INDUCTION OF LOGICAL RELATIONS BASED ON SPECIFIC GENERALIZATION OF STRINGS

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Yasin Uzun

January, 2007

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. İlyas Çiçekli (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Nihan Kesim Çiçekli

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Selim Aksoy

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

INDUCTION OF LOGICAL RELATIONS BASED ON SPECIFIC GENERALIZATION OF STRINGS

Yasin Uzun
M.S. in Computer Engineering
Supervisor: Assist. Prof. Dr. İlyas Çiçekli
January, 2007

Learning logical relations from examples expressed as first order facts has been studied extensively by the Inductive Logic Programming research. Learning with positive-only data may cause overgeneralization of examples leading to inconsistent resulting hypotheses. A learning heuristic inferring specific generalization of strings based on unique match sequences is shown to be capable of learning predicates with string arguments. This thesis outlines the effort showed to build an inductive learner based on the idea of specific generalization of strings that generalizes given clauses considering the background knowledge using least general generalization schema. The system is also extended to generalize predicates having numeric arguments and shown to be capable of learning concepts such as family relations, grammar learning and predicting mutagenecity using numeric data.

Keywords: indective logic programming, machine learning, string generalization, hypotheses, example, background knowledge.

ÖZET

MANTIKSAL İLİŞKİLERİN DİZİLERİN ÖZGÜL GENELLEMESİNE DAYANAN BİR YÖNTEMLE TÜMEVARIMSAL ÇIKARILMASI

Yasin Uzun
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Yar. Doç. Dr. İlyas Çiçekli
Ocak, 2007

Mantıksal ilişkilerin birincil sıra gerçekler olarak ifade edilmiş örneklerden çıkarılması Tümevarımsal Mantık Programlama araştırmalarınca derinlemesine çalışılmış bir konudur. Sadece pozitif örneklerden yola çıkılarak yapılan öğrenmeler aşırı genellemelere neden olup tutarsız hipotezlerin sonuçlanmasına neden olabilir. Tek eşlemeli dizilere dayalı özgül genellemeler çıkaran bir öğrenme yönteminin dizi argümanlı önerileri öğrenebildiği gösterilmiştir. Bu tez, dizilerin özgül genellemeleri fikrine dayalı, en az genel genelleme şemasını kullanma yoluyla geri plan bilgisini de dikkate alarak önerme genelleyen bir tümevarımsal öğrenicinin gerçekleştirilebilmesi için yapılan çalışmayı özetlemektedir. Gerçekleştirilen sistem, ayrıca sayısal argümanlı önermeleri de genelleyebilecek şekilde genişletilmiş ve akrabalık ilişkileri, dilbilgisi öğrenme ve sayısal veri işlemesi gereken mutagenesis tahmini gibi örneklerde başarılı sonuçlar verdiği gösterilmiştir.

Anahtar sözcükler: tümevarımsal mantık programlama, makine öğrenmesi, dizi genellemesi, hipotez, örnek, geri plan bilgisi.

Acknowledgement

I would like to express my gratitude and appreciation to my advisor, Dr. İlyas Çiçekli, for his guidance, invaluable help and supervision during this study.

I thank to Dr. Nihan Kesim Çiçekli and Dr. Selim Aksoy for showing keen interest, and accepting to read and review this thesis.

I acknowledge the Scientific and Technological Research Council of Turkey (TÜBİTAK) for supporting my MSc studies under MSc Fellowship Program.

Special thanks to Ali Cevahir, Murat Ak and M. Ali Başţan for helping me while writing this thesis.

I am forever grateful to my family for their encouragement and continuous support during my education.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As human beings, we start to learn at the time of birth as an infant. In fact, there are fetal psychology foundings such as different reactions given by the fetus to the voices of the mother and other people, indicating that learning process starts even before birth [14]. Throughout our life, we learn about ourselves and the environment in various ways and learning by experience is perhaps the most common method we follow.

It can be said that, learning concepts from examples is a strong way of learning for human beings. There are more radical claims such as

> Example is not another way to teach. It is the only way.

by A. Einstein. For instance, an infant does not learn speaking by using grammar books, what she simply does is to imitate her relatives, mostly her family. We usually follow the same strategy when we are learning reading, writing, speaking a foreign language or performing a particular sport. It can be possible to make use of this idea to build clever machines that can learn certain concepts.

Providing computer systems to learn has been studied in long term and enormous amount of research has been done in this field. Although a Star Wars android does not seem to appear in near future, machine learning studies showed

their efficiency in many real-world domains such as speech recognition, face recognition, computer vision, medical diganosis, bioinformatics [22]. Although there are many learning systems based on different approaches, most of them share the common property of requiring a training set to identify the target concept.

Logic Programming can be defined as use of Mathematical Logic for Computer Programming [49]. Studies on Artificial Intelligence and Automatic Theorem Proving [18] formed theoretical foundations of Logic Programming in 1960's. Efforts on theorem proving in early 1960's inspired Robinson [40] for introducing the resolution inference rule, which could enable computer systems to perform automated deduction. Developed in 1972 by Colmaurer [18], Prolog programming language had great influence in Logic Programming by providing a solid and universal basis for the research.

Logic programs consist of logic formulas and computation is the process of proof construction. The most distinctive feature of a Logic Program comes from the declarative meaning of logic, that is its self-expressiveness and closeness of the notation to real life [49]. That is, it is not necesary to have a deep knowledge of syntax and notation to understand a logic program and express some real life facts in the language.

Although different taxonomies are present, it can be said that Machine Learning paradigms include analytic learning, genetic algorithms, neural networks and inductive learning [3]. Most of the current systems rely on one of these paradigms, though there are some implementations which exploit the advantages of several techniques [36]. There are arguments [20] stating that the knowledge produced by the system should be understandable by humans, which omits Neural systems out.

Inductive Logic Programming, shortly ILP, is a relatively new research area that is between Machine Learning and Logic Programming, and inherits the techniques and theories from both disciplines. The aim of ILP research is to learn logic programs, given examples and background knowledge expressed in Horn clause logic, which correctly define a single concept or multiple related concepts. The learned logic programs are usually expressed in Prolog syntax and declarativeness

of logic programs is the main source of efficiency of ILP.

There are many ILP learners implemented and tested in the literature. These systems can be classified as empricial and interactive in terms of input style or top-down and bottom-up in the aspect of search direction. Empirical learners take their input at once and learn a single predicate; while interactive systems interact with the user and can induce several predicates. Among these systems, MIS [45] is top-down and interactive, FOIL [35] is top-down and empirical, CIGOL [44] is bottom-up and interactive, GOLEM [43] is bottom-up empirical learner. Progol [27] works in the same manner as GOLEM, and is the most common state-of-the art ILP learner.

Common approach in state-of the art ILP paradigm is to produce general clauses from positive examples and restrict their coverage by the help of negative examples. In domains where there is positive-only data, the systems may not be able to learn the concepts correctly because of the absence of negative examples. The problem is so substantial and common that, Progol system is designed to work in a different mode when there is only positive data.

