IEEE Trans. on Knowledge and Data Engineering, 8(6),898–910.

A Metapattern-Based Automated Discovery Loop for Integrated Data Mining
— Unsupervised Learning of Relational Patterns

Wei-Min Shen
Information Sciences Institute and
Computer Science Department
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
shen@isi.edu

Bing Leng
Inference Corporation
4th Floor, 8410 W Bryn Mawr Ave
Chicago, IL 60631
leng@inference.com

Abstract

Metapattern (also known as metaquery) is a new approach for integrated data mining systems. Different from a typical "tool-box" like integration, where components must be picked and chosen by users without much help, metapatterns provide a common representation for inter-component communication as well as a human interface for hypothesis development and search control. One weakness of this approach, however, is that the task of generating fruitful metapatterns is still a heavy burden for human users. In this paper, we describe a metapattern generator and an integrated discovery loop that can automatically generate metapatterns. Experiments in both artificial and real-world databases have shown that this new system goes beyond the existing machine learning technologies, and can discover relational patterns without requiring humans to pre-label the data as positive or negative examples for some given target concepts. With this technology, future data mining systems could discover high-quality, human comprehensible knowledge in a much more efficient and focused manner, and data mining could be managed easily by both expert and less expert users.

Index Terms: Induction, Deduction, Human Interaction, Integration, Unsupervised Learning, Relational Concepts, Metaquery, Metapattern.

1 Introduction

It has been commonly recognized that integration is an essential aspect of building effective data mining systems [22]. However, the existing approaches for integration are still quite primitive. A system may provide a collection of data mining techniques for humans to pick and choose, yet offer little assistance in important activities such as creating hypotheses, preparing relevant data sets, and selecting appropriate analysis tools. As a consequence, such systems behave like "tool boxes" and they put high demands on human users. To use them effectively, a human user must not only be an expert of the domain, but also familiar with the specific databases and data analysis tools.

Metapatterns provide an alternative approach to the integration problem. To discover patterns from a set of databases, a user simply specifies a metapattern (a second-order expression) for the type and format of the patterns that he or she would like to discover. The system will then instantiate the metapattern to a set of appropriate database queries, prepare the data sets based on the query results, and invoke the necessary analysis tools to generate final patterns that satisfy the format and content of the given metapattern. In several existing systems (for example [6, 13, 25, 26, 27]), human-generated metapatterns have been used in an interactive discovery loop to integrate inductive machine learning algorithms, statistic analysis tools, and deductive database technologies.

One of the open problems of the metapattern approach, however, is that the task of generating metapatterns is still a heavy burden for human users. The current machine learning technologies do not provide a satisfactory solution for the task because algorithms for learning relational patterns have severe limitations when applied to databases directly. On one hand, most algorithms that learn relational patterns (see, for example, [18, 20]) require humans to label the data as positive and negative examples for a given target concept. On the other hand, most unsupervised algorithms (see, for example, [3, 14, 28, 29, 31]) hardly go beyond learning attribute-based concepts.

This paper describes recent progress towards automatic generation of metapatterns in an integrated discovery loop that is both interactive and autonomous. Such a discovery loop provides much more feedback and interaction for human users, and it can discover relational patterns directly from databases without requiring humans to pre-label the data. The metapattern generator and the automated discovery loop have been implemented in a prototype system and tested in several artificial databases and one large, real-world database. To demonstrate its capability of unsupervised learning of relational patterns, the system has also been applied to some well-known supervised learning tasks to show that relational patterns can be learned without supervision (i.e., manually pre-labeling the data) as well.

Section 2 of this paper reviews the related work on relational learning techniques and integrated data mining systems. Sections 3 through 7 describe in detail the metapattern generator and the automated discovery loop. Section 8 illustrates the capabilities of the system with several unsupervised relational learning experiments and some experiments in a large, real-world logistic database. Finally, Section 9

concludes the paper with a summary of the work and several directions for future research.

2 Related Work

This section reviews the related work on learning relation-based patterns and integrated data mining systems (in particular, metapatterns and the human-directed discovery loop).

2.1 Learning Relational Patterns

Since learning relation-based patterns from databases is our major concern here, several research projects in machine learning and inductive logical programming are closely related to this paper.

FOIL [20] is a machine learning algorithm that can induce a set of logical Horn rules for a given concept from a set of positive and negative examples of that concept. Based on this algorithm, many algorithms, for example FOCL [16], have used semantic knowledge to enhance the performance of selecting candidate terms for building definitions. These algorithms represent the efforts to integrate static domain knowledge with induction.

The researchers in inductive logical programming have also addressed the problem of learning relation-based patterns. Different from the algorithms described above, these programs (see for example [2, 15, 18]) are focused on inventing new predicates from positive and negative training examples. Some of them (for example [17]) also use semantic knowledge as guidance.

Our research in this area is focused on discovering relation-based patterns from data that are commonly found in databases. These data have no labels to indicate whether they are positive or negative examples of some concepts, and they unavoidably include errors and noise. These imply that the algorithms to learn from these data must not rely on humans to predetermine what target concepts to discover, the patterns discovered must be more flexible than rigid logic rules (i.e., their significance must be statistically characterized to reflect the characteristics of the underlying data). Furthermore, due to the nature of data mining, the domain and semantic knowledge must be incorporated interactively as the discovery loop iterates; they cannot be predetermined at the offset of the application.

Although there are existing systems (see for example [3, 14, 28, 29, 31]) in the literature that can perform unsupervised learning, most of them are not targeted to learning relation-based patterns directly from databases. These systems differ from ours in several aspects, including the goal and context of learning, ways of learning (incremental or constructive), and the form of the final concepts.

CLUSTER/S [28] performs goal-oriented classification on structured objects by using two methods. The first is concept formation by repeated discrimination which requires both positive and negative examples. The second is concept formation by finding classifying attributes. This method attempts to find one or more classifying attributes whose value sets can be split into ranges that define individual classes. It is not clear how this method could learn the relationships among the given attributes.

MOBAL [31] regards concept formation as a process of forming extensional aggregates of objects and then characterizing these aggregates with an intentional definition. Given a knowledge base of facts and rules that may be overly general, MOBAL uses its knowledge revision tool (KRT) to correct those overly general rules. Whenever KRT cannot correctly specialize an incorrect rule, the rule's instances and exceptions are used as positive and negative examples of a new concept. These examples are then passed to its concept learning tool (CLT) which uses a model-driven, most-general learner (RDT) to induce the concept definition. MOBAL requires some rules to start with which are not available in a database. Although RDT can learn concepts, it needs pre-labeled examples.

LABYRINTH [29] incrementally forms concepts on composite objects, while KLUSTER [14] constructively induces structural knowledge on term-subsumption formalisms. Both systems produce concept hierarchy which is a different representation from function-free Horn clauses. Finally, KBG [3] is a knowledge based generalizer in a first order logic representation. It forms generalization tree, rather than implication rules, from a set of examples, each of which is a collection of tuples, by iterative use of clustering and generalization. However, its generalization tree cannot represent recursive concepts, its examples need pre-grouped from database tables.

2.2 Integrated Knowledge Discovery Systems

It has been commonly recognized that the integration of different data mining techniques, such as induction, deduction, and human guidance, is a necessary and crucial step in building an effective discovery system (see for example [22]). Many existing systems indeed contain some or all of these components. For example, systems such as INLEN [12] and IMACS [21] concentrate on inductive learning methods and the use of human knowledge. Systems such as KDW [7], RECON [4], DBMiner [8] connect tightly with databases and provide a set of inductive and deductive tools for humans to choose from.

