CRD:A New Data Mining Method In Deductive Databases *

Yunming Wang Chun Tao Yonggang Zhao
Yang Yang

Computer Science Department of Fudan University
Shanghai 200433, P.R.China
Email: ymwang@fudan.edu.cn

Abstract

Data mining, also known as knowledge discovery in databases, has been popularly recognized as an important research issue with broad applications. Many kinds of data mining methods have been developed in previous studies. However, in order to serve the representation and implementation of deductive databases and intelligent query answering, there have been increasingly pressing calls on more general-purpose data mining techniques. In this paper, we present a new data mining method called CRD(Common Rule Discovery). It is a general-purpose method that is able to extract a wide spectrum of rules, including recursive ones. We also introduce abstract constants, user-defined predicates, built-in predicates in the extracted rules, which greatly enhance the flexibility and representation power for knowledge. CRD is applicable not only in data mining area, but also in inductive logic programming. This method has been implemented in our CRDS with satisfactory results. Some experiments using CRDS are also provided.

Keywords Data mining, Knowledge Discovery in Databases, Deductive Databases, Inductive Logic Programming, CRD, CRDS

1 Introduction

A *deductive database* is a database which consists of data, rules, integrity constraints and a mechanism of deductive reasoning. Roughly speaking, it uses rules to represent (or store) intensional data, and performs deductive reasoning from rules to data when necessary. On the contrary, *Inductive Logic Programming (ILP)* focuses on constructing a logical (intensional) definition of a relation from positive and negative examples. That is, it performs induction from data to rules. They are inverse but closely related. The logic programming school in deductive databases argues that deductive databases can be effectively represented and implemented using logic and logic programming.

Another closely related notion is *Data Mining*, which means a process of nontrivial extraction of implicit, previously unknown and potentially useful information from data[2]. Since most of the extracted knowledge can be expressed in the form of rules, we can also look on data mining as extracting rules from data. Though ILP and data mining lay different emphases and possess different methods, they have many subtle relationships, and their combination is taken into consideration. [3] states an idea of transforming the ILP problem to

*Research is partially supported by Natural Science Foundation of China and 863 National High-Tech Program.

attribute value form, and generating rules by data mining method. As the latter handles noisy data successfully, it seems to be more practical and exhibits many advantages.

However, gap still lies between the rules extracted by ILP or data mining and those in deductive databases. For example, it is obvious that rules in deductive databases can be recursive, but few systems can now extract recursive rules perfectly. [12] provides a method which can generate certain kind of recursive rules, however, the form of extracted rules is also strongly restricted.

Moreover, the data mining methods previously provided usually only fit certain area and generate certain kind of knowledge. In order to discover various kinds of knowledge, we have to develop one algorithm (or subsystem) for each kind of knowledge. Simple integration of these subsystems will lead to too large a system and too high expense to maintain. With the rising of intelligent query answering [10] [8], there are increasingly pressing calls on general-purpose data mining methods that are able to extract a wide spectrum of rules so as to serve the representation and implementation of deductive databases and intelligent query answering.

In this paper, we present a novel data mining method called CRD(Common Rule Discovery), which is able to extract any kind of horn rules from data, including recursive ones. It is a general-purpose method, using a meta-rule to guide the extraction. Users can provide user-defined predicates (act as IDB predicates in the deductive databases), built-in predicates and concept hierarchies in order to make the method fit their special application area better.

This method has been implemented in our CRDS with satisfactory results. CRDS discovers Common Rules in relational or deductive databases, answers advanced queries from users, and maintains the discovered knowledge. In this paper, we also provide some experimental results of CRDS to show some features of CRD method in section 6.

2 Problem description

2.1 Predicates

There are three kinds of predicates which may concern in this paper. Suppose a relation schema of the form:

$$relation\_name(Attr_1 : dom_1, Attr_2 : dom_2, ..., Attr_n : dom_n),$$

where $Attr_i$ is a field of the relation, and dom_i is its corresponding domain. We can also look on the schema as a predicate, whose corresponding facts are the tuples of the relation. Such predicates are called *schema-predicates*, they act as the EDB predicates in deductive databases.

Users can also define their own predicates from such schema-predicates. For example, starting from the schema-predicate(where the domains are omitted):

$$student(Name, Sno, Status, Major, Gpa, Birth\_data, Birth\_place),$$

we can define following predicates:

$$student(Name, S, Status, M, Gpa, Birth\_date, Birth\_place) \rightarrow major(S, M) \qquad (1)$$

$$student(Name, S, Status, Major, Gpa, Birth\_date, Birth\_place) \wedge$$

$$Status = graduate \wedge Gpa > 3.5 \rightarrow fellowship\_candidate(S) \quad (2)$$

$$student(Name, S, Status, Major, Gpa, Birth\_date, Birth\_place) \wedge$$
$$Status = undergraduate \wedge Gpa > 3.2 \rightarrow fellowship\_candidate(S) \quad (3)$$

Such predicates are referred to as *user-defined predicates*. Actually, users can define various predicates to meet their needs. These user-defined predicates act as IDB predicates in deductive databases.