One application area of ILP is learning predicates having string arguments, which can occur in many domains such as Grammar Learning and Machine Translation. The bottom-up method Least General Generalization proposed in [33] may cause overgeneralization in the clause generation in the absence of negative examples. In [4], a specific generalization of two strings is proposed to reduce overgeneralization. To compute SG, unique match sequence, which is a sequence of similiarities and differences is found in initial step and followed by the generalization by replacing differences with variables. In the mentioned work, application of the heuristic in Machine Translation and Grammar Learning is also explained with example cases.

Although [4] proposes a heuristic for generalization of strings, it is far from being an ILP system because of lack of background knowledge processing and generalization of non-string (numeric) arguments. The purpose of the research that is materalized in this thesis was to develop an ILP system based on the specific generalization idea. We achieved this purpose by extending the specific

generalization algorithm taking the background clauses into consideration and constructing a generalization technique for numeric arguments that is similar to SG algorithm based on pairwise distances of numbers. We showed that the system works effectively in several example domains.

The rest of the thesis is organized as follows. Chapter 2 summarizes the main points and paradigms of ILP. Specific generalization of strings, proposed in [4] is discussed in Chapter 3. In Chapter 4, we explain the construction of an inductive learner, which we name InGen, based on the specific generalization heuristic outlined in the previous chapter. Implementation of InGen and experimental results are listed in Chapters 5 and 6, respectively. Chapter 7 concludes the whole thesis with future directions to study.

Chapter 2

Inductive Logic Programming

Inductive Logic Programming is a research field between Machine Learning and Logic Programming that learns logical relations as logic programs as illustrated in Figure 2.1. It relies on the logical theories of Logic Programming and learning techniques of Machine Learning.

2.1 Foundations

Induction can be defined as a way of reasoning from specific to general and inductive learning is described as the process of deriving the formal description of concepts using the given examples [16]. It can also be considered as a search

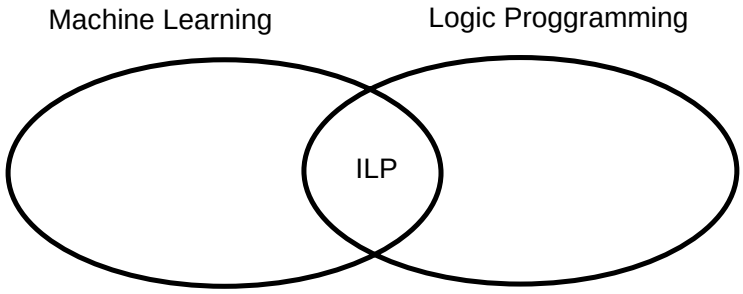

Figure 2.1: Machine Learning, Logic Programming and ILP

of underlying theory behind the facts that are given as examples given in prior.

The success of induction process is closely related with the language that is used to describe concepts and descriptions. A possible choice for object description can be attribute-value representation, in which every object is described with the values assigned to a set of attributes. For instance, all the cards in a deck can be represented by two attributes: *suit* and *rank* of the card. The set of values for the *suit* attribute is $\{hearts, diamonds, clubs, spades\}$. The set of values for the *rank* is: $\{ace, 2, 3, .., 10, jack, queen, king\}$. In this language, an individual card can be represented as: $[Suit = clubs]$ *and* $[Rank = 5]$. In Predicate Calculus, the same card can be described as $card(clubs, 5)$

We must represent concepts together with the objects in the language induction is performed. For instance the concept of a *pair* in a deck of cards can be described in several ways in an attribute-value language. The most compact representation is
pair if $Rank_1 = Rank_2$.
In Predicate Calculus, the same concept can be described as:
$pair(card(Suit_1, Rank_1), card(Suit_2, Rank_2)) \leftarrow Rank_1 = Rank_2$

One of the main issues of inductive learning is to decide whether an object description satisfies the concept description, meaning that the concept *covers* the object. A hypothesis is a possible description of the concept to be learned. An object description is labeled as a positive example if it is an instance of the concept and negative otherwise. For instance for the concept of card pairs in a deck of card:
$pair(card(clubs, 4), card(spades, 4))$ is a positive example.
$pair(card(hearts, ace), card(clubs, ace))$ is a positive example.
$pair(card(diamonds, 8), card(diamonds, 3))$ is a negative example.

Based on the concept and object descriptions, we can define *covers(H,e)* as a boolean function that results true when hypothesis H covers example e and *covers(H, E)* as a function that results the set of examples in example set E, covered by hypothesis H. A hypotheses is said to be *complete* if it covers all the positive examples and *consistent* if it covers no negative examples. In this

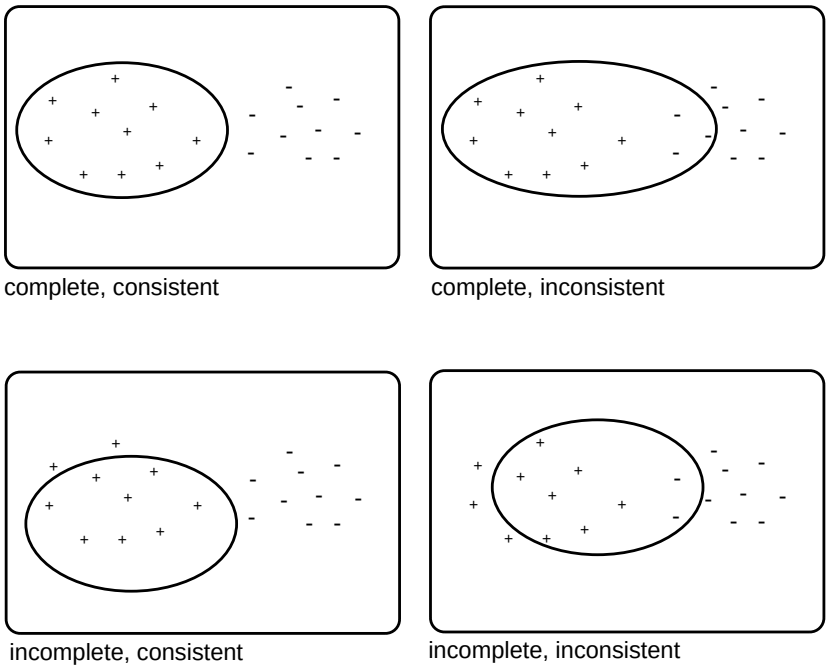

Figure 2.2: Completeness and consistency + and - signs represent positive and negative examples, respectively and the elips represents the coverage set of the hypotheses

context, a hypotheses can be one of four states with respect to a given example set, including positive and negative examples, as illustrated in in Figure 2.1. In this context, learning a concept can be defined as the task of finding a hypotheses H for a concept C, that is both complete and consistent.

