Efficiently Storing and Discovering Knowledge in Databases via Inductive Logic Programming Implemented Directly in Databases

A Thesis
SUBMITTED TO THE FACULTY OF
UNIVERSITY OF MINNESOTA
BY

Ravikanth Repaka

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Dr. Richard Maclin

July, 2015

Acknowledgements

I would like to take this opportunity to sincerely thank Dr. Richard Maclin for giving me an excellent opportunity to work on this thesis and to work closely with him, and also for providing knowledge, continuous support and guidance throughout the course of this thesis. I would like to thank my thesis fellow mate, Akshay Koppula for providing me an invaluable support during this course.

I would like to express my gratitude to my committee members Dr. Haiyang Wang and Dr. Steven Trogdon for evaluating my thesis and for their suggestions.

I greatly indebted to all faculty in UMD for sharing their invaluable knowledge and providing great course work. I would like to extend my appreciation to Lori Lucia, Clare Ford, Jim Luttinen and International Student Services who have been helpful throughout my two years of study.

I would like to thank to all my class friends for their encouragement and support throughout the course of my masters, and for sharing their knowledge and being with me in all stages.

Last but not least, I would like to thank my parents, brother and sister in a special way as I would not be here without their love and support; you all are the true inspiration for me.

Dedication

I would like to dedicate my thesis to my parents Krishna Rao and Lakshmi, and my brother, Satish and my sister, Ratna Devi for their unconditional love and support, without them I would never have done my masters. I would also like to dedicate my thesis to all my classmates in UMD for their continued support and encouragement.

Abstract

Inductive Logic Programming (ILP) uses inductive, statistical techniques to generate hypotheses which incorporate the given background knowledge to induce concepts that cover most of the positive examples and few of the negative examples. ILP uses techniques from both logic programming and machine learning. Research has been evolving from several years in this field and many systems are developed to solve ILP problems and most of these systems are developed in Prolog and take the input in the form of text files or other similar formats.

This thesis proposes to use a relational database to store background knowledge, positive and negative examples in the form of database entities. This information is then manipulated directly uses ILP techniques efficiently in the process of generating hypotheses. The database does the heavy lifting by efficiently handling and storing very large number of intermediate rules which are generated in process of finding the required hypotheses. The proposed system will be helpful to generate hypotheses from relational databases. The system also provides a mechanism to store the given data into database which exists in text files. Sequential covering algorithm is used to find the hypotheses which cover all positive examples and few or none of the negative examples. The proposed system is tested on real world datasets, Mutagenesis and Chess Endgame, and the generated hypotheses and its accuracy are similar to the results of existing systems which were tested on the same datasets. The results are promising and this encourages researchers to use the system in future to discover the knowledge for other datasets or in relational databases.

Contents

List of Tables

List of Figures

1 Introduction

Inductive logic programming (ILP) [1] is a field that combines principles from machine learning with the representation of logic programming with an aim to induce general rules from specific observations and background knowledge. ILP is more powerful as it uses first-order logic framework and the background knowledge. First-order logic is more expressive than the traditional attribute value and it can represent more real world complex problems. The background knowledge is an important factor in generating the hypotheses in all kinds of artificial intelligence applications. ILP concepts and techniques are implemented by a lot of existing systems and these systems can be categorized in several dimensions. These dimensions include whether the system is incremental or empirical, interactive or non-interactive, learns single or multiple predicates, and generate hypotheses from scratch or initial hypotheses, and finally generate new predicates or not.

Many ILP systems are implemented in Prolog language and each take input from a text file or other similar format and produce output in the same format. But, none of these systems consider learning knowledge from the data which exists in databases. Learning knowledge from the databases is an important application as the database contain a lot of interest data and all business applications use databases to store their data. So, creating a system that stores and discovers knowledge in relational database is useful to many people who are willing to discover knowledge in databases. The thesis aims at implementing a system which can store the given knowledge as relational database entities, facts, and examples in terms of tables; rules in terms of views. Then use database techniques in the process of generating hypotheses from the stored data. The generated hypotheses is return in the form of first-order logic and it also stored as a database entity.

The next chapter (Chapter 2) gives the background information required to understand the thesis and it is followed by the system implementation (Chapter 3) and the results (Chapter 4) which are acquired by testing the system on datasets to measure the accuracy.

Finally, the last chapter (Chapter 5) discuss about the conclusion and future work.

2 Background

This chapter discuss concepts which are needed to understand ILP in detail. It starts by defining induction and deduction, and then the concept of ILP by illustrating with different examples. Further, the basic techniques used to solve ILP problems and how different ILP systems are categorized are explained. Finally, traditional systems in ILP are discussed and comparison between those systems is shown.

2.1 Induction vs Deduction

The ILP uses induction technique rather deduction to solve the problems. Let us define induction and deduction, and see how they are different.

Deduction:

The deduction process starts with the given statements called premises and then reasons about what else would have to be true if the given premises to be held as true. For example, consider two below statements

All humans are mortal

I am human

Given this premises, a below statement can be deducted.

I am mortal

Induction:

Induction is the process of generalizing the given premises to reach conclusions that are supported by the examples but may not be true (the examples may be biased towards premises that are not true). Let us understand this concept with below example

Premises:

This car is blue.

That car is blue.

A third car is blue.

From this premises, below conclusion can be induced.

All cars are blue

This hypothesis is certainly reasonable with the given premises. However, induction does not prove that the theory is correct but rather says a complete theory when all the statements that can be derived with the theory are exist in the premises. Also, a theory is said to be consistent when all the statements in the premises are covered by the theory. In the above example, the theory *All cars are blue* is complete and consistent.

2.2 Inductive Logic Programming

Inductive logic programming [1] has been defined as the intersection of inductive learning and logic programming. It is a subfield of machine learning and uses logic programming to generate hypotheses from given logic program which contains positive examples, negative examples and background knowledge. The generated hypotheses tries to cover as many positive examples as possible without covering any negative examples (or few negative examples).

Let us illustrate the ILP and define the keywords used previously with an example as "How age, sex, cholesterol, and blood pressure are affect a person to have heart disease".

A logic program is a set of sentences which contains facts and rules. A fact is a statement which is held to be true, for example if bob's age is 40 then this fact can be represented as *age(Bob, 40)*

In this fact, *age* is called as a predicate or a relation, and *Bob* and, *40* are called its arguments. Below are the predicates which are used in this section to discuss the heart disease example.

age(person, #years) - As described above age represents *person* age in *years*
An example: *age(Bob, 40)* - states that *Bob* age is *40.*

sex(person, type) - describes *person*'s *sex* as *type* (male or female)
An example: *sex(Bob, male)* - states that *Bob* is a *male*

bloodPressure(person, value) - represents *person* blood pressure *value* in mm
An example: *bloodPressure(Bob, 130)* - states that *Bob*'s blood pressure is *130* mm

cholesterol(person, value) - represents *person*'s serum cholesterol *value* in mm/dl
Example: *cholesterol(Bob, 206)* - states that *Bob*'s cholesterol is *206* mm/dl

Before getting into rules, let us introduce variables. A variable in logic programming is different from other programming languages. It is an un intialized variable and it will be instantiated in runtime with facts representing in the dataset. This process of instantiating is called unification. Let us see an example of a variable

age(Bob, X)

Here, *X* is called as a variable and it will be instantiated with *40* as *Bob*'s age is *40* in the above given fact.

Now with this background information, a rule can be defined. A rule is a key concept in logic programming and it allow us to make conclusions in the problem domain. A rule has the form

conclusion(arg1, arg2, …., arg N) :- relation1, relation2, …. , relationM

A rule can be divided into two parts, one is called as head which comes before ':-', and the second one which is the remaining part, comes after the ':-', is called as body. The comma in body of the rule represents logical 'AND'. So this rule can be read as *conclusion* (head) is true only if all the *relations* in the body can be proven as true. Below another example of rule helps in understanding this concept.

heart-disease(Bob) :- age(Bob,40), sex(Bob, male), bloodPressure(Bob,130), cholesterol(Bob, 206)

This rule says that *Bob* has a heart disease if *Bob* is a *male*, has an age as *40*, has blood pressure level as *130*, and has cholesterol as *206*.

All the predicates in the above rule have constant values as arguments, but in general predicates can have both constants and variables as arguments. A clause, which is another important concept in logic programming, can be defined as a fact or rule. Let us move further in defining positive examples, negative examples and background knowledge.

In any ILP task the goal is to generate hypotheses for a target predicate, in our example *heart-disease* is the target predicate. Initially, some examples are given which are held to be true and, false for the target predicate and these examples will be used in finding the consistency of the generating hypotheses for the target predicate. In our example, persons who are suffering and not suffering with heart disease will be given. So the persons who have heart disease are treated as positive examples and persons who do not have heart disease treated as negative examples. For example, if *Bob* has a heart disease then *Bob* is considered as positive example, similarly if *Scott* does not have a heart disease then *Scott* is taken as negative example. Positive examples and negative examples combinedly called as Training Examples.

Finally, background knowledge is the information that is given in form of facts or rules. The final hypothesis is generated in terms of this background konwledge. The background knowledge in our example is all the facts which are given for *age*, *sex*, *bloodPressure*, and *cholesterol* predicates. Our goal is to predict the hypotheses for heart disease with the predicates mentioned in background knowledge. A sample dataset for this heart disease task is shown in table 2.1.

Table 2.1: A sample dataset for heart disease

Training examples		**Background knowledge (facts)**
Positive examples	**Negative examples**	
heart-disease(Bob)	*heart-disease(Scott)*	*age(Bob,60)*
heart-disease(Lynda)		*sex(Bob, male)*
		bloodPressure(Bob, 130)
		cholesterol(Bob, 206)
		age(Lynda,55)
		sex(Lynda, female)
		bloodPressure(Lynda, 125)
		cholesterol(Lynda, 205)
		age(Scott, 40)
		sex(Scott, male)
		bloodPressure(Scott, 140)
		cholesterol(Scott,130)

The hypothesis for the heart disease for the given dataset, mentioned in table 2.1, can be found as *heart-disease(X) :- age(X, >40), sex(X, Y), bloodPressure(X, >120),*

cholesterol(X, >200)

This hypothesis represents a person have a heart disease if age is greater than *40*, *sex* as male or female, blood pressure is greater than *120*, and cholesterol is greater than *200*. This hypothesis has two new variables *X* and *Y*, *X* will be instantiated with a person and *Y* will be instantiated with male or female. This hypothesis covers all positive examples and none of the negative examples in the dataset, so this is the required hypothesis. Consider a positive and, a negative example, and substitute those examples in the rule to the check the accuracy of this rule. As mentioned previously, the head of rule will be *True* only if all predicates in the body of rule will be evaluated as *True*.

Let us instantiate *X* with *Bob* (a positive example) and find the truthfulness of the body as follows:

Table 2.2: Checking the consistency of heart disease hypothesis for a positive example

Predicate in the body of the rule	**Fact in background knowledge**	**Result (True or False)**
age(Bob, >40)	*age(Bob, 60)*	*True*
sex(Bob, Y) :- Instantiate *Y* with *male* *sex(Bob, male)*	*sex(Bob, male)*	*True*
bloodPressure(Bob, >120)	*bloodPressure(Bob, 130)*	*True*
cholesterol(Bob, >200)	*cholesterol(Bob, 206)*	*True*

As shown in table 2.2, all the predicates in the body are evaluated as *True*. This causes the overall body of the rule as *True*, and then it results *heart-disease(Bob)*, head of the rule, as *True*. So this rule identifies positive example correctly, and similarly another positive example can also be satisfied with the hypothesis.

Now instantiate *X* with *Scott* (negative example) and find the truthfulness of the body. This is depicted as follows:

Table 2.3: Checking the consistency of heart disease hypothesis for a negative example

Predicate in the body of the rule	**Fact in background knowledge**	**Result (True or False)**
age(Scott, >40)	*age(Scott, 40)*	*False*
sex(Scott, Y) :- Instantiate *Y* with *male* *sex(Scott, male)*	*sex(Scott, male)*	*True*
bloodPressure(Scott, >120)	*bloodPressure(Scott, 140)*	*True*
cholesterol(Scott, >200)	*cholesterol(Scott, 130)*	*False*

As shown table 2.3, some of the predicates are evaluated as *False*. This causes the overall body of the rule as *False* and then that results *heart-disease(Scott)*, head of the rule, as *False*. This means the hypothesis able to predict the negative example correctly. The generated hypothesis is complete which means it covers all positive examples, and it is consistent as it does not cover any negative examples. The next section will discuss about the techniques which are used in ILP to generate hypotheses for the given dataset.

2.3 ILP Techniques

This section discusses basic ILP techniques to generate hypotheses such as generalization and specialization [2]. The topics in this section will be explained with a family relation example as given *parent, female,* and *male* relations as background knowledge and to generate hypotheses for *mother*. The relations are explained below:

mother(person1, person2) - states that *person1* is a mother of *person2*

parent(person1, person2) - states that *person1* is a parent of *person2*

female(person) - states that *person* is a female

male(person) - states that *person* is a male

The dataset for these predicates is represented in table 2.4.

Table 2.4: A sample dataset for family relation

Training examples		**Background knowledge (facts)**
Positive examples	**Negative examples**	
mother(diana,linda)	*mother(linda,ann)*	*parent(diana,linda)*
mother(ann,john)	*mother(john,scott)*	*parent(ann,john)*
		parent(scott,ann)
		parent(scott,diana)
		female(ann)
		female(diana)
		female(linda)
		male(john)
		male(scott)

The rest of the section will talk about generalization and, specialization, and the operators involved in these concepts.