The third kind of predicates is *built-in predicates*, such as logic operations($>$, $\geq$, etc.) and arithmetic operations (add, subtract, multiply, divide, square, etc.). Their corresponding relations are infinite, hence we have to handle them in a different way in section 4.

Notice that each variable in a predicate has a domain. Domain is not the data type of the field. Different domains may have the same data type. For example, both $Name$ and $Major$ may have the data type char(20), but their domains are different. We decide whether two variables are unifiable by their domains.

2.2 Meta-rule

In order to specify the pattern of the rules a user want to extract, avoid generating uninteresting rules, and improve the efficiency, we can use a *meta-rule* to guide extraction. The system only tries to find rules that comply with the meta-rule. Mete-rule is constructed according to the application environment and the knowledge users want to discover. For examples, if users want to obtain the definition of *son* from predicates *parent* and *male*, they can use meta-rule (4); If users want to obtain the definition of *path* from two (or one) unknown predicates, they can use meta-rule (5); If users want to know the percentage that fellowship candidates account for in each major, they can user meta-rule (6).

$$parent(\_, \_) \wedge male(\_) \rightarrow son(X, Y) \quad (4)$$

$$P \wedge Q \rightarrow path(X, Y) \quad (5)$$

$$major(S, M) \rightarrow fellowship\_candidate(S) \quad (6)$$

In (4), all predicates have been specified. We only need to find proper variables and/or constants to fill in the blanks in the meta-rule to generate target rules. However, the predicates in the body of (5) are not specified. So we have to find appropriate predicates with appropriate arguments in them to fit the meta-predicates P and Q. Notice that we can instantiate a meta-predicate in body to null, i.e. $arc(X, Y) \rightarrow path(X, Y)$ also complies with (5).

We can also specify some of the variables be identical in the meta-rule as (6) shows. How do the results of meta-rule (6) answer the third query will be be made clear in the next section.

2.3 Support and confidence

In order to handle noisy data and get some probabilistic results, *support* and *confidence* are introduced. Suppose a rule :

$$p_1(\overrightarrow{v_1}) \wedge p_2(\overrightarrow{v_2}) \wedge \cdots \wedge p_n(\overrightarrow{v_n}) \rightarrow q(\overrightarrow{u}) \quad (7)$$

Denote the relation defined by the head with H. Notice that H is not necessarily the whole relation for predicate q (denoted with Q). Generally speaking, H is a selection from Q.

The body of the rule also defines a relation with respect to the head, which is denoted with B. Let $\overrightarrow{v} = \cup \overrightarrow{v_i}$. When (7) is safe, we have $\overrightarrow{u} \subseteq \overrightarrow{v}$. It is easy to see that $B = \prod_{\overrightarrow{u}} (\bowtie P_i)$. When it is not safe, $\bowtie P_i$ does not define enough imformation for the head, hence we can join it with Q. As a result, $B = \prod_{\overrightarrow{u}} (\bowtie P_i \bowtie Q)$.

Let $G = B \cap H$, denote the number of tuples in B with $|B|$, and $|H|$, $|G|$ as well. The *support* of the rule (7) is $|G|/|H|$, and the *confidence* is $|G|/|B|$.

Roughly speaking, confidence is the validity of the rule, and support is the weightiness of the rule leads to the head, or, the importance of the reason leads to the result. For example, suppose we get a result rule for meta-rule (6) as (8),

$$major(S, software) \rightarrow fellowship\_candidate(S)\ (51\%, 43\%) \tag{8}$$

The confidence(51%) means that software students are fellowship candidates with possibility 51%, that is, 51% of the students majoring in software are fellowship candidates. The support(43%) means that 43% of the fellowship candidates can be derived by this rule, i.e., 43% fellowship candidates major in software.

Note that the support defined here is different from that in [4], [5], and [1]. This support not only filters out uninteresting rules, but also has clear semantics as confidence does. Thus, each rule discovered by CRD has its 'dual semantics'.

Rules with small values of confidence and support are usually uninteresting. Hence users can provide two thresholds. CRD only tries to discover rules whose confidence and support are no less than corresponding thresholds.

2.4 Problem description

Given some relations, predicates, a support threshold, a confidence threshold, and a meta-rule, our task is to discover the rules that comply with the meta-rule, and whose confidence and support are no less than the confidence threshold and support threshold respectively.

3 Basic ideas of CRD

In this section, we only introduce the basic ideas of CRD. The process of built-in predicates and variable specializations are discussed in section 4 and section 5.

3.1 Primitive rule

CRD algorithm begins with generating a *primitive-rule* according to meta-rule. First, each meta-predicate is substituted with the conjunction of all predicates that comply with it, leaving all the variables blank. Then distinct variables are allotted for these blanks. For example, primitive rule for meta-rule(4) is generated as follows by allotting distinct variables:

$$parent(V_1, V_2) \wedge male(V_3) \rightarrow son(X, Y) \tag{9}$$

Notice that a meta-predicate is substituted with the conjunction of several predicates. For example, we get primitive-rule (10) if candidate predicates for P and Q in (5) can be *arc*

and *path*:

$$arc(V_1, V_2) \wedge path(V_3, V_4) \wedge arc(V_5, V_6) \wedge path(V_7, V_8) \rightarrow path(X, Y) \tag{10}$$

where P is substituted with $arc(V_1, V_2) \wedge path(V_3, V_4)$. However, at most one of $arc(V_1, V_2)$ and $path(V_3, V_4)$ will appear in the result rules. This will be clarified by following sections.