In certain aspect, inductive concept learning can be defined as searching the correct description among the space of all possible concept descriptions [21], which can be very large for difficult problems. The search space may shrink with the usage of additional clauses about the concept, known in prior, namely, *background knowledge*. With the help of background knowledge, the concepts might be expressed closer to the descriptions in human mind. The background clauses might be presented in different forms such as Horn clauses form or First Order Clausal Form. Considering the background knowledge, the *covers* relations must be extended as follows:
$covers(B, H, e) = covers(B \wedge H, e)$, for a single example.
$covers(B, H, E) = covers(B \wedge H, E)$, for an example set.

Examples	Background Clauses
daughter(mine, aylin). $\oplus$	parent(aylin, mine). female(aylin).
daughter(elif, tolga). $\oplus$	parent(aylin, tolga). female(mine).
daughter(tolga, aylin). $\ominus$	parent(tolga, elif). female(elif).
daughter(elif, aylin). $\ominus$	parent(tolga, ibrahim). male(tolga).

Table 2.1: The daughter example

where B is the set of background clauses. The coverage function, which denotes whether a fact can be deduced from a theory or hypotheses, can be implemented using several different ways in Logic and ILP. SLD-Resolution proof [17], is the mostly used procedure for this purpose, and is mainly based on the variable substitution and resolutions using logic rules.

The notions of completeness and consistency need to be redefined considering the background knowledge, where E^+ and E^- denote sets of positive and negative examples, respectively.
A hypothesis H is complete with respect to background knowledge B and examples E if $covers(B, H, E^+) = E^+$.
A hypothesis H is consistent with respect to background knowledge B and examples E if $covers(B, H, E^-) = \phi$.

Learning a relational concept description in terms of given examples and background clauses in the language of logic programs is named as logic program synthesis or inductive logic programming [26], shortly ILP.

Learning daughter relation is a simple ILP problem where the learning task is to define the predicate *daughter(X, Y)*, which describes the case that person X is daughter of person Y. As an example, consider that we have an example set consisting of two positive (denoted with $\oplus$) and two negative examples (denoted with $\ominus$), and background family clauses as in Table 2.1, where *parent(X, Y)* denotes that person X is parent of Y and *female(X)* has its obvious meaning.

We expect an ideal ILP system to induce the following hypothesis:
$daughter(X, Y) \leftarrow female(X) \wedge parent(Y, X)$.
which is the correct description of the concept.

2.2 History of ILP

The history of induction dates back to Socrates' dialogs noted in Plato's Crito [26]. In these dialogs, concepts are developed and refined by means of examples and counter-examples from everyday life. In 17th century, Bacon was the first to give the formal description of inductive scientific method in his book *Novum Organum*. Methods developed for predicting the outcome of chance games formed the basis of statistics, which was used in the evaluation of scientific hypotheses in 18th century.

The discussion on ability of machines to learn from examples first came out when Turing suggested the use of an oracle to derive the incompleteness of logical theories [12, 47, 48]. From the statistical perspective, Carnap developed theories to confirm the correctness of theories expressed in first-order form. Plotkin [33] and Shapiro [45] worked on inductive inference based on Predicate Calculus.

Plotkin's work in his PhD thesis [33] formed the basis of current bottom-up generalization methodology in ILP. Since logic programming was not present at that time, he developed his theories independent of Horn clause logic. He introduced two important concepts that shed light on the generalization research:

- relative subsumption, which defines the generality between two clauses.
- relative least general generalization and its inductive mechanism.

But he also noted the fact that there was no guarantee that least general generalization of two clauses is finite, and this restricted his relative least general generalization implementation. This inefficiency motivated Shapiro to follow a general to specific approach and use algorithmic debugging in MIS [45]. In this technique, the faulty clause that causes the logic program to be incomplete or incorrect was found and replaced with a better clause to make the system consistent.

First area that an ILP system was used in a real life domain is construction of expert systems. Early expert systems were developed by hand coded rules,

which required vast amount of labor to develop and maintain, therefore they were limited in the number of rules and had high costs. GASOIL and BMT were the first expert systems that enjoyed atuomated induction performed by Quinlan's inductive decision tree building algorithm ID3 [34]. These two systems illustrated the great amount of benefit in terms of software engineering that can be gained by automated induction.

Quinlan later introduced FOIL [35], which is an efficient program that induces first-order clauses and is based on general to specific search relying on the entropy of the invented clauses. Quinlan noted that his approach in FOIL is natural extension of ID3 and admitted that his search heuristic may not find the solution for some concepts such as list reverse and integer multiplication.

In [2], a generalization system MARVIN was introduced, which generalizes a single example at a time. Muggleton and Buntine [44] would show that this generalization was a special case of inverting a resolution proof.

To overcome the limitations of Plotkin's LGG, various attempts had been made. Muggleton and Feng [43] developed GOLEM, a system that was based on the inverse resolution which Sammut and Banerji applied a special case in MARVIN.

Recently, Muggleton introduced Progol, which is a sophisticated system that makes use of type and mode declarations to GOLEM to achieve better efficiency. Progol showed its efficiency in many domains and is the most common ILP learner at the moment. The implementation is publicly available for research and licensed for commercial use.

2.3 Classification of ILP paradigms

ILP paradigms can be classified in two aspects: presentation of input and the search strategy. In terms of input presentation, the paradigm may be empirical or interactive [16]. In terms of search strategy, the paradigm may be top-down

or bottom-up [10].

2.3.1 Empirical vs. Interactive

Empirical systems are those that take the input example set and background clauses at once and produce the hypothesis and give it as output. Interactive systems start with an example set, produce a hypothesis and incrementally update it by the answers of questions that are directed to an oracle by the system.

While most empirical systems force the background clauses to be ground, most of the interactive systems allow nonground clauses. Another advantage of the interactive systems is that they can learn multiple predicates while empirical systems can learn only a single predicate in general.

Some examples of empirical ILP systems are, FOIL [35], mFOIL [8], GOLEM [43], Progol [27], LINUS [31], MARKUS [13] and MOBAL [24]. Interactive ILP systems include MIS [45], CLINT [37], CIGOL [44] and MARVIN [2].

2.3.2 Top-down vs. Bottom-up

Top-down ILP methods generate caluses by means of specialization, that is, they start with the most general clause and specialize it by iteratively restricting it by body literals, so that it does not cover any negative examples. Bottom-up methods work by generalization, which is described as process of building a general description from specific examples in order to predict the classification of new data [19].

Most bottom-up approaches take their root from Plotkin's LGG schema, which is the first sound description of the generalization process for inductive inference. Some wellknown bottom-up ILP systems are GOLEM [43], IRES [42], ITOU [41], CLINT [37], CIGOL [44]. Top-down methods generally make use of statistics and refinement graphs to build and select clauses. Some examples of top-down systems are FOIL [35], FOCL [32], MIS and MARKUS [13].

2.4 Applications

Extensive research has been performed in ILP in last decade and it has been applied in many domains. First and most common area is construction of expert systems, as mentioned in 2.2. Another application domain is knowledge discovery in databases [50]. Lastly, ILP is used for scientific discovery, theory formation, experimental design and theory verification [38].