Our research has been focused on a different but important aspect of the integration issue, namely, what are the interdependencies between induction, deduction, and human guidance, and how to exploit them to build integrated data mining systems that are more than just collections of different tools. We recognize that induction, deduction, and human guidance are intrinsically connected in an iterative discovery loop: Induction generates hypotheses from data but counts on other agents to prepare the data and select the proper tools; deduction verifies hypotheses in databases but relies on a source for hypotheses; humans have valuable external knowledge but depend on both induction and deduction components to perform the analysis tasks. All these dependencies must be supported by a common representation so that sharing knowledge, hypotheses, and tasks will be possible.

In order to exploit these interdependencies, new technologies have been developed to link induction, deduction, and human guidance via a notion called metapatterns (also known as metaqueries) [23, 26, 27]. A metapattern is a second-order logical expression with multiple usages: it contains information for accessing real databases deductively to prepare data, for invoking inductive actions (such as constructing relational

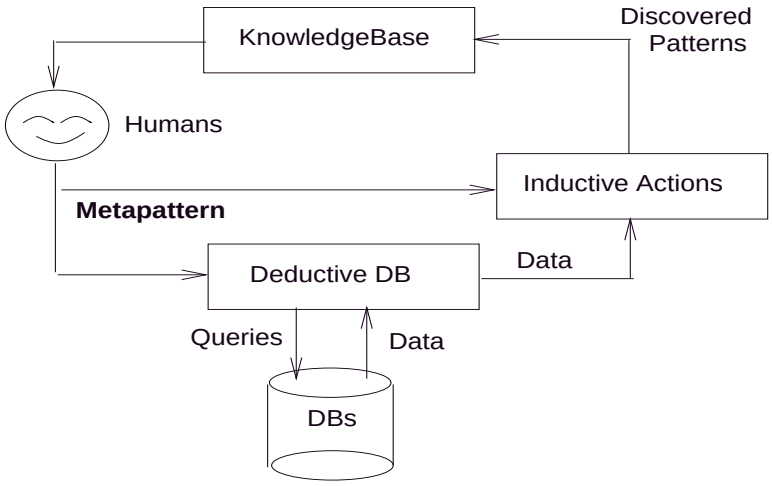

Figure 1: Metapatterns and the Human-Directed Discovery Loop

patterns, supervised classification, unsupervised clustering, regression, and visualization), for instructing the construction of discovered patterns, and for accepting human guidance as part of the iterative discovery loop.

2.3 Metapatterns and the Human-Directed Discovery Loop

The notion of a *metapattern* is proposed in [26, 27] as a template or a second-order expression in a language $\mathcal{L}$ that describes a type of pattern to be discovered. For example, let P, Q and R be variables for predicates, and X, Y and Z be variables for objects, then the metapattern

$$P(X,Y) \wedge Q(Y,Z) \Rightarrow R(X,Z)$$

specifies that the patterns to be discovered are transitivity relations $p(X,Y) \wedge q(Y,Z) \rightarrow r(X,Z)$, where p, q, and r are specific predicates. Two possible patterns satisfying this metapattern are

$$\text{citizen}(X,Y) \wedge \text{officialLanguage}(Y,Z) \rightarrow \text{speaks}(X,Z)\ [0.93]$$
$$\text{parent}(X,Y) \wedge \text{ancestor}(Y,Z) \rightarrow \text{ancestor}(X,Z)\ [0.99]$$

In the first pattern, "citizen", "officialLanguage", and "speaks" are relations bound to P, Q, and R, respectively, in the current database. In the second pattern, "parent" is a relation bound to P, and "ancestor" is a relation bound to both Q and R, and this is a recursive pattern about ancestor relationship. Note that each pattern is associated with a "strength" value, which is the probability of seeing the right-hand side being true when the left-hand side is true.

In general, a metapattern is a two-part specification: the left-hand side is a conjunction of meta-predicates and predicates that specifies a strategy for data gathering, and the right-hand side is an action to be applied to the gathered data and related predicates. For example, the left-hand of the above example, when P and Q are bound to specific predicates p and q respectively, specifies that only those data tuples $[X,Z]$ that satisfy $p(X,Y) \wedge q(Y,Z)$ for some Y should be fetched. The right-hand of that example, when R is bound

to a specific predicate r, is shorthand for an action that computes the strength of a given pattern using the returned tuples $[X, Z]$. In fact, a more precise formulation of the above metapattern is:

$$P(X, Y) \wedge Q(Y, Z) \Rightarrow \mathit{ComputeStrength}(R, [X, Z])$$

where $\mathit{ComputeStrength}(R, [X, Z])$ is an action that computes the strength of the pattern (we will give a precise definition of the strength in a later section). The result of executing a metapattern is a set of patterns whose left-hand sides are instantiated forms of the left-hand side of the metapattern, and whose right-hand sides are the results of the corresponding metapattern action.

As reported in [25, 26, 27], the supported actions for metapatterns are as follows:

- $\mathit{ComputeStrength}(R, [X, ..., Z])$, where R is a predicate variable and $[X, ..., Z]$ is a tuple of object variables. This action returns an expression $r(X, ..., Z)$ and a strength value when a specfic predicate r is bound to R.
- $\mathit{Classify}([X, ..., Z])$. This action classifies the given data tuples and returns a set of class descriptions. By convention, the variable Z is the variable for classes.
- $\mathit{Cluster}([X, ..., Z])$. This action clusters the given data tuples and returns a set of cluster descriptions.
- $\mathit{Plot}([X, ..., Z])$. This action plots the given data and returns a set of graphic commands that represent the graph.
- $\mathit{Regression}([X_1, ..., X_n, Y_1, ..., Y_m])$. This action approximates the function between the independent variables, $X_1, ..., X_n$, and the dependent variables, $Y_1, ..., Y_m$, and returns the approximated function.

In addition to being templates for patterns, metapatterns are also used to create a discovery loop as shown in Figure 1. For the deductive part of the loop, metapatterns outline the data-collecting strategy and serve as the basis for the generation of specific queries. Queries are generated by instantiating the variables in the left-hand side of the metapatterns with relevant table names and column names in the database of interest (the relevant information is defined and stored in an on-line knowledge base) and then run against the database to collect relevant data. Similarly, for the inductive part of the loop, metapatterns serve as generic descriptions of the class of patterns to be discovered: The action of a metapattern determines which inductive action to apply, and the format of a metapattern is the mold for the final results.

Since metapatterns are declarative expressions, they serve as a very important interface between human discoverers and the discovery system. They can be specified by human experts, or alternatively, as we will see later in this paper, generated automatically. Using metapatterns, human experts can focus the discovery process onto more profitable areas of the database; the system-generated metapatterns will provide valuable clues to the human expert regarding good starting points for database searches. They will also serve as the evolutionary basis for the development of more fruitful metapatterns.

Metapatterns and the human-directed discovery loop are currently implemented in a system called DataCrystal, described in [26, 27]. The deductive part of DataCrystal contains a metapattern instantiator and a deductive database technology called $\mathcal{LDL}$++ [19, 30]. The instantiator generates a set of $\mathcal{LDL}$++ rules from a metapattern. These rules are then used by $\mathcal{LDL}$++ to access various types of relational databases. The inductive part of DataCrystal includes a set of inductive data analysis tools that implement the above metapattern actions and a pattern constructor. At present, the action *classify* is a high-performance incremental algorithm called CDL2 [24] for learning decision lists from data. The action *cluster* is a Bayesian-based Cobweb clustering algorithm [5, 27]. The action *ComputeStrength* is an $\mathcal{LDL}$++ procedure, the *Plot* action is a powerful GnuPlot tool, and the *regression* action is a standard polynomial interpolation method augmented with a stopping criterion based on the priciple of Minimal Descrition Length (MDL). The task of the pattern constructor is to package the final results of the induction into a set of patterns that match the form of the current metapattern. The discovered patterns are stored persistently in an on-line knowledge base. The patterns can be interpreted as $\mathcal{LDL}$++ rules themselves, so they can be selected to run directly in the databases as queries. Currently, the discovery loop in DataCrystal is only human-directed; it takes metapatterns directly from human experts.