We take the idea of evolving the primitive rule gradually to target rules by

1. eliminating superfluous predicates.
2. unifying some of the variables in a proper way.
3. instantiating appropriate variables to decent constants.

1 and 2 are solved in this section, and 3 is solved in section 5.

3.2 Equivalent class and partition

As each variable is distinct in the primitive rule, we have to unify some of the variables to generate target rules. For example, when unify Y with V_1, X with V_2 and V_3 in rule (9), we get (11):

$$parent(Y, X) \wedge male(X) \rightarrow son(X, Y) \tag{11}$$

Two variables are defined to have *equivalent relation* if they are to be unified. All the variables in the primitive rule are then divided into several *equivalent classes*. The unification of generating (11) correspond to *partition* $\{\{Y, V_1\}, \{X, V_2, V_3\}\}$.

After unifying the variables in the same class, there may be some superfluous predicates in the primitive-rule. For example, we get (13) by unifying variables in (10) according to partition (12)

$$\{\{X, V_1\}, \{Y, V_8\}, \{V_2, V_7\}, \{V_3\}, \{V_4\}, \{V_5\}, \{V_6\}\} \tag{12}$$

$$arc(X, V_2) \wedge path(V_3, V_4) \wedge arc(V_5, V_6) \wedge path(V_2, Y) \rightarrow path(X, Y) \tag{13}$$

We can easily see that it is the same as (14)

$$arc(X, V_2) \wedge path(V_2, Y) \rightarrow path(X, Y) \tag{14}$$

Hence we maintain a set of *relevant* variables for each partition. Only relevant variables contribute to generating rules. Initially, all the variables are irrelevant. Some irrelevant variables are made relevant when necessary in the next section. We also call a predicate *relevant* if it contains at least one relevant variable.

As stated above, we can define the *application of a partition to primitive rule* as: unify the variables in the same class, and drop predicate(s) that is not relevant. For example, we get (14) by applying (12) to (10) if relevant variables are $\{X, Y, V_1, V_2, V_7, V_8\}$.

The number of possible partitions is large sometimes if there are no adequate constraints. Fortunately, we have several constraints and an elegant method to construct proper partitions. The constructing method is discussed in section 3.6. Here the constraints are discussed.

Constraint 3.1 *All the variables in a class must share the same domain.*

We can see that partition $\{\{X, V_1\}, \{Y, V_2\}\}$ and $\{\{X, V_5\}, \{Y, V_6\}\}$ will lead to the same result when applied to (10). Hence we need only to generate one of them.

Suppose there are two predicates in the primitive rule $p(\overrightarrow{v})$ and $p(\overrightarrow{u})$, all the variables in $\overrightarrow{v}$ and $\overrightarrow{u}$ are free. A *commutation* of a partition is to substitute all the variables in $\overrightarrow{v}$ with those in $\overrightarrow{u}$, and those in $\overrightarrow{u}$ with in $\overrightarrow{v}$. A partition is *symmetrical* to another if it can reach the other through commutation(s). Thus we have :

Constraint 3.2 *Generating one partition prevents generating any of its symmetrical ones.*

We can also see that a partition with relevant variables $\{X, Y, V_1, V_2, V_3, V_4, V_5\}$ is invalid for (10) , because the meta-predicate P corresponds to two relevant predicates. Hence we have:

Constraint 3.3 *Each meta-predicate corresponds to at most one relevant predicate.*

These three constraints are combined in the constructing process elegantly in CRD with very low cost.

3.3 Making a variable relevant

When a variable V is made relevant, all the variables in the predicate to which V belongs are also made relevant. At the same time, some of the other variables may be deleted from the partition. For example, when we make the variable V_1 relevant in the primitive rule (10), we make V_2 relevant as well. What's more, we can conclude that variable V_3 and V_4 can be deleted, for they will never be made relevant any more(due to constraint 3.3).

3.4 Primitive partition

There may be some variables that are identical in the meta-rule as (6) manifests. However, the variables in the primitive rule are distinct. In order to preserving the homology of variables in the meta-rule, we make a *primitive partition* according to the meta-rule and the primitive rule. All the partitions we concern afterwards are based on the primitive partition. For example, the primitive rule and primitive partition for meta-rule(15) is (16) and (17) respectively.

$$P(\_, A) \wedge Q(A, \_) \rightarrow path(X, Y) \tag{15}$$

$$arc(V_1, V_2) \wedge path(V_3, V_4) \wedge arc(V_5, V_6) \wedge path(V_7, V_8) \rightarrow path(X, Y) \tag{16}$$

$$\{\{V_1\}, \{V_2, V_4, V_5, V_7\}, \{V_3\}, \{V_6\}, \{V_8\}, \{X\}, \{Y\}\} \tag{17}$$

