Implementation of Breadth-First Search Method Based on a Randomly Chosen Bottom Clause for Inductive Logic Programming Method

A Thesis
SUBMITTED TO THE FACULTY OF
UNIVERSITY OF MINNESOTA
BY

Puja S Davande

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

Dr. Richard Maclin

March 2017

Acknowledgements

I would like to express my sincere gratitude to my advisor Dr. Richard Maclin for sharing his expertise, and offering invaluable assistance and support. His guidance helped me in all the time of research and writing of this thesis.

I would like to thank my thesis committee members, Dr. Haiyang Wang and Dr. Arlen Severson for their patience, support, and for providing insightful comments on this thesis. I wish to express my sincere thanks to Dr. Pete Willemsen for the continuous encouragement.

I would like to express gratitude to all of the Computer Science Department faculty members for sharing their immense knowledge and valuable guidance. I would like to thank Lori Lucia and Clare Ford for providing help and support in the times of need.

I would like to thank my friends Priyankana Basak and Anicia Dcosta for helping and supporting me throughout my masters, and for sharing their knowledge, skills, and experiences.

Finally, I must express my gratitude to my parents, my in-laws, and my husband for their continuous support and encouragement through the duration of my study. This achievement would have not been possible without them.

Dedication

I would like to dedicate this thesis to my husband, Abhishek, who has been a constant source of support and encouragement during the challenges of graduate school and life. This work is also dedicated to my parents, Shashikant and Yojana Davande, who offered unconditional love and support.

Abstract

Inductive Logic Programming (ILP) is the study of learning methods for data and rules that are represented in first-order predicate logic [Muggleton]. ILP methods mostly use logic programming as a uniform representation language for examples, background knowledge and hypotheses. Background knowledge holds the information about the language used to describe the examples and concepts, such as possible values of variables, hierarchies, predicates, and rules. ILP induces hypotheses from examples represented as first-order predicates and synthesize new knowledge from the examples. There are two standard approaches in ILP, one is bottom-up and second is top-down. Bottom-up programs implemented in systems such as ALEPH (A Learning Engine Processing Hypothesis) start with a very specific clause (also called a bottom clause) generated from a seed positive example and generalize it as far as possible without covering negative examples. The purpose of ILP is to discover definition of target predicates together with background knowledge such that it entails positive examples and not negative examples.

The aim of this research is to implement a bottom-up learning mechanism incorporating a bottom clause for implementing Inductive Logic Programming methods using standard DBMS software to represent data and a Java interface to implement the ILP methods.

Table of Contents

List of Tables

List of Figures

1.0 Introduction

Inductive Logic Programming (ILP) deals with learning a general rule from specific examples and background knowledge. ILP uses first-order logic theories from a set of negative and positive examples and fixed background knowledge [1]. The purpose of ILP is, in its simplest form, to find the definition of a (target) predicate by observing positive and negative examples of it. Using positive and negative examples of the target predicate, other background information (i.e., clauses and facts) may also be provided containing further information relevant to learning the target predicate.

For example, an ILP problem could contain information about a set of patients diagnosed with cancer. For example, we might know things like that the patient is Female, and the patient had a blood test done on a particular date and that the test exhibited a high result. This information could be captured in logic as:

```
name(Patient1,JaneDoe)
sex(Patient1,Female)
testPerformed(Patient1,Test_42)
datePerformed(Test_42,2016,4,21)
testType(Test_42,BloodPanel)
result(Test_42,BloodPanel_Value1,45)
…
```

The data would describe a set of patients, some of whom have cancer, and some of whom do not. For example, an extra set of facts might be as follows:

```
hasCancer(Patient1)
hasCancer(Patient3)
```

```
hasCancer(Patient4)
```

Implicitly, any Patient without cancer would simply not have the `hasCancer` fact be true about them. The job of the ILP system would be to infer a rule or rules that *cover* the data and can be used to recognize (appropriately) all of the patients with cancer and none of the patients without. A rule is phrased as an implication like this:

```
hasCancer(p):- testPerformed(p,t),testType(t,BloodPanel),
   result(t,BloodPanel_Value3,v),v > 20.
```

This rule suggests that an individual `p` has cancer if they had a test performed `t`, and `t` is a `BloodPanel` test and the resulting value of that test `v` is greater than `20`.

Most of the researchers in inductive logic programming have done their work in Prolog, which naturally captures logical expressions. Even though logic-based programs can be represented accurately in Prolog, there are some disadvantages in using it. Prolog and other logic programming language are not able to make much of an impact on the computer industry. Developing large programming applications are difficult in Prolog because not all prolog compilers support modules and there are compatibility issues with the major Prolog compilers. To overcome these issues, this thesis presents the use of a relational database as backend and a programming language such as Java to implement an ILP system. Background knowledge, positive examples, and negative examples are important components for ILP system for generating hypothesis. In this thesis, these components are stored in a relational database. There is one most-specific clause per example, containing all facts known to be true about the example called the bottom clause. The Bottom clause is also called a most specific clause because it puts more restriction on clauses that may be added to the current rule. This thesis generates a bottom clause to restrict the search process generated by using breadth first search. This system uses relational database and Java interface to implement ILP methods.

Chapter 2 focuses on background information including definitions, ILP techniques, and mode declarations. It also gives information on different ILP systems. Chapter 3 discusses about the implementation of our system in detail using Michalski's train problem [4]. Chapter 4 discusses about results obtained from our system on dataset examples used by ALEPH [2]. Chapter 5 discusses about conclusions and future enhancement.

2.0 Background

This chapter discusses concepts which are important to understand ILP in detail. It starts with explaining inductive learning and deductive learning, followed by terms like background knowledge, positive and negative examples, and rule learning. Further, the basic ILP approaches are illustrated with examples such as top-down and bottom-up approaches.

2.1 Inductive Learning and Deductive Learning

Inductive learning starts with specific cases and leads to general statements. On the other hand, deductive learning starts with general statement and results in specific statements. This section explains these two methods with examples.

2.1.1 Inductive Learning

Induction is the process of learning a general theory by studying specific examples where a system tries to discover a general rule from the observed examples. Inductions, as opposed to deductions, are not guaranteed to be correct. Suppose we are given data of patient's records from the hospital which includes each patient's symptoms and diseases. Now the task is to find general rules about which symptoms indicate diseases. The patient's records provided as examples from which we can find out the rules. Consider the viral disease, chicken pox. Fever and red spot are the most common symptoms of the chicken pox. It is a very contagious disease. If everyone in the hospital who has a fever and has red rash suffers from chicken pox. Therefore, we can say:

1) "If the person has a fever and red spot, the person has chickenpox."

We also can say:

2) "If the person has chickenpox, the person will get red spots."

These rules can be used to make prediction about patient's future. When we want to learn some rules we often use background knowledge which is relevant to the learning task. Consider a person is suffering from chickenpox. Suppose we found out the person from same house also suffering from chickenpox. As we know chickenpox is a very contagious disease, we can conclude

3) "If x has a fever, y has chicken pox and x, y lives in the same house as x then x, also has chicken pox.

We can combine the above rule (3) with rule (2), and we can infer that x will get red spots.

From above example we can say that, in inductive learning, premises (given statements) make conclusion probable.

2.1.2 Deductive Learning

The deduction process starts with the given statements called premises and its conclusion is claimed to necessarily follow from its premises. It means if the premises are true and acceptable, then the conclusion must be true and acceptable. Let us consider an example to understand the deduction. Below are the two premises:

1) "All the mammals have lungs."

2) "All dolphins are mammals."

From above premises, we can deduce:

3) "All dolphins have lungs."

From above example we can say that, in deductive learning, premises make conclusion certain.

2.2 Inductive Logic Programming

Inductive Logic Programming (ILP) is the intersection of Logic Programming and Machine Learning.

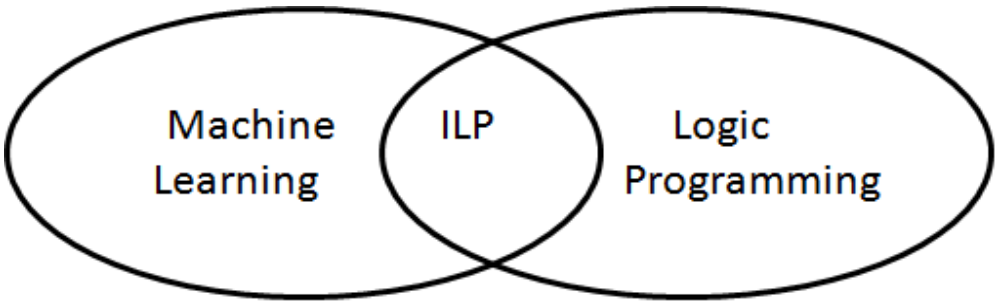


Figure 2.1: Intersection of Machine learning and Logic Programming resulting in ILP

It is the study of learning methods for data and rules that are represented in first-order predicate logic derived from background knowledge and hypothesis [1]. Background knowledge holds the information about the language used to trace the examples and concepts, such predicates, and rules. It is expressed as a set of predicate definitions. ILP induces hypotheses from examples represented as first-order predicates and synthesize new knowledge from the examples. ILP is different from the Machine Learning methods because of the use of a logical representation language and its ability to use prior knowledge in the domain such as this knowledge helps to find from examples an unknown relationship in terms of relations already known from that domain. For given training set of positive

examples E^+, negative example E^-, and background knowledge B with all clauses in First-order logic, the goal of ILP is to find hypothesis H such that,

$B \cup H \Rightarrow E+$ *and*

$B \cup H \not\Rightarrow E-$.