2.3.1 Generalization

Generalization is the process of finding hypotheses using bottom-up approach. The

process starts from the most specific clause which covers a positive example and generalize this further until it does not cover any negative examples. Let us understand this concept with the example of family relation described above and with data presented in table 2.4.
A clause, can be a fact or rule, mentioned below satisfies a positive example.

$$mother(diana, linda)$$

This clause will be generalized as follows:

mother(diana, linda) can be represented with *parent(diana, linda)* as:

$$mother(diana, linda) \leftarrow parent(diana, linda)$$

The above rule can be modified further by including information of *female(diana)*

$$mother(diana, linda) \leftarrow female(diana), parent(diana, linda)$$

Finally, constants can be replaced with variables to further generalize the rule as:

$$mother(X, Y) \leftarrow female(X), parent(X,Y)$$

This clause is the final hypothesis for *mother*.

The following concepts (substitution and θ-subsumption) will help in defining the techniques used by ILP systems which use generalization to generate hypotheses.

Substitution: A substitution [3] is a finite set of the form $\{X1/t1,...,Xn/tn\}$ where *Xi* is a variable and *ti* is term. Substitution replaces all occurrences of *Xi* with *ti*. The substitution is usually represented with θ and if this substitution occurs in a clause *C* then it results

Cθ. Below are some examples of the substitution.

Let us consider clause $C = female(X)$ and substitution $\theta = \{X/ann\}$ then $C\theta$ becomes

$$female(ann)$$

This is obtained by replacing variable *X* with *ann*.

The below example illustrates multiple variables replaced with constants. Consider a clause $C = mother(X,Y) \leftarrow female(X), parent(X,Y)$ and substitution $\theta = \{X/diana, Y/linda\}$ then $C\theta$ is as follows:

$$mother(diana, linda) \leftarrow female(diana), parent(diana, linda)$$

This is obtained by replacing each variable of *X* with *diana* and each variable of *Y* with *linda*.

Similarly, a variable can be replaced with another variable. If clause $C = mother(X,Y) \leftarrow female(X), parent(X,Y)$ and substitution $\theta = \{Y/X\}$ then $C\theta$ becomes

$$mother(X, X) \leftarrow female(X), parent(X,X)$$

This is obtained by replacing each variable of *Y* with *X*.

θ-subsumption: A clause *C1 θ-subsumes* [4] another clause *C2* if and only if there exists a substitution *θ* such that $C1\theta \subseteq C2$. The $C1\theta \subseteq C2$ can be read as $C1\theta$ is a proper subset of *C2*. The following examples are used to illustrate this concept.

Let *C* be the clause as

$$C = mother(X,Y) \leftarrow parent(X,Y).$$

The clause *C*, θ-subsumes the below clause under empty substitution $\theta = \emptyset$

$$mother(X,Y) \leftarrow female(X), parent(X,Y)$$

Similarly, clause C, θ-subsumes the following clause with substitution $\theta = \{X/diana, Y/linda\}$

$$mother(diana,linda) \leftarrow female(diana), parent(diana,linda)$$

Also, if substitution $\theta = \{Y/X\}$ then C θ-subsumes the clause

$$mother(X,X) \leftarrow female(X), parent(X,X)$$

The *θ-subsumption* is used in ILP systems that use both generalization and specialization. The next two subsections will discuss about the basic techniques used by many ILP systems to represent generalization.

2.3.1.1 Relative Least General Generalization

The relative least general generalization (rlgg) [4] is one of the techniques in generalization and it uses least general generalization (lgg) [4]. The lgg of two clauses *C1* and *C2*, denoted by *lgg(C1, C2)*, is the least upper bound of *C1* and *C2* in the *θ-subsumption* lattice. Let *C1* is represented as set of literals as *{L1, L2,…, Lm}* and *C2* is represented as set of literals as *{M1, M2, …, Mn}* then *lgg(C1, C2)* is defined as

$$lgg(C1, C2) = \{lgg(Li,Mj) \mid Li \in C1,\ Mj \in C2 \text{ and } lgg(Li,Mj) \text{ is defined}\}$$

To compute the lgg of clauses, we need to find lgg of literals. If literal is an atom then lgg of atoms which have same predicate can be computed as

$$lgg(p(s1,\ldots,sn),\ p(t1,\ldots,tn)) = p(lgg(s1,t1),\ lgg(s2,t2),\ \ldots.,\ lgg(sn,tn))$$

For example,

$$C1 = mother(diana,\ linda) \leftarrow female(diana),\ parent(diana,\ linda)$$
$$C2 = mother(ann,\ john) \leftarrow female(ann),\ parent(ann,\ john)$$

These clauses can be written as

$$C1 = \{\ mother(diana,\ linda),\ female(diana),\ parent(diana,\ linda)\}$$
$$C2 = \{mother(ann,\ john),\ female(ann),\ parent(ann,\ john)\}$$

Then *lgg(C1, C2)* is represented with this its literals as follows

$$\{lgg(mother(diana,linda),\ mother(ann,john))\ ,$$
$$lgg(female(diana),female(ann)),$$
$$lgg(parent(diana,linda),\ parent(ann,john))\}$$

By computing lgg of literals, the above lgg(C1, C2) is written as

$$lgg(C1,\ C2) = \{mother(lgg(diana,ann),lgg(linda,john))\ ,$$
$$female(lgg(diana,ann)),$$
$$parent(lgg(diana,ann),lgg(linda,john))\}$$

By substituting *lgg(diana,ann)* with *X* and *lgg(linda,john)* with *Y* the above *lgg(C1, C2)* can be written as

$$lgg(C1, C2) = \{mother(X,Y), female(X), parent(X,Y)\}$$

Finally, the clause representation of this *lgg(C1,C2)* is as follows

$$mother(X,Y) \leftarrow female(X), parent(X,Y)$$

where X is lgg(diana, ann) and Y is lgg(linda, john)

The rlgg is an extension of lgg and it is defined as the least general generalization of two clauses *C1* and *C2* (*lgg(C1, C2)*) relative to the background knowledge.

Let us consider positive examples *e1 = mother(diana, linda)* and *e2 = mother(ann, john)* and background knowledge *B* from the family example mentioned in table 2.4. The rlgg for two examples e1 and e2 can be computed as follows:

$$rlgg(e1,e2) = lgg((e1 \leftarrow K), (e2 \leftarrow K))$$

Here *K* represents the conjunction of all literals related to background B which is

parent(ann,john),
parent(diana,linda),
parent(scott,ann),
parent(scott,diana),
female(ann),
female(diana),
female(linda).

For simplicity, the following abbreviations are used: *m-mother, p-parent, f-female, a-ann, d-diana, j-john, l-linda, and s-scott*. By having this representation, *K* can be written as

$$K = p(a,j), p(d,l), p(s,a), p(s,d), f(a), f(d), f(l)$$

The *rlgg(e1,e2)* is computed as

$$m(Va,d,\ Vj,l) \leftarrow p(a,j),\ p(d,l),\ p(s,a),\ p(s,d),\ f(a),\ f(d),\ f(l),$$
$$p(s,\ Va,d),\ p(Vs,d,\ Vd,l),\ p(Vs,d,\ Va,l),\ p(Vs,a,\ Vd,j),\ p(Vs,a,\ Va,j),$$
$$f(Va,d),\ f(Va,l),\ f(Vd,l)$$

Here *Vx,y* represents *rlgg(x,y)* for each *x* and *y*. The number of literals will be generated for the given training examples are high, so a resulting clause can be very large. To reduce these large number of literals, systems that use the rlgg will eliminate irrelevant literals and have restrictions on creating new variables. The final clause for this example is calculated as

$$m(Va,d,\ Vj,l) \leftarrow f(Va,d),\ p(Va,d,\ Vj,l)$$

which stands for *mother(X,Y)* ← *female(X), parent(X,Y)*

2.3.1.2 Inverse Resolution

Inverse resolution [5] is another generalization technique and it can be thought as inverting the resolution [6] rule of deductive inference. Resolution is a sound and complete method used in deductive systems. Inverse resolution is the process of working backwards in resolution trees to learn theories from examples and background knowledge.

Resolution uses substitutions, figure 2.1 explains a simple resolution. A clause *C* is derived from the given two clauses *C1* and *C2* by applying substitutions *θ1* and *θ2* respectively. The *C* can be called as resolvent of *C1* and *C2* and it can be denoted as *res(C1,C2)*. Also, *C1* and *C2* can be called as arm clauses, and *C* is called as base clause.

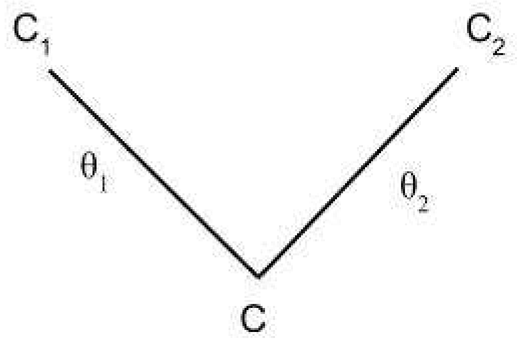


Figure 2.1: Single resolution

To better illustrate the resolution concept, consider our previous family relation data; suppose background knowledge *B* consists of two clauses *b1 = female(ann)* and *b2 = parent(ann,john)* and assume hypothesis *H = {c} = {mother(X,Y) ← female(X), parent(X,Y)}*. Figure 2.2 depicts how a fact *mother(ann, john)* can be derived from the given *B* and *H.* The resolution process for this example is done in below two steps:

- First resolvent, *mother(ann,Y) ← parent(ann,Y),* is derived from background clause *b1* and clause *c* by using substitution *θ1 = {X/ann}*.
- Second resolvent, *mother(ann, john)*, is derived from the previous resolvent and another background clause *b2* by using substitution *θ2 = {Y/john}*.

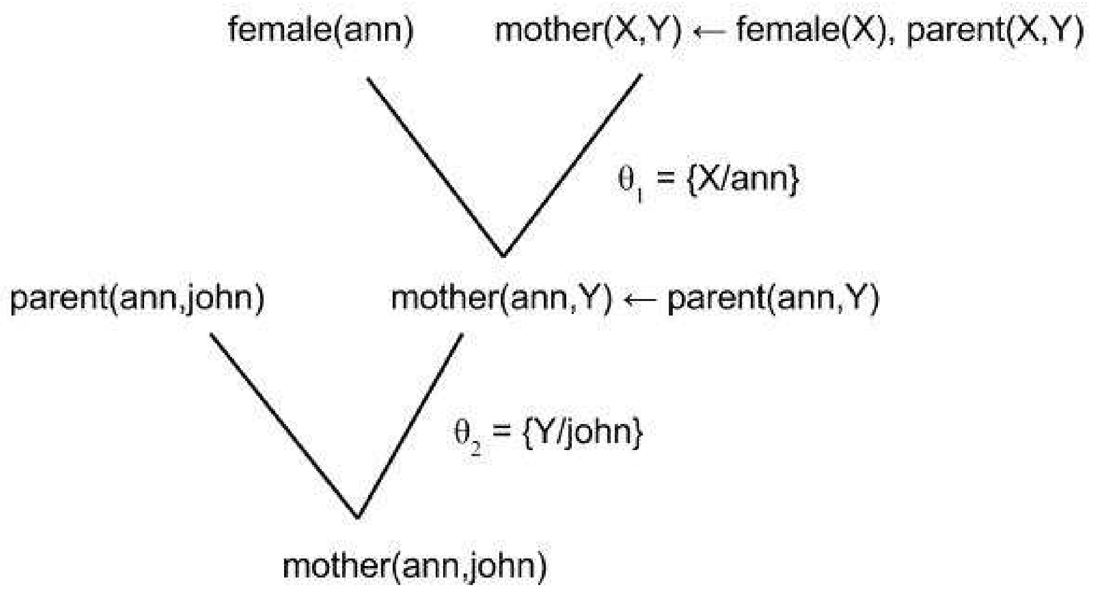


Figure 2.2: Resolution example for family relation

Inverse resolution uses inverse substitution θ^{-1} to map terms to variables. There are four major inverse resolution operators are available. One of the important operators is called as an absorption operator and this is used in ILP systems MARVIN [7] and CIGOL [5] to generate hypotheses. This absorption operator is also called as 'V' operator as it derives a clause at the arm of 'V', given clauses at the other arm and at the base. Let us take literals *C*, *C1*, and *C2* from figure 2.1, this absorption operator derives *C2* given *C1* and *C*.

Let us consider an example to see how absorption operator works for deriving a hypothesis for

$$mother(X,Y) \leftarrow female(X), parent(X,Y)$$

from the background knowledge *B* that consists of two clauses and a fact *b1 = female(ann)*

b2 = parent(ann,john)

mother(ann,john)

The hypothesis can be derived in two steps using absorption operator.

- Initial hypothesis "mother(ann,Y) ← parent(ann,Y)" is derived from the background clause "parent(ann,john)" and the fact "mother(ann,john)" and by using inverse substitution θ1-1 = {john/Y}.
- Final hypothesis "mother(X,Y) ← female(X), parent(X,Y)" is derived from the previous hypothesis and another background clause "female(ann)" and by using inverse substitution θ2-1 = {ann/X}.