The metapattern-based, human-directed discovery loop has already been successfully applied to several real-world applications. These include discovering regularities in a large, common-sense knowledge base [23], finding circuit patterns from a telecommunication database [26], building prediction models from a chemical research database [27], constructing fault detection rules from a database that contains sequences of control data for semiconduct chip manufacture [25], discovering association rules in databases [6], and detecting consuming patterns in retail [13]. Interested readers are refered to these papers for details.

3 The Metapattern Generator

Although metapatterns are powerful tools for data mining, designing the right metapatterns for a given application is not an easy task. If a metapattern is too specific, then it may miss the interesting patterns. If a metapattern is too general, then it may exhaust the computing resources that are available. To generate the right metapatterns, one must not only understand the nature of the underlying data, but also analyze the patterns discovered from the previous metapatterns.

To illustrate how productive metapatterns are generated manually, let us consider our experience in the chemical research domain. Following a suggestion by a chemist, we initially used a metapattern to find the relationship between a set of compounds that have different percentages of the ingredients 'A322' and 'B721' and their chemical properties. However, the patterns returned did not show any trends. After showing the results to the chemist, he discovered that these compounds also contained auxiliary chemicals that may effect the properties in a different way. Given this knowledge, we constrained the metapattern so the compounds that had such auxiliary ingredients were not considered. Sure enough, the resulting patterns showed many

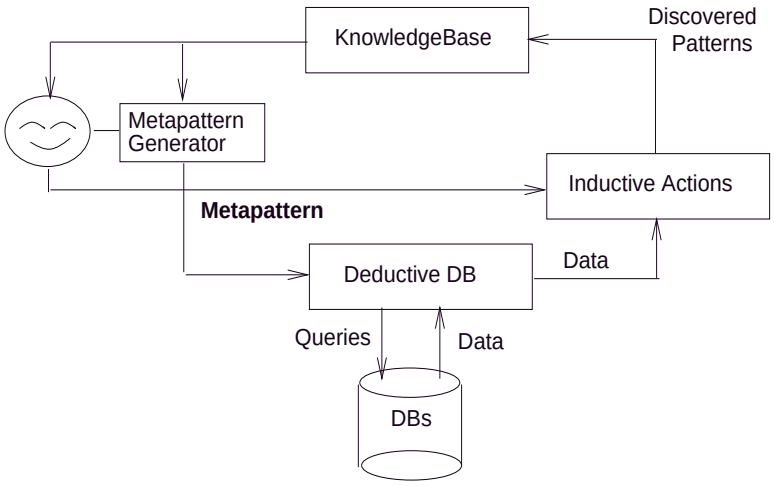

Figure 2: The Automated Discovery Loop with a Metapattern Generator

clear trends. We can see from this example that if domain experts can directly interact with the system (i.e., metapatterns are generated automatically for them to choose and test), then the entire discovery process can be much more efficient and productive. The system should provide suggestions and feedback of metapatterns so the experts can discover new knowledge and try better metapatterns.

For this purpose, a metapattern generator must be developed to automate the discovery loop. As shown in Figure 2, this generator will be placed in parallel with the human user and will generate metapatterns based on either the meta-knowledge of the database, such as the data schema and the data ranges, or the patterns that are discovered with other metapatterns. Human users can interact with the generator by examining, selecting, and executing the metapatterns, and they can also create metapatterns by themselves as before. The motivation for having this generator is to give human users suggestions for new metapatterns so that the more expert users can get inspirations, and the less expert users can learn how to perform data mining in a particular domain by observation.

3.1 The Space of Metapatterns

In order to generate metapatterns, we must first understand the nature of the set of all possible metapatterns. As we described earlier, metapatterns are templates in a language $\mathcal{L}$ that describes types of patterns to be discovered. For simplicity, we shall ignore the actions in metapatterns for the moment, and assume that the left-hand side of a metapattern is a conjunction of second-order predicates and the right-hand side has only one such predicate. (If this type of metapattern is generated, we can always replace the right-hand side with the appropriate actions.) Furthermore, we restrict ourselves only to the relational data model, so a predicate variable can only be bound to relational table names or built-in predicates in $\mathcal{L}$ (such as *equal, greaterThan*, or *lessThan*), and an object variable can only be bound to column names or constant values (such as the integer "20" or the language "English").

For a fixed length, metapatterns can be ordered from the most general to the most specific, depending on how many variables, either for predicates or for objects, are present. For example, among all metapatterns that have three binary (second-order) predicates, the most general ones include:

$$P(X,Y) \wedge Q(Y,Z) \Rightarrow R(X,Z) \quad \text{(MP-1)}$$

$$P(X,Y) \wedge Q(X,Z) \Rightarrow R(X,W) \quad \text{(MP-2)}$$

$$P(X,Y) \wedge Q(Y,Z) \Rightarrow R(X,W) \quad \text{(MP-3)}$$

These metapatterns are the most general because they contain only variables. On the other hand, the most specific metapatterns corresponding to the general ones listed above (MP-1, MP-2, and MP-3) include:

authorOf('Orwell','AnimalFarm') $\wedge$ writtenIn('AnimalFarm','English') $\rightarrow$ canWrite('Orwell','English')

likes('John',*tools*) $\wedge$ hasHobby('John','Carpenter') $\rightarrow$ ownsOne('John',*hammer*)

livesIn('Mary','House1') $\wedge$ costs('House1',900893) $\rightarrow$ Income('Mary','High')

All the variables in these metapatterns are bound to specific table names (e.g., likes), column names (e.g., *tools*), or constant values (e.g. 'John'). We define a *family* of metapatterns to be the set of all metapatterns of length n. The metapatterns in a family are partially ordered. One can traverse a family of metapatterns from the general to the specific by incrementally instantiating the predicate variables with table names and built-in predicates and the object variables with column names and in turn constant values.

Note that *not* all metapatterns in a family are interesting. In order to have some prediction value, a metapattern must be *connected*. That is, the predicate on the right-hand side must share at least one variable with some predicate on the left-hand side. For example, the metapattern $P(X,Y) \wedge Q(Y,Z) \Rightarrow R(U,V)$ is not interesting because its right-hand side is not connected to the left-hand side. Furthermore, predicates like $P(X,X)$ or $p(X,X)$ are not considered interesting because they do not link to others. Since we are using relational database models, one should also notice that the order of parameters for a predicate is not important. For example $P(X,Z)$ is the same as $P(Z,X)$ because in a relational table, columns are not ordered.

With the families of metapatterns so defined, one natural question to ask is whether the length of metapatterns can be arbitrarily long. Fortunately, for any given set of databases, the length of the longest metapatterns is bounded because metapatterns must be connected and there is only a fixed number of tables, columns, built-in predicates, and values that are presented in the databases. (Even if a column is typed "real", the number of distinct values in the column is finite because of the implementation.) Therefore, it is meaningful to define the *space* of all possible metapatterns for an application to be the union of all possible families of metapatterns.

4 Generating Metapatterns Based on Data Schema and Ranges

The space of metapatterns is very large and it is infeasible to enumerate them all. Our approach is to start with the set of most general metapatterns, and incrementally generate interesting ones as the process of discovery continues. Our goal is not to cover all the metapatterns but to guide the discovery process in fruitful directions.

Among all the general metapatterns, the transitivity metapattern (see MP-1 in Section 3.1) is the most interesting. In essence, it subsumes many other types of metapatterns, such as *implication*, *inheritance*, *transfers through*, and *functional dependency* (see [23] for details). In this section, we describe an algorithm to generate the set of all possible transitivity metapatterns for a given database. We believe that similar algorithms can be designed to generate other types of general metapatterns such as MP-2 or MP-3 in Section 3.1.

The set of all possible transitivity metapatterns can be generated based on the data schema and ranges of the databases. The idea is to first identify the sets of columns that are significantly connected, and then use these sets to build metapatterns.