So, for given background knowledge and the hypothesis we attempt to cover all the positive examples (E+) and none of the negative examples (E-). It is not always possible to follow these constraints. In real-world applications this constraint is not strict. The hypothesis tries to cover the maximum number of positive examples and very few negative examples [3]. For example, we are learning a rule with given background knowledge (described below).

Table 2.1: A sample dataset for family relation

Background Knowledge	**Training Examples**	
	Positive Examples (E^+)	**Negative Examples (E^-)**

`mother(anna,bob)`
`father(scott,anna)`
`mother(alex,john)`
`female(anna)`
`male(bob)`
`male(scott)`
`male(john)`
	`parent(anna,bob)`
	`parent(scott,anna)`
		`parent(alex,anna)`
		`parent(john,bob)`

Table 2.2: Rules for family relation

Hypotheses
`parent(X,Y):- mother(X,Y),female(X).`
`parent(X,Y):- father(X,Y),male(X).`

In the above example, the background knowledge includes the facts. The symbols `mother, father, parent, female,` and `male` are predicates. Inside predicates there are constant symbols. Hypotheses contain rules which may have constants but may also have variables. Here are some of the most used terminologies in ILP explained with a family relation example.

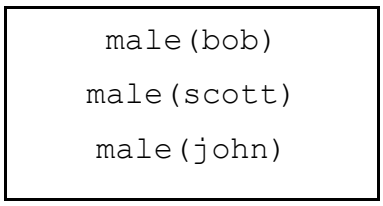
Facts

A logic program is a set of facts and rules. Facts are expressing things known to be true. A fact "anna is a mother of bob" is represented as follows:

```
mother(anna,bob).
```

Table 2.3: Facts representation for family relation

Facts
`mother(anna,bob)`
`father(scott,anna)`
`mother(alex,john)`
`female(anna)`
`female(alex)`



The above diagram represents eight facts which are true. It has relations between two people we often called it has two arguments and write as `mother/2` or `father/2`. The concept of a relation from a database can be thought of as a predicate in logic. It also has relations about single individuals for objects and facts for recognizing whether that person is male or female, and represent with one argument like `male/1` or `female/1`.

Variables

Variables in logic programming are different than other programming languages. Variables simply act like placeholders which are not yet known. Variables are represented in capital letters.

For example, `mother(anna,X)`?

The meaning of above query is "Who is anna mother of ?" and according to the facts `anna` is mother of `bob`, therefore `X=bob`.

Rules

Background knowledge not only expresses facts but also expresses rules. Rules are also called hypotheses.

For example, `parent(X,Y):- mother(X,Y), female(X).`

The above rule says if `X` is a parent of `Y`, then `X` is mother of `Y` and `X` is a female. A rule has two parts, one which comes before ":-" is called as head, and second part after ":-" is called

as body. The logical operator "AND" in the rule is represented as comma. In the above example, `parent(X, Y)` is the head of the clause and mother `(X, Y), female(X)` are the body of the clause.

Positive Example

To understand a positive example, consider `parent(anna,bob).`

According to the given facts, `anna` is the mother of `bob` and `anna` is a `female` which is true. So that is a positive example.

Negative Example

To understand a negative example, consider `parent(alex,anna).`

According to the given facts, `anna` is not a parent of `alex`. The given example does not hold for any of the hypotheses so that is negative example.

Rule Learning

Rule learning is a method for finding new relations in data and to find structured knowledge. We can say it is the process of finding hypothesis in the form of rule which will be complete (i.e., will cover all the examples) and consistent (i.e., it predicts correct class for the examples).

There are two major rule learning tasks [4]:

1. Learning a rule which covers the maximum number of positive examples and no negative examples. This is called maximal requirement for the rule.
2. Learning a rule which covers all the positive examples. This is called minimal requirement for rule.

Let us see how we can come up with the hypothesis for

`Parent(X,Y):- mother(X,Y),female(X).`

1. Take the example `parent(anna,bob).`

2. Find the information relevant to this example using background knowledge:

```
mother(anna,bob),female(anna).
```

3. Form a rule from these facts.

```
parent(anna,bob):- mother(anna,bob),
```

4. Generalize the rule.

```
Parent(X,Y):- mother(X,Y), female(X).
```

5. Check if this rule is valid with respect to the positive and the negative examples.

From the above example we can say that learning a single rule starts with the rule whose body is always true for that particular rule. Then it tries to cover maximum positive examples. If it is still covering negative examples, additional constraints can be added to its body. To find a rule which covers no negative examples, the bottom-up algorithm selects a random positive example from all covered examples. When a rule has been found which covers only positive examples, all covered examples are removed and next rule can be learned from remaining examples. This process repeats until there are no examples to cover. This process guarantees that the learned rule covers all of the given positive examples (completeness) and no negative example (consistency) [5]. The hypothesis is said to be complete with respect to background knowledge and examples if every positive example can be derived from the combination of the hypothesis and background knowledge. And the hypothesis is said to be consistent if no negative example can be derived from given hypothesis and background knowledge.

2.2.1 Inductive Logic and Deductive Logic

ILP has two basic fundamental principles: inductive logic and deductive logic. Inductive reasoning always starts with some facts and derives general conclusion from it. In the above example we induce hypotheses from gives facts.

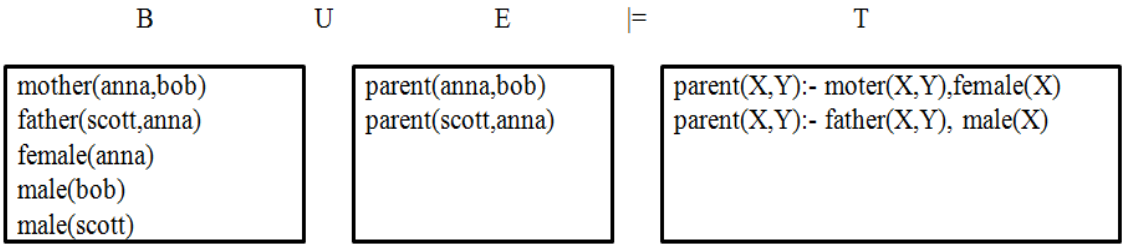


Figure 2.2: Inductive logic

Deductive logic always starts with a hypothesis or rule and derives what is based on the rule as well as the domain knowledge.

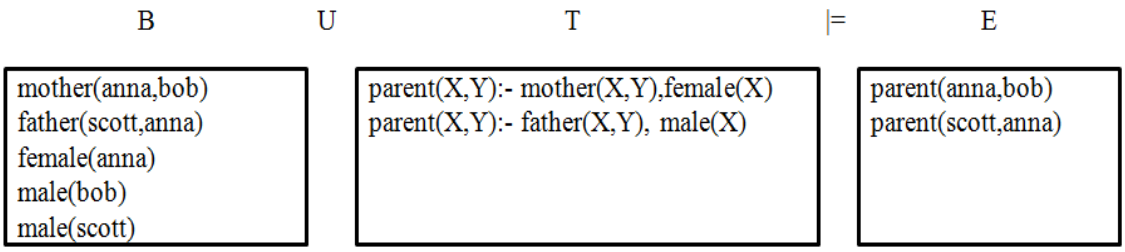


Figure 2.3: Deductive logic

2.3 ILP Techniques

There are two standard approaches in ILP to find a hypothesis: one is top-down and other is bottom-up. This section explains basic ILP approaches such as the top-down and bottom-up approaches. The bottom-up approach also explains the least general generalization concept. The topics in this section will be explained with the `daughter` and `mother` relation example as given `mother`,`female` relations as background knowledge and to generate hypotheses for `daughter`.

2.3.1 Top-Down Approach

In a top-down approach, it works from general to specific. It starts with empty clause for the condition of the hypothesis which entails everything. After that it continuously applies specialization rules (deductive rules) so that it will cover less negative examples. The ILP system PROGOL is an example of top-down search. It searches from top-down manner to find a good rule which is more general than specific clause in the hypothesis space [5]. The top-down technique is also called a specialization technique. Specialization uses two basic operations. First, it applies substitution to the clause and second it adds literals to the body of a clause [6].

Consider the following background knowledge, positive, and negative example to understand the process of top-down approach by finding relation daughter using relations mother and female.

