MYRA: A Java Ant Colony Optimization Framework for Classification Algorithms

Fernando E. B. Otero
School of Computing
University of Kent
Chatham Maritime, Kent, UK
F.E.B.Otero@kent.ac.uk

ABSTRACT

This paper introduces MYRA, an open-source Java framework that provides the implementation of several ant colony optimization classification algorithms. The algorithms are ready to be used from the command-line or can be easily called from custom Java code. The framework is implemented using a modular architecture, therefore algorithms can be easily extended to incorporate different procedures and/or use different parameter values. The paper gives particular attention to the common architecture from which the algorithms are built on, highlighting the shared classes among the different implemented algorithms. The source code and documentation of MYRA are available for download at https://github.com/febo/myra.

CCS CONCEPTS

•**Computing methodologies** → **Machine learning algorithms;**

KEYWORDS

classification, data mining, ant colony optimization

ACM Reference format:
Fernando E. B. Otero. 2017. MYRA: A Java Ant Colony Optimization Framework for Classification Algorithms. In *Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3067695.3082471

1 INTRODUCTION

The advances in computer technologies, allowing the storage of virtually any kind of data, have led to an exponential growth of available information and, consequently, to an increased interest in (semi-)automated data analysis techniques. The research area concentrated in applying computation models to extract knowledge from real-world structured data is called data mining [5]. One of the most studied data mining tasks in the literature is the classification task. In essence, the classification task consists of learning a predictive relationship between input values and a desire output. Each instance (data point) is described by a set of attributes (features)—referred to as predictor attributes—and a class attribute. Given a set of instances, a classification algorithm aims at creating a model, which represents the relationship between predictor attributes values and class values, capable of predicting the class of an instance based on the values of its predictor attributes.

Since the goal of a classification algorithm is to find the best predictive model, a classification problem can be cast as an optimisation problem. A wide range of techniques have been used to design classification algorithms—e.g., statistical algorithms, artificial neural networks, evolutionary algorithms, ant colony optimization, among others. A good collection of machine learning algorithms for classification task can be found in the Waikato Environment for Knowledge Analysis (WEKA) workbench [6], probably the most popular open source data mining tool.

Ant colony optimization (ACO) [4] is a metaheuristic inspired by the behaviour of real ant colonies. Many ant colonies are able to cooperate to perform complex tasks despite the lack of a central control mechanism and a relative simple individual behaviour—e.g., there are ant species with limited or no vision that are able to find the shortest path between the nest and a food source. All interaction between individual ants, and between the environment, is accomplished in an indirect way by dropping pheromone on the ground to create a pheromone trail. Trails with higher pheromone concentration are more likely to be followed by ants. Since the shorter trail is traverse faster, its pheromone concentration is higher, which eventually will result in the colony converging to using predominantly the shorter path. ACO algorithms mimic the behaviour of ant colonies to perform a global search, where the search is guided to better regions of the search space based on the quality of the solutions. In the context of the classification task in data mining, ACO algorithms have the advantage of performing a flexible robust search for a good combination of predictor attributes, less likely to be affected by the problem of attribute interaction [7].

Parpinelli et al. [24] proposed the first ACO algorithm for classification, called Ant-Miner, to create *IF-THEN* classification rules. The *IF* part corresponds to the antecedent of the rule and it contains (attribute, value) terms representing tests on attributes' values; the *THEN* part corresponds to the consequent of the rule and it contains a class value prediction. An instance that satisfies all terms in the antecedent of a rule is said to be covered by the rule and it has the class value in the consequent of the rule predicted. After the introduction of Ant-Miner, research on ACO classification algorithms attracted greater attention—the original Ant-Miner paper has more than 990 citations according to Google Scholar—and a large number of variations have been proposed in the literature [14]. While the vast majority ACO algorithms for classification are focused on creating classification rules, there

GECCO '17 Companion, Berlin, Germany

978-1-4503-4939-0/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3067695.3082471

are also works focused on creating decision trees [2, 22], hierarchical classification [18, 21], learning bayesian network classifiers [25] and training artificial neural networks [1]. Despite the large number of publications proposing ACO classification algorithms, the algorithms' implementations are not publicly released in most cases. The few exceptions to the best of our knowledge are Ant-Miner (https://sourceforge.net/projects/guiantminer/), AntMiner+ [13] (http://www.antminerplus.com) and PSO/ACO2 [10] (https://sourceforge.net/projects/psoaco2)—although PSO/ACO2 is not a *pure* ACO algorithm, since it combines an ACO and a particle swarm optimization procedures.

This paper introduces MYRA, a freely available, open source, object-oriented software framework for ACO classification algorithms, written in Java. The framework provides the implementation for several ACO classification algorithms: Ant-Miner, *c*Ant-Miner [19, 20], *c*Ant-Miner$_{PB}$ [23], Unordered *c*Ant-Miner$_{PB}$ [16, 17] and Ant-Tree-Miner [22]. MYRA has generated 8,950 downloads to date, since its first version released in 2008. The framework has been through two major refactorings—the last one on June 2015—in order to improve its modularity and computational time. The most recent version is 4.5 and it is the one discussed in this paper. The paper gives particular attention to the common architecture from which the algorithms are built on, highlighting the shared classes among the different implemented algorithms. While the framework has been exclusively used to design classification algorithms, there are architectural and design elements that can be used in the design of ACO algorithms for other problems. MYRA source code, documentation and a binary package are available for download at https://github.com/febo/myra, licensed under the Apache License (version 2.0).[1] The framework is self-contained—only depends on standard Java libraries—and requires Java version 1.7+. Javadoc is used throughout the source code and Apache Maven[2] is used to manage the framework's build, reporting and documentation. Note that this paper does not provide a discussion of the results achieved by the algorithms—a detailed statistical analysis can be found in the algorithms' original publications, where they have been compared against state-of-the-art algorithms from the literature.