Two columns, from different database tables, are significantly connected, if they have the same type and have ranges that overlap each other above a user specified threshold o, where $0 < o < 1$. The degree of overlapping is computed as follows. Let C_x and C_y be two columns, and V_x and V_y be their value sets, respectively, then the overlapping of C_x and C_y is the maximum number of the shared values relative to either V_x or V_y, as follows:

$$Overlap(C_x, C_y) = \max(\frac{|V_x \cap V_y|}{|V_x|}, \frac{|V_x \cap V_y|}{|V_y|}).$$

where $|\cdot|$ denotes the cardinality of a set. Here domain knowledge may be used to eliminate unnecessary connections (e.g., height vs. temperature) or suggest and establish syntactically different connections (e.g., color vs. light frequency). Each pair of columns that are connected are then given a reference name, and these connections will be represented in a significant connection table (SCT), where each row is a connection, each column is a table, and each non-empty entry is the name of a connected data field (or column).

To illustrate the idea, consider for example an abstract database shown in Figure 3. In this database, there are four tables, t_1 to t_4, each has some columns c_{ij}. For simplicity, the value ranges of each column are also listed along with the schema. (In reality, value ranges can be obtained by simple SQL queries.)

Given these information, pairs of columns that are connected can be easily determined according to our definition. For example, suppose the threshold o for overlapping is set to 0.6, then column c_{13} in table t_1 and column c_{22} in table t_2 are connected because they have the same data type and their overlapping is 0.9. A reference name, X_1, is then created for this pair of connected columns. After considering every pair of columns, a significant connection table, shown in the upper part of Figure 4, is constructed. As we can see, every connected pair of columns is represented as a row in this SCT. For instance, columns c_{13} and c_{22} are in the first row, where c_{13} is under t_1 while c_{22} is under t_2.

Schema and Data Ranges

Tables	Columns (Name Type[ValueRange])					
t_1	c_{11}	char(2)	c_{12}	integer[2–7]	c_{13}	float[0.4–0.8]
t_2	c_{21}	integer[12–17]	c_{22}	float[0.1–0.7]	c_{23}	char(3)
t_3	c_{31}	integer[13–16]	c_{32}	char(2)		
t_4	c_{41}	char(3)	c_{42}	float[0.0–0.1]	c_{43}	integer[4–7]

Table t_1

c_{11}	c_{12}	c_{13}
jj	5	0.5
nn	5	0.8
ll	7	0.5
qq	5	0.5
kk	5	0.6
pp	4	0.6
mm	2	0.5
nn	4	0.6
kk	4	0.4
nn	5	0.4

Table t_2

c_{21}	c_{22}	c_{23}
14	0.5	mmm
14	0.6	iii
14	0.3	jjj
12	0.7	nnn
12	0.1	lll
15	0.6	ppp
15	0.4	mmm
13	0.6	ooo
16	0.6	ooo
17	0.4	mmm
14	0.4	lll
14	0.6	kkk
15	0.3	mmm
12	0.5	mmm
15	0.4	nnn
15	0.6	ooo
16	0.5	ppp
16	0.7	ppp

Table t_3

c_{31}	c_{32}
14	oo
15	kk
16	mm
15	kk
16	ll
15	ll
15	mm
13	oo
16	oo
14	mm
13	mm
14	mm

Table t_4

c_{41}	c_{42}	c_{43}
nnn	0.0	5
mmm	0.0	6
rrr	0.1	4
mmm	0.0	4
ooo	0.1	7
jjj	0.0	4
kkk	0.0	5
mmm	0.0	5
jjj	0.1	4
mmm	0.0	5
jjj	0.0	5
jjj	0.0	5
lll	0.0	7
nnn	0.1	4

Figure 3: An Example Database

For reasons that will become clear later, we also represent the information in SCT as a graph G, where each node in G is an non-empty entry in the SCT, and each edge connects two non-empty entries that are on the same row or column in the SCT. For example, the graph built from the SCT in Figure 4 is shown in the lower part of Figure 4, where node (t_1, X_1) and node (t_2, X_1) represent two non-empty entries, c_{13} and c_{22}, in the SCT. Since they are in the same row, there is an horizontal edge between them. Similarly, node (t_1, X_1) and node (t_1, X_2) represent two non-empty entries, c_{13} and c_{11}, in the SCT, this time they are on the same column, so there is a vertical edge between them.

The graph G generated above provides a basis for generating all possible transitivity patterns in a given database. The idea is to find all the cycles in the graph with alternated vertical and horizontal edges, and convert each of these cycles into a "cycle" of predicates before generating a set of transitivity patterns.

Finding cycles in a graph can be accomplished by using a standard transitive closure algorithm (see for example [1]) with some simple augmentation to enforce the alternating edge constraint. A graph G cannot have more than $|G|!$ cycles because a cycle, without duplicated nodes, cannot be longer than the size of the graph. To convert a cycle of graph nodes into a cycle of predicates is also a straightforward task; one can simply rewrite each vertical edge in the cycle by the table name. For example, the cycle indicated by thick lines in Figure 4 is

$$(t_2, X_1)(t_2, X_5)(t_4, X_5)(t_4, X_3)(t_1, X_3)(t_1, X_1)(t_2, X_1)$$

and that can be rewritten as a cycle of predicates as follows:

SCT

	t_1	t_2	t_3	t_4
X_1	c_{13}	c_{22}		
X_2	c_{11}		c_{32}	
X_3	c_{12}			c_{43}
X_4		c_{21}	c_{31}	
X_5		c_{23}		c_{41}

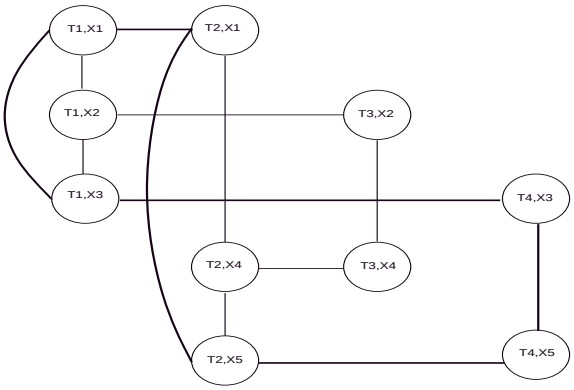

Figure 4: A Significant Connection Table (SCT) and its graph G

$$t_2(X_1, X_5), t_4(X_5, X_3), t_1(X_3, X_1),$$

where $t_2(X_1, X_5)$ is a rewrite of the vertical edge $(t_2, X_1)(t_2, X_5)$, and $t_4(X_5, X_3)$ is a rewrite of $(t_4, X_5)(t_4, X_3)$, and so on. Using this method, we can generate all cycles of predicates from the graph G, as listed in Figure 5.

$$t_2(X_1X_5)t_4(X_3X_5)t_1(X_1X_3)$$
$$t_2(X_1X_4)t_3(X_2X_4)t_1(X_1X_2)$$
$$t_2(X_5X_1)t_1(X_1X_2)t_3(X_2X_4)t_2(X_4X_5)$$
$$t_2(X_4X_5)t_4(X_5X_3)t_1(X_3X_1)t_2(X_1X_4)$$
$$t_1(X_3X_1)t_2(X_1X_4)t_3(X_4X_2)t_1(X_2X_3)$$
$$t_3(X_2X_4)t_2(X_4X_5)t_4(X_5X_3)t_1(X_3X_2)$$
$$t_1(X_2X_3)t_4(X_3X_5)t_2(X_5X_1)t_1(X_1X_2)$$
$$t_2(X_1X_4)t_3(X_4X_2)t_1(X_2X_3)t_4(X_3X_5)t_2(X_5X_1)$$
$$t_1(X_1X_2)t_3(X_2X_4)t_2(X_4X_5)t_4(X_5X_3)t_1(X_3X_1)$$

