hypothesis verification using machine learning techniques: a case study, in: IEEE Automated Software Engineering Conference, 1998.
[37] P. McMinn, Search-based software test data generation: a survey, Software Testing, Verification, and Reliability 14 (2) (2004) 105–156.
[38] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of state calculations by fast computing machines, The Journal of Chemical Analysis 21 (6) (2005) 1953.

[39] W. Pedrycza, G. Succic, Genetic granular classifiers in modeling software quality genetic granular classifiers in modeling software quality, Journal of Systems and Software 76 (3) (2005) 277–285.
[40] J.R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann Publishers Inc, 1993.
[41] D.E. Rumelhart, D.E. Hinton, R.J. Williams, Learning representations by back-propagation errors, Nature 323 (1986) 533–536.
[42] R. Selby, W. Porter, Empirically guided software development using metric-based classification trees, IEEE Software 7 (2) (1990) 46–54.
[43] R. Vivanco, Improving predictive models of software quality using an evolutionary computational approach improving predictive models of software quality using an evolutionary computational approach, in: Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2007), 2007, pp. 503–504.
[44] J. Wegener, A. Baresel, H. Sthamer, Evolutionary test environment for automatic structural testing, Information and Software Technology 43 (14) (2001) 841–854.
[45] H. Zuse, A Framework of Software Measurement, Walter de Gruyter, 1997.