Figure 2.3 depicts this process of generating hypotheses.

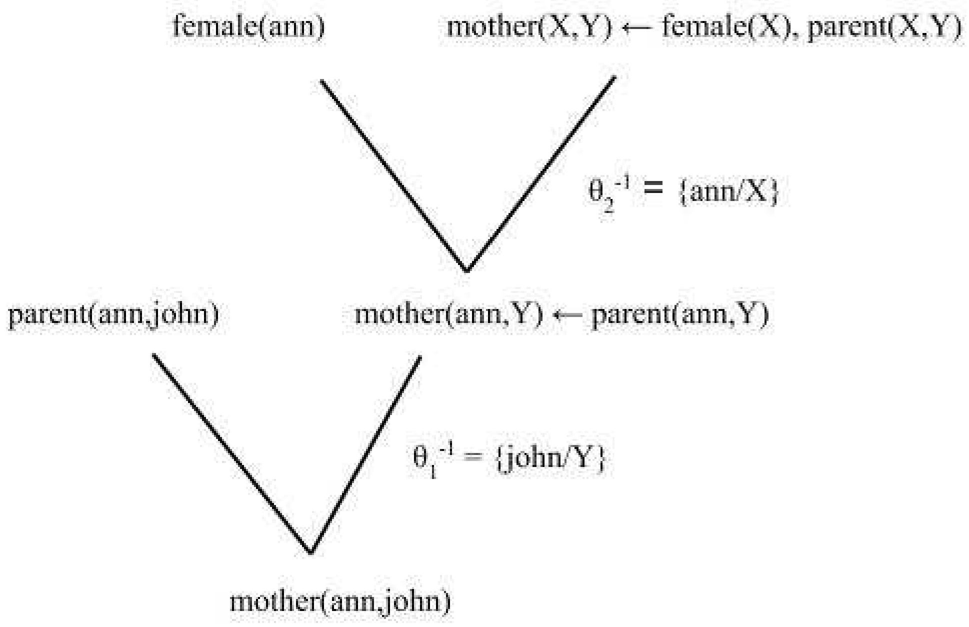

female(ann)
mother(X,Y) ← female(X), parent(X,Y)
θ_2^{-1} = {ann/X}
parent(ann,john)
mother(ann,Y) ← parent(ann,Y)
θ_1^{-1} = {john/Y}
mother(ann,john)

Figure 2.3: Inverse resolution for family relation

The other three inverse resolution operators are identification (another V operator), intra-construction, and inter-construction. The intra-construction and inter-construction operators are also called W operators and these are formed by joining two V operators. These W operators generate clauses using predicates which are not available in the initial dataset.

2.3.2 Specialization

Specialization techniques [2] use a specialization or refinement operator and search the hypothesis space in top-down approach and it starts from the most general clause and specializes the clause continuously until it no longer covers negative examples and cover atleast one positive example. Top-down search of refinement graphs is the basic ILP specialization technique. A refinement graph is a directed and acyclic graph. It is constructed in the generation of hypotheses with nodes as clauses and edges as refinement operators. The refinement operator basically does two operations on a clause - replacing a variable with a term and adding a literal to the body of a clause.

An example of refinement search graph for the family example (for generating hypothesis for *mother*) is shown in figure 2.4. The graph initially starts with most general clause "*mother(X,Y)* ← " and continues the searching by adding refinements to clauses until no negative examples are covered. The first level of refinement add clauses to the general clause such as *mother(X,Y)* ← *parent(X,Y)*, and the second level refinement *mother(X,Y)* ← *female(X), parent(X,Y)* covers all positive examples and none of the negative examples. So this final level refinement is the required hypothesis.

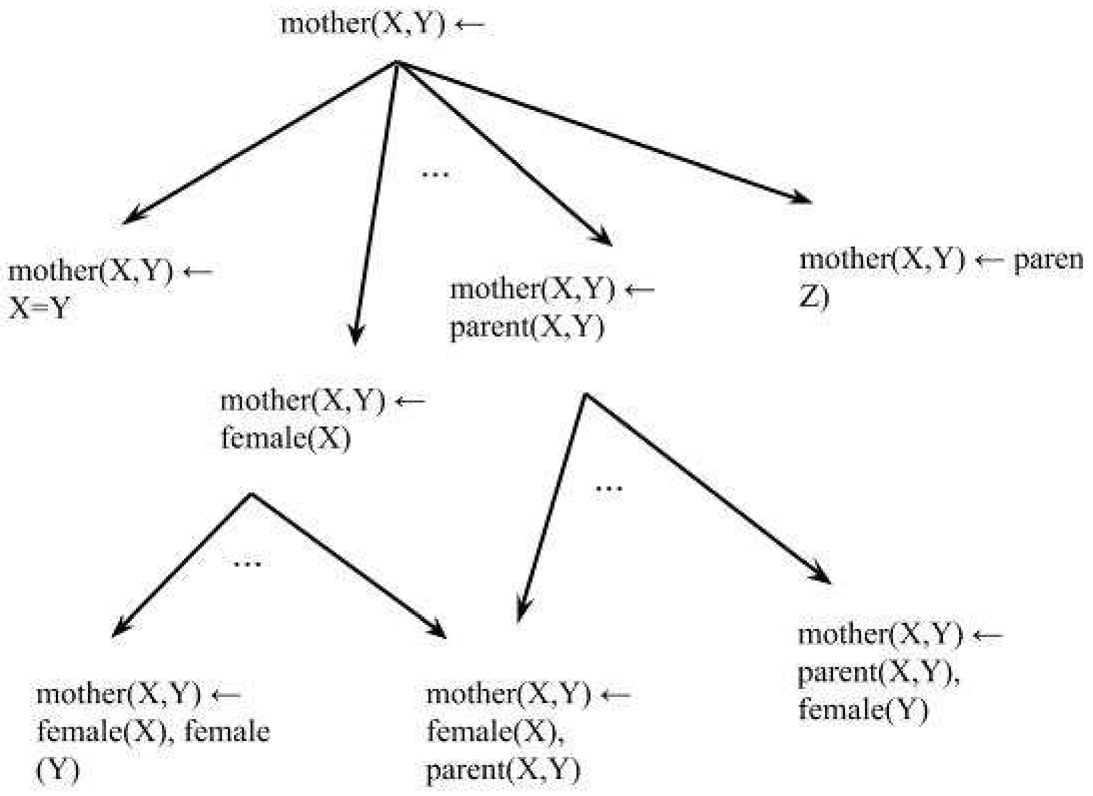

mother(X,Y) ←
...
mother(X,Y) ← X=Y
mother(X,Y) ← parent(X,Y)
mother(X,Y) ← paren Z)
mother(X,Y) ← female(X)
...
...
mother(X,Y) ← female(X), female (Y)
mother(X,Y) ← female(X), parent(X,Y)
mother(X,Y) ← parent(X,Y), female(Y)

Figure 2.4: An example of refinement search graphs

2.4 Dimensions of ILP

Dimensions of ILP describes categorization of an ILP system and there are several dimensions for these systems. First of all, a system is categorized as it can learn either a single predicate (concept) or multiple predicates. Second, all training examples are given ahead to the learning process (called as batch learner) or provide examples one by one (called as incremental learner). Third, a system is called as interactive if it rely on an oracle to validate the generalization and/or to classify examples generated by the system, otherwise system is categorized as non-interactive. Fourth, a system can either generate new predicates (terms) during the learning process to facilitate the hypothesis generation or generate the hypothesis with the only predicates given prior to the system. The final categorization as a system can accept an initial hypotheses (theory) and use it to generate the final hypothesis (called as theory revisors) or learn the concept from scratch.

All these dimensions are independent, however existing systems can be placed in two simple categories. One is called empirical ILP systems, systems which are batch learners, non-interactive, and learn single predicates from scratch. Second type is called as interactive or incremental ILP systems, which are interactive and incremental theory revisors, and learn multiple predicates.

Interactive ILP systems learn multiple predicates (concepts) from small examples and queries to the users. On the other hand, empirical ILP systems learn single predicate from large number of examples, FOIL [8] is one of the most popular systems in this class of systems.

The system implemented in this thesis is type of empirical ILP system and it is similar to FOIL. So the next section will discuss about the FOIL and its related systems.

2.5 Existing ILP Systems

This section discusses about some existing ILP systems. This section mostly focuses on systems such as FOIL [8] and its related systems, and ALEPH [9] as the current system implemented in this thesis is related to them. The current system used sequential covering algorithm to generate hypotheses and this algorithm is explained initially.

2.5.1 Sequential Covering Algorithm

Sequential covering algorithm is one of the most popular algorithms in rule learning. Rule learning is a machine learning technique that generate rules from the target examples. The sequential covering is a general to specific approach which means a rule is formed by adding literals to it in a step by step manner. The algorithm learns one rule at a time (with high accuracy, any coverage) and repeat this process gradually until all positive examples are covered by the rules. The final set of rules are disjunctive set of rules, in the sense that each rule uniquely covers some set of positive examples. The algorithm is as follows:

```
SEQUENTIAL-COVERING(Target_attributes, Attributes, Examples, Threshold):
    1. Initialize learned rules set as empty.
    2. call function LEARN-ONE-RULE(Target_attributes, Attributes, Examples) and
       store the return value as a Rule
    3. while PERFORMANCE(Rule, Examples) > Threshold do
        a. Add the Rule into learned rules set
        b. Remove the examples covered by this Rule
        c. call function LEARN-ONE-RULE again and store the return value as a
           Rule
    4. Sort the learned rules over the performance and return the result
```

```
LEARN-ONE-RULE(Target_attributes, Attributes, Examples):
    1. Initialize a rule with most general rule possible
    2. while negative examples covered by the rule, do the following:
        a. Generate all possible literals and find a best literal among them which
           gives high PERFORMANCE by adding the literal to the rule
        b. Add the best literal to the rule
    3. Return the rule which covers some portion of positive examples and none of the
       negative examples.
```

In the above function finding of a rule, beam search can be used to select a set of best literals instead of a single best literal that can be added to the rule. All the best literals are added to the rule to have set of best possible rules in current level rule and process these current best level rules to find the next best possible set of rules and finally the best rule among the best possible rules is returned. The number of best possible literals is constant throughout the execution and selects only a fixed of best literals in every level of execution, this makes it called as beam search (searching with a constant size of best possibles).

2.5.2 FOIL

First Order Inductive Learner (FOIL) [8] is a type of empirical ILP systems, which means it requires a complete set of examples before the learning process and learns a single concept from scratch and it does not require oracle to validate the hypothesis. FOIL uses sequential covering algorithm and learn rules one by one with high accuracy but with any coverage. The input for the FOIL system is represented in relations and tuples of constants belong to each relation. The system finds the target relation in horn clause format. The tuples for target relation are divided into positive and negative examples. The negative examples could be explicitly specified as part of the problem; alternatively, the closed-world assumption may be invoked to consider all tuples, other than those specified, as negative examples.

Algorithm:

The algorithm with same as sequential covering algorithm but FOIL uses information gain to identify the best literal to add to the clause. The process of finding the target relation R with k arguments is starts with empty set of clauses and the training set. A clause will be generated by adding literals to it until it covers no negative example. The clause is added to the final set of clauses for the target definition by removing positive examples covered by that clause. This process of finding new clauses will continue to happen until all positive examples in the training set are covered. The algorithm is explained in step by step as follows:

```
Initialisation:
    target relation := null
    training set := constant tuples, some labelled as positive, and negative

While positive examples exist in the training set:
  • Find a clause that satisfies part of the target relation:
      o Initialize the clause to $R(V1,V2,V3,...,Vk) \leftarrow$ .
      o Initialize a local training set $T$ with the remaining positive examples and all the negative examples.
      o While $T$ contains negative examples:
          ▪ Find a literal $L$ to add to the right hand side of the clause.
          ▪ Create new training set $T'$ from all tuples of $T$ that satisfy the literal $L$. If the literal introduces new variables then the concatenation of tuple $t$ from $T$ with new variable $b$ gives $t.b$ and add this tuple to $T'$ if it satisfies the literal $L$. This process of concatenation and adding new tuples occur for all tuples in $T$ with all new variables in $L$.
          ▪ Replace $T$ with $T'$.
      o Remove all positive tuples from training set those satisfy this clause.
      o Add this clause to the final set of clauses of target relation.
```

FOIL uses information based estimate in choosing a best possible literal from a possible set of literals. This will be calculated from information values of training set and the new training set. A training set information can be calculated as

$$I(T) = -log_2(T^+ / (T^+ + T^-))$$

Here T^+ means number of positive tuples and T^- means negative tuples in the training set T.

Similarly information contained in the new training set is also calculated by adding a literal to the clause. Finally, Information gain for the literal L is calculated as

$$Gain(L) = T^{++} * (I(T) - I(T')).$$

Here T^{++} represents the number of positive tuples are common in both the training sets T and T'. The literal with high information value will be selected to add to the clause.

FOIL has some restrictions in considering literals in the space of large available literals. A literal to be considered if it qualifies any of the below requirements

- The literal must have at least one existing variable to have some linkage to the previous literals in the clause.
- If the literal relation is same as the relation in left side of the clause then few more restrictions will be imposed to prevent problematic recursions such as infinite recursion.
- The Gain heuristic allows pruning the search space.

Example:

This algorithm and the concept of calculating information gain are explained with a family relation. The task is to find the clause for ‘grandmother’ with below information.

