ARM: **Automatic Rule Miner**

Slim Abdennadher, Abdellatif Olama, Noha Salem, and Amira Thabet

Department of Computer Science, German University in Cairo
[slim.abdennadher, abdellatif.olama, noha.salem, amira.thabet]@guc.edu.eg
http://www.cs.guc.edu.eg

Abstract. Rule-based formalisms are ubiquitous in computer science. However, a difficulty that arises frequently when specifying or programming the rules is to determine which effects should be propagated by these rules. In this paper, we present a tool called ARM (Automatic Rule Miner) that generates rules for relations over finite domains.
ARM offers a rich functionality to provide the user with the possibility of specifying the admissible syntactic forms of the rules.
Furthermore, we show that our approach performs well on various examples, e.g. generation of firewall rules or generation of rule-based constraint solvers. Thus, it is suitable for users from different fields.

1 Introduction

Historically, developing rules has been the province of the human experts. Typically, learning the rules in any application domain requires a long apprenticeship. However, when a new knowledge domain immerses then it becomes actually an unaffordable luxury to spend time developing rules in the conventional manner. Therefore, the trend towards generating the rules in an automated manner is rapidly expanding. To introduce the different available techniques it is important to present the general steps of automatic rule generation.
The first step is the *knowledge acquisition*, which is the action of collecting the data and representing it in the appropriate form as input to the second step. This process is indispensable to build either self-learning or expert systems. The knowledge collection is done with the aid of a field expert and involves a lot of computer science irrelevant details, however the representation of the knowledge has to be thought through in order to better serve the data-mining/rule-inference process. The second and most important step, which is the center of this paper is the *knowledge elicitation*, which involves inferring more information than extensionally provided in the knowledge base, i.e. the generation of general rules that are induced from the given data. This procedure is carried out by experts, who employ usually one of three options:

1. Human manual classification, where experts spend a lot of time studying the technical as well as the practical aspects of the knowledge base and work out the clusters manually.

2. Semi-automatic structuring, where the computer scientists are required to build up an entire expert system manually and then use this expert system afterwards in the induction process.
3. Automatic selection and generation, where there exists some kind of a system that carries out the process with no - or minor - human intervention.

The motivation for automatic rule generation lies in the advantages offered thereby. The automated generation process is indispensable if no knowledge engineers exist to mine the data manually in order to acquire the deep knowledge. Automatic generation of rules is needed in the fields where it is important to assess and *validate* experts knowledge in a faster and more reliable manner, especially in applications where the lack of reliability is dangerous.
Last but not least, the knowledge provided in most of the application fields is incomplete and sometimes it is useful to be able to induce the rules automatically each time the knowledge base has to be updated. Such updates, if done manually, highly affect the cost.
As mentioned before, automatic rule generation - sometimes referred to as data mining - is a way of extracting rules directly from data and presenting them in an easily understood format. Some of the intelligent techniques that are used include *Neural Networks*, *Genetic Algorithms*, *Fuzzy Logic* and also *Neurofuzzy Logic* [10].
In this paper, we present a tool that is based on an algorithm previously proposed to generate rule-based constraint solvers [1, 2]. However, the algorithm turned out to be of great use in different fields that provide crucial services to the various sectors in research as well as in industry and medicine. The power of the tool is accentuated by its high *expressivity* and beneficial *flexibility*. It is more expressive than usual automatic rule generation tools implemented in applications based on Artificial Intelligence since the rules inferred from the knowledge base do not propagate only equality constraints. Using ARM it is possible to propagate *all* sorts of constraints provided that the required constraint solvers exist. Thus, it is possible to customize the generated rules according to any type of application (further elaboration in Section 2).
The paper is organized as follows: In Section 2 the representation of the knowledge base as well as the algorithm is briefly described. Three different applications that benefit from the algorithm implemented in ARM are elaborated in Section 3. In Section 4, an explicit and detailed explanation of the tool ARM is presented, where eventually in Section 5, future perspectives and conclusions are discussed.

2 Algorithm

Knowledge is classified into *facts*, statements that are always true, and *rules*, more complicated and more general statements. Facts are considered to be extensional definitions of some sort of relations. Rules, which denote the intensional definition of the relation, are conditionable; in the sense that they are customized to condition-action situations, like expert and prediction systems.