Knowledge acquisition is a time consuming and difficult task in the process of building expert systems, since it is necessary to observe and interview with domain experts, who usually have difficulty in expressing their experiences in computational formalism. This problem is named as *knowledge acquisition bottleneck* and inductive logic technologies can be helpful for partial automatization of knowledge acquisition phase providing better efficiency than coventional dialogue based techniques [1].

One of the well-known knowledge acquisition tool based on ILP is MOBAL [24], which is a model inference system. This system has three components. First one extracts models from rules, second one classifies the models that has been extracted and the other builds a model hierarchy. Another learning system, DISCIPLE [46] is used for interactively building knowledege bases. DISCIPLE has three learning modules, a knowledge base and a expert system shell.

Database knowledge discovery research is interested in extracting implicit, unknown information from big databases that may have potential good [50]. Conventional Machine Learning systems construct a single relation, attribute-value solution. But ILP makes use of the interdepencies and other relations among the data.

Several ILP systems such as FOIL, GOLEM and LINUS have been applied in database knowledge discovery and gave promising results. But these systems learn a single predicate at a time. In order to capture the relational interdependencies, multiple predicate learners such as MOBAL, MPL [38] and CLAUDIEN [5] should

be preferred.

Scientific knowledge discovery is parallel to building expert systems in the aspect of construction steps [38]. In both processes, new piece of information, namely hypothesis, is extracted by generalizing observations or examples with the help of domain knowledge. ILP can aid scientific discovery process in following steps [16]:

- interactive generation of experiments,
- generating the logical theory from the observations.
- testing the logical theory.

For first step, only interactive ILP systems can be applied, such as MIS and CLINT. For generating the theory, both class of ILP frameworks can be used. An empirical system, GOLEM has been applied and gave sound results that are published in the scientific literature [30]. FOIL and LINUS are other systems that are applied in theory generation.

ILP shows potential use for several application areas. Some are satellite fault diagnosis [11], predicting secondary structure of proteins [29] and finite element mesh design [6].

2.5 Common ILP Systems

Although there are many ILP systems due to vast amount of research as discussed in previous sections, we will discuss five systems, which have major importance and impact in ILP field. These are CIGOL, which is based on inverse resolution, MIS, which relies on a breadth first search of refinement graphs, FOIL, which is based on entropy calculation, GOLEM ,which is build upon the idea of Plotkin's RLGG and Progol, which integrates modes and types to GOLEM.

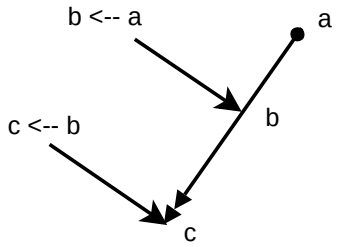

Figure 2.3: Simple resolution procedure

2.5.1 CIGOL

CIGOL (inversely read LOGIC) is an interactive learning system that is built on the basic idea of inverse resolution, which is the inverse of the resolution rule that is used to prove the correctness of logic programs.

2.5.1.1 Resolution

Introduced by Robinson [40] in 1965, resolution rule had great influence in Logic Programming paradigm and has been almost the standard method to prove logical theories. Rather than giving its theoretical definition, we will explain it with an example.

Suppose we have a theory $T = \{c \leftarrow b, b \leftarrow a, a\}$ we want to derive c. Firstly, the fact a resolves with $b \leftarrow a$ to give b. Then b resolves with $c \leftarrow b$, giving c. The resolution procedure is illustrated in Figure 2.3

Although the resolution is simple when clauses are ground, the procedure gets more complex because of need for substitution when there are variables in the theory. Consider we have the daughter relation as the theory:
$H = \{c\} = \{daughter(X, Y) \leftarrow female(X), parent(Y, X).\}$
The background knowledge consists of two facts:
$b_1 = female(mine)$.
$b_2 = parent(aylin, mine)$.
and we want to derive the fact $daughter(mine, aylin)$.

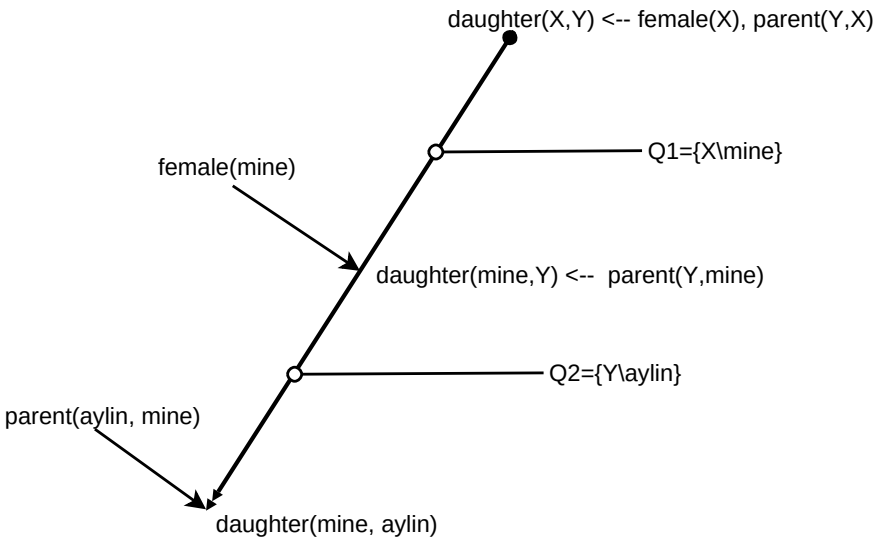

Figure 2.4: The resolution tree for deriving daughter fact

Firstly, clause c is resolved with clause b_1. Therefore, $female(mine)$ and $female(X)$ in the body of the clause are unified and variable X is bound to constant *mine*. The resolution result is:
$c_1 = daughter(mine, Y) \leftarrow parent(Y, mine)$.
Next, c_1 should be resolved with b_2 under the substitution $\{Y/aylin\}$ giving the clause:
$c_2 = daughter(mine, aylin)$.
Therefore the fact is derived. Figure 2.4 shows the resolution tree for this example.

2.5.1.2 Inverse Resolution

Inverse resolution works in the same way but opposite direction with proof resolution procedure. Suppose the background knowledge is same as in the previous example and we encounter the positive example *daughter(mine, aylin)*. Initially, the fact *daughter(mine, aylin)* is inversely resolved with *parent(aylin, mine)* giving the clause $daughter(mine, aylin) \leftarrow parent(aylin, mine)$ as the result.

Applying inverse substitution $\{aylin/Y\}$ results as:
$daughter(mine, Y) \leftarrow parent(Y, mine)$.

