An Ant Colony Optimization Heuristic to Optimize Prediction of Stability of Object-Oriented Components

H. Harmanani, D. Azar, G. Zgheib, and D. Kozhaya
Department of Computer Science and Mathematics
Lebanese American University
Byblos 1401 2010, Lebanon

***Abstract*—The IEEE 729-1983 Standard defines software quality as "the composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer." Assessing software quality in the early stages of design and development is crucial in reducing time and effort. Various *metrics* have been proposed for estimating software quality characteristics from measurable attributes. This paper presents an *Ant Colony Optimization (ACO)* approach that improves the prediction accuracy of software quality estimation models by intensifying the search around the metric neighborhood. The method has been implemented, and favorable results comparisons are reported.**

I. Introduction

Industry and beyond is becoming more and more dependent on software. Software is becoming more complex which makes their evolution time consuming and costly. There is a great need to assess the quality of a software product at the early development stages. Quality of software is measured in terms of characteristics such as maintainability, reusability, stability, etc. However, it is not possible to directly measure such characteristics but they can be inferred from some software attributes such as coupling, cohesion, etc. These can be directly measured and many metrics have been used for this purpose [5]. Examples of such metrics include number of lines of source code, percentage of statements in source code, cyclomatic factor, etc. In order to establish a relationship between the attributes and the software quality characteristic, estimation models are built. These models take the form of mathematical models or logical models. Logical models can be decision trees or rule-based models. The latter are easier to interpret by human experts. It is important to improve their accuracy as they do not perform as well as mathematical models but they are preferable due to their white-box nature.

This work presents an *Ant Colony Optimization* heuristic that defines a promising neighborhood for a metric and intensifies the search in it. We validate our technique using the software quality characteristic stability that is measured at the level of a class in an object-oriented software system. However, the approach can easily be used on any other software quality characteristic.

The remainder of this paper is organized as follows. Section II provides an overview of the related work while section III describes the problem and the formalism used throughout the paper. Section IV provides a brief introduction to *Ant Colony Optimization*. Section V presents our technique and describe our algorithm. Experimental results are presented in Section VI. We conclude in Section VII with a brief recollection of the technique and future paths.

II. Related Work

Machine learning has been widely used for constructing software quality estimation models. Selby *et al.* [10] have used decision trees as estimation models for software quality. Mao *et al.* [6] used C4.5 to build models that predict reusability of a class in an object-oriented software system. De Almeida *et al.* [1] use C4.5 rule sets to predict the average isolation effort and the average effort. Briand et al. [4] investigate 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. Various search-based software engineering approaches have been proposed. For example, Azar *et al.* [2] presented two genetic algorithm-based approaches that optimize the accuracy of these models on new data. Pedrycz et al. [8] represent classifiers as hypberboxes and uses genetic algorithms to modify these hyperboxes. Vivanco [11] uses a genetic algorithm to improve a classifier accuracy in identifying problematic components. Azar et. al [3] used an adaptive approach that takes predictive models that have already been built from common domain data and adapts them to context-specific data.

III. Predictive Model

The predictive model starts with a vector of attributes $v_i = (a_1, a_2, ..., a_n)$ as an input, and outputs a classification label c_i. In the context of this work, the attributes $a_1, a_2, ..., a_n$ are metrics (such as number of methods, number of computational statements in the code, etc.) that are considered to have a certain impact on the software quality factor being predicted (stability). The classification label $c_i \in \{0(stable), 1(unstable)\}$ represents this software quality factor. Each vector v_i describes a component i.e. a class in an object-oriented software system. A class in

978-1-4673-6656-4/15 $31.00 © 2015 IEEE
DOI 10.1109/IRI.2015.45

Rule 1: Lines $> 100 \wedge$ %Comments $\leq 5 \Rightarrow 1$
Rule 3: Lines $\leq 100 \wedge$ Calls $\leq 2 \Rightarrow 0$
Default Classification: 0

Figure 1. Example Classifier

an object-oriented software system is said to be "stable" if it keeps the same public interface between two different versions of the system. Otherwise, the class is said to be "unstable." The objective is to find a function f with a low error rate. The evaluation of f is done on a data set D of the form $D = (v_1, c_1), ..., (v_m, c_m)$. 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 a classifier C that minimizes the error on D_{train}. C is then evaluated on D_{test}. C4.5 [9] is an example of a machine learning algorithm that uses this principle by building classifiers in the form of decision trees. The decision trees can be transformed later on to rule sets which we call rule-based classifiers (or classifiers for short) or rule-based models (or models for short) in this work. A rule-based classifier is a disjunction of rules and a default classification label where each rule is a conjunction of conditions or tests performed on the metrics a_i.