3.5 Original partitions

There are three kinds of equivalent relations, that is, relation between a head variable and a body variable, between head variables, and between body variables. An *original partition* is a partition without the third kind of equivalent relations. In our method, we first construct original partitions of the primitive rule, and then construct new partitions according to existing ones recursively. All possible equivalent relations of the first and the second kind are taken into accounts when constructing original partitions. Hence we will only consider the third kind of equivalent relations in the next section. The corresponding body variables and head variables that are unified in the original partition are made relevant. For example, the

original partition $\{\{V_1, X\}, \{V_2\}, \{V_3\}, \{V_4\}, \{V_5\}, \{V_6\}, \{V_7\}, \{V_8, Y\}\}$ of primitive rule (10) makes $\{V_1, V_8, X, Y\}$ relevant. As a result, $\{V_2, V_7\}$ are also made relevant and $\{V_3, V_4, V_5, V_6\}$ are deleted (Hence it becomes $\{\{V_1, X\}, \{V_2\}, \{V_7\}, \{V_8, Y\}\}$) .

3.6 Constructing partitions

We construct further partitions from existing ones recursively by merge operation. A *merge operation* is to make two variables V_1, V_2 equivalent, where V_1, V_2 satisfy:

- They share the same domain but they are not equivalent
- Both V_1 and V_2 are body variables
- V_1 is a relevant variable

Merge operation also makes V_2 relevant after making V_1 and V_2 equivalent. For example, the original partition $\{\{V_1, X\}, \{V_2\}, \{V_7\}, \{V_8, Y\}\}$ will lead to $\{\{V_1, X, V_2\}, \{V_7\}, \{V_8, Y\}\}$, $\{\{V_1, X\}, \{V_2, V_7\}, \{V_8, Y\}\}$, etc..

Theorem 3.1 *Suppose we get P_1 by exerting a merge operation on P_2, we get R_1 (or R_2) by applying P_1 (or P_2) to the primitive rule, and the support of R_1 (or R_2) is S_1 (or S_2), then $S_1 \leq S_2$. (Proof omitted)*

As mentioned above, we get a set of candidate partitions, denoted with SCP. They are pruned and checked one by one in next sections.

3.7 Pruning partitions

Suppose there are two partitions P_1 and P_2, for each equivalent class C in P_1, there exists a class C' in P_2, such that $C \subseteq C'$, then we say P_2 *contains* P_1, denoted with $P_1 \subseteq P_2$.

In order to eliminate some improper partitions, we maintain a set of *forbidden partitions*, denoted with SFP. When we have constructed a partition, we check whether it contains any of the partition(s) in SFP, if it does, it can be pruned immediately. The original SFP of a primitive rule consists of partitions that will render some body predicates the same as the head after unification. For example, original SFP for (10) is $\{\{\{X, V_3\}, \{Y, V_4\}\}, \{\{X, V_7\}, \{Y, V_8\}\}\}$. We will also add some partitions to SFP when checking them in the next section.

In order to avoid generating successive symmetrical partitions, we also maintain a set of symmetrical partitions, denoted with SSP. Put all of its symmetrical partition into SSP when one candidate partition is generated. When the lately generated partition contains any of the partition(s) in SSP, it is also pruned. The difference between SFP and SSP is that partitions in SFP are never deleted, but SSP is emptied at the beginning of each pass of construction.

Actually, constructing and pruning partitions can be combined into one process. We discuss them separately in order to clarify the process.

3.8 Checking candidate partitions

For each candidate partition generated as previously stated, we will get a rule by applying it to the primitive rule. We check these rules one by one according to their supports and confidences as follows.

1. If neither confidence nor support is less than its threshold, this is just a rule we want to discover, so we output it. As merge operation to a partition will only make the rule more strict, we move the partition from SCP to SFP in order to avoid generating uninteresting partitions in subsequent constructing.

2. If the support is no less than the threshold, and the confidence is less than its threshold, it means that the partition should be merged to improve its confidence, we do nothing here, leaving this partition in SCP as a base for further construction.

3. For those rules whose supports are less than their thresholds, we can assert that their corresponding partitions need not to participate in further construction of SCP (Theorem 3.1). Hence we have to move the corresponding partitions from SCP to SFP.

3.9 CRD Algorithm

As a conclusion, we describe the algorithm as follows:

```
generate primitive rule according to the meta-rule
generate primitive partition according to the meta-rule and primitive rule
generate original SFP
generate original SCP and prune it
repeat
         empty SSP
         check each partition in SCP
         construct new SCP from old SCP by merge operation and prune new SCP
until new SCP is empty
```

3.10 Performance analysis

At a first glance, an intuition may emerge that this algorithm has a considerable search space. However, with the three constraints, some other optimization arguments, and the recursive constructing mechanism, CRD actually checks only a very small fraction of the possible space. As an example, let's consider the primitive rule (10) for meta-rule (5). Any two variables can be unified because they share the same domain. Hence the number of all possible partitions is 115975. But CRD never checks more than 35 of them without any loss of possible rule (It varies according to the thresholds and data). Which is about 0.03% of total space. For other experiments with many different domains in the dataset, this is still much smaller. Therefore, we can safely suppose that the number of partitions to be checked is a constant compared with the large dataset.