Background knowledge:

parent(diana,linda), parent(linda,john), parent(diana,scott), parent(scott,ann)
parent(ann,bob), female(ann), female(diana), female(linda), male(john),
male(scott), male(bob)

Positive Examples:

grandmother(diana,john), grandmother(diana,ann)

Negative Examples:

grandmother(ann,ann), grandmother(ann,bob), grandmother(ann,diana),
grandmother(ann,linda), grandmother(ann,john), grandmother(ann,scott),
grandmother(bob, ann), grandmother(bob, bob),grandmother(bob, diana),
grandmother(bob, linda),grandmother(bob, john),grandmother(bob, scott),
grandmother(diana, bob), grandmother(diana, diana), grandmother(diana, linda),
grandmother(diana, john), grandmother(diana, scott),
grandmother(linda, ann), grandmother(linda, bob),grandmother(linda, diana),
grandmother(linda, linda),grandmother(linda, john),grandmother(linda, scott),
grandmother(john, ann), grandmother(john, bob),grandmother(john, diana),
grandmother(john, linda),grandmother(john, john),grandmother(john, scott),
grandmother(scott, ann), grandmother(scott, bob),grandmother(scott, diana),
grandmother(scott, linda),grandmother(scott, john),grandmother(scott, scott),

The algorithm starts by initializing the clause as

grandmother(X,Y) ←

Some of possible literals to add are

parent(X,Y) - Both are existing variables, so it is valid literal
parent(X,Z) - It has a new variable (*Z*) but has one existing variable, so it is valid literal
parent(Y,X) - Valid literal
male(X) - Valid literal
female(X) - Valid literal
female(Z) - Only has one variable but that is new variable, so this is not valid literal

Upon the all possible valid literals, *female(X)* is selected as the best literal for first time as it has high the information gain. Let us see how the information gain is calculated for this literal. Initially, information values for both old and new clauses will be calculated and then the information gain will be calculated by using those two values.

Information for initial training set with positive examples (2) and negative examples (34) as

$$I1 = -log(2 / (2+34)) = 1.25$$

Information for training set caused by adding literal *female(X)* to the initial clause. The new clause formed as *grandmother(X,Y) ← female(X)*. This covers 2 positive examples (as grandmother is a female) and 16 negative examples (all negative examples with first argument is female). So the information with this new values as

$$I2 = -log(2 / (2+16)) = 0.95$$

The information gain of this literal *female(X)* as

$$\begin{aligned} Gain(female(X)) &= (\#\ of\ common\ positives) * (I1 - I2) \\ &= 2 * (1.25 - 0.95\ ) = 0.6 \end{aligned}$$

Similarly, information gain for the literal *parent(X,Z)* can also be calculated. In this literal *Z* means new variable and it matches any value. The clause adding by this literal, covers 2 positive examples and 22 negative examples (all negative examples with first argument is same as first argument in parent)

$$I3 = -log(2 / (2+22)) = 1.07$$

The information gain of this literal *parent(X,Z)* as

$$\begin{aligned} Gain(parent(X,Z)) &= (\#\ of\ common\ positives) * (I1 - I3) \\ &= 2 * (1.25 - 1.07) = 0.36 \end{aligned}$$

Similarly, all the remaining possible valid literals information gain will be calculated. The best literal with highest gain is selected and added to the initial clause. In this example, literal *female(X)* has the highest score and it is selected. This leads to the new clause as

$$grandmother(X,Y) \leftarrow female(X)$$

Now, the negative examples dataset is reduced to the examples which are selected by the previous clause (16 in this case). Then the process continues until it does not cover any negative examples. On the second level, the clause will be modified as

$$grandmother(X,Y) \leftarrow female(X),\ parent(X,Z)$$

Finally, a literal *parent(Z,Y)* will be added to complete to make the clause does not cover any negative examples.

$$grandmother(X,Y) \leftarrow female(X),\ parent(X,Z),\ parent(Z,Y)$$

This clause is added to the set of final rules and removes all positive examples covered by this clause. As no more positive examples are exist in the dataset, the algorithms stops here and returns the previous clause as the final hypothesis.

FOIL is tested on different kind of learning tasks in machine learning literature and results were obtained for them. The results demonstrate that FOIL is a powerful learning mechanism that uses function-free horn clauses.

In summary, FOIL has both advantages and limitations. The advantages are, FOIL adapted efficient methods from attribute-value learning systems, it uses Horn clause logic which express complex concepts and objects, it can generate recursive definitions, and it does not require an oracle. On the other hand, the limitations are: FOIL does not have any mechanism to choose if one or more literals have the same Gain heuristic information. Also, it uses greedy search approach in the selection of variables and once a literal selected from a list of literals then it does not explore other alternatives. Sometimes selecting other alternative would give better results compared to the selected literal. Additionally, the restriction on literal attributes in recursive definitions to prevent infinite recursion is incomplete. Some other techniques can be used like empirical testing to check whether trial cases could cause the issue. Finally, it requires the training set which contains both positive and negative examples prior to the learning process.

2.5.3 FOIL Related Systems

Many systems are similar to FOIL by enhancing it further and this section explains those systems.

2.5.3.1 FOCL

FOCL [10] [11] (First Order Combined Learner) is an extension of FOIL to take domain knowledge into consideration when generating clauses for the target relation. FOCL uses Empirical Learning technique and Explanation-based Learning (EBL) technique by utilizing a domain theory; thus it is an integrated learning system. By integrating different forms of background knowledge in the FOIL system leads to increase in the class of problems it can solve and increase in accuracy of results and decrease in the heuristic search space generated in the process of searching literals. FOCL uses same Gain information metric to evaluate the literals to be added to a concept. The literals can be generated in inductive manner or explanation based manner or combination of both. It extends FOIL in three ways, one extension is by allowing some constraints to reduce the search heuristic space. A second extension incorporates predicates which are defined as extensionally (i.e., predicates defined by a collection of examples) and intentionally (i.e., predicates defined by a rule instead of a collection of examples) to use as a background knowledge which is similar to domain theory in EBL. Final extension is to have a partial or incorrect definition for the target relation as an input. Choosing of whether to use inductive approach or empirical approach to select literals is based on the likelihood to producing an accurate hypothesis as measured by the information gain metric. FOCL is similar to FOIL in terms of finding target relation by reducing the covered positive examples and not covering the negative examples. But it is different from FOIL by allowing background knowledge in different ways and uses both inductive and empirical learning techniques to generate rules.

2.5.3.2 mFOIL

Another ILP system mFOIL [12] is an extension of FOIL. It improves FOIL to handle noisy data better way. The search space is reduced compared to FOIL by having constraints on variables and it is dependent on the program clauses and predicates defined in the background knowledge. A main difference in mFOIL is that it uses an accuracy based estimate such as the Laplace estimate or the m-estimate [13] instead of FOIL's information based estimate. This employs better in noisy data. Another significant change in mFOIL is achieved by replacing FOIL's greedy hill-climbing search with beam search by storing small set of best clauses found so far and updates this set in each search step. This technique increases the chance of finding a better clause and eliminates the chance of getting local maxima. Also mFOIL uses statistical significance test based on likelihood ratio to find out when the clause should be stopped for growing further and this is another change with FOIL as it uses MDT criterion.

2.5.3.3 HYDRA

HYDRA [14] is a system that was developed to generate clauses for multiple classes with noisy data. It differs from FOIL in three ways. First, it finds a probabilistic clause for each different class and then finds best clause to classify the test example of each class by considering only examples that match for the class as positive examples and treating all remaining examples as negative examples. Second, after all clauses have been learned, reliability of each individual clause will be calculated using likelihood ratios. In this calculation each class of positive examples can be divided into two splits as positive examples that are satisfied by the clause and as negative examples that are not satisfied by the clause. These positive and negative example counts are compared with total positive and negative examples count of that class to estimate logical sufficiency for each clause in each class. The clause with highest score is treated as most reliable clause. The final difference is that HYDRA uses likelihood improvement to select the next literal to be added to the clause rather than FOIL's information gain metric. It will stop adding further literal when the likelihood score does not have improvement or the current clause does not cover any negative examples. HYDRA predicts more accurately than FOIL if the input data is noisy.

2.5.3.4 GRASSHOPPER

Learning search control rules is a way of generating a plan. This process of generating plan can be viewed as selecting a partial clause to make a final clause from the available alternatives and choosing an object from the domain to include to the clause. The trial and error method can generate this plan but it has much complexity. Problem solving experience in the domain can be used to learn search control rules and thereby reducing the time in the decision making process. This additional knowledge should be made in such way that it does not increase the overall complexity in the search process, some examples in increasing the complexity are when knowledge itself is not applicable for the current decision or processing the knowledge is a complex task. Explanation-based learning (EBL) techniques have been used in learning search control knowledge but these techniques have disadvantages such as knowledge-intensive and uses a single example at a time in the learning.

GRASSHOPPER [15] uses inductive learner such as FOIL to learn search control knowledge and to eliminate the problems caused by EBL methods. It is designed to learn about goal, operator and binding decisions. The process starts by extracting interesting decisions from planning search tree and then cluster these based on both type of the decision problem and outcome of the problem such as positive or negative. It then learns a unique description for each cluster of decisions and it uses FOIL to learn this generalized description. Further, a new or updated search control will be created by combining the characterisation and advice from a cluster of decisions. Finally, it estimates the utility of new rules and decides whether new rules are to be added to the planner. The results demonstrated that time required to generate a plan in this system is much smaller than the systems used EBL methods and it is achieved this with less number of rules thus the utility of these rules are high.

2.5.3.5 MPL

Existing ILP systems both empirical and explanation based systems have not been learned rules for multiple predicates. However FOIL algorithm allows to learn multiple predicates by taking each one as an independent of another. MPL [16] is developed to learn rules for multiple predicates parallely. It also identified the problems with existing systems when multiple predicates are given as input, and those are consideration of clauses at local level, order of learning the predicates, extensional notion coverage, overgeneralization, and infinite recursion for some cases in mutual recursion. All these problems are handled in this system. This is developed based on FOIL and it learns rules parallely by considering global and as well as local consistency. The algorithm works by removing globally inconsistent and irrelevant clauses from the hypothesis and adding locally consistent clauses. This process runs in a loop and it stops when the hypothesis is globally complete and consistent. Experiments conducted on this system proved that this system was able to solve some of problems which were out of scope of other empirical systems.

2.5.4 ALEPH

A Learning Engine for Proposing Hypotheses (ALEPH) [9] is developed initially to understand the inverse entailment and then eventually included some of the functionality of existing ILP systems. ALEPH code is written in prolog and it is an open source.

The basic ALEPH algorithm has 4 steps.

- Select an example to generalize and this algorithm runs until all examples are done.
- Construct a bottom clause with literals that entails the selected example. The clause should follows the given language constrains. This step is called saturation step.
- Find a more general clause from the selected bottom clause. This step is called reduction step and it is implemented by a slight variation of branch-and-bound [17] algorithm.
- The clause found in reduction step is added to the current theory and all examples which are made redundant by this clause are removed. Return to step 1.

The system takes background knowledge and positive and negative examples in three different files and then construct the hypothesis. The background knowledge file also specifies language and search restrictions for ALEPH and these constraints are usually expressed in modes, types, and determinations.

Mode:

Mode declares the mode of call for predicates in hypotheses. This is represented in the form

mode(RecallNumber, PredicateMode)

Here, *RecallNumber* specifies number of instances to the predicate, and it can be a non-negative integer (greater than equal to 1) or * (represents no limits on the number of instances). *PredicateMode* indicates the predicate format and it is usually in the form

p(ModeType, ModeType, ……)

Here *ModeType* can be either a simple or structured type. Simple *ModeType* represents type of the variable and it could be input or output or constant type. It uses '*+*' to represent input type, '*-*' for output and '*#*' for constant. Structured type is function type that is an either simple type or structured type.

Type:

Type is another constraint and this specifies the type of the argument in each predicate.

Let us take an example of mode as below

:- mode(1,parent(+person,-person))

Here *RecallNumber* is 1 and predicate *parent* has two arguments one as input and another as output, and the type of argument is *person*.

Determinations:

These specify the predicates that can be used in construction of hypotheses. One target predicate can have many determinations. It has the below format

determination(TargetName/Arity, BackgroundName/Arity)

TargetName is predicate name of that appear in head of the clause and *BackgroundName* is the predicate name that appears in body of the clause. *Arity* is number of times the predicate can appear in the clause.

The basic Aleph algorithm can be changed by changing the parameters in the system and this let a user to have more advanced features. Some of the additional features that Aleph allow to choose are select more than one example to generalize or change search algorithm instead of default branch-and-bound or change the order of generation of rules or change evaluation function.

3 Implementation

This chapter discusses our system implementation in detail. The main goal of this system is to determine how efficiently relational databases can be used to generate ILP hypotheses for the given background knowledge and training data. It uses sequential covering algorithm and generate clauses until all positive examples are covered. The system uses MySQL as a relational database and Java as a front end to have interactions with user and the database. The system is type of empirical ILP system, means it generates hypotheses from scratch with the prior given training set, it does not need an oracle to validate the hypotheses, it learns a single predicate, and it does not generate new predicates in the process of learning.

This section starts with explaining the database representation of given background knowledge, positive and negative examples. Next, the process of generating hypotheses is explained.