Background knowledge represented as follows:

```
mother(anna,mary),
mother(alex,linda),
mother(scott,george),
mother(bob,jack),
female(anna),
female(alex),
female(linda),
female(mary).
```

The predicates are as follows:

`daughter(X,Y)` - X is a daughter of Y,

`mother(X,Y)` - X is a mother of Y,

`female(X)` - X is a female.

Positive Examples (E^+):

```
daughter(mary,anna),
daughter(linda,alex).
```

Negative Examples (E^-):

```
daughter(george,scott),
daughter(jack,bob).
```

The below diagram illustrates the top-down technique in ILP. Initially, the head of the hypothesis added to an empty rule. This rule is extended by adding all the positive literals in the body of the rule. The best rule among all of these is selected and processed further. Because all positive examples are not covered, we add a new general clause to the hypothesis. We iterate the process of adding clause until all the possible examples are covered. The hypothesis obtained after the processing is as follows:

```
daughter(X,Y) :- mother(Y,X), female(X)
```

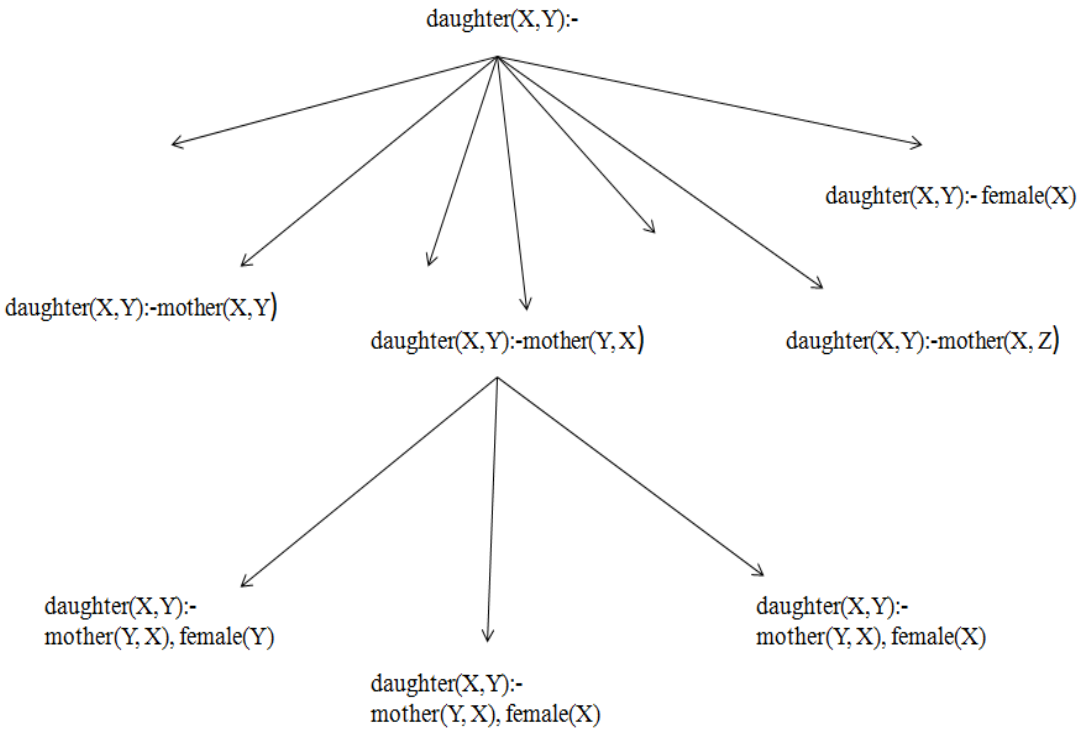


Figure 2.4: Top-down approach of ILP

2.3.2 Bottom-Up Approach

Bottom-up approach starts from examples and background knowledge. It starts with most specific hypothesis and then repeatedly applies generalization rules (inductive rules) so that it generates a hypothesis which entails more and more positive examples and no negative examples. GOLEM is the example of bottom-up search. It forms a starting clause using relative least general generalization of randomly chosen pair of positive example. This starting clause generalized by selecting more and more positive examples. Bottom-up technique is also called generalization technique. Generalization uses two basic operations. First, it applies substitution that maps terms to variables. Second, it removes a literal from the body of the clause [6]. To avoid the risk of covering negative examples by hypothesis ILP system uses notion of least general generalization (lgg).

The notion of least general generalization is based on subsumption. It assumes that if clause c1 and clause c2 are true, then lgg(c1,c2) will also be true. The least general generalization of two clauses c1 and c2 denoted by lgg(c1,c2), is the least upper bound of c1 and c2 in the θ-submission lattice [7].

If c1 and c2 are set of literals, c1= {L1, L2, ..., Ln} and c2={N1, N2, ..., Nn} then lgg (c1, c2) is given as

lgg(c1, c2) = { lgg(Li, Nj) | Li ∈ c1 , Nj ∈ c2, lgg(Li, Kj) is defined }.

For example,

c1 = { `daughter(mary,anna) mother(anna,mary) female(mary)` },
c2 = { `daughter(linda,alex) mother(alex,linda) female(linda)`}.

then the lgg of c1 and c2 is represented as follows:

```
lgg(c1, c2)= { (daughter(mary,anna), daughter(linda,alex)),
               (mother(anna,mary), mother(alex,linda)),
                                   (female(mary), female(linda)) }
```

lgg(c1,c2) is represented as

lgg(c1, c2) = { `daughter(X, Y), mother(Y, X), female(X)` }.

Here X= lgg(mary, linda) and Y= lgg (anna, alex).
The first argument in the `daughter` clause, the second argument in `mother` clause and, and the only argument in the `female` clause can be replaced by a variable `X`. The second

argument in the `daughter` clause and the first argument in `mother` clause can be replaced by a variable `Y`.

2.4 Declarative Bias

There are two types of declarative bias in ILP: Syntactic Bias (often called as language bias) and Semantic Bias. Syntactic Bias puts the restrictions on the syntax or form of the positive ground unit clauses in hypotheses. Semantic bias restricts a meaning or behavior of the hypotheses [1].

2.4.1 Language Bias

ILP systems try to alleviate the complexity problem by putting constraints on the candidate hypotheses and all these constraints are nothing but language bias. A set of all possible hypotheses for given rule can be infinite. We can put restriction on such infinite set of hypothesis with the help of language bias. ILP system reduces complexity of the ILP problem by applying all sorts of constraints on candidate hypothesis [12]. Language Bias simply is the set of predicates, which can help to decide how to do search and how deep it can go. The goal of the language bias is to put restrictions on hypothesis as much as possible otherwise data will grow infinitely and behavior become unpredictable [1]. So, the more constraints we put on the clause search space become smaller and search will finish faster. Using general first-order literals as conditions on the body of the rule may result in an infinite search space. The FOIL system, which allowed all possible combinations of variables, led through increase in search space exponentially. This majorly influences accuracy and efficiency [5].

2.4.2 Mode Declarations

Modes and declarations (defined below) are used to improve the efficiency of prolog compilers. They are also used to restrict the set of hypotheses in Inductive logic programming.

Modes are useful in Inductive logic programming for two reasons [1]:

1) If we are learning a single predicate, with correctly specified background predicates and their modes then we can assume that whatever hypothesis it will generate that will be comply with the mode.

2) We can optimize the search while generating the hypothesis.

The most common language bias is called mode declarations. Mode declarations are a well-known process from logic programming to describe the possible input-output behavior of a predicate definition [12]. The purpose of mode declarations in an ILP system is to bias and delimit the hypothesis search space. This type of restriction significantly reduces the hypothesis space. A mode declaration characterizes the format of valid hypothesis. These aspects are taken care of by the bottom clause construction algorithm. The declaration mode describes the kind of argument of each predicate:

```
mode(Recall,PredicateMode)
```

The main parts of predicate mode declaration are `modeh` predicates and `modeb` predicates, and determination. The mode head declaration, `modeh`, which is the target predicate the ILP system is supposed to induce. A `modeh` declaration occurs in the head of a hypothesis clause. It is head of hypothesis. A `modeb` declaration is body of the clause and consists of clauses in the body. A determination predicate consist head and body literal. Mode body declarations refer to the predicates defined in the background knowledge [11].

The `modeh` and `modeb` predicates consist of input variable, output variable, and recall. Let's take an example from ALEPH [2].

```
:- modeh(1,eastbound(+train))
:- modeb(1, next_car(+train,-car))
:- modeb(1,shape(+car, #shape))
```

The first clause is a head clause containing predicate `eastbound(X)` where `X` is type of train. The second clause is a body clause containing predicate `next_car(X,Y)` where `X` is a type of train and `Y` is a type of car. The third clause is body clause containing predicate `next_car(X,Y)` where `X` is of type train and `Y` is of type shape. The types of every argument of all the predicates have to be specified so that it can be used while constructing a hypothesis. These types are nothing but facts. The "+" types are used to provide input argument indicate whether argument should be a variable in the head of the rule or one from previous body literals in the rule. The "-" types are used for an output argument indicate that a new fresh variable. "#" type is a constant type. Below are the examples of facts.