The reminder of the paper is structured as follows. Section 2 discusses the core ACO classes of the framework. Section 3 discusses the implementation of ACO classification algorithms included in MYRA. Instructions on how to run the algorithms and visualise their output are given in Section 4. Finally, Section 5 presents conclusions and future development ideas.

2 ACO ALGORITHMIC COMPONENTS

In this section, we discuss the core ACO classes present in MYRA and the rationale for their design choice. ACO algorithms have three main caracteristics [12]: (i) they are population-based algorithms, where m ants create solutions to the problem at hand, mimicking a colony of ants; (ii) solutions are created by a probabilistic procedure, where solution components are selected based on pheromone and heuristic information values; (iii) pheromone values are updated at each iteration using the quality of the candidate solutions as (positive) feedback.

1. Initialise();
2. **while** termination condition not met **do**
3. ConstructAntSolutions();
4. ApplyLocalSearch();
5. UpdatePheromones();
6. **end while**

Figure 1: High-level pseudocode of an ACO algorithm.

Figure 1 presents the high-level pseudocode of an ACO algorithm. There are four main operations:

(1) *Initialise*: this procedure is responsible to initialise all data structures (e.g., pheromone values, heuristic information) and parameters of the algorithm;
(2) *ConstructAntSolutions*: this procedure mimics the movement of artificial ants over the construction graph to create candidate solutions. Solutions are created in a probabilistic fashion, where solution components are selected based on pheromone and heuristic information values—the higher the values, the more likely is the selection probability;
(3) *ApplyLocalSeach*: this is an optional procedure used to further refine candidate solutions. Its main purpose is to introduce small modifications to solutions in order to explore neighbouring solutions. Modifications that lead to a decrease in quality are usually discarded;
(4) *UpdatePheromones*: this procedure updates the pheromone associated with solution components. There are two stages in the update: (i) pheromone values are decreased due to evaporation, allowing ants to forget choices made at earlier stages of the search; (ii) pheromone values of the best candidate solution(s) are updated to increase the selection probability of good components (positive feedback).

The basic structure of an ACO algorithm, shown in Figure 1, is represented by the `Scheduler` class (package `myra`) in the framework. A `Scheduler` implementation is inspired by a *template method* design pattern [9], where the order of operations is defined but their individual implementations are delegated to the `Activity` interface. Therefore, the implementation of an ACO algorithm is defined by an `Activity`, which is then executed by the `Scheduler`. A `Scheduler` will execute an `Activity` in its `run` method:

```
public void run() {
   initialise();

   while (!terminate()) {
      create();

      search();

      update();
   }
}
```

[1] Previous MYRA 1.x, 2.x and 3.x versions can be found at https://sourceforge.net/projects/myra/. Readers interested in hierarchical classification algorithms [18, 21] (MYRA 3.x) should check this repository.
[2] http://maven.apache.org

Each method on the `Scheduler` (`initialise`, `terminate`, `create`, `search` and `update`) will delegate the call to the encapsulated `Activity` object. As can be seen in the code listing, the `run` method closely implements the pseudocode shown in Figure 1.

One motivation for decoupling the implementation of the operations and the structure (order of operations) of the algorithm is that different `Scheduler` implementations can be used to execute the same `Activity`. For example, the framework includes a `ParallelScheduler` that executes an `Activity` in a parallel fashion, taking advantage of multi-threading environments—the only requirement is that the implementation of the `Activity` is thread-safe. The `Activity` interface defines the following methods:

```
public interface Activity<T extends Comparable<T>> {

   // creates a single solution to the problem
   public T create();

   // applies local search to candidate solutions
   public boolean search(Archive<T> archive);

    // performs the initialisation step
   public void initialise();

   // indicates if the algorithm should stop
   public boolean terminate();

   // updates the state of the activity
   public void update(Archive<T> archive);

}
```

The generic parameter type `T` is the type of the solution created by the algorithm; and an `Archive` holds the solutions ordered by quality—if adding a new solution exceeds its capacity, the new solution is only added if it is better than the lowest solution in the archive. The framework provides an abstract `Activity` implementation, the `IterativeActivity` class. This class can be used as a base class for algorithms that are controlled by a maximum number of iterations. It provides an implementation for the `terminate` and `update` methods: in the `terminate`, it checks whether the maximum number of iterations has been reached; and in the `update`, it increments the number of iterations.

The `ParallelScheduler` is a subclass of the `Scheduler` and it overrides its `create` method implementation. While the `Scheduler` implementation creates a solution for each ant in the colony sequentially by iterating over the number of ants calling the `create` method of the encapsulated `Activity`, the `ParallelScheduler` uses a `CountDownLatch` (package `java.util.concurrent`) to execute the creation in parallel. In a `CountDownLatch`, multiple operations are executed in threads while the current (main) thread awaits. The rationale of using a `CountDownLatch` is to encapsulate each call to the `create` method as a `Runnable` object and use a set of threads to execute them. Note that only the `create` method perform parallel operations—i.e., only the creation of solutions is in parallel. This is a straightforward way to parallelise ACO algorithms and more complex strategies are likely to deliver better performance gains. The choice between the sequential `Scheduler` and `ParallelScheduler` is done as a command-line argument, as it is the number of executor threads.