3.1 Database Representation

This section explains how background knowledge such as predicates and clauses, positive examples, and negative examples are stored and represented in relational database. Also, it explains the process of storing recursive clauses which is different from the process of clauses which do not have recursion. The system takes a file which can contains both facts and rules, with restriction as a single line contains a fact or rule. If a line in the input file contains a ':-' then the line data is taken as rule, otherwise it is a fact.

3.1.1 Storing Facts

A fact is stored as in a relational table in the database. Predicates with same name but with different number of arguments can exist in the background knowledge. To uniquely identify a predicate with its number of arguments, the table name is defined as *tbl_PredicteName_NumOfArgumentsArgs*. The system assumes that column names are given as a fact in the input file, and also the fact should be the first one in a set of facts representing for a predicate. As mentioned previously, the system takes the information of facts as a text file with restriction as each line contains a single fact.

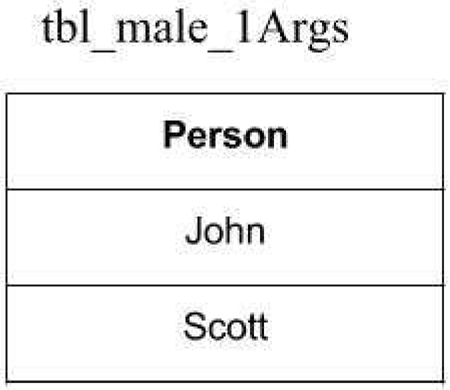
The system reads each line in the input and do the following:

- Checks whether a table (*tbl_PredicteName_NumOfArgumentsArgs*) exists in database which representing the fact.
- If table does not exist then create a new table with column names as the arguments.
- If table exists, then store the argument values of the fact as a new row in the table.

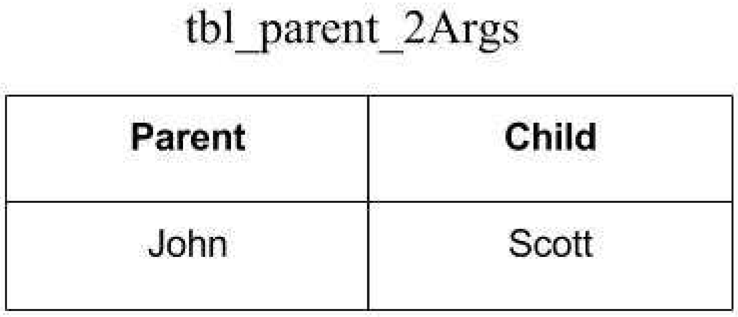
Let us consider below facts and see these are stored in database. Assume that database is empty and it does not has any data.

male(Person)
male(John)
male(Scott)
parent(Parent, Child)
parent(John, Scott)

In the above example, first line contains a fact *male(Person),* up on reading this fact, system checks whether a table with name *tbl_male_1Args* exists in the database. By getting confirmation that the table does not exist, it creates a new table with the name as *tbl_male_1Args* and with column *Person*. Further, the next line contains *male(John)*, by reading this fact, system stores *John* in the first row of the table. Continuing further, third fact will be read and stored as a second row in the same table. This process continues further, system reads the line *parent(Parent, Child)* and creates a new table *tbl_parent_2Args* with column names as *Parent and Child.* The last fact will be stored as *John* in the first column and *Scott* in the second column of *tbl_parent_2Args*. At the end, database has two tables with data as shown in figure 3.1.


tbl_male_1Args

Person
John
Scott


tbl_parent_2Args

Parent	Child
John	Scott

Figure 3.1: Relational table representation for facts

3.1.2 Storing Rules

Background information may also be given as rules and storing these rules is a crucial task. Rules are stored as views in database. A view in database is a virtual table and it stores only a query definition to generate the result set. View does not store the actual data but rather use its stored query to get data from one or more tables. The other characteristics of a view are: it takes very less space as it only store the query definition, it can join multiple tables into a single virtual table, it runs dynamically and returns data and this means even if underlying table(s) data changes there is no impact on the view definition.

Rules are better suited to be represented as views because a rule can contain multiple predicates and the result of the rule is depend on the data of the predicates in the body of the rule. This means rule result will be calculated dynamically from all the predicate tables, so views are chosen to represent the rules.

The input rule format is in horn clause logic. In implementing the algorithm to store rules as views, below cases need to be handled.

- Predicate arguments in the body of the rule can be
    - Same as the arguments of predicate in head
    - New arguments
    - Constants
- A same predicate with same number of arguments can appear more than once in the body

The input of the rule to be stored as view is given input text file. The algorithm to create a view for the rule, with considering all those above cases, is as follows:

- Find predicates in the body of the rule which do not have table representations in the database and create tables for them as explained in the previous section (3.1.1).
- Find all unique arguments in the body of rule and make a list of them.
- For each argument in this list, identify all predicates which have this argument. Also, identify the argument number in each predicate and this argument number is useful as it same as the column number in the table. Create a new list with each argument and that maps to all identified predicates of this argument, call this list as *argument-predicate* list.
- For each argument in the above *argument-predicate* list do the following:
    - Combine all predicates as a *JOIN* statement of tables. The tables are joined with a condition as selected column in each table should be equal.
    - If the selected argument exists in arguments of the predicate in the head then do the following. Select one column from one of its tables as one of the *select columns* of the query.

    - If the selected argument is a constant then create a *where condition* with column name, and its constant value.
- Add *NULL* in select column list if any arguments in the head do not appear in any predicate in the body.
- Combine *select column list*, *join statement* and *where clause* to create a final view query.
- Create a view with this generated query.

This algorithm is illustrated with four different rule examples as follows:

Example 1: (A simple rule with different predicates and with same arguments as like head predicate arguments)

mother(Mother, Child) :- female(Mother), parent(Mother, Child)

View Query:

```
SELECT table1.Person AS Mother, table2.Child AS Child
FROM tbl_female_1Args AS table1
JOIN tbl_parent_2Args AS table2
ON table1.Person = table2.Parent
```

This query can be explained as combination of two tables *female* and *parent* (specified with *JOIN* keyword) such that *female* table's first column data should match with *parent* table's first column data (specified with *ON*). Upon combining these tables, select *female* table's first column and *parent* table's second column as the final result (specified in *SELECT* list).

This query definition is stored with view name as *vw_mother2args_12_female1_parent12*. As similarly with predicate names, view name is defined to be unique. In the above view name *parent12* represents *parent* literal's two arguments are in the same order as of mother, similarly *female1* represents *female* literal's first argument matches with *parent* literal first argument.

Example 2: (A rule with a predicate appears more than once in the body and with a new argument)

grandmother(Grandmother, Grandchild) :- female(Grandmother),
parent(Grandmother, Z), parent(Z, Grandchild)

In this rule *Z* is called as a new variable as it does not exist in arguments of predicate in the head.

View Query:

SELECT table2.Parent AS Grandmother, table3.Child AS Grandchild
FROM tbl_female_1Args AS table1
JOIN tbl_parent_2Args AS table2
JOIN tbl_parent_2Args AS table3
ON table1.Person = table2.Parent AND table2.Child = table3.Parent

This query is stored as a view with name as *vw_grandmother_2args_12_female1_parent13_parent32*. In this name, *3* represents as a new argument. As seen in the query, *parent* table has two instances (*parent* predicate appears two times in the rule) and those are joined with a condition as first instance second column and second instance first column should be equal. Also, *female* table and first instance of *parent* table are joined on first column of each table. Selecting columns for the final result is an important task and in this example first column can be *female* table's first column or first instance of *parent* table's first column (*parent* table's column is shown above) and second column should be the second column of *parent* table's second instance.

Example 3: (A rule with constant arguments)

father(Father, Child) :- male(Father), parent(Adam, Child)

The view query for this rule is

```
SELECT table1.Person AS Father, table2.Child AS Child
FROM tbl_male_1Args AS table1
JOIN tbl_parent_2Args AS table2
WHERE table2.Parent = 'Adam'
```

In this query, observe that *ON* clause is not mentioned for *JOIN* statement as no common argument exist in both the predicates in body. Also, notice that constants are specified in *Where clause* as its value is fixed in the rule.

Example 4: (A rule with a missing variable in the body)

father(Father, Child) :- male(Father)

The view query for this rule is

```
SELECT table1.Person AS Father, NULL AS Child FROM
tbl_male_1Args AS table1
```

As noticed in the given rule, variable *Child* is not in the predicates of the body. The query represents this missing variable by selecting *NULL* value as a column.

3.1.2.1 Handling Recursion

A rule has a recursion when a predicate in the body is same as the predicate of the head. If a rule has recursion then it should be handled separately as it cannot be fit with the previous algorithm. The rule has two internal rules and one of them is called base case rule and another one is called recursive rule. Base case rule is the rule that is used as terminating condition for recursion and it does not have recursion in it. As the name says that recursive rule has recursion and this recursive rule will be executed until it met the basecase rule. Both these rules are combined with '+' to indicate that rule has a recursion and given as input for the system. An example of rule with recursion is as follows:

ancestor(Ancestor, Child) :- parent(Ancestor, Child) +
ancestor(Ancestor, Child) :- parent(Ancestor, Z), ancestor(Z, Child)

The above rule means that ancestor can be a parent (basecase rule), grandparent (parent-parent) or grand-grandparent, etc. (generated by joining parent with ancestor - recursive rule).

Below is the algorithm to represent recursive rule in database and this algorithm uses the previous algorithm internally. This algorithm is divided into 2 stages: Rule queries generation, creating a view.

Generating rule queries:

1. Divide the given rule based on '+' and store the first part as basecase rule and second part as recursive rule.
2. Generate a view query for basecase rule by using the previous algorithm mentioned in 3.1.2.
3. Assume a temporary table with name as recursive predicate name exists in the database. Also assume that the table initially contains the data obtained by basecase rule query.

4. Modify the recursive rule to generate its view query as
    a. Find all predicates in the body of recursive rule that appear in basecase rule body.
    b. Replace basecase rule predicate arguments with the arguments of the predicates, found in previous step, with exact order.
    c. Find a new basecase rule head for these new predicates.
    d. Replace all predicates in the body of recursive rule except recursive predicate with the predicate (found in previous step). After this step, all predicate names in recursive rule are same but with different arguments.
5. Generate a view query for the above rule with the previous algorithm mentioned in 3.1.2. This query contains the logic of recursive rule. The table name exists in this query is the recursive predicate name and which is same as the previously assumed temporary table name.
6. Create a join query to join the temporarily table with above generated query with join condition as all columns are equal. This join query select rows which are new and that does not exist in the table.
7. Store the recursive predicate name, basecase rule query, join query in a separate table.

Creating a view for the rule:

1. Create a view for retrieving the stored queries.
2. Calling the below mentioned stored procedure by passing these queries as input.

The above algorithm store queries for a recursive rule. The result of the revulsive rule is obtained by calling the stored view which internally calls the below stored procedure. The view retrieves the queries and, the recursive predicate name which are specific to the given recursive rule and pass those queries information to the stored procedure which is common to all recursive rules. This means the stored procedure has no information about the recursive rule, it just runs the given queries and return the result to the view. The stored procedure runs a loop to simulate the recursion.

Stored Procedure:

1. Create a table with recursive predicate name.
2. Run the base case rule query and store its result set in the created table.
3. Execute the join query and store the result set into the table.
4. Run the above step until it returns empty set.

Let us consider the above *ancestor* recursive and see how this algorithm finds queries for this rule.

Base case rule and its query:

ancestor(Ancestor, Child) :- parent(Ancestor, Child)

SELECT table1.Parent AS Ancestor, table2.Child AS Child
FROM tbl_parent_2Args AS table1

Recursive rule and its query: (The initial recursive rule *ancestor(Ancestor, Child) :- parent(Ancestor, Z), ancestor(Z, Child)* is modified as new recursive rule)

ancestor(Ancestor, Child) :- ancestor(Ancestor, Z), ancestor(Z, Child)

SELECT table1.Ancestor AS Ancestor, table2.Child AS Child
FROM tbl_ancestor_2Args AS table1
JOIN tbl_ancestor_2Args AS table2
ON table1.Child = table2.Ancestor

The above two queries are combined to make the below final query.

SELECT table3.Ancestor AS Ancestor, table3.Child AS Child FROM
(SELECT table1.Ancestor AS Ancestor, table2.Child AS Child

```
FROM tbl_ancestor_2Args AS table1
JOIN tbl_ancestor_2Args AS table2
ON table1.Child = table2.Ancestor) AS table3
LEFT JOIN tbl_ancestor_2Args AS table4
ON table3.Ancestor = table4.Ancestor AND table3.Child = table4.Child
WHERE table4.Ancestor = NULL
```

The final query joins a table (here *tbl_ancestor_2Args)* with itself and finds the new records based on recursive conviction that does not exist initially the table. The final query, base case query, and table name (*tbl_ancestor_2Args*) are stored in a table. A view is created to execute the stored procedure by retrieving the above stored data.

3.1.3 Representing Positive and Negative Examples

The system assumes that positive and, negative examples are stored as two different tables in database and those table names are given as input to the system. If positive and, negative examples are given as facts then we can store them in two different tables with the help of algorithm mentioned in 3.1.1.

3.2 Hypotheses Generation

By having background knowledge, positive and negative examples stored in database this section deals with how hypotheses are generated in this system. The system uses Sequential Covering algorithm, discussed in 2.5.1, and the process is also similar to FOIL algorithm (explained in 2.5.2). It generates set of hypothesis and each hypothesis covers some or all positive examples and few or none of the negative examples. This is a top-down approach and hypotheses are generated sequentially.