For example,
```
train(east1).
train(east2).
car(car_11).
car(car_12).
```

Determination

A determination is used to construct the hypothesis. The format for determination is as follows:

```
determination(head hypothesis/arguments, body
hypothesis/arguments)
```

For example,

```
:-determination(eastbound/1,next_car/2).
```

In the above example, the first argument is a hypothesis with one argument and the second argument is a body with two arguments.

Recall

Recall can be any number greater than or equal to 1 or "*". The recall of a predicate is the maximum number of times the predicate is allowed to succeed (i.e., its maximum number of solutions) when constructing the most-specific clause. If we know there are particular instantiations of predicates then we can mention it in recall. For example, if we know each train has only one car following it then we can give recall as 1 in `next_car` in the mode declaration. If we have `parent` predicate in mode declaration, we can give recall as 2 because everyone has at most two parents. If we consider `grandparent` predicate in mode declaration then we can give recall as 4.

"*" means there is no limit on instantiation of predicate. For example, if we do not know how many ancestors people have in that case we might give recall "*" in the `ancestor` mode declaration.

2.4.3 Michalski's trains problem

To understand the use of mode declarations we can use Michalski's trains problem as an example [4].

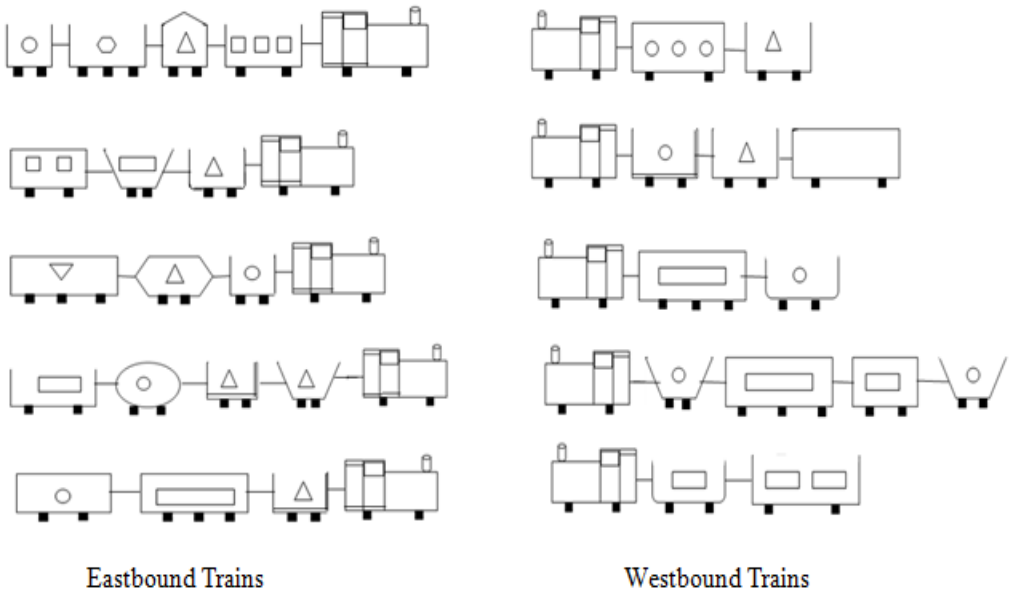


Figure 2.5 Michalski's trains problem

The above figure shows two set of trains. The ones on the right travel eastbound and the ones on the left travel west bound. The decision of whether a train is travelling east or west is made by an unknown rule. The goal of the Michalski problem is to find this unknown rule.

The mode declarations help which literals we should include in a candidate hypothesis. Below is a possible set of mode declarations which include literals for properties of train and its carriages. The modeh declaration says that the target predicate is `eastbound/1` and has

one argument of type `train`. The mode body `has_carriage/2` declaration expects a term of type `train` as input and returns a term of type `carriage`. Here are the Mode declarations for the Michalski's trains problem:

```
modeh(1,eastbound(+train))
modeb(1,open(+carriage))
modeb(*,has_carriage(+train,-carriage))
modeb(1,wheels(+carriage,#int))
modeb(1,closed(+carriage))
modeb(1,infront(+train,-carriage))
modeb(1,short(+carriage))
modeb(1,infront(+carriage,-carriage))
modeb(1,long(+carriage))
modeb(1,load(+carriage,#shape,#int))
```

The '*' symbol is called recall and can be instantiated an infinite number of times. The `wheels/2` mode body declaration takes two arguments. It takes input as a type `carriage` and returns an integer constant, succeeding only once. Other body mode declarations work as similar to above example.

2.4.3.1 Bottom clause generation

The background knowledge to ILP contains facts which are true about that particular example. In Michalski's trains problem the background Prolog program is rather simple. It is simply a list of the facts that are true for each example. Here is the background knowledge for example `east2` (second train on the left in Figure 2.5 Michalski's trains problem).

```
has_carriage(east2,car_21)
```

```
shape(car_23,rectangle)
has_carriage(east2,car_22)
open(car_21)
has_carriage(east2,car_23)
open(car_22)
infront(east2,car_21)
closed(car_23)
infront(car_21,car_22)
load(car_21,triangle,1)
infront(car_22,car_23)
load(car_22,rectangle,1)
short(car_21)
load(car_23,circle,2)
short(car_22)
wheels(car_21,2)
short(car_23)
wheels(car_22,2)
shape(car_21,u_shaped)
wheels(car_23,2)
shape(car_22,u_shaped)
```

One of the most important concepts in ILP is a most specific clause also called as a bottom clause or $\perp$. Bottom clause contains all true facts about that example. Ground and variablized are the two versions of most-specific clause. Below representation shows the ground most-specific clause for example `eastbound(east2).`

```
eastbound(east2):-
has_carriage(east2,car_23), has_carriage(east2,car_22),
has_carriage(east2,car_21), infront(east2,car_21),
closed(car_23),
open(car_22), open(car_21), short(car_23), short(car_22),
short(car_21),
load(car_23,circle,2), load(car_22,rectangle,1),
load(car_21,triangle,1),
wheels(car_23,2), wheels(car_22,2), wheels(car_21,2),
infront(car_22,car_23), infront(car_21,car_22).
```

If we replace each unique constant from ground most-specific clause by a different variable it gets converted into a variablized most-specific clause. The original constants in the mode body declaration remain same as constants. Below figure shows example of variablized most-specific clause. In the below example, in the predicate `load(car_23,circle,2)`, `car_23` is replaced by variable `B` and `circle` and 2 are constants so they remain same in the predicate. We use the variablized most-specific clause in this thesis. Here is the variablized most-specific clause for example `eastbound(east2)`:

```
eastbound(A):-
has_carriage(A,B), has_carriage(A,C),
has_carriage(A,D), infront(A,D), closed(B),
open(C), open(D), short(B), short(C), short(D),
load(B,circle,2), load(C,rectangle,1), load(D,triangle,1),
wheels(B,2), wheels(C,2), wheels(D,2),
infront(C,B), infront(D,C).
```

It is the variablized form of the most-specific clause. Particularly, in a top-down ILP system ⊥ is at the bottom of the hypothesis space, whereas in a bottom-up system ⊥ is at the top of the search.

One more important parameter we considered while constructing the most-specific clause, that is, the number of levels of variables. Level 0 contains the head clause. When i = 1, input variables of the head (level 0) are added to the most-specific clause. At layer i only literals having input variables appearing in layer i − 1 (as output or input variables) can be formed. One cannot use output variables of the head as input variables for a literal unless they already introduced as the output variable of previous layer. The below description shows the variablized most-specific clause of the above example, `eastbound(east2)`, when it is constructed with i = 0 (First level) and i=1 (Second level).

Below is the variablized most-specific clause for example `eastbound(east2)` when i=0 (First level)

```
eastbound(A):-
```

variablized most-specific clause for example `eastbound(east2)` when i=1(Second level)

```
eastbound(A):-
has_carriage(A,B), has_carriage(A,C),
has_carriage(A,D), infront(A,E).
```

variablized most-specific clause for example `eastbound(east2)` when i=2 (Third level)

```
eastbound(A):-
has_carriage(A,B), has_carriage(A,C),
has_carriage(A,D), infront(A,D), closed(B),
```

```
open(C), open(D), short(B), short(C), short(D),
load(B,circle,2), load(C,rectangle,1), load(D,triangle,1),
wheels(B,2), wheels(C,2), wheels(D,2),
infront(C,B), infront(D,C).
```

The length of the most-specific clause can potentially grow exponentially with i. The depth value of i thus directly leads to the size of the hypothesis space. With a low value for i the target concept may not be present in the hypothesis space as the required literals may not occur in the most-specific clause. For instance, the target concept for the instance of the Michalski's trains problem [4] presented is

```
eastbound(X) ← has_carriage(X,Y), closed(Y), short(Y).
```

This target concept is absent in the hypothesis space defined by a most-specific clause constructed with i = 1 because the predicates `closed/1` and `short/1` will not occur in a most-specific clause at i = 1. These both predicates are present at level 2 (i.e., i = 2) as they require an input variable of type `carriage` which is itself only introduced at i. The predicate `has_carriage` which requires an input variable of type train introduced at i=0.