Figure 5: All predicate cycles found in the example DB

From the list of all possible cycles of predicates, we can now generate a complete set of transitivity metapatterns by generalizing table names and reference names and introducing an implication in each cycle. In our current example database, the result is a set of metapatterns listed in Figure 6. For example, MP-4 is a generalization of the first two predicate cycles in Figure 5; MP-5 is a generalization of cycles 3 through 7; and MP-6 is a generalization of the last two cycles. This set is complete because it includes all possible transitivity metapatterns in our example database.

$$P_1(Y_1, Y_2) \wedge Q_1(Y_2, Y_3) \Rightarrow R_1(Y_1, Y_3) \quad \text{(MP-4)}$$
$$P_2(Y_1, Y_2) \wedge Q_2(Y_2, Y_3) \wedge W_2(Y_3, Y_4) \Rightarrow R_2(Y_1, Y_4) \quad \text{(MP-5)}$$
$$P_3(Y_1, Y_2) \wedge Q_3(Y_2, Y_3) \wedge W_3(Y_3, Y_4) \wedge V_3(Y_4, Y_5) \Rightarrow R_3(Y_1, Y_5) \quad \text{(MP-6)}$$

Figure 6: All transitivity metapatterns found in the example DB

5 Discovering and Evaluating New Patterns

Generating all transitivity metapatterns is not the end of story. Depending on the strength or interestness of the patterns that are found with these metapatterns, the discovery system should generate more metapatterns that are deemed to be plausible. To do so, let us first examine how discovered patterns are evaluated.

When a metapattern is selected for execution, the system first instantiates it into a set of specific patterns that are possible in the current databases, and then evaluates them to see if they have sufficient support from the actual data. Given the information produced in the process of generating metapatterrns, instantiating a metapattern is a straightforward procedure. It simply replaces the variables in the metapattern with specific table names and column names. For example, the variables P_1, Q_1, and R_1 in MP-4 in Figure 6 can be bound to table names t_1, t_2, t_3, and t_4 (these are the table names involve in the first two cycles of predicates in Figure 5), and reference variables X_1, X_2, X_3, X_4, and X_5 can be bound to corresponding columns according to the SCT in Figure 4, as follows:

$$\begin{aligned}
t_2(X_1X_5)t_4(X_5X_3) &\rightarrow t_1(X_1X_3)\\
t_1(X_3X_1)t_2(X_1X_5) &\rightarrow t_4(X_3X_5)\\
t_4(X_5X_3)t_1(X_3X_1) &\rightarrow t_2(X_5X_1)\\
t_2(X_1X_4)t_3(X_4X_2) &\rightarrow t_1(X_1X_2)\\
t_1(X_2X_1)t_2(X_1X_4) &\rightarrow t_3(X_2X_4)\\
t_3(X_4X_2)t_1(X_2X_1) &\rightarrow t_2(X_4X_1)\\
&\vdots
\end{aligned}$$

Notice that not all instantiated patterns are supported by the data in the underlying database. We say a pattern is "interesting" only if its strength is above a user specified threshold. In our approach, each pattern p is evaluated against the underlying database U_{db} by two values: the *strength* value p_s, which is the probability of seeing the right-hand side of p being true when the left-hand side of p is true, and the *base* value p_b, which estimates how likely the left-hand side of p occurs in databases that have the same schema of U_{db}.

Based on the famous "Laplace's Rule of Succession", the strength value of p can be computed as

$$p_s = Prob(RHS|LHS, U_{db}, I_0) = \frac{|S_{rhs}| + 1}{|S_{lhs}| + 2}$$

where LHS represents the left-hand side of p, RHS the right-hand side of p, S_{lhs} the set of tuples in U_{db} that

satisfy LHS, S_{rhs} the set of tuples in S_{lhs} that satisfy RHS, and I_0 the assumption that the prior distribution of S_{rhs} in S_{lhs} is uniform. Intuitively, this p_s value is the probability of seeing a tuple that satisfies RHS given the condition that the tuple satisfies LHS. Interested readers may find the complete derivation of Laplace's Rule of Succession in Chapter 6 of [10].

It is difficult to compute the exact likelihood of the occurrence of the left-hand side of p in databases that have the same schema of U_{db}, so we use the following formula to estimate the base value p_b of a pattern:

$$p_b = \frac{|S_{lhs}|}{DOM(LHS)}$$

where $DOM(LHS)$ is the product of sizes of the tables that appear in LHS. Note that this value is bound in $0 \leq p_b \leq 1$ because the number of tuples in S_{lhs} can never be greater than DOM(LHS). In fact, the value of p_b is often small and very rarely close to 1.

In the discovery process, a pattern's strength and base values, p_s and p_b, are compared with two user specified thresholds s and b. When $p_s \geq s$ or $p_s \leq 1-s$, the pattern is accepted. Otherwise, if the base value is still above its threshold (i.e., $p_b \geq b$), then the pattern is considered plausible. Such a pattern still has enough tuples to be constrained to increase (or reduce) the strength value, and is recorded for further search (see Section 6 for more discussion). A pattern is discarded when $p_b < b$ and $1 - s < p_s < s$. As an example, suppose that the user specified thresholds are $b = 0.1$ and $s = 0.7$, and the following three patterns, with their base and strength values, are found in the example database in Figure 3:

$$t_1(X_2 X_1) t_3(X_4 X_2) \rightarrow t_2(X_4 X_1)\ [0.17,\ 0.7]$$
$$t_3(X_4 X_2) t_1(X_2 X_3) t_4(X5 X_3) \rightarrow t_2(X_4 X5)\ [0.02,\ 0.5]$$
$$t_2(X_4 X_1) t_1(X_3 X_1) t_4(X5 X_3) \rightarrow t_2(X_4 X5)\ [0.15,\ 0.4]$$

Among these patterns, the first one will be accepted because it has high enough strength value. The second one will be discarded because both it has a low base and yet an uncertain strength. The third one will be kept as a plausible pattern because it has high enough base although its strength is still uncertain. Notice that for any given database, users may need several trial-and-errors to find suitable thresholds.

6 Generating Metapatterns Based on Plausible Patterns

We have seen that with the initial set of metapatterns, a large set of actual patterns may be generated from the database. Some of these patterns are accepted, some discarded and some are still plausible. Interestingly, the plausible patterns provide the basis for dynamically generating more metapatterns. In particular, if a metapattern is associated with many plausible patterns, it will be used to generate more metapatterns by adding additional (meta)constraints to its left-hand side.

Adding constraints to generate new metapatterns is accomplished as follows. Given a candidate pattern, the system will add to its left-hand side a new (meta)constraint of the form $S(X, W)$, where S is a predicate

variable and W is an object variable, while X must be a variable that already exists in the pattern in order to link the constraint in. For example, one can add $S_3(Y_2, Y_3)$ to MP-6 in Figure 6 to get:

$$P_3(Y_1, Y_2) \wedge Q_3(Y_2, Y_3) \wedge W_3(Y_3, Y_4) \wedge V_3(Y_4, Y_5) \wedge S_3(Y_2, Y_3) \Rightarrow R_3(Y_1, Y_5) \qquad \text{(MP-7)}$$

or add $S_1(Y_2, O)$, where O is an object variable, to MP-4 in Figure 6 to get:

$$P_1(Y_1, Y_2) \wedge Q_1(Y_2, Y_3) \wedge S_1(Y_2, O) \Rightarrow R_1(Y_1, Y_3) \qquad \text{(MP-8)}$$

The motivation for this generation is to have the system search for an actual constraint, instantiated from S, that can yield patterns that have higher strengths.