Input: Training Instances
1. RuleList ← {};
2. **while** |Instances| > threshold **do**
3. Rule ← LearnOneRule(Instances);
4. Instances ← Instances − Covered(Rule, Instances);
5. RuleList ← RuleList ∪ Rule;
6. **end while**

Figure 2: High-level pseudocode of the iterative rule learning process.

While MYRA was not designed to be a generic ACO framework, a `Scheduler` can execute any ACO algorithm, as long as it is implemented as an `Activity`; the `Archive` can store any type of solution, as long as the class representing the solution type implements the `Comparable` interface so the `Archive` is able to maintain the order of solutions.

3 CLASSIFICATION ALGORITHMS

This section discusses the implementation of ACO classification algorithms included in MYRA. The algorithms presented can be divided into three categories in terms of their strategy to create a classification model: iterative rule learning (Ant-Miner and *c*Ant-Miner), Pittsburgh-based rule learning (*c*Ant-Miner$_{PB}$ and Unordered *c*Ant-Miner$_{PB}$) and decision tree learning (Ant-Tree-Miner). An overview of the different strategies is presented together with the details of the algorithms' implementations.

A classification algorithm is represented as a subclass of the `Classifier` class (package `myra.datamining`). It aims at training a classification model (`Model` interface). The `Classifier` class is an abstract class—subclasses are required to implement the `description` and `train` methods. A `Dataset` object is given as a parameter to the `train` method, representing the input (training) data. It contains a collection of `Instance` objects, each describing the values of the predictor and class attributes (`Attribute` class). The `train` method is responsible for executing an `Activity`, which represents the ACO procedure.

3.1 Iterative rule learning

Both Ant-Miner and *c*Ant-Miner follow an iterative rule learning approach to create classification rules from the training data. In this approach, a list of rules is created in a sequential covering fashion: at each iteration a rule that covers some training instances is created; training instances covered by the rule are removed and the rule is added to the list; the process is then repeated until the number of training instances is lower than a user-specified threshold. Figure 2 presents the high-level pseudocode of the iterative rule learning. In Ant-Miner and *c*Ant-Miner, the `LearnOneRule` procedure is implemented as an ACO procedure to search for the best rule given a set of instances.

In MYRA, the `SequentialCovering` class (package `myra.rule.irl`) implements the iterative rule learning approach. The rule creation procedure is implemented as `FindRuleActivity`, which is a subclass of the `IterativeActivity` class, and it is executed by a `Scheduler` at each iteration of the sequential covering. Each iteration is actually executing an independent ACO procedure to create a classification rule and, since the training instances are different, they create different classification rules.

The `FindRuleActivity` makes use of four important components: (1) `Graph`, representing the ACO construction graph from where ants choose solution components; (2) `RuleFactory`, representing the probabilistic procedure to select components from the construction graph to create a rule; (3) `PheromonePolicy`, representing the strategy to control pheromone values during the ACO execution; and (4) `Pruner`, representing a procedure to remove irrelevant terms from the antecedent of rules.

The `Graph` (package `myra.rule`) is created using the information from the predictor attributes. Given a set of nominal attributes $\mathcal{X} = \{x_1, \ldots, x_n\}$, where the domain of each nominal attribute x_i is a set of values $\mathcal{V}_i = \{v_{i1}, \ldots, v_{id_i}\}$ (where d_i equals to the number of values in the domain of attribute x_i), the construction graph consists of an almost fully connected graph. For each pair of nominal attribute x_i and value v_{ij} (where x_i is the i-th nominal attribute and v_{ij} is the j-th value belonging to the domain of x_i), a vertex representing the term $x_i = v_{ij}$ is added to the construction graph. Then, vertices are connected by edges to every other vertex of the construction graph, with the restriction that there are no edges between vertices referring to the same attribute to avoid inconsistent terms such as '*gender* = *male* AND *gender* = *female*' being included in the same rule; otherwise the rule would cover no examples. Ant-Miner only copes with nominal attributes, therefore any attempt to load a dataset with continuous attributes will generate an exception. *c*Ant-Miner, on the other hand, can cope with both nominal and continuous attributes. For each continuous attribute y_i, a vertex is added to the graph representing the term y_i. Then, the newly created vertex y_i is connected to all previous vertices of the construction graph. Note that the `Graph` class supports both nominal and continuous vertices, therefore there is a single `Graph` implementation in the framework. Each vertex has associated a pheromone and heuristic values (as `double` values). Heuristic information can be specified implementing the `Heuristic` interface.