2.5 ILP Systems

This section discusses some existing ILP systems. This section mostly explains systems such as FOIL and ALEPH [2] as they are related to system implemented in this thesis.

2.5.1 Sequential Covering Algorithm

A sequential covering algorithm is a one of the most popular rule based algorithms. It follows the process of developing set of hypotheses that covers all the positive examples with none of the negative examples. Sequential learning algorithm learns one rule at a time and repeat the process gradually to cover all the all the positive examples [14]. The algorithm for Sequential Covering [15] is as follows:

Procedure Sequential _Covering(Target_rule, Attributes, Examples, Threshold)

Learned_rules ← Empty_Set

Rule ← LEARN-ONE-RULE(Target_rule, Attributes ,Examples)

While Performance(Rule, Examples) > Threshold do

Learned_rules ← Learned_rules + Rule

Examples ← Examples - examples_correctly_classified_by_Rule}

Rule ← LEARN_ONE_RULE(Target_rule, Attributes, Examples)

Learned_rules ← sorted _Learned_rules

return Learned_rules

The above algorithm takes four arguments as input. The target rule is rule to be generated, attributes are the facts used in generating rule, examples are positive and negative examples, and threshold is the performance up to which the process has to continue. Initially learned rules are empty. Sequential covering uses the Learn-One-Rule method to find out best rule. Learn One Rule algorithm is follows [15]:

Procedure Learn_One_Rule(Target_rule, Attributes, Examples, Threshold)

Pos ← positive_Examples

Neg ← negative_Examples

if Pos then

```
        NewRule ← Most_General_Rule
        NewRuleNegatives ← Neg
        while NewRuleNeg do
                Literals ←Generate_Literals()
                Best_literal ← Best_Performance_literal()
                NewRule ← New Rule + Best_literal
                NewRuleNeg ← Negative_NewRule
    return NewRule
```

Learn One Rule follows a top-down approach to learn rules. It starts with most general rule and keeps adding literals to the rule until it doesn't find specific rule which covers all positive examples. Beam search is used to search a set of best literals that can be added to the rule. The Generate_Literals() function returns all the possible combinations of literals for that rule. Best_Performance_Literal() function used to decide which literals give best rule based on their positive and negative score. Beam search is a heuristic search algorithm where only a set of threshold limit of rules with high accuracy are extended. Only those rules are selected and new literals are added into them.

2.5.2 ALEPH

The Aleph (A Learning Engine for Proposing a Hypothesis) is an Inductive Logic Programming System [2]. Aleph was developed to be a prototype to explore ideas in ILP. It is written in Prolog and maintained by Ashwin Shrinivasan at the University of Oxford. The Aleph has a powerful representation language that allows representing complex expressions and simultaneously incorporates new background knowledge very easily [2]. Aleph uses three files to construct a theory (described below).

Background knowledge file (file.b)

This file contains the background knowledge. This background knowledge is in the form of Prolog clauses. It also contains language and search restrictions for Aleph along with modes, types, and determinations.

For example,

```
:- determination(eastbound/1,has_car/2).
:- determination(mult/3,mult/3).
```

Positive example file (file.f)

All the positive examples are written in the file. The positive examples are simple a ground facts.

For example,

```
eastbound(east1).
eastbound(east2).
eastbound(east3).
```

Negative example file (file.n)

All the negative examples are written in the file. The negative examples are simply ground facts.

For example,

```
eastbound(west1).
eastbound(west2).
eastbound(west3).
```

Prolog compiler is needed in order to use Aleph. To compile Aleph, the SWI or YAP platforms [2] can be used. Both of these compilers are open source and available on internet.

Basic Aleph Algorithm

To construct a theory Aleph uses the following algorithm. Aleph follows following four steps [2]:

1. **Select example:** Select an example to be generalized. If there are not more examples then stop.

2. **Build most-specific-clause:** Construction of the specific clause is based on example selected and language restriction. This is usually a definite clause with many literals called as a "bottom clause." This is also called saturation.

3. **Search:** In this step, find a clause which is more general than the bottom clause. This can be done by searching for some subset of the literals in the bottom clause that has the best score.

4. **Remove redundant:** Add the best clause to the theory, remove all redundant examples and return to step 1.

ALEPH uses mode declarations and can be changed as per the requirements. ALEPH allows the choice of more than one example to be generalized. If we choose more than one example, it creates a bottom clause for each of the example and after reduction step, the best clause is added to the theory. It allows us to choose other search strategies instead of branch and bound approach.

2.5.3 FOIL

First-Order Inductive Learner (FOIL) is rule based algorithm proposed by Quinlan [13]. FOIL uses an algorithm that tries to find a rule which covers as many positive examples as possible while covering no negative examples. The clause is initially empty. Clauses are

added into existing hypothesis. Then FOIL removes the positive example covered by the new rule. The process of finding clauses keeps repeating until all the positive examples are covered by that clause. FOIL uses top-down approach. It starts with most generalized rule and continues to more specific rule. It starts with generalized rule and continues adding literals until no negative examples are covered. New variables which are not present in the left-hand side of the rule are introduced in the process of adding literals. The process of adding literals to the rule is explained as follows [13]:

- Initialize the rule with target relation and training set (T) with positive examples that are not covered by any previous rules and also negative examples. Initialize training set T with remaining positive examples and all the negative examples.

- While T contains negative examples:
    - Find a literal L to add to the body of the rule.
    - Create new training set T1 by adding all the examples that are covered by the rule. If rule introduces new variable, then all the examples covered by new variable also added to T1.
    - Replace T with T1.
    - Remove all the positive examples from training set those satisfy the clause
    - Add this clause to the final set of clauses of target relation.

FOIL puts some restrictions on the search space of literals:

- The literal must have at least one existing variable to have some connection to the previous clauses.
- If the literal relation is same as the head of the rule then more restriction will be imposed to prevent unexpected and uncontrolled recursions.
- The gain heuristic should allow pruning of search space.

Following example explains the FOIL algorithm. The task is to find rule for 'grandfather' using below background knowledge.

Background knowledge:

```
parent(scott,jack),
parent(jack,anna),
parent(alex,john),
parent(john,bob),
parent(jack,mike),
male(mike),
male(jack),
male(john),
male(scott),
female(alex),
female(anna),
```

Positive Examples (E^+):

```
grandfather(scott,anna).
grandfather(Scott,mike).
```

Negative Examples (E^-):

```
grandfather(scott,alex), grandfather(scott,john),
grandfather(scott,alex), grandfather(scott,bob),
grandfather(jack,anna), grandfather(jack,mike),
grandfather(jack,scott), grandfather(jack,alex),
grandfather(jack,john), grandfather(jack,bob),
grandfather(anna,scott), grandfather(anna,jack),
```

```
grandfather(anna,alex), grandfather(anna,john),
grandfather(anna,bob), grandfather(anna,mike),
grandfather(alex,scott), grandfather(alex,jack),
grandfather(alex,anna), grandfather(alex,john),
grandfather(alex,bob), grandfather(alex,mike),
grandfather(john,scott), grandfather(john,jack),
grandfather(john,anna), grandfather(john,alex),
grandfather(john,bob), grandfather(john,mike),
grandfather(bob,scott), grandfather(bob,jack),
grandfather(bob,anna), grandfather(bob,alex),
grandfather(bob,john), grandfather(bob,mike),
grandfather(mike,scott), grandfather(mike,jack),
grandfather(mike,anna), grandfather(mike,alex),
grandfather(mike,john), grandfather(mike,bob).
```

To generate the rule, let us assume that the head of the rule is,

```
grandfather(X,Y).
```

If we select first literal as `parent(X,Y)`, then rule only covers negative examples without covering positive examples. Some possible literals are:

`parent(Y,X)` - This literal is valid as both are existing variables, but does not cover any positive examples.

`parent(X,Z)` - It has one new variable `(Z)` and one existing variable `(X)`, so literal is valid. it covers both the positive examples and few negative examples.

`male(X)` - It is a valid literal and covers both positive and few negative examples.

`male(Y)` - It covers one positive examples with few negative examples.

From the above options, `parent(X,Z)` and `male(X)` would be the good choices. Suppose if the first literal chosen is `parent(X,Z)` then rule will be

```
grandfather(X,Y) :- parent(X,Z).
```

Now we have following options from search space:

`parent(Z,Y)` - It covers both the positive examples and one negative example.

`parent(X,Y)` - It covers more negative examples than positive examples.

`male(X)` - It covers both positive examples with few negative example.