3.2.1 Input Format

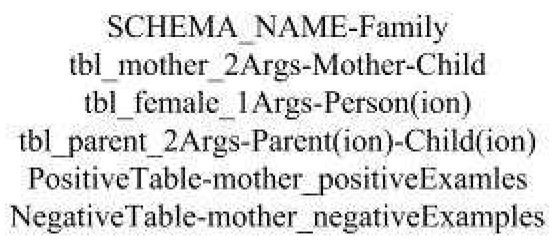
The input to generate hypotheses is described in text file and each line (except first line and last two lines) contains a table name and its column names which are separated with '-'. The first line contains the schema (database) name of which contains the given tables. The second line tells us a predicate name and with its argument names for which hypotheses need to be generated. Further lines, except the last two lines, represent table names that can be used in the body of the hypothesis clause along with column names. The order of column names is important as it matches the arguments order in the predicate. This means first argument in a predicate will get data from the first column name, and second argument will match with second column data and so on.

Each column in these tables should contain argument's mode with in the brackets after the column name. The argument's mode represents the argument type in the predicate and it can have a combination of below five individual characters

- i – input variable
- c – constant variable
- o – old variable
- n - new variable
- f – fixed column

The mode has a restriction that each character should appear only once, and 'i' and 'f' cannot be in a single mode. Next section (3.2.2) explains the mode in detail. The last two lines give information about tables which contain positive and negative examples.

A sample input file to generate hypothesis is shown in figure 3.2.



```
SCHEMA_NAME-Family
tbl_mother_2Args-Mother-Child
tbl_female_1Args-Person(ion)
tbl_parent_2Args-Parent(ion)-Child(ion)
PositiveTable-mother_positiveExamles
NegativeTable-mother_negativeExamples
```

Figure 3.2: A sample input file to generate hypothesis for *mother* predicate

In the above figure (3.2), the first line represents that the hypothesis to be generated is for *mother* with two columns. The next two lines represent table names *female* and *parent* that can be used in the body of the hypothesis clause.

3.2.2 Generation of All Possible Literals

After each argument's mode has been read by the system, the systems generates all possible literals that can be used to append the rule which is in the process of creation. The system generates rules in a similar manner of sequential covering algorithm (2.5.1). So we need a mechanism for generating all possible literals that can be added to a rule. This means generate different set of arguments for each predicate in the background knowledge.

The process of generating all possible literals is depend on the arguemnt modes which is given by user as input. The mode can be a combinations of below types.

1) **Input Argument ('i') :**

   If an argument's mode has 'i' in its one of its characters then that argument is allowed to have all input arguments. The input arguments are the arguments of the predicate in the head, and these are the column names represented in second line in the input text file. Consider an example of generating hypotheses for *mother* with using *female* as background table and *female* predicate allows only input augment, then the input will be represented as (only necessary lines shown):

tbl_mother_2Args-Mother-Child

tbl_female_1Args-Person(i)

In this example, *female* only allows one of the input arguments (*Mother or Child)* in its argument. If this information given, the only possible literals that can be generated in the system are:

female(Mother)

female(Child)

2) **Fixed Argument ('f') :**
As similar to the previous case, an argument can be fixed if its mode has 'f'. The fixed argument represents the augment is allowed with only its column name. Consider an example, similar to the above, of generating hypotheses for *mother* with using *female* as background table and *female* predicate allows only fixed augment, then the input will be represented as (only necessary lines shown):

tbl_mother_2Args-Mother-Child

tbl_female_1Args-Person(f)

In this example, *female* only allows *Person* in its argument. This means only below possible literal is possible in this example.

female(Person)

The system has a restriction on argument's mode as a it should not be defined as both input and fixed arguments.

3) New Variable ('n') :

If an argument is allowed to be a new variable then it should be defined with 'n' in the mode. This new variable represents that the value can match with any value in that particular column in the dataset. Initially, new arguments are represented with *NEW* and then those will be replaced with *newVar#* before the literal is added to a rule. The number in the *newVar#* starts with 1 and increments it sequentially by 1. For example,

tbl_mother_2Args-Mother-Child
tbl_parent_1Args-Parent(i)-Child(n)

In this example, *parent* predicate is allowing first argument as type of input argument (one of the *Mother* or *Child*) and second argument as type of new variable. The below possible literals can be generated for this example:

parent(Mother, NEW)
parent(Child, NEW)

These will be replaced later as follows:

parent(Mother, newVar1)
parent(Child, newVar2)

In generating all new variables, system has a restriction of generating all new variables for all arguments in a predicate. The restriction is because, the predicate with all new variables can match all rows in a table as it does not have any match with arguments in the rule.

4) **Old Variable (‘o’):**

Old argument can be set with ‘o’ and it means the argument can have any previous new arguments used in the rule. If no previous new variables are exist in the rule then the current literal will be ignored. Consider the below example if input file:

tbl_mother_2Args-Mother-Child
tbl_parent_1Args-Parent(i)-Child(on)

In this example, *parent* predicate can have new variable in its first argument, and input augment or old variable in its second argument. Initially old variables are represented with *OLD* and those will be replaced later. The below possible literals can be generated for this example:

parent(Child, NEW)
parent(Mother , NEW)
parent(Child, OLD)
parent(Mother, OLD)

We need to have a rule to replace the *OLD*. Consider current rule as

mother(Mother,Child) ←

Initially, the first two literals can be added to the above rule to make new rules but the last two literals (*parent(Child, OLD)* and *parent(Mother, OLD)*) cannot be added as there no previous new arguments in the rule (before adding this current literal). To add the first two literals to the above rule, *NEW* has to be replaced with *newVar#*. The new literals are as follows

parent(Child, newVar1)

parent(Mother, newVar2)

By adding those two literals new rules can be formed as below:

mother(Mother,Child) ← *parent(Child, newVar1)*
mother(Mother,Child) ← *parent(Mother, newVar2)*

Further, these two rules can be extended by adding the initial four literals. Consider a single rule above *mother(Mother,Child)* ← *parent(Child, newVar1)* and this can be extender further as follows:

mother(Mother,Child) ← *parent(Child, newVar1), parent(Child, newVar3)*
mother(Mother,Child) ← *parent(Child, newVar1), parent(Mother, newVar4)*
mother(Mother,Child) ← *parent(Child, newVar1), parent(Child, newVar1)*
mother(Mother,Child) ← *parent(Child, newVar1), parent(Mother, newVar1)*

Notice that *OLD* is replaced with *newVar1* as it the previous new variable in the rule. As explained in the previous, after replacing *OLD* with new variables if a literal has all new variables that the literal will be ignored.

Below are the all new possible literals that can be generated for the example mentioned in figure 3.2.

parent(Mother, Child)
parent(Child, Mother)
parent(Child, NEW)
parent(Child, OLD)
parent(OLD, Child)
parent(NEW, Child)
parent(Mother, OLD)

parent(Mother, NEW)

parent(OLD, Mother)

parent(NEW, Mother)

parent(OLD, OLD)

parent(OLD, NEW)

parent(NEW, OLD)

female(Mother)

female(Child)

female(OLD)

Notice that literals *female(NEW)*, and *parent(NEW, NEW)* are not in the above list as they are not valid because they contain all arguments as type of new.

Dealing with constants:

Predicates can also allow constants in its arguments. The predicate which allow constants is represented with '*c*' in the argument's mode of the column. For example, if the parent table allow constants on its first column then the parent table representation in the input format would be as follows:

tbl_parent_2Args-Parent(ion)-Child(ionc)

To create rules with constants, the system creates a new table with columns that are represented as constants. The table data is filled with all unique combinations of constants that are possible with those columns data. This data is pulled from table and then literals are created with all values in the data. Each constant value is prepended with *const_* keyword to identify that an argument in a literal is a constant.

3.2.3 Algorithm to Generate Hypotheses

The algorithm is a sequential covering approach and learns each rule one by one. It does this process sequentially.

- Create a temporary positive table and insert positive table data into this table
- Do the following until any tuples exist in the temporary positive table
    - Execute LEARN-ONE-RULE algorithm.
    - Store the return rule from the above algorithm into final set of rules.
    - Generate a rule query from the above rule with the help of algorithm mentioned in 3.1.2.
    - Remove records from temporary positive table which are selected from the above rule query.
    - Repeat.

LEARN-ONE-RULE:

This is algorithm is called with an empty body clause. It generates a best rule which covers all or some of positive examples and none or few of negative examples. This algorithm stores all intermediate generated rules in a table and finally picks the best rule from those rules. The algorithm is as follows:

- Generate all possible arguments of predicates as mentioned in 3.2.2.
- If constants are allowed then generate all possible arguments with constants. Combine these predicates with predicates generated in previous step.
- For each rule in current level do the following:
    - Append each predicate from the generated predicates to create a new rule.
    - Generate a view query for this rule by using the algorithm mentioned in 3.1.2.
    - Create new queries to join the above query with positive table and negative table.
    - Execute these queries to get positive and negative scores.

- Store the rule and its positive and negative scores in a table if positive score is greater than zero.
- If the negative score is zero and the positive score is same as the remaining positive examples in the positive table then stop processing further as it is the required rule.

- If some additional criteria is given such as maximum depth of a clause (number of predicates) then check whether the given criteria is satisfied, if so stop processing further and return the best rule found so far.
- Else, Pick the top best rules of newly added and treat them as current level rules. The criteria to pick the top best score rules is a rule with highest value of positive score plus negative score. If more than one rule have the same score then pick a rule with low negative value. By default top 10 best rules are selected for the next level and this simulates the beam search technique.
- Repeat above three steps until a best rule found. Finally return the best rule.

Example:

Let us illustrate this algorithm with an example for generating hypothesis for 'mother', with two arguments, from 'parent' and 'female' predicates. Below tables, from 3.1 to 3.4, represent background data, and positive and, negative tables.

Table 3.1: Background knowledge table representation for *female*

Person
Ann
Diana
Linda

Table 3.2: Background knowledge table representation for *parent*

Parent	**Child**
Diana	Linda
Ann	John
Scott	Ann
Scott	Diana

Table 3.3 Table representation for positive examples of *mother*

Mother	**Child**
Diana	Linda
Ann	John

Table 3.4 Table representation for negative examples of *mother*

NotMother	**Child**
Ann	Ann
Ann	Diana
Ann	Linda
Ann	Scott
Diana	Ann
Diana	Diana
Diana	John

Diana	Scott
Linda	Ann
Linda	Diana
Linda	Linda
Linda	John
Linda	Scott
John	Ann
John	Diana
John	John
John	Linda
John	Scott
Scott	Ann
Scott	Diana
Scott	Linda
Scott	John
Scott	Scott

Initially the hypotheses is empty and LEARN-ONE-RULE function is called with the above tables' data. It generate all possible valid arguments for the background predicates and create new rules as mentioned in table 3.5.

Table 3.5 Rules generated in first level

Possible New Literal	**Rule**	**Positive Score**	**Negative Score**
parent(Mother, Child)	*mother(Mother, Child) :- parent(Mother, Child)*	*2*	*2*
parent(Mother, newVar1)	*mother(Mother, Child) :- parent(Mother, newVar1)*	*2*	*8*
parent(newVar2, Mother)	*mother(Mother, Child) :- parent(newVar2, Mother)*	*2*	*10*
parent(Child, Mother)	*mother(Mother, Child) :- parent(Child, Mother)*	*0*	*2*
parent(Child, newVar3)	*mother(Mother, Child) :- parent(Child, newVar3)*	*0*	*10*
parent(newVar4, Child)	*mother(Mother, Child) :- parent(newVar4, Child)*	*2*	*8*
female(Mother)	*mother(Mother, Child) :- female(Mother)*	*2*	*8*
female(Child)	*mother(Mother, Child) :- female(Child)*	*1*	*10*

All these rules with those scores are stored in database except two rules, *mother(Mother, Child) :- parent(Child, Mother)* and, *mother(Mother, Child) :- parent(Child, newVar3)*, as they have positive score as zero.

As mentioned in the above algorithm, rule query is joined with positive and negative tables to calculate positive and negative scores. Let us consider the first rule (*mother(Mother, Child) :- parent(Mother, Child)*) from the above table and discuss how these queries are formed.

Rule query:

```
SELECT table1.Parent AS Mother, table1.Child AS Child
FROM tbl_parent_2Args AS table1
```

This query is joined with positive and negative tables.

Positive score query:

```
SELECT COUNT(*) FROM mother_positiveExamples AS positiveTable JOIN
(SELECT table1.Parent AS Mother, table1.Child AS Child
FROM tbl_parent_2Args AS table1) AS ruleQuery
ON positiveTable.Mother = ruleQuery.Mother
AND positiveTable.Child = ruleQuery.Child
```

Negative score query:

```
SELECT COUNT(*) FROM mother_negativeExamples AS negativeTable JOIN
(SELECT table1.Parent AS Mother, table1.Child AS Child
FROM tbl_parent_2Args AS table1) AS ruleQuery
ON negativeTable.NotMother = ruleQuery.Mother
AND negativeTable.Child = ruleQuery.Child
```

Based on the sum of positive and negative scores, best rules are selected from the above generated rules. These rules are in order:

mother(Mother, Child) :- parent(Mother, Child)

mother(Mother, Child) :- parent(Mother, newVar1)

mother(Mother, Child) :- parent(newVar4, Child)

mother(Mother, Child) :- female(Mother)

mother(Mother, Child) :- parent(newVar2, Mother)

mother(Mother, Child) :- female(Child)

As all these best rules are still covering few negative examples, LEARN-ONE-RULE algorithm is called for each best rule to extend those rules further. The first best rule, *mother(X,Y) :- parent(X,Y)*, is extended by appending all possible predicates to it and all the possible rules generated in the next level shown in table 3.6.