The `RuleFactory` (package `myra.rule.irl`) is a probabilistic procedure to select vertices from the construction graph to create a rule, mimicking the action of an ant traversing the graph. Each ant starts with an empty rule—i.e., a rule with an empty antecedent—and iteratively selects vertices to add to its partial rule based on their values of the amount of pheromone τ and a problem-dependent heuristic information η. The probability of ant k selecting a particular vertex c_i at each iteration of the rule construction process is given by

$$p_{c_i}^k = \frac{[\tau_{c_i}]^\alpha \cdot [\eta_{c_i}]^\beta}{\sum_{c_j \in \mathcal{N}^k} [\tau_{c_j}]^\alpha \cdot [\eta_{c_j}]^\beta}, \quad \text{if } c_i \in \mathcal{N}^k, \tag{1}$$

where τ_{c_i} and τ_{c_j} are the amount of pheromone associated with neighbouring vertices c_i and c_j; η_{c_i} and η_{c_j} are the values of heuristic information associated with neighbouring vertices c_i and c_j; $\mathcal{N}^k$ is the feasible neighbourhood of ant k—i.e., the set of vertices that (1) the ant k has not visited and (2) do not correspond to terms associated with attributes already used by current terms in the antecedent of ant k; α and β are user-defined weight parameters that indicate the relative importance of the pheromone and heuristic information. This process is repeated until all feasible vertices have been visited or any vertices added to the antecedent would make the rule cover fewer training instances than a user-defined minimum value. The latter restriction is used to avoid too specific and unreliable rules.

After a `Rule` (package `myra.rule`) is created, it undergoes a pruning procedure (`Pruner` class). The pruning aims at removing irrelevant terms from the antecedent of a rule, added as a consequence of the probabilistically nature of the rule construction. Terms are removed while the quality of the rule does not decreases. The pruning procedure can be seen as an ACO local search operator. There are two pruning procedures implemented in the framework: a greedy pruner (`GreedyPruner` class), which removes one-term-at-a-time term of the rule until the rule quality decreases [24]; and a backtrack pruner (`BacktrackPruner` class), which removes the last term of the rule until the rule quality decreases [20]. The quality of a rule is determined using an evaluation function represented by a `RuleFunction` object—the framework provides several implementations of evaluation function in the `myra.rule.function` package. The `PheromonePolicy` (package `myra.rule.irl`) is responsible for the pheromone update and evaporation. Pheromone update is implemented by increasing the pheromone values by a value proportional to the quality of the iteration-best rule. Pheromone evaporation is implemented by normalising pheromone values after the update—values that have not been increased during the update will decrease as a result of the normalisation.

At the end of the `FindRuleActivity`, usually controlled by a maximum number of iterations, the best rule created by the ACO procedure is returned to the iterative rule learning procedure. Given the modular architecture of the framework, any of these implementation classes can be easily replaced by custom ones. Therefore, it is straightforward to create variations of the algorithms. Figure 3 presents a simplified class diagram illustrating the dependencies among main classes used in Ant-Miner and *c*Ant-Miner implementation.

3.2 Pittsburgh-based rule learning

As aforementioned, the creation of each rule is an independent execution of an ACO procedure in the iterative rule learning. In a Pittsburgh-based approach, a complete list of rules is created by each ant instead of single rule. It has two main advantages when compared to iterative rule learning: (i) it copes better with rule interaction problem[3]; (ii) the ACO algorithm is guided by the quality of a complete list of rules. MYRA includes the implementation of two Pittsburgh-based classification algorithms: *c*Ant-Miner$_{\text{PB}}$ and Unordered *c*Ant-Miner$_{\text{PB}}$. While both algorithms follow a Pittsburgh-based strategy, as their name suggest, they differ on the

[3]The problem of rule interaction in iterative rule learning derives from the order in which rules are created—the outcome of a rule influences the rules that can be created by subsequent iterations [8].

model representation: *c*Ant-Miner$_{PB}$ produces a `RuleList` (package `myra.rule`), where the order of rules is important to make predictions; Unordered *c*Ant-Miner$_{PB}$ produces a `RuleSet`, where the order of rules is not important. These algorithms are implemented reusing many classes from the *c*Ant-Miner implementation—the main difference is on their `Activity` implementation.

Ordered Rules. The *c*Ant-Miner$_{PB}$ procedure to create a complete list of rules is implemented by the `FindRuleListActivity` class (package `myra.rule.pittsburgh`). Differently than the iterative rule learning approach, the ACO procedure implemented by `FindRuleListActivity` is responsible to create a complete list of rules. Therefore, no multiple executions of the ACO procedure are required. A `FindRuleListActivity` works in a sequential covering fashion, but rules created at each iteration do not necessarily correspond to the best rule. Therefore, the `LearnOneRule` is replaced by a probabilistically construction process (`LevelRuleFactory` class) that is not optimised to create the best possible rule since the list of best rules is not necessarily the best list of rules. The quality quality of the individual rules is not important in *c*Ant-Miner$_{PB}$, as long as the quality of the list of rules improves, since the entire list is created at once and the best list is chosen to guide the ACO search. The `FindRuleListActivity` uses a `LevelRuleFactory` to create rules at different positions (levels) of the list. In essence, a `LevelRuleFactory` works as a `RuleFactory`, with the difference that it uses pheromone values indexes by the position of the rule—i.e., the first rule uses pheromone values at index 0, the second at index 1 and so forth. In this way, ants are able to identify good vertices for multiple rules.

Given the use of multiple pheromone values, a `LevelPheromonePolicy` class is used. It updates the pheromone values in the same way as the `PheromonePolicy` from Ant-Miner, with the difference that it takes a `RuleList` object as a parameter, corresponding to the iteration-best list of rules. Each rule in the list is used to increment the pheromone values of its associated level. The increment is proportional to the quality of the iteration-best list of rules, measured by a `ListMeasure`—the framework provides different implementation of measures in the `myra.classification.rule` package. Pheromone evaporation is implemented by decreasing the pheromone values by a user-defined factor ρ. Figure 4 presents a simplified class diagram illustrating the dependencies among main classes used in *c*Ant-Miner$_{PB}$ implementation. As can be seen, there are many classes shared between *c*Ant-Miner$_{PB}$ and *c*Ant-Miner (Figure 3) implementations.