`male(Y)` - It covers one positive with few negative examples.

From above options, `parent(Z,Y)` would be the good option to choose as it covers both positive examples and very few negative examples. This literal is using variable which introduced in previous literal. After adding this literal to existing rule, the rule will become,

```
grandfather(X,Y) :- parent(X,Z), parent(Z,Y).
```

Again the process continues and the `male(X)` literal is chosen which covers all positive examples and no negative examples. The final rule is as follows:

```
grandfather(X,Y) :- parent(X,Z), parent(Z,Y), male(X).
```

3.0 Implementation

This chapter explains the system implementation in detail. Background knowledge and examples are the core components of the ILP system. The main goal of the system is to generate hypotheses for given background knowledge and training data. The bottom clause, as applied to ILP problems allows putting constraints on clauses generated by breadth first search. The system uses MySQL relational database as a backend and Java as a front to interact with user and database.

This first section explains the database representation of given background knowledge, positive and, negative examples. It followed by the process of generating hypotheses using breadth first search.

3.1 Database Representation

This section explains how background knowledge such as predicates, clauses, positive examples, and negative examples are stored and represented in relational databases.

3.1.1. Storing facts in database

A fact is stored in a table in the relational database. Background knowledge can contain the same predicate with different arguments. To identify a predicate with its number of arguments, the table name is defined as **tbl_PredicateName_NumOfArgumentsArgs**. Facts are stored in the text file and system takes this text file as an input for system. Column names are given as a fact in the input file. Each line in the text file should contain single fact.

The system reads each line in the input and follows the steps as mentioned below [17]:

- It checks whether a table representing the fact exists in the database.
- If table does not exist in the database, it creates new table with column names same as the arguments.
- If table exists, it stores facts as a new row in the table.

Let us consider below facts and see how these are stored in database.

```
eastbound(train),
eastbound(east1),
eastbound(east2),
has_car(train,carriage),
has_car(east1,car_12),
shape(carriage,shape_of_carriage),
shape(car_12,rectangle).
```

In the above example, first fact contains `eastbound(train)`, after taking this fact as an input system checks whether a table **tbl_eastbound_1args** exists in the database. If it does not exist, it creates a new table with name **tbl_eastbound_1args** with column train. The table contains information about name of the trains which goes eastbound. After reading the second fact which is `eastbound(east1)`, system stores fact value east1 in the first row of the table. Further reading the third fact, it will get stored in the second row of the same table. System reads the line `has_car(train, carriage)` and creates a new table **tbl_has_car_2args** with column names as train and carriage. Next the fact will be stored as east1 in first column and car_12 in second column.

Similarly, the system reads `shape(carriage,shape)` and creates a new table **tbl_shape_2args**(carriage, shape) with column names as carriage and shape_of_carriage. car_12 will be stored in carriage column and rectangle will be stored in shape_of_carriage column. Below Figure 3.1 shows a table representation of the facts.

tbl_eastbound_1args

train
east1
east2

tbl_has_car_2args

train	carriage
east1	car_12

tbl_shape_2args

carriage	shape_of_carriage
car_12	rectangle

Figure 3.1: Representation of train in relational database

3.1.2. Storing Rules

Rules are stored in the form of views in the database. A view is a virtual table and one can query on a view like table. A view does not store the data, but can combine data from two or more tables using joins or can contain subset of data which makes them to hide the complex functionality. Rules are an important part of any ILP algorithm. Rules are represented in the form of views because a rule can contain many predicates present in the body of the rule. All of the predicates tables will be used to find the result of the rule. So, views are better to represent the rules. The following steps are used to generate view for the rule [17]:

- Separate the rule into two parts. The first part is head of the rule and the second part is body of the rule.
- Find the predicates in the body of the rule and create table representations in the database.
- If there are any common arguments in the predicates, create a join query for those predicates (predicates are represented as tables in database) on the respective columns. The view must have a select statement that selects the column from the tables on the body of the rule.
- Search for arguments that are in the body as well as head of the rule.
- Create a select query that selects common arguments in the head and body of the rule using the predicates in the body of the rule.
- Append join query to the select query to generate the final query for the view.
- Create the view and store in the database.

The name of the view contains table name from the head of the rule along with the number of arguments followed by name of the tables in the body of the rule along with the number of arguments.

Following examples explains above steps:

```
A(X):- B(X)
```

In the above rule, `A(X)` is a head of rule and `B(X)` is a body of the rule. There is only one predicate in the body of the rule so there will be no join in this case. The view for above rule will be simply as follows:

CREATE VIEW vw_A_1args_B_1args AS
SELECT column1 FROM tbl_B_1args.

```
eastbound(X):- has_car(X, Y),short(Y)
```

The above rule represented as X is a eastbound train if it has Y carriage and Y is short. There are two arguments in the body of the rule and have one common argument Y. We can join two arguments using the common argument. So, the view will be

CREATE VIEW vw_eastbound_1args_has_car_2args_short_1args AS
SELECT table1.column1, table1.column2 FROM tbl_has_car_2args table1 JOIN tbl_short_1args table2 ON table1.column2=table2.column1

The above query joins the has_car and short tables. Both the columns are selected from the has_car table because after tables get joined it will give the same output even if we select column1 from table2 instead of column2 from table2.

```
eastbound(X):- has_car(X, Y), shape(Y, Rectangle)
```

The above rule represented as X is an eastbound train if it has Y carriage and shape of X is rectangle. The view will be as follow,

CREATE VIEW vw_eastbound_1args_has_car_2args_shape_2args AS
SELECT table1.column1, table1.column2 FROM tbl_has_car_2args table1 JOIN tbl_shape_2args table2 ON table1.column2=table2.column1

```
has_car(X,Y):-short(X)
```

The above rule represents missing variable in the body. The view query will be

CREATE VIEW vw_has_car2args_short_1args AS
SELECT table1.column1, NULL AS column2 from short_1args

As argument `Y` is not present in the body predicates. The query puts NULL value in the column to represent missing variable.

3.1.3. Representing Positive and Negative Examples

Positive and Negative examples are stored in the two different tables as Positive_table and Negative_table in the database. If we have positive and negative examples as facts we can also store them in different tables with the help of storing facts mechanism mentioned in Section 3.1.1.

3.2 Hypotheses Generation

The previous sections discussed background knowledge, and positive and negative examples stored in the database. This section focuses on how to generate hypotheses in the system.

3.2.1 Input

The process of learning rule starts with input from the user. The input for a hypothesis is stored in a text file. Each line contains the table name followed by number of columns. The '-' separates each part of input line. The first contains predicate name and arguments of head part for which we need to generate hypothesis. Further lines after first line are considered as the body part of hypothesis. It also contains predicates and number of arguments. The input to generate a rule eastbound table will be as follows:

1. tbl_eastbound_1Args-1
2. tbl_has_car_2Args-1-2
3. tbl_shape_2Args-1-2

The above example has the rule to be generated in line 1. Line 2 and line 3 contain the names of the tables in the database used to generate hypothesis. Each column can have different mode. Mode describes the type of values that a column can take. The modes of input are explained below:

Mode i: This mode stands for input. It means column values in **tbl_has_car_2Args** can take values either from eastbound or has_car.

Mode n: This mode stands for NEW, means it can take a new variable. For example, tbl_has_car can take new variable as the value of column.

Mode o: This mode stands for OLD, means that the value of particular column can take value that has been introduced in previous predicates. For example, if a new variable is introduced in has_car then that new variable can a part of tbl_shape.

Mode c: This mode is for constants. It means it can take constant values which are present in tables.

This stored input is processed in the code and all the required elements to generate the rule are fetched.

3.2.2. Generation of All Possible Literals

After reading input file by the system, the system generates all different possible combination of arguments among the input tables. Generation of all the possible arguments is one of the important steps in hypothesis generation. It helps to test all the possible cases for a rule. It also calculates positive and negative score for each rule with different combinations. Therefore, it becomes easier to choose which rule and combination of arguments cover more positive examples. All the possible argument for has_car is as shown in following example:

For example,

The input for has_car table is as follows:

tbl_has_car_2Args-1-2

Here are all the different possible combination for has_car table can be generated as follows. Here we consider different modes of input. Table has_car has two columns.

```
has_car(column1, column2),
has_car(column2,new),
has_car(column1,old),
has_car(column2, column1),
has_car(column2, new),
has_car(column2, old),
has_car(new, column1),
has_car(new, column2),
has_car(new, old),
has_car(old, column1),
has_car(old, column2),
has_car(old, new),
has_car(old, old),
```

new in the column means that new variable will take the place in column. *old* means the column will contain a variable that has been already introduced in the previous predicates.

```
eastbound(column1):- has_car(column1,newVar1), shape(
newVar1,column2)
```

For the above rule, `v1` is a new variable which is introduced in the first predicate `has_car`. After introducing in `has_car`, `v1` becomes old variable when being used as an argument in second argument `shape`. So, the first predicate is looks similar to `has_car(column1,new)` and second predicate derived from the combination of `shape(old,column1)`.