We illustrate rule-based classifiers using the example shown in Figure 1. The example predicts the stability of a component based on the metrics *Lines* (actual physical lines in source code), *Comments* (percentage of comments in the source code), and *Calls* (number of statements that include method calls). The classifier consists of two rules and a default classification label. The first rule classifies a class with number of lines greater than 100 and the percentage of comments in the source code less than or equal to 5 as unstable. The second rule classifies a class with number of lines no more than 100 and number of statements which include method calls less than or equal to 2 as stable. The classification is sequential i.e. the first rule whose left hand side is satisfied by a class classifies the class. If no such rule exists, the default classification label is used to classify the class (default classification 0). The quality of the classifier is measured in terms of its accuracy. The latter is measured in terms of correctness i.e. percentage of classes correctly classified as given by the following equation:

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

where n_{ij} is the number of classes that were predicted to have classification label i but have classification label j. Our goal is to maximize the correctness on the testing set.

IV. Ant Colony Optimization

Ant colony optimization (ACO) is a metaheuristic that finds solutions for complex combinatorial optimization problems [7]. The algorithm is based on the behavior of real ants that can find the shortest path *from* the nest *to* the food source and back *to* the nest by depositing a chemical substance called *pheromone*. By sensing pheromone trails, *foragers* can follow the path to food discovered by other ants. With time, shortest routes accumulate a higher concentration of pheromone and almost all ants end up using them. ACO models the behavior of real ants where each ant builds a solution to the problem by applying a step-by-step decision that assigns an initial amount of pheromone to trails under which lie promising solutions. The amount of deposited pheromone is updated based on a probability that indicates the desirability of the step that was made. Finally, centralized actions that cannot be performed by single ants are applied.

V. ACO Predictive Algorithm

An efficient implementation strategy for an ACO should include the following steps: (1) A graph problem representation that facilitates the concurrent and asynchronous step-by-step solutions building by the ants. An ant moves from one state to an adjacent one on the graph. The move is based on the pheromone and the heuristic value associated with the underlying edge that connects the two states; (2) the update of the pheromones trails by either increasing or decreasing the trails value; (3) the daemons that implement centralized actions that cannot be performed by single ants.

A. Problem Representation

We model the problem as a search for best path using a directed graph, $G = (V, E)$. Each vertex $v \in G$ corresponds to a metric that may or may not be selected, and is assigned a weight that represents the amount of pheromone that an ant may deposit on this part of the graph. The resulting graph is traversed by the ants based on a probabilistic approach where each traversal results with a candidate solution (Figure 2).

Let T be the training set where the metrics show a *normal Gaussian distribution*. For each metric in T, we compute the *mean* as well as the *standard deviation* of all values for which the data instance has classification label "1". Let μ be the mean and σ the standard deviation for a particular metric m. We associate with m the following four ranges: $]-\infty, \mu-\infty[$, $[\mu-\infty, \mu[$, $[\mu, \mu+\infty[$, and $[\mu+\infty, +\infty[$. Consequently, most of the values of the metric that are between $\mu - \sigma$ and $\mu + \sigma$ favor a classification label of 1 while those outside this range favor a classification label of 0.

B. Solution Construction

There are various factors that affect the rules construction such as the metrics included in the rule and the ranges of the chosen metrics. Each ant explores the construction of one rule with the objective of contributing to the final *rule set*. The construction of a single rule proceeds as follows. An ant starts by computing the statistical ranges and then scans all the metrics one at a time in order to decide whether or

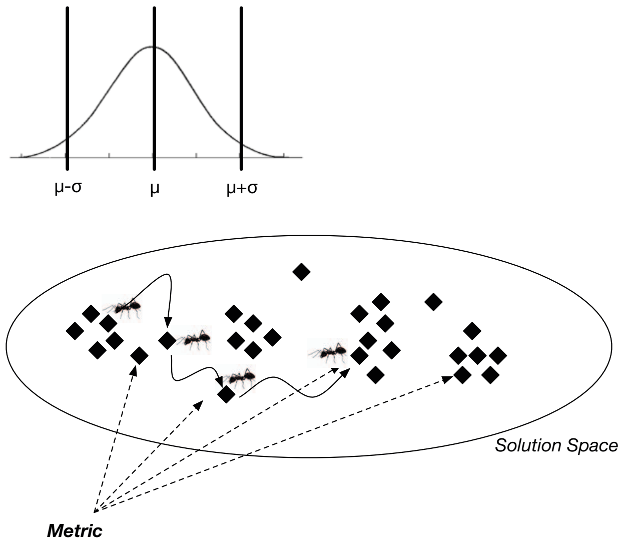

Figure 2. Illustrating the ACO Algorithm

not to include the metric in the rule. A metric is included in the rule with a probability of 20% based on the following:

$$P_{ij} = \frac{\tau_{i,j}^{\alpha} * \mu_{i,j}^{\beta}}{\sum_{i,j}(\tau_{i,j}^{\alpha} * \mu_{i,j}^{\beta}}) \quad (2)$$

Where μ is the randomness element, t is the pheromone value, α is the power of pheromone, β is the power of randomness, i is the index of attribute, and j is the range.

An ant generates a rule once it completes the scanning of all metrics. The rule will have a subset of all metrics along with a range for each. The classification label of the rule is resolved by scanning the whole rule and examining at the ranges of the metrics in these rules. If the majority of the ranges are between $[\mu-\sigma, \mu+\sigma]$ then the classification label will be set to "1"; otherwise it will be set to "0."

C. Solution Validation

Once the rule has been generated, it is tested on the training data set by computing the number of data points that can be classified by this rule, N_C, as well as the number of correctly classified data points, N_{CO}. The algorithm computes the rule's *efficiency* as well as the rule's *accuracy* both of which are included in the objective function (Equation 3) that is used to evaluate the rule:

1) Efficiency = ($N_C - N_{CO}$)/ (total # of data points);
2) Accuracy = N_{CO} / (total # of data points).

If the objective function of the rule is better than that of the worst rule, it is accepted and replaces the worst rule.

D. List of Rules

At the end of every iteration, each ant generates a rule. The best rules are stored throughout the process in a list. The length of this list is equal to the total number of ants. Initially, the list is empty. During the first iteration, each ant will add its rule to the list. Then, throughout the iterations, each ant will scan this list for its weakest rule. If this weakest rule is worse than the rule currently generated by the ant, then this new rule will replace it in the list. Otherwise, the ant's new rule is discarded. While running experiments, we noticed that a higher accuracy might be achieved by ensuring the survival of "good" rules with classification label "0." Therefore, a certain priority is provided to these particular rules while updating the list of rules.

Algorithm 1 Ant Colony Predictive Algorithm

```
Input: Set of Metrics

 1: $\mu$ = mean
 2: $\sigma$ = standard deviation
 3: Find ranges for each $x$ in a set of metrics
 4: for all Generations do
 5:     for all t do
 6:         for all Ants do
 7:             Sort all rules in ascending order by accuracy
 8:             Set default class to 1
 9:             Test Rule efficiency
10:             if f(r) > f(Worst Rule) then
11:                 replace worst rule with r
12:             end if
13:              Update pheromone
14:              Update List of Rules
15:         end for
16:         for all Vertices do                  ▷ Evaporate pheromone
17:             $M_{ij} \leftarrow M_{ij} - (1-p)$                 ▷ $0 < p < 1$
18:         end for
19:     end for
20:     Generate the Rule Set from the list of Rules
21:     /* Test Rule Set Efficiency*/
22:     $N_C \leftarrow$ number of data points that can be classified
23:     $N_{CO} \leftarrow$ number of data points correctly classified
24:     efficiency $\leftarrow (N_C - N_{CO})/|T|$
25:     accuracy $\leftarrow N_{CO}/|T|$
26:     $f = c_1 * accuracy + c_2 * efficiency$
27: end for
```

Algorithm 2 Generate Rule Set

```
Input: Set of Metrics

 1: for all Ants do
 2:     for all metrics do
 3:         include the metric in the rule with $p = 0.2$
 4:         if (metric included) then
 5:             pick a range to choose a value from
 6:         end if
 7:         if range chosen is $[\mu-\sigma, \mu+\sigma]$ then
 8:             assign 1 as a classification label to rule
 9:         else
10:             assign 0 as a classification label to rule
11:         end if
12:     end for
13: end for
```

Fold	Accuracy - Training			Accuracy - Testing		
	C4.5	ACO	% Improv	C4.5	ACO	% Improv
0	0.63	0.67	5.58	0.59	0.62	4.73
1	0.63	0.66	5.15	0.63	0.66	5.76
2	0.63	0.67	5.70	0.63	0.66	5.59
3	0.63	0.67	5.96	0.63	0.66	4.07
4	0.63	0.66	5.14	0.63	0.68	8.01
5	0.63	0.66	5.51	0.63	0.66	5.31
6	0.63	0.67	5.78	0.63	0.67	6.46
7	0.63	0.66	5.32	0.63	0.67	5.57
8	0.63	0.66	5.45	0.63	0.67	5.57
9	0.63	0.66	5.12	0.63	0.67	6.83