Table 3.6 Rules generated in second level

Possible New Literal	Rule	Positive Score	Negative Score
parent(Mother, Child)	mother(Mother, Child) :- parent(Mother, Child), parent(Mother, Child)	2	2
parent(Mother, newVar5)	mother(Mother, Child) :- parent(Mother, Child), parent(Mother, newVar5)	2	2
parent(newVar6, Mother)	mother(Mother, Child) :- parent(Mother, Child), parent(newVar6, Mother)	2	2
parent(Child, Mother)	mother(Mother, Child) :- parent(Mother, Child), parent(Child, Mother)	0	0
parent(Child, newVar6)	mother(Mother, Child) :- parent(Mother, Child), parent(Child, newVar7)	0	2
parent(newVar8, Child)	mother(Mother, Child) :- parent(Mother, Child), parent(newVar8, Child)	2	2
female(Mother)	mother(Mother, Child) :- parent(Mother, Child), female(Mother)	2	0
female(Child)	mother(Mother, Child) :- parent(Mother, Child), female(Child)	1	2

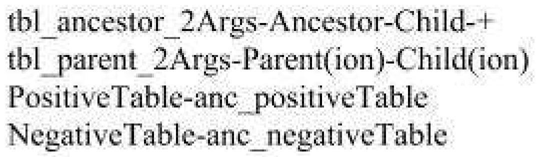
Similarly all other rules are extended by appending new predicates. Once all these second level rules are stored in database, best rules are found again among these newly generated rules. The best rule among the second level rules is *mother(Mother, Child) :- parent(Mother, Child), female(Mother)*

which covers all positive examples and none of the negative examples. This rule will be returned as the final best rule for the given mother predicate. By deleting the covered examples of above rule from the positive table, empty set results in the table and thereby it causes to terminate the program without calling the LEARN-ONE-RULE algorithm further.

3.2.3.1 Dealing with Recursion

Generation of hypothesis involving recursion is similar to the above algorithm and it uses above algorithm to generate base case. The base case rule is the rule which covers some of the positive examples and none of the negative examples. Once a base case rule is generated, all different possible arguments are created for the predicate in head of clause and appended with base case rule to create new recursive rules. Query for the each recursive rule is generated from the algorithm mentioned in 3.1.2.1 and it is joined with positive and negative tables to calculate scores. The best rule which covers all positive and none of the negative examples is chosen as the required hypothesis.

The input file contains a '+' sign at the end of the first line to represent that generating hypotheses involves recursion. Figure 3.3 shows the input file for the ancestor. By identifying this flag, recursion algorithm will run to generate the hypotheses.



```
tbl_ancestor_2Args-Ancestor-Child-+
tbl_parent_2Args-Parent(ion)-Child(ion)
PositiveTable-anc_positiveTable
NegativeTable-anc_negativeTable
```

Figure 3.3: A sample input file to represent recursion

3.2.4 Improvements

In the implementation of this system few improvements are added to speed up the hypothesis generation.

- Multithreading has been implemented by distributing current level rules to different processes and extend each current level rule in each process simultaneously. Connection pool is also configured to limit the number of connections to be used across all processes and to reuse the idle connections.
- Indexing is created for each column that can be used in the body of the clause to improve the running time for queries. As each query nearly involves join with other table this indexing on columns has increased the overall runtime very drastically.

4 Results

This chapter presents results obtained with this system on two different tasks that are tested with many ILP systems. The proposed system got similar accuracies compared to the other existing systems and acquired the required hypotheses for each task. Apart from the learning tasks mentioned in this chapter, this system is also tested to learn family relations with manually synthesized data for *parent*, *female*, and *male* tables.

Hypothesis for *mother*:

mother(Mother, Child) ← parent(Mother, Child), female(Mother)

Hypothesis for *grandmother*:

grandmother(Grandmother, Grandchild) ← parent(Grandmother, newVar259),
parent(newVar259, Grandchild), female(Grandmother)

A recursion rule for *ancestor* is also learned as follows:

ancestor(Ancestor, Child) ← parent(Ancestor, Child)
ancestor(Ancestor, Child) ← parent(Ancestor, newVar2392),
ancestor(newVar2392, Child)

4.1 Mutagenesis

The task in this mutagenesis data set is to find mutagenicity of 230 drug compounds. Finding the mutagenesis is an important task as it can help in understanding and prediction of carcinogenesis which can cause damage to DNA. Each compound contains bond and atom information and the task is to learn the mutagenesis in terms of bond and atom. Progol system [18] generated the hypotheses for this task. The goal of this experiment is to generate hypotheses and to compare its accuracy with progol system output.

The structure information of bond and atom are given as prolog facts with following forms.

bond(compound, atom1, atom2, bondtype) - states that the compound has a bond of bondtype between the atoms atom1 and atom2.

An example: *bond(d2, d2_1, d2_2, 7)*

atm(compound, atom, element, atomtype, charge) - states that compound has an atom with an element as element and with type as atomtype and with charge as charge.

An example: *atm(d2,d2_1,c,22,0.067)*

Total of 12203 facts were generated for all the compounds to represent structure information both bond and atom. Out of the 230 compounds, 138 have positive levels of log mutagenicity (as mentioned in [19]) and labeled as active and taken as positive examples. The remaining 92 are labeled as inactive and taken as negative examples. The data is further categorized as two sets such as regression friendly and regression unfriendly. The system tested on regression friendly data which contains information of 188 compounds of which 125 are positive and, 63 are negative examples.

Language Specification:

Progol system restricts the language with few conditions to generate hypotheses for this dataset. The current system is also uses the same language speciation to generate hypotheses and to have valid comparison between the systems. The specifications are mentioned for each argument in both bond and atom predicates. The specifications are

- atom should have a new variable in column1, a new or an old variable in column2, constants in column3 and column4, a constant in column5 if there is only one predicate or a old variable in column5.
- bond should have a new variable in column1, an old or a new variable in column2, an old or a new variable in column3, a constant in column4.

Below is the input file given to the system by considering all the above constrains for this task.

SCHEMA_NAME-mutagenesis
tbl_act_2Args-compound
tbl_atm_5Args-compound(f)-atom(fon)-element(c)-atomType(c)-charge(cn)
tbl_bond_4Args-compound(f)-atom1(fon)-atom2(fon)-bondType(c)
PositiveTable-mutagenesis_positive
NegativeTable-negativeTable

It represents predicate atom has following arguments: *compound* is a fixed argument, *atom* can have a fixed argumetn or an old variable or a new variable, *element* and, *atomType* are constants, and the last argument *charge* is a constant or new variable. Similarly, arguments in *bond* have following constraints: *compound* is fixed argument, *atom1* and *atom2* can have a fixed or an old variable or a new variable, the last argument *bondType* is a constant.

The system generated following hypotheses with data of 188 compounds. For the sake of readability, new variables in the hypotheses are replaced with capital letters (from *B* to *Z*), and *A* is being used to represent *compound*.

1. *act(A) :-bond(A,B,C,1), atm(A,B,c,22,D), atm(A,C,c,10,E)*
   Accuracy = 88   Coverage = 28
2. *act(A) :-bond(A,B,C,1), atm(A,D,c,27,E), bond(A,C,D,7)*

Accuracy = 86   Coverage = 19

3. *act(A) :-atm(A,B,o,40,-0.384)*

   Accuracy = 79   Coverage = 9

4. *act(A) :-atm(A,B,h,3,0.144)*

   Accuracy = 89   Coverage = 6

5. *act(A) :-atm(A,B,h,3,0.142)*

   Accuracy = 89   Coverage = 6

6. *act(A) :-atm(A,B,h,3,0.147)*

   Accuracy = 83   Coverage = 4

7. *act(A) :-atm(A,B,c,14,C), bond(A,D,B,1)*

   Accuracy = 67   Coverage = 5

8. *act(A) :-bond(A,B,C,1), bond(A,C,D,7), atm(A,B,n,32,E)*

   Accuracy = 67   Coverage = 5

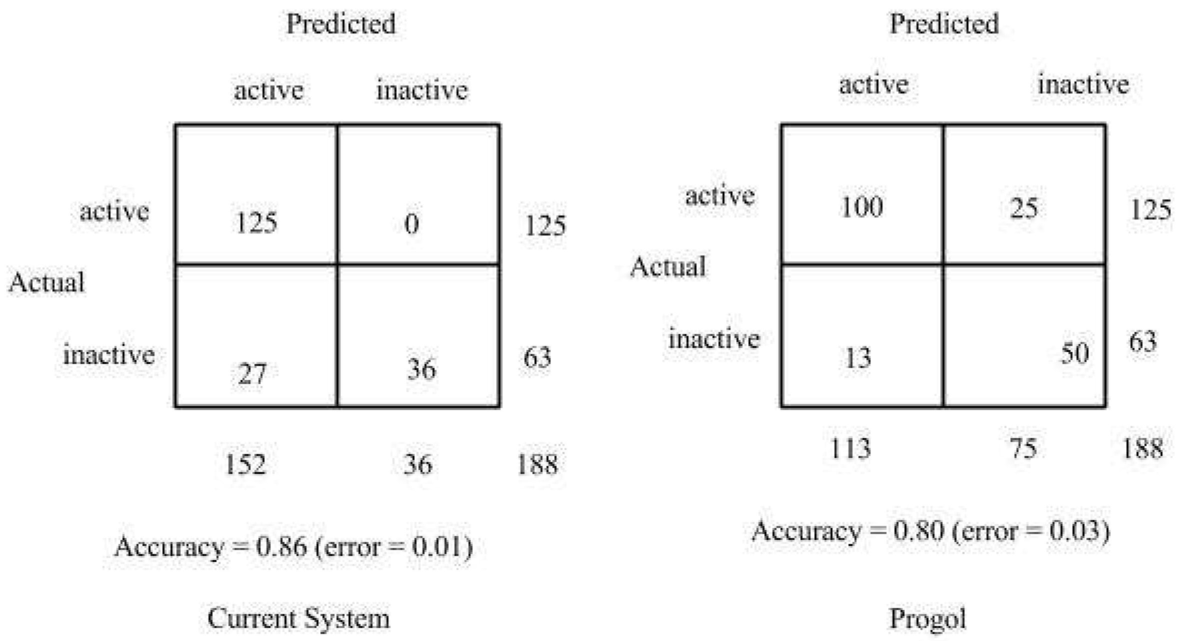
9. *act(A) :-atm(A,B,c,195,C)*

   Accuracy = 100   Coverage = 3

10. *act(A) :-atm(A,B,h,3,0.149)*

    Accuracy = 80.0   Coverage = 3.2

The above system generated hypotheses are similar to as the progol, 6 rules among these 10 rules are same as the progol rule and the remaining 4 rules are similar. The above hypotheses are subset of actual hypotheses and each hypothesis coverage is greater than 3%. The remaining set has 10 rules with each of it coverage less than 3. In overall, total hypotheses cover entire positive examples in the given dataset and it represents that it has 100% coverage. Also, as each hypothesis cover some of the negative examples, over all accuracy is less than 100%. Figure 4.1 shows the comparison of accuracies between current system and progol. By comparing the results, it is noticeable that system presented in this thesis is feasible and comparable to the progol system.



Current System

		Predicted active	Predicted inactive	
Actual	active	125	0	125
	inactive	27	36	63
		152	36	188

Accuracy = 0.86 (error = 0.01)

Progol

		Predicted active	Predicted inactive	
Actual	active	100	25	125
	inactive	13	50	63
		113	75	188

Accuracy = 0.80 (error = 0.03)

Figure 4.1: Accuracy comparison between current system and Progol

4.2 Chess Endgame

The chess endgame learning task is to predict how white wins over black given that White King, White Rook, and Black King are exist in the chess board. This dataset is also called as KRK (King Rook King). The task generate rules for how chess piece locations determine the number of moves for Black King to take for White to win or draw. The number of moves can be in between 0 to 16, so the main task is divided into subtasks such that each subtask will generate rules for each depth of move. The format of the input is given as seven arguments, such as

arg1,arg2,arg3,arg4,arg5,arg6,arg7

arg1 and, *arg2* represents position of White King in column and row values; *arg3* and, *arg4* specifies position of White Rook in column and row values; further, *arg5* and, *arg6* represents the position of Black King in column and row values. Finally, *arg7* represents the depth of win for White from 0 to 16 moves, otherwise draw.

Each column value is between *a* to *h* and each rowvalue is in between *1* to *8*. Consider below facts from the dataset.

chess(a,1,b,3,c,2,draw)

This fact says that when the White King is at position *a1*, White Rook is at *b3*, Black King at *c2* then it results in a draw.

Consider another fact.

chess(d,3,h,1,d,1,zero)

This fact says that when the White King is at position *d3*, White Rook is at *h1*, and Black King at *d1* then White has won the game without moving Black King further.