There are two main different types of rules that could be generated given a specific knowledge base: *propagation rules* and *simplification rules*. Simplification rules, as the name suggests, simplify the knowledge base by removing one or more facts and replacing them by other simpler ones, whereas propagation rules induce a process of deriving new facts from given ones and adding them to the existing knowledge.
Our algorithm generates first only propagation rules. Often, some propagation rules can be transformed into simplification rules. Thus, in a second step a post-processing approach based on a confluence test is performed [2]. The algorithm for generating propagation rules has been developed based on previous work done in the field of *knowledge discovery*.
The algorithm of the tool at hand allows the user to define the form of the rules to be generated. As mentioned before, this tool accepts any type of constraints on both sides of the rule. Technically a rule consists of two parts, called the left-hand side (LHS) and the right-hand side (RHS). The antecedent is written first and called the *head* of the rule, the consequent is called the *body*.
Simplification rules are rules of the form $LHS \Leftrightarrow RHS$ and propagation rules are rules of the form $LHS \Rightarrow RHS$, where LHS and RHS are sets of constraints. Using ARM the user has the possibility to specify the admissible syntactic forms of the rules. The user determines the relation for which rules have to be generated, i.e. the LHS, and chooses the candidate constraints to form conjunctions together with the left hand side. Usually, these candidate constraints are simply equality constraints. For the right hand side of the rules the user specifies also the form of candidate constraints she/he wants to see there. Finally, the user determines the semantics of the constraint on the LHS by means of its extensional definition which must be finite, and provides the semantics of the candidate constraints and the candidate RHS by two constraint theories. Furthermore, it is assumed that the constraints defined are handled by an appropriate constraint solver.
To compute the rules the algorithm enumerates each possible LHS constraint (noted C_{lhs}) and for each determines the corresponding RHS constraint (noted C_{rhs}).
For each LHS C_{lhs} the corresponding RHS C_{rhs} is computed in the following way:

1. if C_{lhs} has no solution then $C_{rhs} = \{false\}$ and we have the failure rule $C_{lhs} \Rightarrow \{false\}$.
2. if C_{lhs} has at least one solution then C_{rhs} is the set of all atomic constraints which are candidates for the RHS part and that are true for all solution of C_{lhs}. If C_{rhs} is not empty we have the rule $C_{lhs} \Rightarrow C_{rhs}$.

The algorithm uses pruning strategies to reduce the number of rules generated. This way it becomes much more efficient if during the enumeration of all possible rule LHS, a given LHS is considered before any of its supersets. So a specific ordering for this enumeration is imposed in the algorithm. Moreover, this ordering allows to discover early covering rules avoiding then the generation of many uninteresting covered rules.

3 Application

Many applications require data-mining and rule generation. In this section, three applications from three different domains are presented to endorse the generality of the proposed tool for automated generation of rules.

3.1 Firewall design

The function of a firewall is to examine each packet that attempts to enter a private network and decide whether to accept the packet and allow it to proceed or to discard it. A typical firewall design consists of a sequence of rules. To make a decision concerning some packets, the firewall rules are compared, one by one, with the packet until one rule is found to be satisfied by the packet: this rule determines the fate of the packet. The first method ever for designing the sequence of rules in a firewall to be consistent, complete, and compact using Firewall Decision Diagram was proposed by [7]. Using ARM it is possible to use the extensional definitions of a packet to generate the firewall rules. There are usually five primary fields that describe the packet in any firewall and are used for deciding the course of a packet: `discard` or `accept`. The following is an excerpt of the knowledge base of a firewall with simplified representation of a network packet `pack(F0,F1,A)` with only two parameter fields, `F0` and `F1` together with the action `A` to be taken upon the arrival of this packet, where `a` stands for accept and `d` stands for discard:

```
pack(4, 2, a).   pack(4, 3, a).   pack(4, 5, a).   pack(4, 6, a).
pack(4, 7, a).   pack(4, 0, d).   pack(4, 1, d).   pack(4, 4, d).
pack(4, 8, d).   pack(4 ,9, d).   pack(5, 2, a).   pack(5, 3, a).
pack(5, 5, a).   pack(5, 6, a).   pack(5, 7, a).   pack(5, 0, d).
pack(5, 1, d).   pack(5, 4, d).   pack(5, 8, d).   pack(5, 9, d).
pack(6, 2, a).   pack(6, 3, a).   pack(6, 5, a).   pack(6, 6, a).
pack(6, 7, a).   pack(6, 0, d).   pack(6, 1, d).   pack(6, 4, d).
pack(6, 8, d).   pack(6, 9, d).   pack(7, 2, a).   pack(7, 3, a).
pack(7, 5, a).   pack(7, 6, a).   pack(7, 7, a).   pack(7, 0, d).
pack(7, 1, d).   pack(7, 4, d).   pack(7, 8, d).   pack(7, 9, d).
pack(0, 0, d).   pack(0, 1, d).   pack(0, 2, d).   pack(0, 3, d).
pack(0, 4, d).   pack(0, 5, d).   pack(0, 6, d).   pack(0, 7, d).
pack(0, 8, d).   pack(0, 9, d).   pack(1, 0, d).   pack(1, 1, d).
pack(1, 2, d).   pack(1, 3, d).   pack(1, 4, d).   pack(1, 5, d).
pack(1, 6, d).   pack(1, 7, d).   pack(1, 8, d).   pack(1, 9, d).
pack(2, 0, d).   pack(2, 1, d).   pack(2, 2, d).   pack(2, 3, d).
pack(2, 4, d).   pack(2, 5, d).   pack(2, 6, d).   pack(2, 7, d).
pack(2, 8, d).   pack(2, 9, d).   pack(3, 0, d).   pack(3, 1, d).
pack(3, 2, d).   pack(3, 3, d).   pack(3, 4, d).   pack(3, 5, d).
pack(3, 6, d).   pack(3, 7, d).   pack(3, 8, d).   pack(3, 9, d).
pack(8, 0, d).   pack(8, 1, d).   pack(8, 2, d).   pack(8, 3, d).
pack(8, 4, d).   pack(8, 5, d).   pack(8, 6, d).   pack(8, 7, d).
pack(8, 8, d).   pack(8, 9, d).   pack(9, 0, d).   pack(9, 1, d).
pack(9, 2, d).   pack(9, 3, d).   pack(9, 4, d).   pack(9, 5, d).
pack(9, 6, d).   pack(9, 7, d).   pack(9, 8, d).   pack(9, 9, d).
```