Unordered Rules. The Unordered *c*Ant-Miner$_{PB}$ implementation follows a similar structure, with the difference that the order in which the rules are created is not important to make predictions—i.e., it creates a set of rules. This is implemented by the `FindRuleSetActivity` class (package `myra.classification.rule.unordered`). Rules are created in a (non-optimised) sequential covering fashion, as in *c*Ant-Miner$_{PB}$. The `FindRuleSetActivity` uses a `FixedClassRuleFactory` to create rules for each class value. The prefix '`Fixed`' in the class name indicates that the class value of the rule is decided in advance, before the rule is created, and it guarantees that the algorithm will create at least one rule for each class value. While the order of rules is not important to make predictions, pheromone values are indexed by the position of the

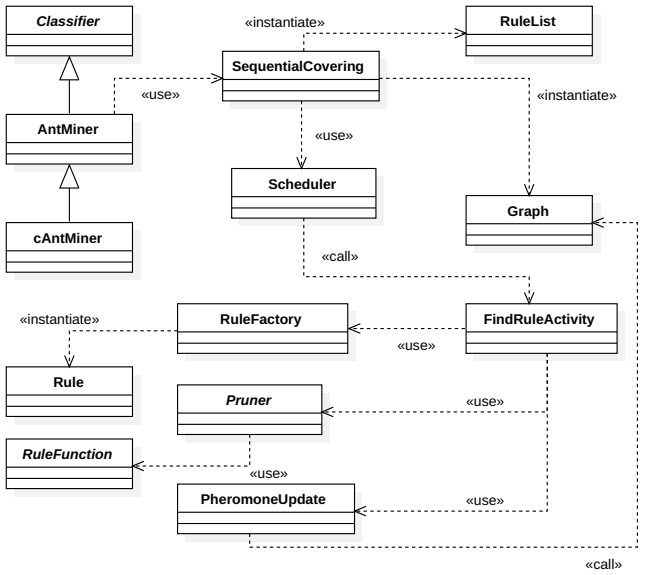

Figure 3: A simplified class diagram illustrating the dependencies among main classes used in Ant-Miner and *c*Ant-Miner implementation.

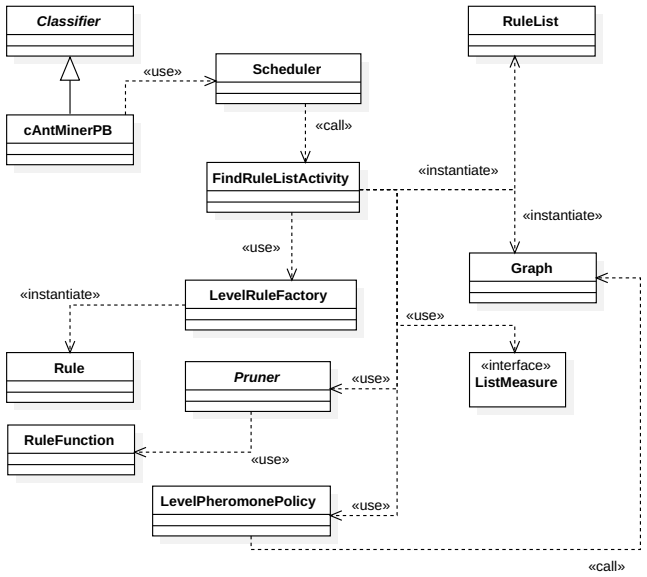

Figure 4: A simplified class diagram illustrating the dependencies among main classes used in *c*Ant-Miner$_{PB}$ implementation.

rule, the same way as *c*Ant-Miner$_{PB}$. It also uses a `LevelPheromonePolicy` to control the update of pheromone values. It should be noted that a `RuleSet` is a subclass of `RuleList`, therefore the same `ListMeasure` measures can be used to evaluate a `RuleSet`.

The implementation of Unordered *c*Ant-Miner$_{PB}$ includes an option to use a `FunctionSelector` (package `myra.classification.rule.function`) to dynamically select a `RuleFunction` for pruning [15]. The motivation for this is that evaluation functions

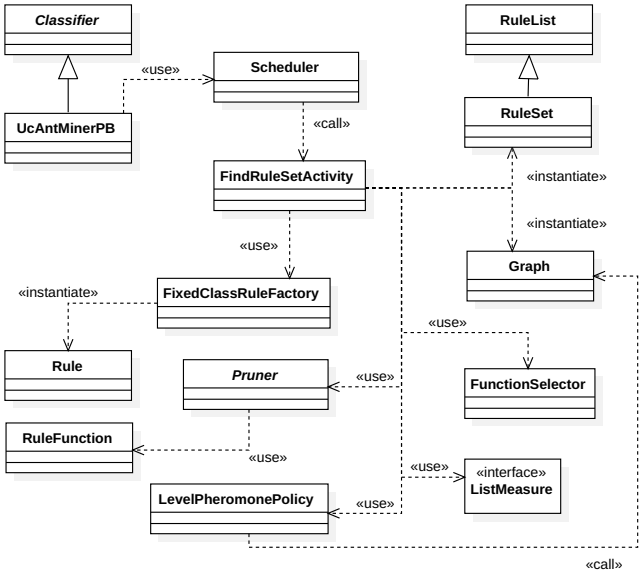

Figure 5: A simplified class diagram illustrating the dependencies among main classes used in Unordered *c*Ant-Miner$_{\text{PB}}$ implementation.