In the current system, the first six arguments are stored as background information about positions and depending on the depth-of-win to which hypothesis is to be generated, those positions are taken as positive examples and remaining all positions are taken as negative examples. For example, zero depth-of-win positions will be positive examples and remaining all positions are as negative examples if the hypothesis is to be generated is for zero depth-of-win.

The results obtained by the current system are compared with the results generated by Bain [20]. The system can produce rules for all kinds of depths but rules for depth of zero and one are compared as these results are given Bain thesis. The input for this dataset is as defined as follows by considering the constrains in this dataset.

SCHEMA_NAME-Chess

tbl_zero_6Args-whiteKingColumn-whiteKingRow-whiteRookColumn-whiteRookRow-blackKingColumn-blackKingRow

tbl_chess_6Args-whiteKingColumn(con)-whiteKingRow(con)-whiteRookColumn(con)-whiteRookRow(con)-blackKingColumn(con)-blackKingRow(con)

PositiveTable-positiveTable

NegativeTable-negativeTable

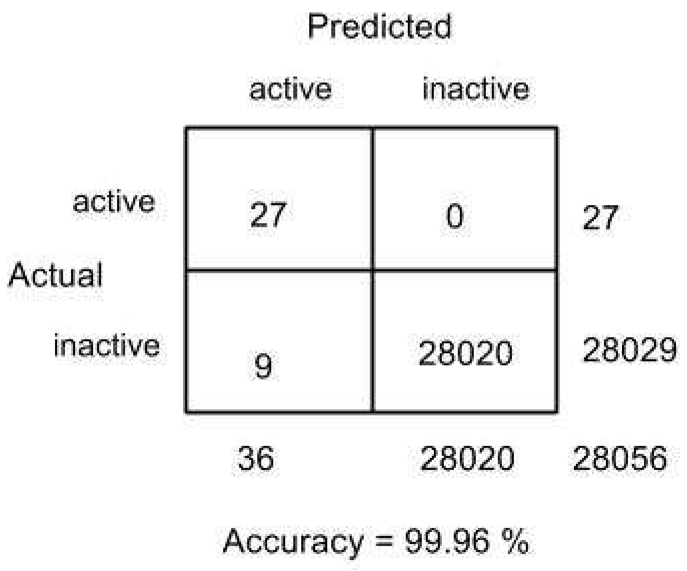
The table *tbl_chess_6Args* is defined with six columns as *whiteKingColumn, whiteKingRow, whiteRookColumn, whiteRookRow, blackKingColumn, blackKingRow.* Each argument in the predicate can be a constant or a new variable or an old variable. Hypotheses generated for both depth zero and one are shown below.

Depth Zero Hypotheses:

The depth zero position means white wins the game without any further moves done by black king. It has 27 positive examples and which are covered using below 3 rules. For the sake of readability, new variables in the hypotheses are replaced with capital letters (in between *G* to *Z*), and letters *A* to *F* are used to represent six columns respectively.

- zero (A, B, C, D, E, F) :- chess(c, G, a, H, a, 1)
    - Accuracy: 85.71%, Coverage: 44.44%
- zero (A, B, C, D, E, F) :- chess(G, 3, H, 1, I, 1)
    - Accuracy: 71.43%, Coverage: 37.04%
- zero (A, B, C, D, E, F) :- chess(c, G, a, H, a, I)
    - Accuracy: 62.5%, Coverage: 18.52%

Figure 4.2 shows the accuracy calculation table for the rules generated for zero depth-of-win.



		Predicted		
		active	inactive	
Actual	active	27	0	27
	inactive	9	28020	28029
		36	28020	28056

Accuracy = 99.96 %

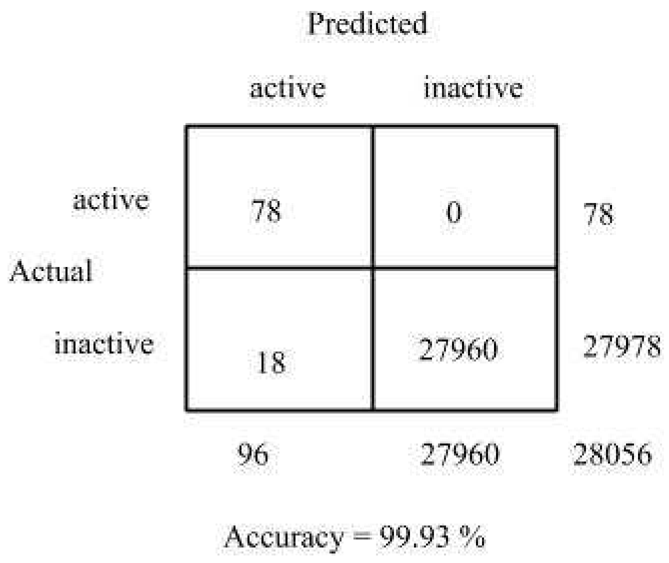
Figure 4.2: Accuracy for zero depth-of-win

Bain generated two clauses for this zero depth-of-win with coverage and accuracy as 100%. The current system generated above three clauses with coverage as 100% and accuracy as 99.96% (shown figure 4.2). The Bain system used new machine generated predicates in the hypotheses and it helps in getting the 100 percent accuracy. Current system does not have this feature of generating new predicates. An example of such a new predicate as the difference between two variables G and H should be greater than 1 and by using this predicate in rule 2 in the above hypothesis gives an accuracy of 100 percent. Even though only two clauses are generated in the Bain's hypotheses, but considering all the new generated predicates gives the overall number of rules identified in the Bain's hypothesis is ten. An experiment is conducted to generate rules with 100 percent accuracy by having a condition on each rule that it should not cover any negative examples. This experiment was successful and it identifies a total of 16 rules. Most of the generated rules are similar to each other with a small modification in one or two arguments and these rules can be grouped to have the same hypotheses generated by Bain.

Depth One Hypotheses:

As similar to the above, another experiment is conducted to find the rules for white to win by moving the black king only once. There are 78 positive example positions in the given dataset for which depth-of-win is one. Below are hypotheses generated by the current system.

- one (A, B, C, D, E, F) :-chess(c, G, H, 3, a, 2)
    - Accuracy - 81.25%   Coverage - 16.67%
- one (A, B, C, D, E, F) :-chess(c, 2, G, 4, a, 1)
    - Accuracy - 87.5%   Coverage - 8.97%
- one (A, B, C, D, E, F) :-chess(c, 2, G, 5, a, 1)
    - Accuracy - 87.5%   Coverage - 8.97%
- one (A, B, C, D, E, F) :-chess(c, 2, G, 6, a, 1)
    - Accuracy - 87.5%   Coverage - 8.97%
- one (A, B, C, D, E, F) :-chess(c, 2, G, 7, a, 1)
    - Accuracy - 87.5%   Coverage - 8.97%
- one (A, B, C, D, E, F) :-chess(c, 2, G, 8, a, 1)
    - Accuracy - 87.5%   Coverage - 8.97%
- one (A, B, C, D, E, F) :-chess(c, G, a, H, b, 1)
    - Accuracy - 75%   Coverage - 7.69%
- one (A, B, C, D, E, F) :-chess(c, G, e, H, d, 1)
    - Accuracy - 75%   Coverage - 7.69%
- one (A, B, C, D, E, F) :-chess(c, 2, G, 4, a, 3)
    - Accuracy - 75%   Coverage - 7.69%
- one (A, B, C, D, E, F) :-chess(d, 3, b, G, c, 1)
    - Accuracy - 75%   Coverage - 7.69%
- one (A, B, C, D, E, F) :-chess(d, 3, f, G, e, 1)
    - Accuracy - 75%   Coverage - 7.69%



		Predicted		
		active	inactive	
Actual	active	78	0	78
	inactive	18	27960	27978
		96	27960	28056

Accuracy = 99.93 %

Figure 4.3: Accuracy for one depth-of-win

As similar to the previous hypotheses, the hypotheses generated above is similar to hypotheses generated by Bain. Avian, with the use of new predicates in Bain system helps in getting both accuracy and, coverage as 100%. The hypotheses generated by the current system has coverage 100% but accuracy is 99.93% (shown in figure 4.3). As similar to the previous experiment, hypotheses can be generated without covering any negative examples, and then grouped to form hypotheses as same as the hypothesis generated by Bain.

As the current system uses greedy approach, sequential covering, and does not support to generate new predicates, the hypotheses generated in this task is slightly variant with the Bain system. Overall, it is proven that the current system can generate hypotheses with similar to traditional systems. The results also proven that the system's ultimate goal of generating hypotheses from relational databases with covering all positive examples and few or none of the negative examples.

5 Conclusions and Future Work

A lot of systems have been implemented in the field of Inductive Logic Programming to predict the hypotheses by using background knowledge to cover the most positive examples and the few negative examples with the help of induction technique. These systems have been successful to generate hypotheses for the most prominent datasets. However, existing systems use prolog language which is not compatible with databases where the most of real world data is stored. This thesis aimed at implementing a new system that learns knowledge in database using ILP techniques.

The system implemented in thesis gives a representation of storing all the required data in relational database and uses relational database techniques to efficiently combine multiple database entities to predict the hypotheses. The system supports storing the given text files data into database and it also supports multi-threading to improve the overall running time. Sequential covering algorithm has been implemented to generate each rule one by one which covers some or all of the positive examples and few or none of the negative examples. The system tested on two different datasets Mutagenesis and Chess Endgame. The results are proved accuracies and hypotheses are similar when compared to the existing systems which were tested on the same datasets. The results are confirming the overall accuracy of the system and this gives a confidence to the future researchers to use and enhance the current system.

As the concept of learning knowledge directly in the database is new to the ILP, there are many ways for the future work. A user friendly interface can be implemented by letting the user to simply select the relational tables and columns on which hypotheses want to be generated. The current system only supports Sequential Covering algorithm, so different set of algorithms can be added to the current system. Current system can be enhanced to support predicting multiple predicates, and generating new predicates. Future research can answer the following questions as well: Is there any better way of representing the given data in database? Can system automatically finds the best scoring mechanism depending on the dataset? Can system work completely in relational database without any interaction from the front end language such as Java?

Bibliography

[1] Muggleton, Stephen. "Inductive logic programming." New generation computing8, no. 4 (1991): 295-318.

[2] Dzeroski, Sašo, and Nada Lavrac. "Inductive logic programming: Techniques and applications." (1994).

[3] Lloyd, J. Foundations of Logic Programming. Berlin: Springer-Verlag, 1987.

[4] Plotkin, G. A note on inductive generalization, In Meltzer, B. and Michie, D., editors, *Machine intelligence* 5, pages 153-163. Edinburgh University Press, Edinburgh, 1969.

[5] Muggleton, Stephen, and Wray Buntine. "Machine invention of first-order predicates by inverting resolution." In *Proceedings of the fifth international conference on machine learning*, pp. 339-352. 1992.

[6] Robinson, John Alan. "A machine-oriented logic based on the resolution principle." *Journal of the ACM (JACM)* 12, no. 1 (1965): 23-41.

[7] Sammut, Claude, and Ranan B. Banerji. "Learning concepts by asking questions." *Machine learning: An artificial intelligence approach* 2 (1986): 167-192.

[8] Quinlan, J. Ross. "Learning logical definitions from relations." Machine learning 5, no. 3 (1990): 239-266.

[9] Srinivasan, A. "A learning engine for proposing hypotheses." Aleph (1999).

[10] Pazzani, Michael J., Clifford A. Brunk, and Glenn Silverstein. "A knowledge-intensive approach to learning relational concepts." In Proceedings of the Eighth International Workshop on Machine Learning, pp. 432-436. 1991.

[11] Pazzani, Michael, and Dennis Kibler. "The utility of knowledge in inductive learning." Machine learning 9, no. 1 (1992): 57-94.

[12] Dzeroski, Saso, and Ivan Bratko. "Handling noise in inductive logic programming." In Proceedings of the 2nd International Workshop on Inductive Logic Programming, pp. 109-125. Report ICOT TM-1182, 1992.

[13] Cestnik, Bojan. "Estimating probabilities: a crucial task in machine learning." InECAI, vol. 90, pp. 147-149. 1990.

[14] Ali, Kamal M., and Michael J. Pazzani. "HYDRA: A noise-tolerant relational concept learning algorithm." In IJCAI, pp. 1064-1071. 1993.

[15] Leckie, Christopher, and Ingrid Zukerman. "An inductive approach to learning search control rules for planning." In IJCAI, pp. 1100-1105. 1993.

[16] Cohen, William W. "Recovering software specifications with inductive logic programming." In AAAI, vol. 94, pp. 1-4. 1994.

[17] Steiglitz, Kenneth, and Christos H. Papadimitriou. "Combinatorial optimization: Algorithms and complexity." Printice-Hall, New Jersey (1982).

[18] Srinivasan, Ashwin, Stephen Muggleton, Ross D. King, and Micheal JE Sternberg. "Mutagenesis: ILP experiments in a non-determinate biological domain." In *Proceedings of the 4th international workshop on inductive logic programming*, vol. 237, pp. 217-232. 1994.

[19] Debnath, Asim Kumar, Rosa L. Lopez de Compadre, Gargi Debnath, Alan J. Shusterman, and Corwin Hansch. "Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital

energies and hydrophobicity." *Journal of medicinal chemistry* 34, no. 2 (1991): 786-797.

[20] Bain, Michael. "Learning logical exceptions in chess." PhD diss., University of Strathclyde, Glasgow, 1994.