The added meta-constraint can be instantiated to either a table that connects (i.e., share a reference name in SCT) to at least one predicate in the pattern or any of the build-in predicates (e.g., *equal*) with some variables that are already in the pattern. It is interesting to notice that the extended metapatterns enable the system to discover patterns that are beyond transitivities. For example, instantiating the metapattern MP-8, if P_1 and R_1 are bound to "ancestor", Q_1 to "parent", S_1 to "gender", and O to "male", then the following "male-ancestor" pattern may be discovered

$$ancestor(Y_1, Y_2), parent(Y_2, Y_3), gender(Y_2, male) \rightarrow ancestor(Y_1, Y_3).$$

In general, adding a new constraint to the left-hand side of a pattern will reduce the number of tuples that satisfy the left-hand side, thus the number of plausible patterns will decrease as the length of their left-hand side increases. Furthermore, we only consider to add a constraint to a pattern when it reduces the number of tuples that satisfy the left-hand side of the pattern, so the process of "finding other types of metapatterns" will eventually terminate.

7 The Automated Discovery Loop

With the metapattern generator, the discovery loop can now be fully automated to provide additional assistance to human users. Using the automated loop, users can now select and examine existing metapatterns, design and insert new metapatterns based on system feedback, and execute metapatterns at their will. If they choose, the system can be executed autonomously and be stopped anytime for further inspections.

The control algorithm for the automated discovery loop is illustrated in Figure 7. Given a relational database D, its schema S, and three thresholds o, b, and s, the system first generates a list M of all possible transitivity metapatterns, as described in Section 4. This M list is then presented to the user for examination. At this point, the user can insert, modify, and reorder the list. For example, a user can select a variable in a displayed metapattern, and a list of possible new metapatterns obtainable by instantiating that variable will be shown and ready for selection. (Recall that each variable is associated with a set of column reference names, and those in turn are linked to column names.) For example, if the user selects R_1 in MP-4 in Figure 6, then the system will show a list of more specific metapatterns as follows:

Inputs: A relational database D, its schema S, and three thresholds o, b, and s (between 0 and 1);
Outputs: Relational patterns that have strength no less than s or no greater than $1 - s$;
Procedure:
1. $M \leftarrow MPGenerator(D, S, o)$;
2. Loop
3. Order and display M;
4. Let users to examine, create, and reorder M;
5. Select m from M;
6. For each pattern p instantiated from m;
7. Computer the strength value p_s and the base value p_b,
8. If $p_s \geq s$ or $p_s \leq 1 - s$, then output p,
9. else if $p_b \geq b$,
10. then select a set C of constraints for p;
11. create new metapatterns by adding $c \in C$ to the left-hand side of p;
12. insert the new metapatterns into M;
13. Until M is empty or the user instructs to stop.

Figure 7: The Control Algorithm of the Automated Discovery Loop

$$
\begin{aligned}
P_1(Y_1, Y_2) \wedge Q_1(Y_2, Y_3) &\Rightarrow t_1(Y_1, Y_3) \\
P_1(Y_1, Y_2) \wedge Q_1(Y_2, Y_3) &\Rightarrow t_2(Y_1, Y_3) \\
P_1(Y_1, Y_2) \wedge Q_1(Y_2, Y_3) &\Rightarrow t_3(Y_1, Y_3) \\
P_1(Y_1, Y_2) \wedge Q_1(Y_2, Y_3) &\Rightarrow t_4(Y_1, Y_3) \\
&\vdots \\
P_1(Y_1, Y_2) \wedge Q_1(Y_2, Y_3) &\Rightarrow equal(Y_1, Y_3) \\
&\vdots
\end{aligned}
$$

As we can see, according to the list of transitivity relations that are derived from this general metapattern, the possible bindings for R_1 are table names t_1, t_2, t_3, and t_4 (see the table names involved in the first two cycles of predicates in Figure 5) and built-in predicates such as *equal* and others. From this new list, the user can select and insert any of them into the metapattern list M. Furthermore, the user can order the metapattern list M manually according to some criteria, or let the system order them automatically (e.g., the simple ones first).

Once the user interactions are completed, the system picks a metapattern from the list for execution. During the execution, the system examines each discovered pattern and decides if any new metapatterns should be generated (see Section 6) and inserted into the M list. The cycle of discovery continues until M is empty or the user instructs to stop.

As we can see, such a discovery system can be autonomous. If users choose not to interfere (at Line 4), the system can repeat the cycle of ordering, selecting, executing, analyzing, and generating metapatterns, until no more metapatterns are available. Of course, such an automatic cycle can always be stopped promptly if a user desires to. Nevertheless, with this control algorithm, human analysts are always informed by a list of metapatterns that are currently under investigation. This list reflects the most fruitful search directions

based on the knowledge accumulated from previous cycles of the discovery loop.

8 Demonstration with Examples

The automated discovery loop also adds values to the extant Machine Learning technologies, for it provides a domain-independent algorithm for unsupervised learning or discovery of relational patterns directly from databases. In this section, we demonstrate this capability of the automated discovery loop by showing that some of the well-known examples in the relational concept learning literature can be learned without supervision.

8.1 A Small Network Example

The first example is a small network example used by Quinlan in his FOIL system [20]. Using this example, Quinlan has shown that given a concept "canReach", and its positive and negative examples, FOIL can learn recursive definition for the concept. We, however, will use the same example, with no pre-specified concept and pre-labeled examples, to learn the same concept definition and, possibly, some other concept definitions as well.

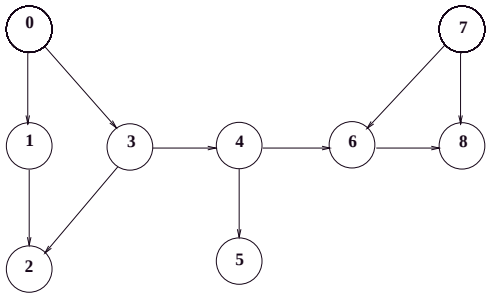

Figure 8: A small network example

The network is shown in Figure 8 and there are nine nodes, labeled from 0 to 8, and ten directed edges between nodes. If the network is represented in relational database tables, two tables are resolved, one is "linkedTo", the other is "canReach", both shown in Figure 9.

Both tables have two columns recording the nodes in the network. A row in the table "linkedTo" records a pair of nodes connected by a single arrow in the network. For example, the row (0 1) represents the fact that node 0 is connected to node 1 by a single arrow. A row in table "canReach" records a pair of nodes connected by a sequence of arrow(s). For example, the row (3 8) represents the fact that node 3 is connected to node 8 through a sequence of three arrows.

Given the data tables and schema, the automated discovery loop first checks if any pair of columns, from different tables, are significantly connected. Since all the columns are the same type, every pair is considered. Given that the significant connection threshold is set to $c = 0.6$,[1] the system finds out that

[1] The threshold 0.6 is used because it is a little above average, but not too restrict. In fact, in this domain, there is no

Schema and Data Ranges

Tables	Columns (Name Type[ValueRange])			
canReach	A_1	integer[0–7]	A_2	integer[1–8]
linkedTo	B_1	integer[0–7]	B_2	integer[1–8]

linkedTo	
B_1	B_2
0	1
0	3
1	2
3	2
3	4
4	5
4	6
6	8
7	6
7	8

CanReach	
A_1	A_2
0	1
0	2
0	3
0	4
0	5
0	6
0	8
1	2
3	2
3	4
3	5
3	6
3	8
4	5
4	6
4	8
6	8
7	6
7	8

Figure 9: The network database

SCT

	canReach	linkedTo
X_1	A1	B1
X_2	A2	B1
X_3	A2	B2

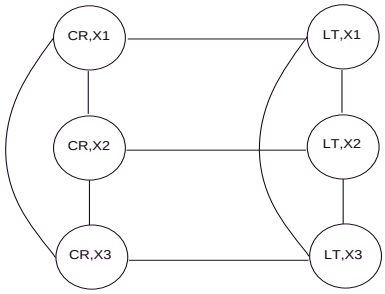

Figure 10: The SCT and G for the network database

column A1 is connected to B1, A2 is connected to B1, and A2 to B2. These connection information is then used to construct the SCT in Figure 10.