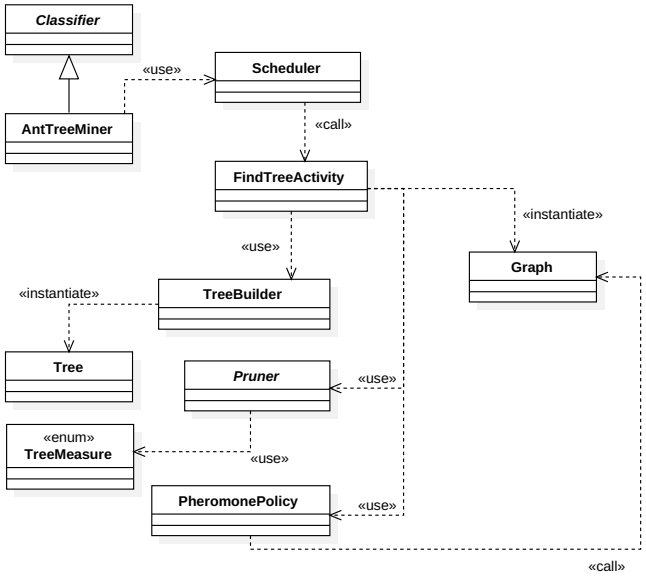

Figure 6: A simplified class diagram illustrating the dependencies among main classes used in Ant-Tree-Miner implementation.

have different bias and capture different aspects of a rule. The procedure to choose an evaluation function is similar to a vertex selection, where the pheromone associated with each function bias the selection. Hence, the algorithm is not only identifying which vertices are more suitable to create rules, but also the evaluation function more suitable to prune each individual rule.

Figure 5 presents a simplified class diagram illustrating the dependencies among main classes used in Unordered *c*Ant-Miner$_{\text{PB}}$ implementation. It is clear that there are many shared classes between *c*Ant-Miner$_{\text{PB}}$ (Figure 4) and the unordered version.

3.3 Decision Trees

In addition to algorithms that produce classification rules as a model, MYRA includes the implementation of Ant-Tree-Miner, an ACO algorithm to create decision trees. Ant-Tree-Miner follows the traditional top-down approach, often referred to as *divide-and-conquer*. In this approach, a decision tree is created iteratively, from the top (root node) to the bottom (leaf nodes). In the first iteration, an attribute is selected to represent the root node of the tree. Then, a branch is created for each different value in the domain of the attribute. Each branch represents a test on a particular attribute value. The data is then split into subsets according to the instances' values of the selected attribute—each subset is then associated with its corresponding branch. The attribute selection is then recursively repeated for each branch created until one of the following conditions is met: (i) all instances on a subset are associated with the same class value; or (ii) the number of instances in a subset is smaller than a user-defined minimum value. At this point, a leaf node predicting the majority class value—the class value associated with the majority of instances in the subset—is added to the tree and the recursive process stops.

As can been seen, the crucial step in the top-down strategy is the selection of attributes to create the nodes of the decision tree. In other words, the problem of creating a decision tree is divided into smaller problems of selecting an appropriate attribute given a subset of data. Ant-Tree-Miner employs an ACO procedure to select attributes during the decision tree construction. This is implemented by a `FindTreeActivity` (package `myra.classification.tree`). The first difference between Ant-Tree-Miner and Ant-Miner (and variations) is the construction graph structure: vertices in Ant-Tree-Miner's graph represent attributes instead of (attribute, value) terms. At each step of the tree construction procedure, an ant probabilistically selects vertex to visit based on the amount of pheromone and the heuristic information, in the same way as Equation 1. When an ant creates a candidate decision tree (`Tree` class), internal nodes are represented by `InternalNode` objects and leaf nodes are represented by `LeafNode` objects. A `Pruner` class is used to remove nodes that lead to an improvement in the quality of the tree. While Ant-Tree-Miner uses a different `PheromonePolicy` implementation, since the construction graph and the solution representation are different, it works in a similar fashion: each internal node is used to increment the pheromone values of its associated level (depth of the node in the decision tree). The increment is proportional to the quality of the decision tree, measured by a `TreeMeasure` enum.

Given that Ant-Tree-Miner is a population-based algorithm, multiple decision trees are created at each iteration of the ACO procedure. Similarly to Ant-Miner and variations, the `FindTreeActivity` is controlled by a maximum number of iterations. At the end, the best `Tree` object created is returned as the output of the algorithm.

Figure 6 presents a simplified class diagram illustrating the dependencies among main classes used in Ant-Tree-Miner implementation. It should be noted that Ant-Tree-Miner implementation has little overlap to the implementation of other rule-based algorithms, apart from ACO algorithmic components discussed in Section 2.

Table 1: Main class for the algorithms included in MYRA.

Algorithm / Main Class
Ant-Miner `myra.classification.rule.impl.AntMiner`
*c*Ant-Miner `myra.classification.rule.impl.cAntMiner`
*c*Ant-Miner$_{PB}$ `myra.classification.rule.impl.cAntMinerPB`
Unordered *c*Ant-Miner$_{PB}$ `myra.classification.rule.impl.UcAntMinerPB`
Ant-Tree-Miner `myra.classification.tree.AntTreeMiner`

Its `Graph`, `PheromonePolicy` and `Pruner` are different, since Ant-Tree-Miner uses a `Tree` as a solution representation instead of a list/set of rules.