These combinations above are used to generate rules and scored to find out which rule is good. Similarly all the predicates in the body follow same procedure. We do not consider `has-car(new,new)`. It does not make any sense to introduce two new variables in the predicate as there is nothing to connect to other predicates.

3.3 Algorithm to generate hypotheses

After processing the input file, the system calls our breadth-first search algorithm. The initial rule consists of the head of the rule and following process adds every possible literal to the body of the rule and finally picks the best one. The best rule is that which covers all or some of the positive examples and none or few negative examples. This algorithm stores all intermediate generated rules and finally picks the best rules from those rules that act as a bottom clause for the hypothesis. This algorithm works similar to ALEPH algorithm. A Breadth-First Search strategy is used to implement this algorithm. First we will expand head rule, then all successors of the head rule, then their successors, and so on until we find the best rule.

The algorithm is as follows:

- For each combination of arguments generated in section 3.2.2, append it to the rule as a new literal. Add all these rules to the list of generated rules.
- For each rule in current level do the following:

- Generate a view query for this rule by using the algorithm mentioned in 3.2.1
- Score is calculated for each rule by creating view and passing it to the database. A stored procedure has written to store the rule in the database along with its score in a table.
- Create a new query to join the above query with positive table and negative table.
- Execute these queries to get positive and negative score. There are different methods to calculate score for a rule. Positive score is number of examples covered in positive table and negative score is number of examples covered in negative table.
- Create an empty queue to store the rule in the queue if the positive score is greater than zero. If the positive score is 0, do not add that rule in the queue.
- The rules which are we added into the queue are specific clauses for the hypotheses.
- Pop the top rule from the queue and take that rule to the next level. Taking rule to the next level means repeat the procedure again for selected new rule until we get the most general rule than specific rule.

- If system reaches to maximum depth of clause then stop processing.

Let's illustrate the above algorithm using an example for generating hypothesis for eastbound with one argument using has_car and shape predicates.

Table 3.1 Table representation for eastbound table

column1
east1
east2
east3
east4

Table 3.2 Table representation for has_car table

column1	column2
east1	car_11
east1	car_12
east1	car_13
east1	car_14

Table 3.3 Table representation for long table

column1
car_11
car_13
car_33

Table 3.4 Table representation for closed table

column1
car_11
car_12
car_23
car_32
car_33
car_43

Table 3.5 Positive examples table for eastbound rule

column1	column2
east1	car_11
east1	car_12
east1	car_13
east2	car_21
east1	car_14

Table 3.6 Negative examples table for eastbound rule

column1	column2
east2	car_12
east2	car_13
east2	car_11
east2	car_14

The input for the system will be as follows:

tbl_eastbound_1Args-1

tbl_has_car_2Args-1-2

tbl_long_1Args-1

tbl_closed_1Args-1

After reading this input, system will generate all possible arguments for each predicates as discussed in section 3.2.2.

Learning process starts with head part of the rule. So the head of the rule is,

```
eastbound(column1) :-
```

In the first stage, all different possible combinations for each predicate is generated and added to the rule and scored.

```
eastbound (column1) :- has_car(column1,newVar1).
```

For the above rule the positive query is:

SELECT count(distinct PositiveTable.column1, PositiveTable.column2) from positiveTable AS PositiveTable join (SELECT distinct table1.column1 AS column1 FROM tbl_has_car_2Args table1) AS ruleExamples ON ruleExamples.column1 = PositiveTable.column1

This query returns the positive score. There are different ways to calculate positive and negative scores. Number of rows returned by this query will be the positive score of the rule.

The positive score for above query is 7.

Examples covered are:

```
has_car(east1, car_11)
has_car(east1, car_12)
has_car(east1, car_13)
has_car(east1, car_14)
```

```
has_car(east2, car_21)
has_car(east2, car_22)
has_car(east2, car_23)
```

For the above rule negative query is:

SELECT count(distinct NegativeTable.column1, NegativeTable.column2) from negativeTable AS NegativeTable join (SELECT distinct table1.column1 AS column1 FROM tbl_has_car_2Args table1) AS ruleExamples ON ruleExamples.column1 = NegativeTable.column1

This query returns the negative score. Number of rows returned by this query will be the negative score of the rule. Negative score for the above query is 4.

Examples covered are:

```
has_car(east2, car_11)
has_car(east2, car_12)
 has_car(east2,car_13)
 has_car(east2,car_14)
```

Here are some of the rules generated in first level:

- eastbound(column1) :- closed(column1)
  Positive score: 0 Negative score: 0

- eastbound(column1) :- long(column1)
  Positive score: 0 Negative score: 0

- eastbound(column1) :- has_car(column1,newVar1)

Positive score: 7 Negative score: -4

- eastbound(column1) :- has_car(newVar2, column1)
  Positive score: 0 Negative score: 0

As the rule `eastbound(column1):- has_car(column1,newVar1)` has positive score greater than negative score it is added into the queue and send to the next level.

Then `eastbound(column1):- has_car(column1,newVar1)` rule popped from the queue as there is only one rule, it looks for its successors, scores them and adds them into the queue.

Rules generated in the second level are:

- eastbound(column1) :- has_car(column1, newVar1), closed(column1)
  Positive score: 0 Negative score: 0

- eastbound(column1) :- has_car(column1, newVar1), closed(newVar1)
  Positive score: 7 Negative score: -4

- eastbound(column1) :- has_car(column1, newVar1), long(column1)
  Positive score: 0 Negative score: 0

- eastbound(column1) :- has_car(column1, newVar1), long(newVar1)
  Positive score: 4 Negative score: 0

- eastbound(column1) :- has_car(column1, newVar1), has_car(column1, newVar5)
  Positive score: 7 Negative score: -4

- eastbound(column1) :- has_car(column1, newVar1), has_car(column1, newVar1)

Positive score: 7 Negative score: -4

- eastbound(column1) :- has_car(column1, newVar1), has_car(newVar6,column1)
  Positive score: 0 Negative score: 0

- eastbound(column1) :- has_car(column1, newVar1), has_car(newVar7,newVar1)
  Positive score: 7 Negative score: -4

- eastbound(column1) :- has_car(column1, newVar1), has_car(newVar1,column1)
  Positive score: 0 Negative score: 0

- eastbound(column1) :- has_car(column1, newVar1), has_car(newVar1,newVar8)
  Positive score: 0 Negative score: 0

From above rules the following rules can be added into the queue:

- eastbound(column1) :- has_car(column1,newVar1), closed(newVar 1)

- eastbound(column1) :- has_car(column1, newVar1), has_car(newVar 7,newVar1)

- eastbound(column1) :- has_car(column1, newVar 1), long(newVar 1)

- eastbound(column1) :- has_car(column1, newVar 1), has_car(column1, newVar5)

- eastbound(column1) :- has_car(column1, newVar 1), has_car(column1, newVar1)

As newVar5 variable introduced in the first level, it is used as an old variable in other predicates in second level.

Below is the rule we get in the second level which covers the only positive example:

```
eastbound(column1) :-has_car(column1, newVar1),
long(newVar1)
```

It covers 4 positive examples and no negative examples.

Rules generated at the third level are:

- eastbound(column1) :- has_car(column1, newVar 1), closed(newVar 1), closed(column1)
  Positive score: 0 Negative Score: 0

- eastbound(column1) :- has_car(column1, newVar 1), closed(newVar 1), closed(newVar 1)
  Positive score: 7 Negative Score: -4

- eastbound(column1) :- has_car(column1, newVar 1), closed(newVar 1), long(column1)
  Positive score: 0 Negative Score: 0

- eastbound(column1) :- has_car(column1, newVar 1), closed(newVar 1), long(newVar 1)
  Positive score: 4 Negative Score: 0

- eastbound(column1) :- has_car(column1, newVar 1), closed(newVar 1), has_car(column1, newVar 9)
  Positive score: 7 Negative Score: -4

- eastbound(column1) :- has_car(column1, newVar 1), closed(newVar 1), has_car(column1, newVar 1)
  Positive score: 7 Negative Score: -4

- eastbound(column1) :- has_car(column1, newVar 1), closed(newVar1), has_car(newVar 10, column1)
  Positive score: 0 Negative Score: 0

- eastbound(column1) :- has_car(column1, newVar 1), closed(newVar1), has_car(v11, newVar 1)
  Positive score: 7 Negative Score: -4

- eastbound(column1) :- has_car(column1, newVar 1), closed(newVar 1), has_car(newVar1,column1)
  Positive score: 0 Negative Score: 0

- eastbound(column1) :- has_car(column1, newVar 1), closed(v1), has_car(newVar 1, newVar 12)
  Positive score: 0 Negative Score: 0

From the above rules the following rules can be added into the queue:

- eastbound(column1) :- has_car(column1, newVar 1), has_car(newVar 7, newVar1)

- eastbound(column1) :- has_car(column1, newVar 1), closed(newVar 1), has_car(newVar 11, newVar 1)

- eastbound(column1) :- has_car(column1, newVar 1), long(newVar 1)

- eastbound(column1) :- has_car(column1, newVar 1), has_car(column1, newVar5)

- eastbound(column1) :- has_car(column1, newVar 1), has_car(column1, newVar1)

- eastbound(column1) :- has_car(column1, newVar 1), closed(newVar 1), closed(newVar 1)

- eastbound(column1) :- has_car(column1, newVar 1), closed(newVar 1), long(newVar 1)
- eastbound(column1) :- has_car(column1, newVar 1), closed(newVar 1), has_car(column1, newVar 9)

- eastbound(column1) :- has_car(column1, newVar 1), closed(newVar 1), has_car(column1, newVar 1)

Below is the rule we get in the third level which covers only positive examples:

```
eastbound(column1) :-has_car(column1, newVar 1),
closed(newVar 1), long(newVar 1)
```

It covers 4 positive examples and no negative examples.