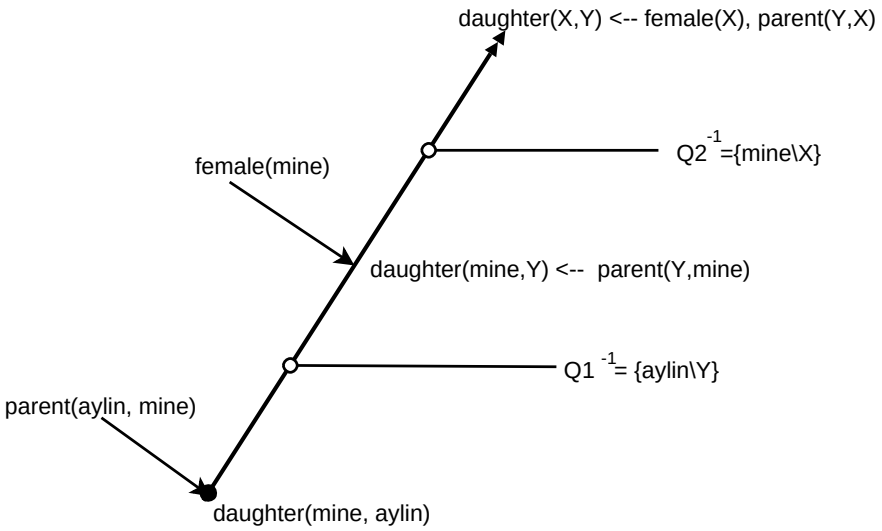

Figure 2.5: Inverse resolution of daughter relation

In the next step, this clause is inversely resolved with *female(mine)* to give $daughter(mine, Y) \leftarrow parent(Y, mine), female(mine)$.
Finally the inverse substitution $\{mine/X\}$ takes place and we get the hypothesis $H = \{c\} = \{daughter(X, Y) \leftarrow parent(Y, X) female(X).\}$
which is the generalization of the example with respect to background knowledge. Figure 2.5 illustrates the inverse resolution procedure.

CIGOL is mainly based on inverse resolution principle. The operation carried in the previous example is called the absorption and represented with symbol 'V'. There are also other operators used in CIGOL. One of them is intra-construction, which is denoted by 'W' and is capable of inventing predicates that are not encountered among the example predicate and background predicates. This may be a very important and useful feature for some concepts to be learned.

Like CIGOL, we build clauses in a bottom-up manner (from specific to general) in our system, but our heuristic based on the specific generalization rather than the inverse resolution and we do not invent new clauses.

```
Hypotheses H ← φ
loop
  Process the next example
  while H is incomplete or inconsistent do
    if H covers a negative example e then
      Delete the clauses causing H to cover e.
    end if
    if There exist a positive example e not covered by H then
      Develop a clause c that covers e by a breadth-first search through the
      refinement graph.
      Add clause c to H.
    end if
  end while
end loop
```

Figure 2.6: MIS Algorithm

2.5.2 MIS

Developed by Ehud Shapiro in 1983, MIS (standing for Model Inference System) was one of the first attempts for inductive logic program synthesis making use of logic programming. MIS employs refinement graphs, which are directed, acyclic graphs that contain the most general clause at the root and the output clauses at the leaves. The arcs represent refinement operators which are either addition of a literal or substitution of a variable with a term. The fundamental MIS algorithm is listed in Figure 2.6.

We will explain how MIS works by using the family example in Table 2.1. Since MIS [45] is an interactive system, the examples will be processed in turn. Initially the hypothesis set consists of the empty clause, which is a contradiction. When first example $e_1 = daughter(mine, aylin)$ is processed, the most general definition of daughter predicate
$daughter(X, Y) \leftarrow$.
is asserted. At this stage, the hypothesis includes a single clause:
$H = \{c\} = \{daughter(X, Y) \leftarrow .\}$
which covers example e_1. Then the second example is presented. This clause also covers example $e_2 = daughter(elif, tolga)$, so it is left intact. Next, the negative

example $e_3 = daughter(tolga, aylin)$ is processed. The example is covered by c although it is negative, therefore the clause needs to be refined by adding a literal to its body. There are two types of literals that can be added at this stage:

- The literals having variables appearing in the head of the clause. These are: $X = Y, female(X), female(Y), parent(X, X), parent(Y, Y), parent(X, Y), parent(Y, X)$.
- The literals introducing new variables. These are: $parent(X, Z), parent(Z, X), parent(Y, Z), parent(Z, Y)$.

where X, Y and Z are variables with different contents.

First, the literal $X = Y$ is tried, but
$daughter(X, Y) \leftarrow X = Y.$
covers none of the examples, therefore it is eliminated. Second, the clause
$daughter(X, Y) \leftarrow female(X)$
is considered. This clause covers two positive examples e_1, e_2 and does not cover negative example e_3. Therefore it is kept as the output of the third step and hypothesis is:
$H = \{c\} = \{daughter(X, Y) \leftarrow female(X).\}$

Then we return to the outer loop and process the negative example $e_4 = daughter(elif, aylin)$. Since clause c covers e_4, it is deleted from hypothesis and search is reinitiated to cover positive examples as in the previous step. Neither of the refinements of
$daughter(X, Y) \leftarrow$.
discriminates examples, therefore refinements of its children are considered. First, refinements of
$daughter(X, Y) \leftarrow X = Y.$
are tried, but obviously none of them cover example e_1, so they are discarded. Second, refinement of
$daughter(X, Y) \leftarrow female(X).$
is considered, and it is discovered that the clause

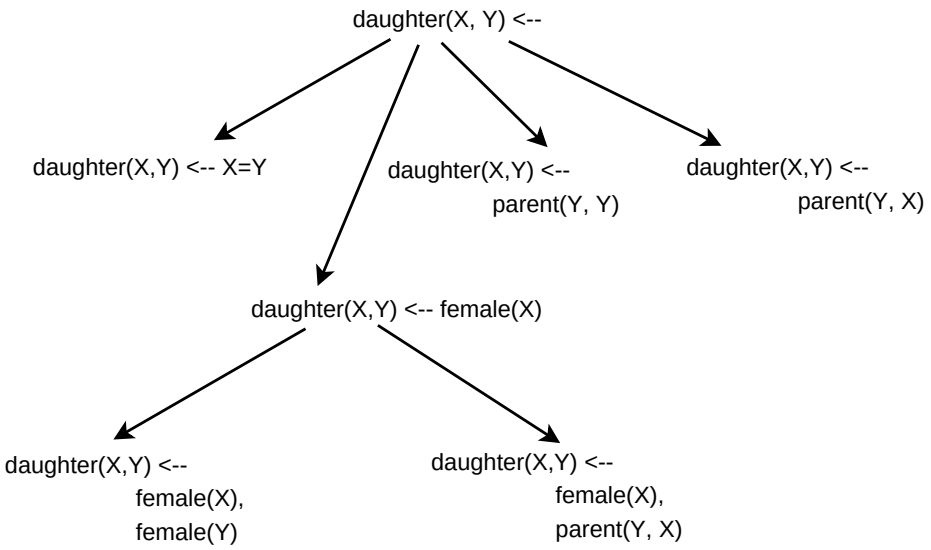

Figure 2.7: The refinement graph for inducing daughter relation

$daughter(X, Y) \leftarrow female(X), parent(X, Y).$
is both complete and consistent with respect to the given example set and it is put into the hypothesis. Finally, our hypothesis will be:
$H = \{daughter(X, Y) \leftarrow female(X), parent(Y, X).\}$
which describes the concept correctly.