4 RUNNING THE ALGORITHMS

The framework includes main classes to run the algorithms directly from the command-line. These are presented in Table 1. In order to run the algorithms from the command-line, the following template can be used:

`java -cp myra-<version>.jar <main class> -f <file>`

where `<version>` is MYRA jar version number (e.g., 4.5), `<main class>` is the main class name of the algorithm and `<file>` is the path to the attribute-relation file format (ARFF)[4] to be used as training data. The minimum requirement to run an algorithm is a training data file. If no training data file is specified, the list of command-line switches is printed. Figure 7 shows the command-line options for Ant-Miner algorithm.

The parameters of an algorithm can be tweaked using command-line options. Experiments can be reproduced by setting the same seed value (`-s` option), since ACO algorithms are stochastic and a pseudorandom number generator is used during their execution. Note that when running the algorithm in parallel (`--parallel` option), there is no guarantee that it will have the same behaviour even if the same seed value is used, since the thread allocation is not controlled in the code. Figure 8 shows an example of a run of *c*Ant-Miner. The output generated includes the values of the parameters to facilitate replicating an execution.

The algorithms can also be executed by instantiating the corresponding main class and calling the `train` method, which is specified by the abstract `Classifier` class. This method returns a `Model` object that can be used to make predictions.

5 CONCLUSIONS

This paper introduced MYRA, a Java ant colony optimization framework for classification algorithms. It provides the implementation of several popular ant colony optimization algorithms. The algorithms are ready to be used from the command-line or can be easily called from your own Java code. They are implemented using a modular architecture, so they can be easily extended to incorporate different procedures and/or use different parameter values.

The current version 4.x is a complete rewrite from version 3.x, although it was not possible to maintain backward compatibility. The overall architecture of the framework is very similar, but most data structures have changed. The computational time has been significantly improved—tasks that used to take minutes, now are done in seconds. In addition, there is an initial support for ant colony optimization regression algorithms [3].

Not all algorithms and features from version 3.x are implemented in current version. Namely, hierarchical multi-label algorithms, support for output of predictions and the GUI interface are not

[4] http://www.cs.waikato.ac.nz/ml/weka/arff.html

```
Usage: AntMiner -f <file> [-t <test file>] [options]

The minimum required parameter is a training file to
build the model from. If a  test file is specified,
the model will be tested at the end of training. The
results are presented in a confusion matrix.

The following options are available:

  -c <size>             specify the size of the
                        colony

  -g                    enables the dynamic
                        heuristic computation

  -h <method>           specify the heuristic
                        method

  -i <number>           set the maximum number
                        of iterations

  -m <number>           set the minimum number of
                        covered examples per rule

  -p <method>           specify the rule pruner

  -r <function>         specify the rule quality
                        function

  -s <seed>             Random seed value

  -u <number>           set the allowed number of
                        uncovered examples

  -x <iterations>       set the number of iterations
                        for convergence test

  --parallel <cores>    enable parallel execution in
                        multiple cores
```

Figure 7: Command-line options of Ant-Miner algorithm. When options are not specified, the algorithm is executed using the default values.

```
cAnt-Miner rule induction [build v4.5]
-----------------------------------------------

Training file: /uci/datasets/iris/iris.arff

[Runtime default values]
-s 1490115439698
-c 60
-i 1500
-m 10
-u 10
-x 10
-p backtrack
-r sen_spe
-h gain
-d mdl

Relation: iris
Instances: 150
Attributes: 4
Classes: 3
Random seed: 1490115439698


=== Discovered Model ===

IF petal-width <= 0.8 THEN Iris-setosa
IF petal-length > 5.15 THEN Iris-virginica
IF petal-width <= 1.45 THEN Iris-versicolor
IF petal-width > 1.75 THEN Iris-virginica
IF <empty> THEN Iris-versicolor

Number of rules: 5
Total number of terms: 4
Average number of terms: 0.80

Classification accuracy on training set: 97.33%

Running time (seconds): 0.24
```

Figure 8: Output of *c*Ant-Miner when executed on the iris data from the UCI Machine Learning repository [11].

present. These will eventually be refactored into a future release. Currently the framework does not have a standard way of organising experiments—e.g., run an algorithm over multiple datasets, perform *n*-fold cross-validation or visualise results of multiple runs. Users are required to provide their own wrapper code/scripts. It would be interesting to incorporate a facility to perform multiple experiments into the framework. Additionally, the framework does not provide any feedback regarding the progress of the execution of an algorithm—this is an important future development. Another important future development is to include MYRA as a package into WEKA.

REFERENCES

[1] C. Blum and K. Socha. 2005. Training feed-forward neural networks with ant colony optimization: An application to pattern classification. In *CD-ROM Proceedings of Hybrid Intelligent Systems Conference (HIS-2005)*.

[2] U. Boryczka and J. Kozak. 2010. Ant Colony Decision Trees – A New Method for Constructing Decision Trees Based on Ant Colony Optimization. In *Proceedings of the ICCCI*. 373–382.

[3] J. Brookhouse and F.E.B. Otero. 2015. Discovering Regression Rules with Ant Colony Optimization. In *Proceedings of the International Conference on Genetic and Evolutionary Computation Companion (GECCO'15 Companion)*. ACM, 1005–1012.