This is the process of generating hypotheses. The hypotheses generated by the system are covering all positive examples and no negative examples

4.0 Results

The system described in this thesis is tested on a dataset used to test the ALEPH system. Aleph is written in prolog. It is difficult to handle databases in prolog. This System uses breadth-first search to generate a hypotheses using a relational database and Java programming language.

A rule is generated for an eastbound train using different properties of train like what kind of carriage train has? Is the carriage closed? Is the carriage short?

The task was to find the rule for eastbound train using tables, has_car, shape, closed, long, and short.

The rule generated by system for eastbound train in first level is:

eastbound(A) :- has_car(A, B)

Rules generated by system for eastbound train in second level are:

- eastbound(A) :-has_car(A,B), closed(B)
- eastbound(A) :-has_car(A, B), has_car(C,B)
- eastbound(A) :-has_car(A, B), long(B)
- eastbound(A) :-has_car(A, B), has_car(A, C)
- eastbound(A) :-has_car(A, B), has_car(A , B)

From above rules system chooses below rule in level 2 as it covers all positive examples

```
eastbound(A) :-has_car(A, B), long(B)
```

It means that the train A is going eastbound if it has carriage B and carriage B is long. Rules generated by system for eastbound train in third level are:

- eastbound(A) :-has_car(A, B), has_car(C, B)
- eastbound(A) :-has_car(A, B), closed(B), has_car(E, B)
- eastbound(A) :-has_car(A, B), long(B)
- eastbound(A):-has_car(A,B),has_car(A,C)
- eastbound(A) :-has_car(A, B), has_car(A, B)
- eastbound(A) :-has_car(A, B), closed(B), closed(B)
- eastbound(A) :-has_car(A, B), closed(B), long(B)
- eastbound(A) :-has_car(A, B), closed(B), has_car(A, F)
- eastbound(A):-has_car(A, B), closed(B), has_car(A,B)

From above rules system chooses below rule in level 3 as it covers all positive examples

```
eastbound(A) :-has_car(A, B), closed(B), long(B)
```

It means that the train A is eastbound if it has a carriage A and carriage A is closed and long.

Table 4.1 represents above rules generated by system in level 1, level 2, and level 3, along with their positive and negative score:

Table 4.1 Rules generated in level 1, level 2, and level 3

Generated Rules	**Positive Score**	**Negative Score**
eastbound(column1) :-has_car(column1,v1)	7	-4
eastbound(column1) :-has_car(column1,v1), closed(v1)	7	-4
eastbound(column1) :-has_car(column1,v1), long(column1)	4	0
eastbound(column1) :-has_car(column1,v1), has_car(column1,v5)	7	-4
eastbound(column1) :-has_car(column1,v1), has_car(column1,v1)	7	-4
eastbound(column1) :-has_car(column1,v1), has_car(v7,v1)	7	-4
eastbound(column1) :-has_car(column1,v1), closed(v1), closed(v1)	7	-4
eastbound(column1) :-has_car(column1,v1), closed(v1), has_car(column1,v9)	7	-4
eastbound(column1) :-has_car(column1,v1), closed(v1), has_car(column1,v1)	7	-4

eastbound(column1) :-has_car(column1,v1), closed(v1), has_car(v10,column1)	4	0
eastbound(column1) :-has_car(column1,v1), closed(v1), has_car(v11,v1)	7	-4

The results show that, the results generated by the proposed system are similar to the traditional ILP system Aleph. Slight variations in the results are expected as the system uses different approach. As the hypotheses generated by the system are satisfied, slight variations in the results can be ignored.

5.0 Conclusions and Future Work

Inductive logic programming is an intersection of machine learning and logic programming. Different systems have been implemented in ILP to derive hypotheses by using known background knowledge, positive and negative examples. An ILP system ALEPH has a big influence on the system proposed in this thesis that was implemented as a prototype for exploring new ideas in ILP. The system implemented in this thesis presents a bottom-up learning mechanism by generating a bottom clause using standard database management system.

The system uses relational database for storing facts and rules. Background knowledge, positive examples, and negative examples are stored in the form of database tables. Rules are stored as database views by joining common argument present between predicates. Breadth first search algorithm is implemented to generate rules which cover all positive examples and very few negative examples. Results obtained from the system are promising and similar to results given by existing system ALEPH. Though the results are not exactly the same, differences in results are due to different approaches implemented in two systems.

While this thesis has demonstrated the potential way of generating hypotheses for ILP system there are many opportunities for expanding the scope of this thesis. The system can be tested on variety of datasets to make it more efficient and robust. A user-friendly graphical user interface (GUI) can be implemented where user can select tables from relational databases and a particular column from tables to generate hypotheses. More work is necessary on handling constants in the predicates. Apart from these enhancements, future research can be done on generating new predicates which are not provided in the input.

Bibliography

[1] Stephen Muggleton, Luc De Raedt. “Inductive logic programming: Theory and Methods” *J. LOGIC PROGRAMMING* 1994: 19, 20:629679

[2] Srinivasan, Ashwin. "The aleph manual." (2001). (http://www.cs.ox.ac.uk/activities/machinelearning/Aleph/aleph)

[3] DiMaio, Frank, and Jude Shavlik. "Learning an approximation to inductive logic programming clause evaluation." *Inductive Logic Programming*. Springer Berlin Heidelberg, 2004. 80-97.

[4] Michalski, Ryszard S., Jaime G. Carbonell, and Tom M. Mitchell, eds.*Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[5] Fürnkranz, Johannes. "Separate-and-conquer rule learning." *Artificial Intelligence Review* 13.1 (1999): 3-54.

[6] Poveda, Jordi Poveda, and Jordi Turmo Borràs. “Inductive logic programming and its application to the temporal expression chunking problem”. Diss. Ph. D. Programme on Artificial Intelligence (UPC). LSI Department Technical Report–January, 2007.

[7] Cussens, James, and Saso Dzeroski. *Learning language in logic*. No. 1925. Springer Science & Business Media, 2000.

[8] Wrobel, Stefan. "Inductive logic programming for knowledge discovery in databases." *Relational data mining*. Springer Berlin Heidelberg, 2001. 74-101.

[9] De Raedt, Luc. "An inductive logic programming query language for database mining." *Artificial Intelligence and Symbolic Computation*. Springer Berlin Heidelberg, 1998.

[10] Duboc, Ana Luísa, Aline Paes, and Gerson Zaverucha. "Using the bottom clause and mode declarations in FOL theory revision from examples."*Machine learning* 76.1 (2009): 73-107.

[11] França, Manoel VM, Gerson Zaverucha, and Artur S. d'Avila Garcez. "Fast relational learning using bottom clause propositionalization with artificial neural networks." *Machine learning* 94.1 (2014): 81-104

[12] Flach, Peter A. "The logic of learning: a brief introduction to Inductive Logic Programming." *Proceedings of the CompulogNet Area Meeting on Computational Logic and Machine Learning*. 1998.

[13] Quinlan, J. Ross. "Learning logical definitions from relations." *Machine learning* 5.3 (1990): 239-266.

[14] *Sequential Covering*. http://www.cse.unsw.edu.au/~cs9417ml/Rule/sequential.html Accessed: 2016-02-18

[15] *Sequential Covering Algorithm.* https://www.d.umn.edu/~rmaclin/cs8751/Notes/L08_Rule_Learning.pdf Accessed: 2016-07-28

[16] *Wikipedia. Prolog*. https://en.wikipedia.org/wiki/Prolog Acessed:2016-07-28

[17] Repaka Ravikanth. "Efficiently Storing and Discovering Knowledge in Databases via Inductive Logic Programming Implemented Directly in Databases." Master's thesis, University of Minnesota, Duluth, 2015

[18] Jose Carlos Almeido Santos. “Efficient Learning and Evaluation of Complex Concepts in Inductive Logic Programming” PhD diss., Imperial College, London, 2010