Since other operations are executed in memory, the major cost in this algorithm is the computation of support and confidence when checking the SCP. In our recursive constructing mechanism, this can be performed within at most one join and two selection operations. Then it is easy to see that the overall cost of this algorithm is $O(Nlog_2N)$, where N is the maximum size of the relations.

4 Process of built-in predicates

Built-in predicates have no (or have infinite) relations. Hence they need particular process in the algorithm.

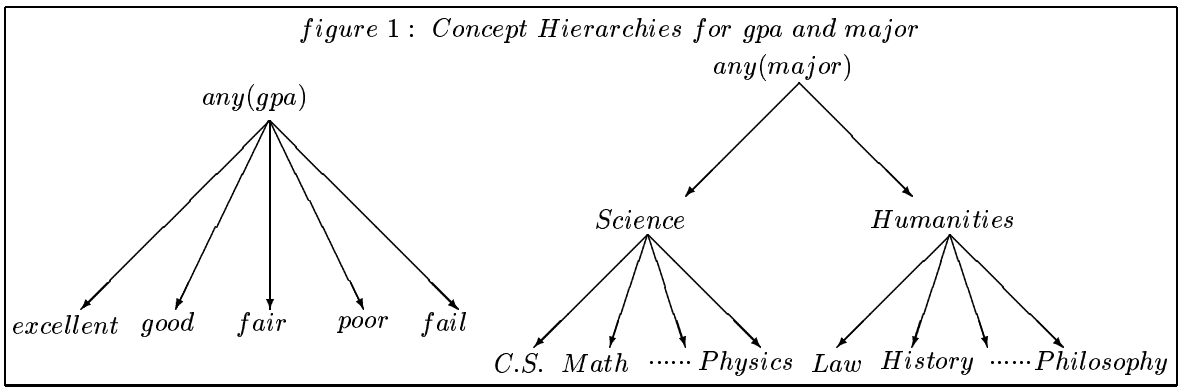
We divide the variables in a built-in predicate into two kinds: *operand variables* and *result variables*. All the variables in *logic predicates* such as $>$, $\geq$ are *operand variables*. But one of the variables in the *arithmetic predicate* is a *result variable*, whose meaning is self-evident. For example, the predicate $add(V_1, V_2, V_3)$ means $V_1 + V_2 = V_3$, hence V_3 is the result variable and V_1, V_2 are operand variables.

We say an operand variable is *available* if it is unified to a variable not in a built-in predicate or an available operand variable or a calculated result variable. The scopes of available variables are bounded. A built-in predicate is calculated only when all of the operand variables become available.

5 Constants in rule

5.1 Concept hierarchies

We may also instantiate some of the variables to generate some more rules. In [5] and [6] concept hierarchies is presented, which represent the relationships among the concepts of domains at different levels. For example, for the domains of the attribute *gpa* and *major*, we can construct concept hierarchies as *figure* 1 shows. Concept hierarchies can be provided by knowledge engineers or domain experts, or be discovered (semi-)automatically using knowledge discovery tools.



Given some concept hierarchies, some of the variables can be instantiated to abstract constants. For example, the second variable of $major(V_1, science)$ in (18) is intantiated to *science* , which means V_1 is a *science* student, but no more information is given whether V_1 is a student majoring in C.S. or Math. Rules at multiple concept levels may lead to the discovery of more accurate and interesting knowledge [5]. CRD also adopts concept hierarchies to discover rules at multiple concept levels.

$$fellowship\_candidate(V_1) \wedge major(V_1, science) \wedge V_5 = cs\_course$$
$$\rightarrow grade(V_1, V_5, excellent)\ (83\%) \qquad (18)$$

5.2 Fair partition

A partition is called a *fair partition* if it satisfies the following three conditions:

- has sufficient support but scarce confidence

- fails to generate any partition that remains in the next SCP after it is pruned and checked
- fails to generate any partition that leads to a result rule.

For example, the primitive rule for (19) is (20). The partition (21) has sufficient support but scarece confidence, and no merge operation can be exerted to it. Hence (21) is a fair partition.

$$fellowship\_candidate \wedge major \rightarrow grade(P, Q, excellent) \tag{19}$$

$$fellowship\_candidate(V_1) \wedge major(V_2, V_3) \rightarrow grade(V_4, V_5, V_6) \tag{20}$$

$$\{\{V_1, V_2, V_4\}, \{V_3\}, \{V_5\}\{V_6\}\} \tag{21}$$

Fair partitions can be collected during the process of constructing and checking SCP. Though fair partitions have sufficient supports, their corresponding confidences can not be improved to exceed the threshold by merge operation any more. Then CRD tries to specialize some of the variables in them properly to increase their confidences. A bottom-up approach as in [4] or [6] can be modified to perform this work. However, CRD uses a top-down approach. It takes advantage of the definition of support, and prunes uninteresting branches quickly. Due to the length of the paper, we only brief through the idea.

5.3 Specialization Scheme

Some variables in fair partition are chosen to be specialized. In the example of (21), we can choose V_3 and V_5. As V_6 have been specified to be excellent in the meta-rule, we do not choose V_6.

A *specialization* is to specialize one variable to a constant at a certain level. Since two or more variables can be specialized simultaneously, We define a *specialization schemes* as a set of specializations. We also use *scheme* as a shorthand of specialization scheme afterwards.