[4] M. Dorigo and T. Stützle. 2004. *Ant Colony Optimization*. MIT Press. 328 pages.

[5] U.M. Fayyad, G. Piatetsky-Shapiro, and P. Smith. 1996. From data mining to knowledge discovery: an overview. In *Advances in Knowledge Discovery & Data Mining*, U.M. Fayyad, G. Piatetsky-Shapiro, P. Smith, and R. Uthurusamy (Eds.). MIT Press, Cambridge, MA, USA, 1–34.

[6] E. Frank, M.A. Hall, and I.H. Witten. 2016. *The WEKA Workbench: Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"* (4th ed.). Morgan Kaufmann.

[7] A.A. Freitas. 2001. Understanding the crucial role of attribute interaction in data mining. *Artificial Intelligence Review* 16, 3 (2001), 177–199.

[8] A.A. Freitas. 2002. *Data Mining and Knowledge Discovery with Evolutionary Algorithms*. Springer-Verlag. 264 pages.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley. 395 pages.

[10] N. Holden and A.A. Freitas. 2008. A hybrid PSO/ACO algorithm for discovering classification rules in data mining. *Journal of Artificial Evolution and Applications (JAEA), special issue on Particle Swarms: The Second Decade* (2008). http://dx.doi.org/10.1155/2008/316145 11 pages.

[11] M. Lichman. 2013. UCI Machine Learning Repository. (2013). http://archive.ics.uci.edu/ml

[12] M. López-Ibáñez, M. Dorigo, and T. Stützle. 2015. Ant Colony Optimization: A Component-Wise Overview. Technical Report TR/IRIDIA/2015-006, IRIDIA, Université Libre de Bruxelles, http://iridia.ulb.ac.be/IridiaTrSeries/link/IridiaTr2015-006.pdf. (2015).

[13] D. Martens, M. De Backer, R. Haesen, J. Vanthienen, M. Snoeck, and B. Baesens. 2007. Classification With Ant Colony Optimization. *IEEE Transactions on Evolutionary Computation* 11, 5 (2007), 651–665.

[14] D. Martens, B. Baesens, and T. Fawcett. 2011. Editorial survey: swarm intelligence for data mining. *Machine Learning* 82, 1 (2011), 1–42.

[15] M. Medland, F.E.B. Otero, and A.A. Freitas. 2012. Improving the *c*Ant-Miner$_{PB}$ Classification Algorithm. In *Swarm Intelligence (Lecture Notes in Computer Science)*, Marco Dorigo, Mauro Birattari, Christian Blum, Anders Lyhne Christensen, Andries P. Engelbrecht, Roderich Groß, and Thomas Stützle (Eds.), Vol. 7461. Springer Berlin Heidelberg, 73–84.

[16] F.E.B. Otero and A.A. Freitas. 2013. Improving the Interpretability of Classification Rules Discovered by an Ant Colony Algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'13)*. ACM Press, 73–80.

[17] F.E.B. Otero and A.A. Freitas. 2016. Improving the Interpretability of Classification Rules Discovered by an Ant Colony Algorithm: Extended Results. *Evolutionary Computation* 24, 3 (2016), 385–409.

[18] F.E.B. Otero, A.A. Freitas, and C.G.Johnson. 2009. A Hierarchical Classification Ant Colony Algorithm for Predicting Gene Ontology Terms. In *Proceedings of the 7th European Conference on Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics (EvoBio 2009), LNCS 5483*. Springer-Verlag, 68–79.

[19] F.E.B. Otero, A.A. Freitas, and C.G. Johnson. 2008. *c*Ant-Miner: an ant colony classification algorithm to cope with continuous attributes. In *Proceedings of the 6th International Conference on Swarm Intelligence (ANTS 2008), Lecture Notes in Computer Science 5217*, M. Dorigo, M. Birattari, C. Blum, M. Clerc, T. Stützle, and A.F.T. Winfield (Eds.). Springer-Verlag, 48–59.

[20] F.E.B. Otero, A.A. Freitas, and C.G. Johnson. 2009. Handling continuous attributes in ant colony classification algorithms. In *Proceedings of the 2009 IEEE Symposium on Computational Intelligence in Data Mining (CIDM 2009)*. IEEE, 225–231.

[21] F.E.B. Otero, A.A. Freitas, and C.G. Johnson. 2010. A Hierarchical Multi-Label Classification Ant Colony Algorithm for Protein Function Prediction. *Memetic Computing* 2, 3 (2010), 165–181.

[22] F.E.B. Otero, A.A. Freitas, and C.G. Johnson. 2012. Inducing decision trees with an ant colony optimization algorithm. *Applied Soft Computing* 12, 11 (2012), 3615–3626.

[23] F.E.B. Otero, A.A. Freitas, and C.G. Johnson. 2013. A New Sequential Covering Strategy for Inducing Classification Rules with Ant Colony Algorithms. *IEEE Transactions on Evolutionary Computation* 17, 1 (2013), 64–74.

[24] R.S. Parpinelli, H.S. Lopes, and A.A. Freitas. 2002. Data Mining with an Ant Colony Optimization Algorithm. *IEEE Transactions on Evolutionary Computation* 6, 4 (2002), 321–332.

[25] K.M. Salama and A.A. Freitas. 2013. Learning Bayesian network classifiers using ant colony optimization. *Swarm Intelligence* 7, 2 (2013), 229–254.