Table I
RESULTS COMPARISONS

E. Rule Set Generation

When all the iterations are completed, the rules are sorted in ascending order of accuracy. This results in the rule with the highest accuracy getting the highest priority for classifying the data points. This will reduce the number of misclassified data points and will lead to a higher number of correctly classified cases and a higher overall accuracy. The rules of the “sorted” list are used to create the rule set, while the default classification label is set to “1.”

F. Pheromone Update

Unlike classical ACO approaches, our algorithm stores the pheromones on the vertices instead of the edges of the graph. The ants generate rules during every iteration, at the end of which the process of updating the pheromone begins. Thus, each ant updates the nodes it visited. The amount of deposited pheromone is proportional to the objective function of the respective rule. This update is based on the following equation: $\tau_{ij} = \tau_{ij} + f$ where f is the objective function, i is the index of the attribute, and j is the range.

To ensure that the algorithm does not fall into a local optimum, a process of evaporation is performed. At the end of every iteration, the algorithm decrements the amount of pheromone by using an evaporation rate, ρ, according to the following function: $\tau_{ij} = \tau_{ij} * (1 - \rho)$ where $0 < \rho < 1$.

G. Objective Function

The objective function is defined as:

$$F = c_1 * accuracy + c_2 * efficiency \qquad (3)$$

Where c_1 and c_2 are two parameters tuned during runtime.

VI. EXPERIMENTS AND RESULTS

The proposed algorithm has been implemented and tested based on a 10 fold cross-validation. Thus, the data set was broken down into 10 subsets of roughly equal size. Nine of these were merged to form the training set and the remaining one was left to test the algorithm. When the algorithm was trained, it generated a rule set. This rule set was evaluated on the training set as well as on the testing set. All experiments were repeated 30 times in order to account for the random element inherent in the algorithm. Averages and standard deviations were computed. The algorithm has several parameters, so tuning was performed in order to cover the wide range of possibilities, and to come up with the set of parameters that allows the algorithm to reach a higher accuracy. The experimental results were compared to the performance of C4.5 and they are both reported, along with the percentage improvement. We can see from Table I that the proposed algorithm outperforms C4.5 on all folds, on both the training (5.47%) and the testing sets (5.79%).

VII. CONCLUSION

We have presented an ACO algorithm that improves the accuracy of software quality estimation models. The algorithm determines promising neighborhoods for metrics used to estimate the stability of a software component, and intensifies the search in those neighborhoods.

REFERENCES

[1] M. A. De Almeida, H. Lounis, and W. Melo. An Investigation on the Use of Machine Learned Models for Estimating Software Correctability. *Journal of Software Engineering and Knowledge Engineering*, 1999.

[2] D. Azar and D. Precup. An Adaptive Approach to Optimize Software Component Quality Predictive Models: Case of Stability. In *New Technologies, Mobility and Security*, Springer, 2007.

[3] D. Azar and J. Vybihalb. An Ant Colony Optimization Algorithm to Improve Software Quality Prediction Models: Case of Class Stability. *Information and Software Technology*, 53(4):388–393, 2011.

[4] L. Briand, P. Devanbu, and W. Melo. An Investigation into Coupling Measures for C++. In *Proc. Int. Conference on Software Engineering*, 1997.

[5] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Approach*. Prentice Hall, 1994.

[6] Y. Mao, H.A. Sahraoui, and H. Lounis. Reusability Hypothesis Verification Using Machine Learning Techniques: A Case Study. In *IEEE ASE Conf.*, 1998.

[7] M. Dorigo and T Stützle. *Ant Colony Optimization*. MIT Press, 2004.

[8] W. Pedrycza and G. Succic. Genetic granular classifiers in modeling software quality Genetic Granular Classifiers in Modeling Software Quality. *Journal of Systems and Software*, 76(3):277–285, 2005.

[9] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993.

[10] R. Selby and W. Porter. Empirically Guided Software Development Using Metric-Based Classification Trees. *IEEE Software*, 7(2):46–54, 1990.

[11] R. Vivanco. Improving predictive models of software quality using an evolutionary computational approach. In *IEEE ICSM 2007*, 2007.