Unlike MIS, our system is placed in the empirical category, and it processes the input literals in pairs, rather than one by one.

2.5.3 FOIL

Inheriting its information based heuristic search, First-Order Inductive Learner (FOIL in short) is natural extension of ID3, as Quinlan comments [35]. It also follows similar covering approach to AQ, as described in Figure 2.8 and top-down search similar to MIS, as discussed in Section 2.5.2 [35].

FOIL accepts function-free ground facts as examples and background knowledge. Negative examples are optional, since the initialization step produces negative examples by relying on closed-world assuption, that is all the possible inputs

$E_{cur} := E$
$H := \phi$
while There are positive examples uncovered **do**
  initialize clause $c := T \leftarrow$.
  $c := specialization(c, E_{cur})$
  $c := postprocess(c)$
  $H := H \cup c$
  $E_{cur} = E_{cur} - cover(B, c, E_{cur})$
  Break if encoding constraint is violated
end while

Figure 2.8: FOIL covering algorithm inherited from AQ family

except positive examples are labeled as negative. The hypothesis language of FOIL consists of function-free program clauses where there is no constants or compound terms. Predicates of body literals of the output clauses can be background predicates or the target predicate, meaning that the recursive clauses can be induced. No new predicate is invented in the procedure and no free variable is allowed, that is, at least one of the variables in the body of an output clause must also appear in its head or some other literal.

Like other top-down approaches, FOIL operates in three steps:

1. Pre-processing of example set
2. Construction of hypothesis
3. Postprocessing of hyhpothesis

Negative examples are produced in first step, if not given. Hypothesis, which may contain several clauses with same predicate, is constructed with main covering algorithm. Last step eliminates the errors that may arise because of noise. The implemented covering algorithm is basically as in Figure 2.8.

The specialization function finds the best literal with repect to selection criteria and constructs the clause by adding a literal to the body of the clause in a loop. The specialization algorithm is as in Figure 2.9.

while $cover(c, B, E^-_{cur}) \neq \phi$ and encoding constraints are not violated **do**
  Find the best literal L to add the body of $c = T \leftarrow Q$
  $c := T \leftarrow Q, L$.
end while
return c

Figure 2.9: FOIL specialization method

The best literal is found by using weighted information gain, which is calculated by computing the entropy of adding a literal as follows:
Let c_i denote the state of clause at step i, and c_i^+, c_i^- denote number of positive and negative examples represented by this clause at step i, respectively. Information needed to signal positivity of an example is with this clause is:
$I(c_i) = -log_2(c_i^+/(c_i^+ + c_i^-)$

In this context, let c_{i+1} denote the state of the clause after adding literal L_i to the body of the clause c_i, and c_i^{++} denote the number of positive examples covered by both c_i and c_{i+1}. Weighted information gain that is obtained by adding literal L_i to the clause body is calculated by:
$Gain(L_i) = WIG(c_i, c_{i+1}) = c_i^{++} * (I(c_i) - I(c_{i+1})))$
In each state of the specialization algorithm, the literal that offers highest weighted information gain is added to the body of the clause.

The essential shortcoming of FOIL is that it searches the clauses greedily with one literal look-ahead. There may be cases when two single literals have zero gain but their conjuction may have high gain and may be necessary to produce the correct result. In this case, FOIL may prefer another literal that has a nonzero gain and no further specializations can be made. This defficiency is called "local pleteau problem" [39] and arises from the fact that FOIL is a hill climbing method.

We also follow AQ covering approach in our system as FOIL. But we use a different specialization algorithm, in which body literals are appended using the differing arguments of the other literals.

2.5.4 GOLEM

GOLEM is a bottom-up learner that is based on Plotkin's LGG schema. In its input language, functional terms are allowed for examples and background clauses, but they are still restricted to ground form. The underlying methodogy for generalization is as follows: The $\models$ operator denoting logical entailment, let B denote the set of background clauses and clause C the least general generalization of examples e_1 and e_2 relative to B and is used only once in the derivation of both e_1 and e_2

$B \wedge C \models e_1$

$C \models B \rightarrow e1$

$\models C \rightarrow (B \rightarrow e_1)$

$\models C \rightarrow (\neg B \vee e1)$

$\models C \rightarrow (\neg(b1 \wedge b2 \wedge ...) \vee e_1)$

$\models C \rightarrow ((\neg b1 \vee \neg b2 \vee ...) \vee e_1)$

Following the same procedure for e_2, we get

$\models C \rightarrow ((\neg b1 \vee \neg b2 \vee ...) \vee e_2)$

If we let

$C_1 = ((\neg b1 \vee \neg b2 \vee ...) \vee e_1)$ and $C_2 = ((\neg b1 \vee \neg b2 \vee ...) \vee e_2)$

Then

$\models C \rightarrow C_1 \models C \rightarrow C_2$

and we get:

$\models C \rightarrow lgg(C_1, C_2)$

The mehodology can be better illustrated with an example. Consider learning to identify a bird. The examples are:

bird(hawk).

bird(eagle).

which are both positive. The background clauses are:

haswings(hawk).

haswings(eagle).

flies(hawk).

flies(eagle).

Using the reasoning presented above, the findings are:
$C_1 = bird(hawk) \vee (\neg haswings(hawk) \vee \neg haswings(eagle) \vee \neg flies(hawk) \vee \neg flies(eagle))$.
$= bird(hawk) \leftarrow haswings(hawk), haswings(eagle), flies(hawk), flies(eagle)$.
$C_2 = bird(eagle) \vee (\neg haswings(hawk) \vee \neg haswings(eagle) \vee \neg flies(hawk) \vee \neg flies(eagle))$.
$= bird(eagle) \leftarrow haswings(hawk), haswings(eagle), flies(hawk), flies(eagle)$.
The generalization results as:
$lgg(C_1, C_2) = bird(X) \leftarrow haswings(X), haswings(hawk), haswings(eagle), flies(X), flies(hawk), flies(eagle)$.
Removing the redundant literals we get:
$bird(X) : -haswings(X), flies(X)$.

Unlike the learning case presented in this simple example, the generalized clause can contain too many literals and become extremely large to process. Therefore restrictions are imposed for variables appearing in the body of induced clauses. For this aim, authors introduce determinism, which forbids body variables that can not be determined uniquely using the values of the variables in the head of the lgg.

GOLEM picks example pairs randomly at initial step, computes their lggs and chooses the lgg that covers maximum number of examples. Then it computes the lgg of the selected clause and other positive examples. The loop continues until the generalization does not extend the coverage set. At this point, the clause is post-processed to eliminate the redundant literals to provide additional generalization.