For this knowledge base of the firewall, ARM will automatically generate among others the following propagation rule provided the user specifies that the right hand side of the rules may consist of a conjunction of equality constraints.

```
pack(F0,9,A) ⇒ A=d.
pack(F0,8,A) ⇒  A=d
...
pack(7,7,A)⇒ A=a.
pack(7,6,A)⇒ A=a.
...
```

The first rule means that for any values for `F0`, if the second field has the value `9`, then the packet should be discarded.
The ARM tool can generate a more compact representation of the rules, if the user specifies to have membership constraints in the left hand side of the rules:

```
pack(F0,F1,A), F1 in[0,1,4,8,9]⇒ A=d.
pack(F0,F1,A), F0 in[0,1,2,3,8,9] ⇒ A=d.
pack(F0,F1,A), F0 in [4,5,6,7], F1 in [2,3,5,6,7] ⇒ A=a.
```

3.2 Generation of Constraint Solvers

Originally the algorithm presented in Section 2 was introduced to generate rule-based constraint solvers for finite constraints given their extensional representation [1]. The generated rules can be executed using the Constraint Handling Rules framework [5, 6].
For example, for the logical operator *and* that can be defined extensionally by the triples $\{(0,0,0),(0,1,0),(1,0,0),(1,1,1)\}$ and for the logical operator *neg* that can be defined by the pairs $\{(0,1),(1,0)\}$, where 1 stands for truth and 0 for falsity, the algorithm can generate, among other, the following rules:

$$\begin{aligned} and(0,Y,Z) &\Leftrightarrow Z{=}0. \\ and(1,Y,Z) &\Leftrightarrow Y{=}Z. \\ and(X,X,Z) &\Leftrightarrow X{=}Z. \\ neg(X,0) &\Leftrightarrow X{=}1. \\ neg(X,X) &\Rightarrow false. \\ and(X,Y,Z),\ neg(X,Y) &\Leftrightarrow Z{=}0,\ neg(X,Y). \end{aligned}$$

The algorithm performs well on various examples, including Boolean constraints, multi-valued logic, Allen's qualitative approach to temporal logic and qualitative spatial reasoning with the Region Connection Calculus [1].

4 Arm Features

4.1 How to run ARM

ARM can be run as an application. The application version of ARM requires the installation of SICStus Prolog version 3.8.6 or later as well as the Java to Prolog

interface package Jasper (se.sics.jasper). If ARM was started successfully, the start screen shown in Figure 1 appears. On the start screen one has to choose either of the two available options which will determine how ARM will specify the domain for each of the parameters of a relation. Choosing the `N-to-1` option will result in setting a single domain which is applied to each of the parameters of the relation based on the values that were used in the specified tuples. The `N-to-N` option sets a separate domain for each parameter of the relation. This feature will suppress the generation of a huge number of failure rules. For example, for the firewall design example, if the user will choose the `N-to-1` option, then rules like

```
pack(F0,F1,4)  ⇒  false.
```

will be generated, although it is clear from the begining that the third argument of the predicate `pack` cannot take the value `4`. Thus for the firewall design example, the user should specify the values of each parameter using the `N-to-N` option. For the constraint solving example, all arguments have the domain $\{0, 1\}$, thus the user should choose the `N-to-1` option.