From the SCT, the automated discovery loop builds the corresponding graph as shown in Figure 10, and generates the following set of cycles of predicates:

$$\text{linkedTo}(X_1X_3)\ \text{canReach}(X_1X_3)$$
$$\text{canReach}(X_1X_2)\ \text{linkedTo}(X_2X_3)\ \text{canReach}(X_1X_3)$$
$$\text{linkedTo}(X_3X_1)\ \text{canReach}(X_1X_2)\ \text{linkedTo}(X_2X_3)$$

A set of transitivity patterns are then generated and evaluated against the database. If the base value threshold is set to $b = 0.1$ and the strength threshold $s = 0.7$, we get the same concept as FOIL:

difference for what threshold to use, except that when o is set to 0.5, one redundant pattern comes back: linkedTo(X_1, X_2) canReach$(X_2, X_4) \rightarrow$ canReach(X_1, X_4) [0.14, 1.0].

$$linkedTo(X_1 X_3) \rightarrow canReach(X_1 X_3)\ [1.0,\ 1.0]$$
$$canReach(X_1 X_2) \wedge linkedTo(X_2 X_3) \rightarrow canReach(X_1 X_3)\ [0.14,\ 1.0]$$

If we use a lower threshold, e.g., $s = 0.1$, our system has found the following patterns as well:

$$canReach(X_1 X_3) \rightarrow linkedTo(X_1 X_3)\ [1.0,\ 0.5]$$
$$canReach(X_1 X_2) \wedge canReach(X_1 X_3) \rightarrow linkedTo(X_2 X_3)\ [1.0,\ 0.1]$$
$$canReach(X_1 X_3) \wedge linkedTo(X_2 X_3) \rightarrow canReach(X_1 X_2)\ [0.63,\ 0.4]$$

The first pattern reflects the fact that half the entries in table "canReach" are the same entries in table "linkedTo". The second pattern says that if two nodes can be reached from the same node (a fork shape), then they are linked by a single arrow. At this point, however, its strength, 0.1, is low. Similarly, the third pattern is about two paths that have the same ending node.

8.2 A Family Tree Example

The second illustration of the automated discovery loop is a family tree example. It was first used by Hinton in his neural network system [9], and then used by Quinlan in his FOIL system. Again, using this example, Quinlan shown that given a concept and its positive and negative examples, FOIL can learn its relational definition. Here, we show that the concept can be learned without supervision[2].

In this example, there are two isomorphic family trees, each with twelve members, as shown in Figure 11. There are twelve relations in the trees — wife, husband, mother, father, daughter, son, sister, brother, aunt,

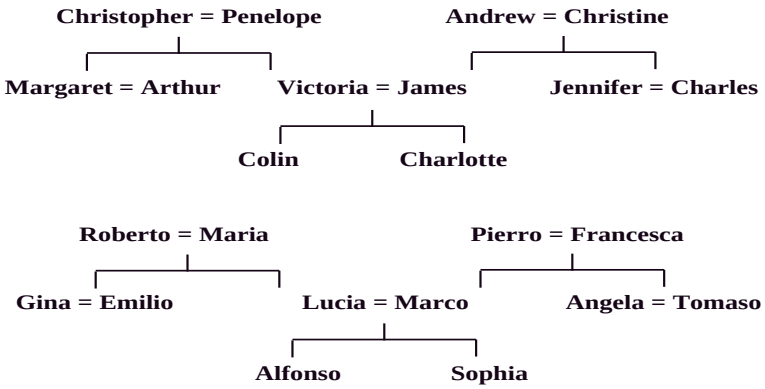

Figure 11: A family tree example, where "=" means "married to."

uncle, niece, and nephew. And two trees can be represented as twelve tables, each corresponding a relation. For example, table Wife corresponds to relation wife.

Once again, our system goes through the same procedure to construct significant connection table, to convert the table to graph, to generate transitivity patterns from the graph, and to evaluate the patterns

[2] Quinlan's paper didn't give the actual concept definitions that learned by FOIL. It reports only a better accuracy 78/80 obtained by FOIL over 7/8 obtained by Hinton's network. Since our goal is to show that relational concept definitions can be learned without supervision, we will just report the concept definitions learned by the automated discovery loop.

Schema and Data Ranges

Tables	Columns (Name Type[ValueRange])			
Wife	A_1	char(11)	A_2	char(11)
Husband	B_1	char(11)	B_2	char(11)
Mother	C_1	char(11)	C_2	char(11)
Father	D_1	char(11)	D_2	char(11)
Daughter	E_1	char(11)	E_2	char(11)
Son	F_1	char(11)	F_2	char(11)
Brother	G_1	char(11)	G_2	char(11)
Sister	H_1	char(11)	H_2	char(11)
Aunt	I_1	char(11)	I_2	char(11)
Uncle	J_1	char(11)	J_2	char(11)
Niece	K_1	char(11)	K_2	char(11)
Nephew	L_1	char(11)	L_2	char(11)

Table *Wife*

A_1	A_2
penelope	christopher
christine	andrew
margaret	arthur
victoria	james
jennifer	charles
maria	roberto
francesca	pierro
gina	emilio
lucia	marco
angela	tomaso

Figure 12: The family tree database

(with length up to 3) against the database. The system has found the definitions of all the twelve concepts that are expected, and many other concepts as well. Some of those additional concepts, 41 patterns, are true in general, such as the following:

$$husband(X_2, X_1) \rightarrow wife(X_1, X_2)\ [1.0, 1.0]$$
$$brother(X_{66}, X_{65}), niece(X_{65}, X_{78}) \rightarrow nephew(X_{66}, X_{78})\ [0.22, 1.0]$$
$$daughter(X_{56}, X_{57}) brother(X_{57}, X_{60}) \rightarrow son(X_{56}, X_{60})\ [0.33, 1.0]$$

Others (about 12 of them) are only true for this particular databases. For example,

$$brother(X_{63}, X_{64}) \rightarrow sister(X_{64}, X_{63})\ [1.0, 1.0]$$

is true here because all the families in the database have siblings of opposite gender.

8.3 A Real-World Logistic Database

This real-world logistic database has 104 tables. The largest table has 54 columns and 72,894 rows, and an average table has 16.37 columns and 5533.42 rows. Since the data in this database are collected directly from real-world applications, they contain noise and errors. To get a feeling of how our system would perform on such a real-world database, we have selected the following four tables: Table7, which has 18 columns and 932 rows, TableSum, 16 columns and 45,207 rows, TableDAF, 11 columns and 32,866 rows, and TableUIC, 14 columns and 766 rows.

An application the method described in this paper to these four tables (with the threshold for overlap set to 0.0) has found 10 pairs of columns that are significantly connected in about 56 minutes of real time on a HP730 machine. Out of these ten connections, 37 cycles have been discovered and three metapatterns (of length 2, 3 and 4) are constructed. This step took less than 2 minutes of real time. Execution of these metapatterns (the average execution time for an instantiated pattern is about 20 minutes in real time) has revealed many interesting patterns, including, for example, the following two:

Table7(NI,IU), TableUIC(IU,DAC) → TableSum(NI,DAC)

Table7(NI,IU), TableDAF(IU,DAC) → TableSum(NI,DAC)

The first pattern was known to us because it was suggested by an expert in order to provide a mapping from the feature IU of the objects in Table7 to the feature DAC of the objects in TableSum. The second pattern, however, comes as a surprise, and in fact it provides an alternative (through TableDAF instead of TableUIC) and more reliable mapping between Table7.IU and TableSum.DAC. This pattern turns out to be very useful in integrating the information in the database to a domain model that is used by information mediator. It provides the crucial mapping between the set of objects in Table7 and their superset in TableSum. Note that other experiments in this database are still in progress, but we are quite encouraged by the results obtained so far.