As an example, scheme samples of (21) are $\{V_3 = science, V_5 = cs\_course\}$ and $\{V_3 = science\}$. Notice that in this example, though V_5 is a head variable, it is specialized for the body. That is, auxiliary predicate(s) will be added to the body when generating rules. e.g., $\{V_3 = science, V_5 = cs\_course\}$ leads to rule (18). We call the condition that no variable is specialized for the head as *head fixed condition*.(However, we can still specify a variable in the head to be a constant in the meta-rule.) In the following discussion, we suppose the head fixed condition holds. Some small modifications will release this condition.

Theorem 5.1 *Suppose the head fixed condition holds, and there are two schemes $S_1 \subseteq S_2$. We get R_1 (or R_2) when S_1(or S_2) and the corresponding fair partition are applied to the primitive rule. The support of R_1(or R_2) is $support_1$ and $support_2$, then we have $support_2 \leq support_1$. (Proof omitted)*

Let $SS[i, j]$ denotes the set of all the interesting schemes that specialize j attributes at the i-th level. We will construct $SS[i, j]$ recursively in the next section.

5.4 Constructing $SS[i, j]$

It is trivial to see that $SS[1, 1]$ contains all the specialization schemes that specialize one attribute at the first level(The root of a concept hierarchy is at the 0th level).

For $i > 1$, we construct $SS[i,1]$ from $SS[i-1,1]$. For each scheme in $SS[i-1,1]$, we descend one level lower and generate schemes for $SS[i,1]$.

For $j > 1$, we construct $SS[i,j]$ from $SS[i,j-1]$. Suppose for each scheme S in $SS[i,j-1]$, the $j-1$ specializations in S are sorted alphabetically according to the variable name. We call two scheme S_1, S_2 a *couple* if :

- They are identical after dropping the first specialization in S_1 and the last one in S_2.
- The first specialization of S_1 and the last one of S_2 are on different variables.

We construct $SS[i,j]$ by merging each couple in $SS[i,j-1]$.

In order to improve the efficiency and drop some uninteresting schemes, we can prune each $SS[i,j]$ immediately after constructing it.

5.5 Pruning and checking SS

Firstly, according to theorem 5.1, we have a corollary that if a scheme S in $SS[i,j]$ has enough support and is interesting, then all of its (j-1)-element subsets must be in $SS[i,j-1]$. Hence we prune all those schemes in $SS[i,j]$ that contain a (j-1)-element subset not in $SS[i,j-1]$.

Then we apply each scheme and its fair partition to the primitive rule, and check the support and confidence of the result rule as follows:

1. If both confidence and support are no less than the corresponding threshold, the rule is output, and the scheme is removed to prevent generating more specific rules.
2. If the support is no less than the threshold, but the confidence is less than its threshold, the scheme remains in the SS for further construction.
3. If support is less than the thresholds, we simply remove the scheme.

5.6 Algorithm of specialization phase

We conclude this phase by giving the algorithm as follows:

```
For each fair partition do
    For i:=1 to as many as possible do
        For j:=1 to as many as possible do
            Construct SS[i, j], prune and check it;
```

We can generate an intermediate relation according to the fair partition. Thus the checking process can be implemented by scanning the intermediate relation once. Since the number of fair partitions, the maximum levels of concept hierarchies, and the number of variables chosen to be checked are all small compared with the large dataset, we can analyze the cost of this phase as $O(N)$, where N is the size of the intermediate relation. As the intermediate relation is usually much smaller than the original ones, the cost of this phase is very low.

6 Features of CRD

CRD method has been implemented in our CRDS with satisfactory results. In some experiments, we find it has some welcome features. Here we relate some of these properties with necessary experimental results.

6.1 Wide spectrum

CRD aims to discover general Horn Clause, with introduction of probability manipulation and abstract constants. The rules that can be discovered by CRD cover a wide spectrum, including recursive rules. Here we give an example to shows that CRD can extract recursive rules successfully. We use two relations: the $arc(Node1, Node2)$ denoting all the arcs in a directed graph and the $path(Node1, Node2)$ denoting all the paths. CRDS generates the following five rules for meta-rule (5) when $confidence = 0.9$ and $support = 0.3$. Three of them are recursive:

$arc(V_1, V_2) \rightarrow path(V_1, V_2)$ $(100\%, 62\%)$
$path(V_3, V_4) \wedge path(V_4, V_8) \rightarrow path(V_3, V_8)$ $(100\%, 38\%)$
$path(V_3, V_4) \wedge arc(V_4, V_6) \rightarrow path(V_3, V_6)$ $(100\%, 38\%)$
$arc(V_1, V_2) \wedge path(V_2, V_8) \rightarrow path(V_1, V_8)$ $(100\%, 38\%)$
$arc(V_1, V_2) \wedge arc(V_2, V_6) \rightarrow path(V_1, V_6)$ $(100\%, 31\%)$

6.2 Strong expressive power

In some previous systems, generated rules are expressed in the form:

$$speciality = `Software' \wedge course\_name = `DataStructure'$$

$$\rightarrow performance = `excellent' \ (40\%)$$

which means that a 'Software' student scores 'Data Structure' excellently with possibility 40%. The predicates in the rule are all of the form $fieldname = `Somevalue'$, whose semantics is very vague, especially when the data is selected from several different tables. For example, if users specify the dataset for fellowship candidates during data selection, the result rule may be:

$$speciality = `Software' \wedge course\_name = `DataStructure'$$

$$\rightarrow performance = `excellent' \ (80\%)$$

which means that a 'Software' fellowship candidate get 'excellent' in 'Data Structure' with possibility 80%. However, we can not distinguish them from the rules themselves. This results from the lack of expressive power of the rule.