Our system also generalizes the input clauses using Plotkin's least general generalization schema as performed in GOLEM. Unlike GOLEM, once we append a literal to the body, we never remove it from the clause.

B	E	$\perp$
$animal(X) \leftarrow pet(X).$ $pet(x) \leftarrow dog(X).$	$nice(X) \leftarrow dog(X).$	$nice(X) \leftarrow dog(X), pet(X), animal(X).$
$hasbeak(X) \leftarrow bird(X).$ $bird(X) \leftarrow vulture(X).$	$hasbeak(tweety).$	$hasbeak(tweety); bird(tweety);$ $vulture(tweety)$
$white(swan1).$	$\leftarrow black(swan1).$	$\leftarrow black(swan1), white(swan1).$
$sentence([],[]).$	$sentence([a,a,a],[]).$	$sentence([a,a,a],[]) \leftarrow sentence([],[]).$

Table 2.2: Most-specific clause for different example clause and background knowledge set pairs

2.5.5 PROGOL

In [27], the authors approach the generic ILP problem as finding the simplest hypotheses H that explains example set E, together with background knowledge B, in the finite or infinite search space of possible solutions, that is,
$B \wedge H \models E$
The authors denote that B, H and E can be arbitrary logic programs. Each clause in H must cover some positive examples, otherwise there is a simpler hypotheses H' to replace H. Considering the H and B each a single clause, using the inference as in GOLEM, the relation is converted to:
$B \wedge \bar{E} \models \bar{H}$
Then, the authors introduce the most specific clause, namely $\perp$, where $\bar{\perp}$ denotes the conjuction of all literals which are true in every model of $B \wedge \bar{E}$. Since $\bar{H}$ is true in the same model, it follows that the literals of $\bar{H}$ are the subset of the literals in $\bar{\perp}$, that is $\bar{H}$ can be deduced from $\bar{\perp}$. The relation is as follows:
$B \wedge \bar{E} \models \bar{\perp} \models \bar{H}.$
therefore for every possible solution H,
$H \models \perp.$
In this context, possible solutions can be computed by considering clauses which θ-subsume $\perp$. Some examples listed by the authors illustrating the relation between E, B and $\perp$ are listed in Table 2.2.

The first case follows from the absorption rule mentioned in CIGOL. The second case relies on the identification rule of in the same system. In the third clause, it is learnt that a swan can not be black and white at the same time,

which demonstrates how negative facts can be extracted. The last example is a special case of the grammar rule $sentence([a|X], Y) \leftarrow sentence(X, Y)$.

Progol reduces the search space by using mode declarations for the target predicate. In this context, type of every variable should be declared by the user. For instance, if there are examples such as:
$class(dog, mammal)$.
$class(shark, fish)$.
Then, the user must specify the types of the variables as follows:
$animal(dog)$.
$animal(shark)$.
$class(mammal)$.
$class(fish)$.
Furthermore, the structure of the target predicate must be declared as follows:
$class(+animal, \#class)$.
where animal and class are variable types that can occur in the argument and + symbol denoting input variable, # denoting constant (- denoting output).

Mode declarations also permit the user to declare the recall number, which specifies the number of alternatives to be tried to instatiate an atom. Declarations are used for both head and body literals. For instance
$modeh(1, class(+animal, \#class))$,
describes the head of the target predicate,
$modeb(1, has_eggs(+animal))$,
describes the structure of a possible body literal, where the integer 1 stands for the recall number.

Having clauses like these as input at hand, Progol can produce clauses like:
$class(X, mammal) :- hasEggs(X), hasMilk(X)$, or
$:- class(X, mammal), class(X, fish)$,
meaning that an *animal* can have only one class.

Progol uses an A*-like algorithm to find the hypotheses, which finds the correct one if it is reachable. It chooses the hypotheses having the greatest Occam compression, using total number of atom occurences as encoding measure, when

there are several solutions. Progol system is implemented in C programming language and available for academic research via world wide web [25].

Our system requires neither type, nor mode declarations. But it can provide better generalizations when type information is provided in background knowledge.

Chapter 3

String Generalization

3.1 Introduction

Learning by positive-only data is a difficult task in ILP due to the possible over-generalization caused by the lack of restriction induced by negative examples. But in real-life, we have many domains where we have only positive examples such as Grammar Learning and Machine Translation. There has been attempts [7, 23, 28] to propose a solution for learning from positive-only data such as statistical techniques using prior probabilities or closed world assumption. In closed world assumption approach, every possible ground clause not given in the positive example set is produced by the system and labeled as negative.

Predicates defined on string arguments occur in many domains such as Grammar Learning and Machine Translation. In [4], the authors propose a solution for learning predicates that have string arguments in domains having no negative examples.

The proposed methodology is based on the notion of unique match sequence, which is based on similarities (subsequences occurring in both strings) and differences (subsequences differing among strings) of two strings. The unique match sequence is generalized using Plotkin's LGG schema.

Suppose we have two positive examples with predicate *endsWith* in Prolog notation, where lists represent strings:

```
endsWith([a,b], [x,y]).
endsWith([c,d,b], [w,z,y]).
```

Although these two predicates share the common property that first argument is a list ends with b, and second argument is a list ends with y, GOLEM, which also uses LGG schema, overgeneralizes this pair with result:

```
endsWith([A,B|C],[D,E|F]).
```

which accepts all *endsWith* predicates with list pair having length at least two as input.

The output of Progol, which is based on similar principles with GOLEM is:

```
endsWith([a,b], [x,y]).
endsWith([c,d,b], [w,z,y]).
```

which overfits on the examples and covers nothing more.

The string generalization technique proposed in [4] learns the following clause with the same example pair:

```
endsWith(L1,L2) :- append(X,[b],L1), append(Y,[y],L2).
```

which accepts clauses with predicate *endsWith*, and the last elements of the first and second arguments are b and y. respectively. This corresponds to *p(Xb, Yy)* in string case.

3.2 Preliminaries

The mentioned methodolgy makes generalizations by processing similarities and differences of strings. A match sequence is the sequence of similarities and differences between two strings. Informally, a similarity between two strings is common subsequence of symbols and a differences are the subsequences between similarities. For a string pair $(abcd, abe)$; ab is the similarity and (cd, e) represents the difference.

Although the string pair $(abcd, ecfg)$ has a single match sequence $(ab, e)c(d, f)$, the pair $(abc, dbebf)$ has two match sequences $(a, d)b(c, ebf)$ and $(a, dbe)b(c, f)$ since b appears twice in the second string.

In the article, a specific case of a match sequence, the notion of unique match sequence is defined with two additional restrictions on a match sequence:

- Symbols occuring similarities and differences constitute two disjoint sets. This rule enforces that, a symbol occuring in one of the similarities can not occur in any difference.
- Symbols of first and second constituents of differences constitute two disjoint sets. This rule enforces that, common symbols can only occur in similarities.

These two restrictions together provide only string pairs whose common symbols occur same number of times in the same order to have a unique match sequence.