9 Conclusions and Future Research

This paper has presented an approach to combine a metapattern generator with an existing human-directed discovery loop in order to build an integrated data mining system that can automatically provide useful feedback and interaction for human users. Experimental results have shown that with this approach an integrated data mining system can become both interactive and autonomous.

The most significant contribution of this work is the notion of metapatterns and its role in automatically exploiting the interdependencies between induction, deduction, and human guidance. Since metapatterns are a general mechanism, we expect researchers and developers in data mining to use them to integrate their favorite deductive and inductive techniques with human guidance. Using this technology, future discovery systems can discover higher quality knowledge in a much more efficient and focused manner, they can be more productive and easier to use (i.e., users are not required to have specific knowledge about the underlying techniques of induction and deduction), and the results of the discovered knowledge will be more comprehensible to humans.

The automated discovery loop also provides an algorithm that can learn relational patterns directly from databases without humans to label the data as positive and negative for some given target concepts. This new learning algorithm advances the state-of-art of relation-based concept learning from raw data in several respects. Humans will not need to specify which hypotheses and examples to learn; nor will it be necessary to preclassify the items in databases as positive or negative. Furthermore, just like the latest developments in this field (see for example, [11]), our patterns have estimated statistical significance for reflecting the characteristics of the data (such as exceptions and noise); and our mechanism has built-in connections to real databases and can accept human guidance interactively during the discovery process.

This research has also revealed several important directions for future research. We would like to conduct more theoretical analysis and practical experiments for our unsupervised relational learning algorithm. We also like to investigate more constraints, in addition to those from the schema, the value ranges, and the previously learned patterns, on how to control the search. Constraints used by FOCL, such as multiple argument constraints and partial operational concept definitions, can be used as a start point. Furthermore,

we need to provide a more systematic and interactive way for setting the thresholds. For example, one may allow users to start with high thresholds and gradually decrease them if too few patterns are found. Finally, we are applying this automated discovery loop to large real-world databases, including the logistic database and a chemical research database from a large chemical company. We have high expectations for both these applications.

Acknowledgments

We would like to thank Yigal Arens, Jose-Luis Ambite, and Weixiong Zhang for their valuable comments and suggestions. Special thanks to Eastman Chemical Company, Motorola Inc., and USC/Information Sciences Institution for providing tasks and resources. This work is supported in part by Rome Laboratory of the Air Force Systems Command and the Advanced Research Projects Agency under Contract Number F30602-94-C-0210, in part by a Technology Reinvestment Program award funded by the Advanced Research Projects Agency under Contract Number MDA972-94-2-0010 and by the State of California under Contract Number C94-0031, and in part by the National Science Foundation under Grant No. IRI-9529615. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing the official opinion or policy of RL, ARPA, CA, NSF, the U.S. Government, or any person or agency connected with them.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[2] J.M. Aronis and Provost F.J. Efficiently constructing relational features from background knowledge for inductive machine learning. In *Proceedings of 1994 AAAI Workshop on Knowledge Discovery in Databases*, 1994.

[3] G. Bisson. Conceptual clustering in a first order logic representation. In *The Proceedings of the 10th European Conference on AI*, pages 458–462, 1992.

[4] Simoudis E., Livezey B., and Kerber R. Integrating inductive and deductive reasoning for database mining. In *Advances in Knowledge Discovery and Data Mining*, chapter 14. MIT Press, 1995.

[5] D.H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2:139–172, 1987.

[6] Y.J. Fu and J.W. Han. Meta-rule-guided mining of association rules in relational databases. In *DOOD95 Workshop on the Integration of Knowlege Discovery with Deductive and Object Oriented Databases*, Singapore, 1995.

[7] Piatetsky-Shapiro G. and Matheus C. Knowledge discovery workbench for exploring business databases. *International Journal of Intelligent Systems*, 7, 1992.

[8] J.W. Han and Y.J. Fu. Exploration of the power of the attribute-oriented induction in data mining. In *Advances in Knowledge Discovery and Data Mining*, chapter 16. MIT Press, 1995.

[9] G.E. Hinton. Learning distributed representations of concepts. In *Proceedings of the 8th Annual Conference of the Cognitive Science Society*, 1986.

[10] E.T. Jaynes. *Probability Theory — The Logic of Science.* Cambridge University Press, (To Appear) 1996. (Contact etj@howdy.wustl.edu for an online version).

[11] Ali K. and Pazzani M. Learning multiple relational rule-based models. In Fisher D. and Lenz H., editors, *Learning from Data: Artificial Intelligence and Statistics, Vol. 5.* Springer-Verlag, 1995.

[12] K. Kaufman, Michalski R., and Kerschberg L. Mining for knowledge in databases: Goals and general description of the INLEN system. In *Proceedings of the 1991 AAAI Workshop on Knowledge Discovery in Databases*, pages 35–51. AAAI Press, 1991.

[13] B. Kero, L. Russell, S. Tsur, and W.M. Shen. An overview of data mining technologies. In *DOOD95 Workshop on the Integration of Knowlege Discovery with Deductive and Object Oriented Databases*, Singapore, 1995.

[14] J.U. Kietz and K. Morik. A polynomial approach to the constructive induction of structural knowledge. *Machine Learning*, 14(2), 1994.

[15] Wim Van Laer, Luc Dehaspe, and Luc De Raedt. Applications of a logical discovery engine. In *Proceedings of 1994 AAAI Workshop on Knowledge Discovery in Databases*, 1994.

[16] Pazzani M. and Kibler D. The utility of knowledge in inductive learning. *Machine Learning*, 9(1):57–94, 1991.

[17] R.J. Mooney and M.E. Califf. Induction of first-order decision lists: Results on learning the past tense of english verbs. *Journal of Artificial Intelligence Research*, 3:1–24, 1995.

[18] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, Tokyo, Japan, 1990. Ohmsha.

[19] Naqvi and D. Tsur. A Logical Language for Data and Knowledge Bases. W. H. Freeman Company, 1989.

[20] R. J. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[21] Brachman R., Selfridge P., Terveen L., Altman B., Halper F., Kirk T., Lazar T., McGuiness D., Resnick L., and Borgida A. Integrated support for data archaelogy. *International Journal of Intelligent and Cooperative Information Systems*, 2(2):159–185, 1993.

[22] Brachman R. and Anand T. The process of knowledge discovery in databases: A first sketch. In *Proceedings of AAAI Workshop on Knowledge Discovery in Databases*, 1994.

[23] W.M. Shen. Discovering regularities from knowledge bases. *International Journal of Intelligent Systems*, 7(7):623–636, 1992.

[24] W.M. Shen. *Autonomous Learning from the Environment.* W. H. Freeman, Computer Science Press, 1994.

[25] W.M. Shen, B. Leng, and A. Chatterjee. Applying the metaquery framework to time sequence analysis. Technical report, USC-ISI-95-117, 1995.

[26] W.M. Shen, K. Ong, B. Mitbander, and C. Zaniolo. Using metaqueries to integrate and inductive learning and deductive database technology. In *Proceedings of AAAI Workshop on Knowledge Discovery in Databases.* AAAI Press, 1994.

[27] W.M. Shen, K. Ong, B. Mitbander, and C. Zaniolo. Metaqueries for data mining. In *Advances in Knowledge Discovery and Data Mining*, chapter 15. MIT Press, 1995.

[28] R. Stepp and R.S. Michalski. Inventing goal-oriented classifications of structured objects. In *Machine Learning: An Artificial Intelligence Approach, vol. II*, pages 471–498. Tioga, 1986.

[29] K. Thompson and P. Langley. Incremental concept formation with composite objects. In *Proceedings of the 6th International Workshop on Machine Learning*, pages 373–374. Morgan Kaufmann, 1989.

[30] S. Tsur, N. Arni, and K. Ong. The LDL++ user's guide. Technical report, MCC Technical Report Carnot-012-93(P), January 1993.

[31] S. Wrobe. Concept formation during interactive theory revision. *Machine Learning*, 14(2):169–192, 1994.