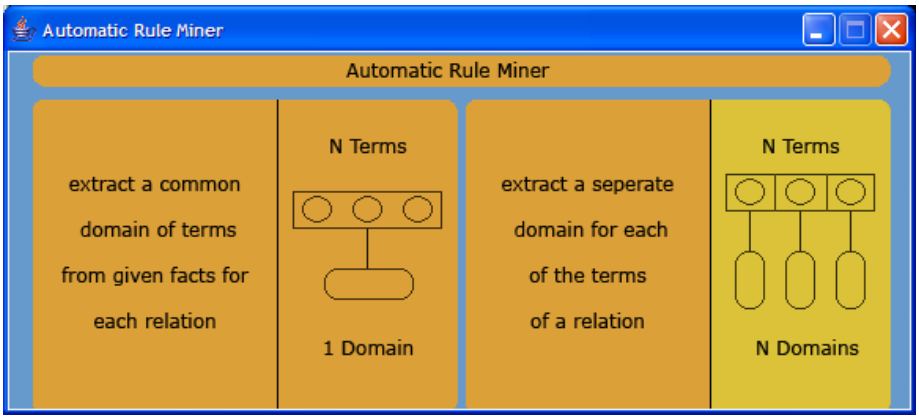

Fig. 1. Start-up Window

After choosing either option by clicking the appropriate button, the main view of ARM will appear as shown in Figure 2.
From the `relations` menu one can add rules to the `relations list`. By clicking on the `add` button a pop-up window will appear where the name of the relation can be entered. By clicking the name of a relation from the `relations list` the tuples associated with the selected relation will be displayed in the `tuples list`, part of the `tuples menu`. Tuples associated with a relation can be added or removed using the `add` and `remove` buttons. By clicking the `add` button of the

`tuples menu` a pop-up window containing a text field will appear where a space-separated list of values should be entered to represent a tuple to be associated with the highlighted relation of the `relations menu`.

Each of the two drop-down lists at the bottom of the main view window contains available constraints that could be added to the list of constraints to be added to the right-hand-side and left-hand-side of the generated rules. Choosing a constraint from the drop-down list and clicking the `add` button will add the constraint to the specified side of the rules.

After finalizing the selection of relations, associated tuples and constraints, the `generate rules` button should be clicked to display the result of generating the rules according to the specified input.