If these rules are generated by CRDS, however, they will be expressed as follows.

$$speciality(V_1, software) \wedge course(V_3, data\_structure, V_5, V_6, V_7)$$

$$\rightarrow grade(V_1, V_3, excellent) \ (40\%, 1.2\%)$$

$$fellowship\_candidate(V_1) \wedge speciality(V_1, software)$$

$$\wedge course(V_4, data\_structure, V_6, V_7, V_8) \rightarrow grade(V_1, V_4, excellent) \ (80\%, 0.7\%)$$

We can now easily tell their accurate semantics from the rules themselves.

What's more, the support defined in CRDS serves more than a filter of uninteresting rules. It has its own semantics. This makes each rule have 'dual semantics'. In section 2.3, we have discussed this point.

Table 1: Extracted rules according to different predicates and meta-rule

Predicates	Meta-rule	Results rules
$parent$, $son, male$	$parent \wedge ? \rightarrow son$	$parent(V_1, V_2) \wedge male(V_2) \rightarrow son(V_2, V_1)$ $(100\%, 100\%)$ $parent(V_1, V_2) \wedge son(V_2, V_6) \rightarrow son(V_2, V_1)$ $(100\%, 100\%)$
$parent$, $son, gender$	$? \wedge ? \rightarrow son$	$parent(V_1, V_2) \wedge gender(V_2, male) \rightarrow son(V_2, V_1)$ $(100\%, 100\%)$ $parent(V_1, V_2) \wedge son(V_2, V_6) \rightarrow son(V_2, V_1)$ $(100\%, 100\%)$
$parent$, $son, male$	$? \rightarrow parent$	$son(V_3, V_4) \rightarrow parent(V_4, V_3)$ $(100\%, 62\%)$
$parent$, $son, male$	$? \rightarrow male$	$son(V_3, V_4) \rightarrow male(V_3)$ $(100\%, 75\%)$
$parent$, $son, male$	$? \rightarrow ?$	$son(V_3, V_4) \rightarrow parent(V_4, V_3)$ $(100\%, 62\%)$ $son(V_3, V_4) \rightarrow male(V_3)$ $(100\%, 75\%)$
$parent$, $son, male$	$? \wedge ? \rightarrow ?$	$son(V_3, V_4) \rightarrow parent(V_4, V_3)$ $(100\%, 62\%)$ $son(V_3, V_4) \rightarrow male(V_3)$ $(100\%, 75\%)$ $male(V_5) \wedge parent(V_6, V_5) \rightarrow son(V_5, V_6)$ $(100\%, 100\%)$ $son(V_3, V_4) \wedge parent(V_6, V_3) \rightarrow son(V_3, V_6)$ $(100\%, 100\%)$

6.3 Flexibility

One important criterion for evaluation of a Data Mining system is how accurately it answers the needed knowledge required by users. A favorable system should provide needed knowledge as much as possible and irrelevant information as less as possible. CRDS let users define predicates, concept hierarchies, and let users provide a meta-rule to guide the discovery. Hence users of various interests can usually get just what they want. $Table1$ gives some simple examples to show what users get according to what they want(which is implied in the meta-rule and predicates).

Meaningful rules that do not easily call to our mind are usually interesting to users. CRD never miss them. For example, the rule

$$parent(V_1, V_2) \wedge son(V_2, V_6) \rightarrow son(V_2, V_1)\ (100\%, 100\%)$$

in the table means that if V_1 is a parent of V_2 and V_2 is a son of someone(V_6, which actually means that V_2 is a male), then V_2 is a son of V_1.

6.4 Wide Applicability

As stated above, CRD method can extract a wide spectrum of knowledge, and has good flexibility, it works well with relational and deductive databases in almost any area, such as supermarket, enterprise, stock, etc..

In this paper, we do not try to give examples in those areas. Instead, we will briefly give an example for finding function expressions. Suppose two function $Y = f_1(X)$ and $Y = f_2(X)$, their corresponding table is $Table2$. In our discovery, we denote $Y = f_i(X)$ with predicate $f_i(X, Y)$. Given meta-rule $P(\_, \_) \wedge Q(\_, \_) \wedge R(\_, \_, \_) \rightarrow f_1$ and $P(\_, \_) \wedge Q(\_, \_) \wedge R(\_, \_, \_) \rightarrow f_2$, we get (22), (23) and (24), which indicate that

Table 2: Function f_1 and f_2

f_1	X	1	2	3	4	5	6
	Y	1	2	6	24	120	720
f_2	X	-2	-1	0	2	4	5
	Y	11	7	5	7	17	25

$f_1(X) = f_1(X-1) * X$ or $f_1(X) = \frac{f_1(X+1)}{X+1}$ and $f_2(X) = X^2 - X + 5$.