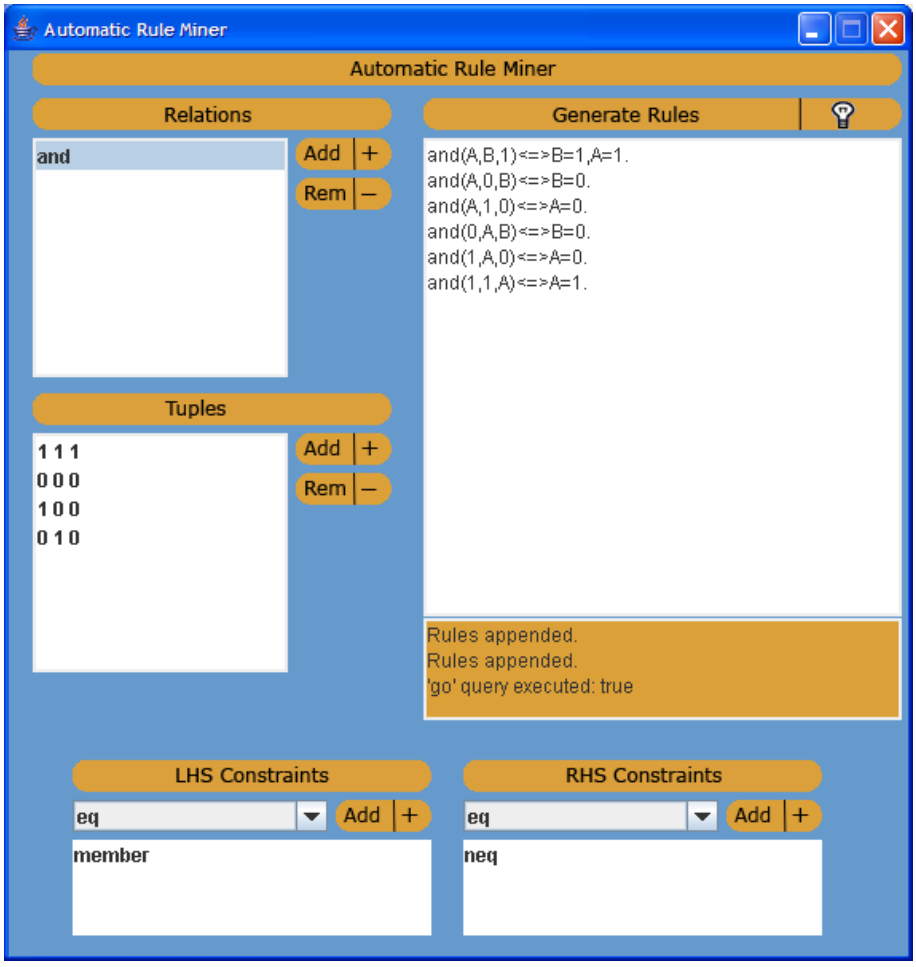

Fig. 2. ARM's Main View

4.2 Implementation of ARM

ARM's graphical user interface is implemented in Java using the Java Software Development Kit (J2SDK) version 1. The underlying generation of rules is implemented partly in Java, however the main part is implemented in SICStus Prolog.
As described in Section 2, ARM needs to enumerate LHS constraints. Our implementation follows the idea of direct extraction of association rules by exploring a tree corresponding to the LHS search space. This tree is expanded and explored using a depth first strategy, in a way that constructs only necessary LHS candidates and allows to remove uninteresting candidates by cutting whole branches of the tree. The branches of the tree are developed using a partial ordering on the LHS candidates such that the more general LHS are examined before more specialized ones. The partial ordering used in our implementation is the θ-subsumption [9] ordering commonly used in ILP to structure the search space (e.g., the WARMR algorithm [4] to mine frequent DATALOG queries). To prune branches in the tree, one of the two main criteria has been inspired by the CLOSE algorithm [8] devoted to the extraction of frequent *itemsets* in dense[1] data sets.
The interaction between Java and Prolog is provided using the bidirectional interface Jasper. Jasper is used as a Java package (se.sics.jasper) for the purposes of ARM since the interaction is needed only in one direction, more specifically, the Java graphical user interface will query the SICStus Prolog and obtain a result which will be displayed by Java again.
SICStus Prolog performs the role of a base layer for communication between the knowledge base on one side and the inference engine and constraint solver on the other side. As a rule-based programming language, Prolog helps in simplifying this task. SICStus provides several choices for developing user interfaces, however Java stands out among other alternatives like C and Tcl/Tk especially because of the portability issue which is overcome by default when using Java.
Through the Java-Prolog interaction, the user of ARM will be able to generate rules. The rules will be generated based on the facts provided by the user. The construction is done through the graphical interface which will trigger the formation of an underlying knowledge base of relations together with associated tuples. The knowledge base will then be formatted by Java, the appropriate query will be generated. The result is submitted through Jasper to the Prolog interpreter which will in turn respond with the corresponding list of rules.

5 Conclusion

ARM is a tool for generating rules from relational data. We have shown that this tool can be used in different application fields: generation of firewall rules and rule-based constraint solvers.
The tool can be run as an application under `http://cs.guc.edu.eg/arm`

[1] e.g., data sets containing many strong correlations.

Future work includes the extension of the tool to generate rules for relations defined intensionally eventually over non finite domains. A first preliminary step in this direction has recently been proposed in [3].

References

1. S. Abdennadher and C. Rigotti. Automatic generation of propagation rules for finite domains. In *6th International Conference on Principles and Practice of Constraint Programming, CP'00*, LNCS 1894. Springer-Verlag, 2000.
2. S. Abdennadher and C. Rigotti. Automatic generation of rule-based constraint solvers over finite domains. *ACM Transactions on Computational Logic*, 5(2), 2004.
3. S. Abdennadher and C. Rigotti. Automatic generation of CHR constraint solvers. *Journal of Theory and Practice of Logic Programming (TPLP)*, 5(2), 2005.
4. Luc Dehaspe and Hannu Toivonen. Discovery of frequent DATALOG patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.
5. T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1-3), October 1998.
6. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer-Verlag, 2003.
7. X. A. Liu M. G. Gouda. Firewall design: Consistency, completeness, and compactness. In *24th IEEE International Conference on Distributed Computing Systems*, 2004.
8. Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Efficient mining of association rules using closed itemset lattices. *Information Systems*, 24(1):25–46, 1999.
9. Gordon Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.
10. P. Smyth U. Fayyad, G. Piatestsky. From data mining to knowledge discovery in databases. *American Association for Artificial Intelligence*, 1996.