$$pre(V_1, V_2) \wedge f_1(V_2, V_8) \wedge multiply(V_8, V_1, V_{11}) \rightarrow f_1(V_1, V_{11}) \ (100\%, 83\%) \tag{22}$$

$$pre(V_1, V_2) \wedge f_1(V_1, V_8) \wedge multiply(V_9, V_1, V_8) \rightarrow f_1(V_2, V_9) \ (100\%, 83\%) \tag{23}$$

$$sqr(V_1, V_2) \wedge add5(V_9, V_{10}) \wedge sub(V_2, V_1, V_9) \rightarrow f_2(V_1, V_{10}) \ (100\%, 100\%) \tag{24}$$

6.5 Guideline(proof omitted)

CRD takes it as a guideline to discover more general knowledge, ignoring specific rules that are 'contained' in the others. For example, if (25) is generated, then (26) and (27) will not be generated in our method.

$$p(X, Y) \wedge q(Y, Z, U) \rightarrow r(X, Z) \tag{25}$$

$$p(X, Y) \wedge q(Y, Z, Z) \rightarrow r(X, Z) \tag{26}$$

$$p(X, Y) \wedge q(Y, Z, special\_value) \rightarrow r(X, Z) \tag{27}$$

6.6 completeness(proof omitted)

All the rules that comply with the meta-rule, and whose confidences and supports are no less than their corresponding thresholds, and none of whose variables is instantiated to constants, can be discovered by CRD, or implied by another rule having been discovered by CRD(refer to guideline).

7 Conclusions

In this paper, we provide a new data mining method in deductive databases. In this method, users can define predicates and concept hierarchies according to their application environments. Also, users can submit a meta-rule to specify their required knowledge. Thus, CRD is flexible and is able to answer queries accurately. CRD aims to extract general Horn Clauses from data. As the results manifest, it discovers a wide spectrum of knowledge, even some that previous systems fail to discover. What's more, the rules extracted by CRD method have strong expressive power. All these are accomplished within a single algorithm, which saves the large expenditure of system integration. Both complexity analysis and experimental results reveal that CRD has satisfactory efficiency and consequently, a promising future.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In Proc. of ACM SIGMOD Conference on Management of Data, pp 207-216, Washington D.C., May 1993.

[2] M.L. Brodie and S.Ceri. On Intelligent and Cooperative Information System: A Workshop Summary. International Journal of Intelligent and Cooperative Information System, Vol. 1 No. 2 pp233-248. 1992

[3] Saso Dzeroski and Nada Lavrac, Inductive Learning in Deductive Databases, IEEE Transactions on Knowledge and Data Engineering, Vol.5, No.6. December 1993

[4] Y.FU and J.Han, Meta-Rule-Guided Mining of Association Rules in relational databases in Proc 1995 Intl Workshop on Knowledge Discovery and Deductive and Object- Oriented Databases(KDOOD'95), Singapore, December 1995

[5] J.Han and Y.Fu, Discovery of multiple-level association rules from large databases, In Proc 1995 Int. conf. Very Large Data Bases. Zurich, Switzerland. Sept. 1995

[6] J. Han and Y. Fu. Exploration of the power of attribute-oriented induction in data mining. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, Advances in Knowledge Discovery and Data Mining, pp 399-421. AAAI/MIT Press, 1996.

[7] J. Han, and Y. Fu. Dynamic generation and refinement of concept hierarchies for knowledge discovery in databases. In Proc. AAAI'94 Workshop on Knowledge Discovery in Databases(KDD'94), pp157-168, Seattle, Wa.: AAAI Press

[8] J. Han, Y. Huang, N. Cercone and Y. Fu. Intelligent query answering by knowledge discovery Techniques, IEEE Transaction on Knowledge and Data Engineering. Mar. 1995

[9] X. Hu, N. Cercone and J. Han, An attribute- Oriented Rough set Approach for Knowledge Discovery in Databases, in W.P. Ziarko(eds.), Rough Sets, Fuzzy Sets and Knowledge Discovery, Springer-Verlag, pp90-99, 1994,

[10] T. Imielinski. Intelligent query answering in rule based system. J. Logic Programming , Vol.4 , pp229-257, 1987

[11] V.Phar and A. Tuzhilin, Abstract-Driven Pattern Discovery in Databases, IEEE Transactions on Knowledge and Data Engineering, Vol.5, No.6. December 1993

[12] W. Sheng, Bing Leng. A Meta-Pattern- Based Automated Discovery Loop for Integrated Data Mining- Unsupervised Learning of Relational Patterns. IEEE Transactions on Knowledge and Data Engineering, Vol. 8. No. 6. Dec. 1996.

[13] W.Shen, K.Ong, B.Mitbander and C.Zaniolo, Meta-queries for data mining, in U.M. Fayyad, G. Piatetsky-Shapiro, P.Smyth, and R.Uthurusamy, editors. Advances in knowledge Discover and Data Mining. AAAI/MIT Press 1995

[14] —, Learning Logical Definitions from Relations. Machine learning Vol. 5. No. 3. pp239-266, 1990