Some examples that can help to clarify the notion of unique match sequence are:

- *UMS(abceb, fgbhb) = (a,fg)b(ce,h)b.*
- *UMS(ab, ab) = ab.*
- *UMS(abc, xyz) = (abc, xyz).*

- *UMS(abcb, dbebf) = (a,d)b(c,e)b(ϵ,f).*
- *UMS(abc, abdb) = ϕ.*
- *UMS(ab, ba) = ϕ.*

The authors introduce the notions of seperable and seperation differences are to provide further capturing of similar patterns. In short, difference (D_1, D_2) is said to be seperable by difference (d_1, d_2) if d_1 and d_2 occur same number of times and grater than zero in D_1 and D_2, respectively. We say that a difference (D_1, D_2) is divided by another difference (d_1, d_2) with separation factor n where n is the number of times d_1 occurs in D_1 and d_2 occurs in D_2.

For instance, the difference *(aba,cdc)* is seperable by difference *(a,c)* with factor 2. Hovewer, the difference *(aba,cd)* is not seperable by difference *(a,c)* since a occurs twice in the first constituent while c occurs in the second constituent only once.

Seperation of a difference (D_1, D_2) with seperation difference (d_1, d_2) is the sequence $(\alpha_1, \beta_1)(d_1, d_2)(\alpha_2, \beta_2)(d_1, d_2) \dots (d_1, d_2)(\alpha_n, \beta_n)$, where D_1 consists of the sequence $\alpha_1 d_1 \alpha_2 d_1 \dots d_1 \alpha_n$ and D_2 consists of the sequence $\beta_1 d_2 \beta_2 d_2 \dots d_2 \beta_n$, and empty differences are dropped. Seperation of a match sequence with a difference is the sequence of similarities and seperation of all differences with that difference.

In the framework terminology, the seperation differences that seperate all the differences in that match sequence and increase the number of differences more than once after the seperation of a difference are discriminated as useful. As an instance of this concept, while *(a,b)* is a useful separation difference for match sequence *(ac,bde)g(a,b)* since the total number of differences which occur more than once is increases from 0 to 2 after the seperation, *(ab,d)* it is not a useful seperation difference for this difference since the same parameter does not increase after the seperation.

For a match sequence to be seperated, the authors describe the most useful seperation difference as the one among useful seperation differences that seperates the match sequence with the greatest factor. If there are more than one useful

$specInstance \leftarrow ums(\alpha_1, \alpha_2)$
while there is a $MUSD$ that seperates specInstance with factor ≥ 2 **do**
  $specInstance \leftarrow seperation(specInstance, MUSD)$
end while
return $specInstance$

Figure 3.1: Finding Specific Instance

seperation differences seperating with the greatest factor n, the seperation of the match sequence with most useful seperation difference should be still seperable by the other differences with factor n.

There can be many useful separation differences for a match sequence but there is at most one most useful separation difference. For instance, the most useful separation difference for match sequence *(cac,bdb)g(cf,bg)* is *(c,b)* with separation factor 3. For match sequence *(ab,c)g(ab,c)*, there is no most useful separation difference, because neither of *(a,c)* and *(b,c)* has the superiority over the other.

3.3 Methodology

3.3.1 Finding Specific Generalization

Once unique match sequence of a string pair is found (if there is), the best (not always most) specific instance of the sequence is computed by the algorithm in Figure 3.1. In this algorithm, specific instance of a match sequence is computed by dividing the match sequence iteratively by the most useful seperation difference. The iterations continue until none of the useful separation differences can be favored among others.

The specific generalization of strings α_1 and α_2 is computed (if exists) by the algorithm in Figure 3.2. In this algorithm, inverse substitution step is the operation of replacing differences with variables, with the restriction that same differences correspond to same variables in the result.

if $ums(\alpha_1,\alpha_2)$ does not exist **then**
  There is no possible generalization
else
  $UMS \leftarrow uniqueMatchSequence(\alpha_1,\alpha_2)$
  $SIofUMS \leftarrow specInstance(UMS)$
  $SG \leftarrow InverseSubsitute(SIofUMS)$
end if

Figure 3.2: Finding Specific Generalization

As an instance that shows how specific generalization works, consider the generalization of a string pair *abcdfc* and *abghefg*. The common subsequences of these strings are *ab* and *f*. Therefore the unique match sequence of the pair is *ab(cd,ghc)f(c,g)*. For this match sequence, *(c,g)* is the mosy useful seperation difference with seperation factor 2. The seperation of the sequence with this difference gives the new sequence: *ab(c,g)(d,he)f(c,g)*. Since there is no most useful seperation difference for this new sequence, we conclude that *ab(c,g)(d,he)f(c,g)* is the most specific instance for the generalization of the string pair. Applying the inverse substitution process, we get the generalized string *abXYfX* as the result of the specific generalization procedure.

A generalized string is a sequence of characters and variables such as *abX*, which represents all strings starting with *ab*. The generalized set *GS* of a generalized string is all the possible strings that are represetnted by that string. For instance, *GS(abX) = All strings starting with ab.*

3.3.2 Generalizing Predicates

The proposed method for generalizing predicates is a coverage procedure based on specific generalization of strings. Every generalization rule includes append predicate implicitly in their bodies. For instance, a predicate definition noted as *p(Xa)* corresponds to

```
p(L) :- append(X,[a],L)
```

GEN(S)	ba	cda	a	aa	faga
Examples used	{1}	{2}	{3}	{4}	{5}
EG set	{1}	{2}	{3}	{4}	{5}

Table 3.1: String generalization, initialization step

GEN(S)	*Xa*	*XaYa*	*ba*	*cda*	*a*	*aa*	*faga*
Examples used	{1, 2, 3}	{4, 5}	{1}	{2}	{3}	{4}	{5}
EG set	{1, 2, 3, 4, 5}	{4, 5}	{1}	{2}	{3}	{4}	{5}

Table 3.2: String generalization, computing generalizations

in Prolog notation.

Two clauses having string arguments are generalized using specific generalization of their arguments if exists. The generalization of two strings α_1, α_2 is their specific generalization, if their specific generalization exists, and it is not a (most general) single varaible X.

Assume that S is a set of ground strings $\alpha_1, \alpha_2, \ldots, \alpha_n$. EG($\alpha$) represents set of ground strings represented by α, where α is a ground or generalized string. To construct the generalized set GEN(S) for a set of strings S, generalizations of all string pairs are computed and put into GEN(S). In the second step, among the generalizations that cover the same examples, the more specific one is kept and the other is removed from the set. Next, the generalizations whose coverage sets are subset of coverage of another generalization are removed from the set. Lastly, if there are generalizations such that all the examples that it covers are also covered by another subset, they are removed from the generalization set. Then S is initialized to GEN(S) and the whole procedure is repeated until there is no possible generalization that can be computed.

To illustrate how the algorithm works, consider the example clause set $\{p(ba), p(cda), p(a), p(aa), p(faga)\}$. Firstly, GEN($S$) is initialized to the set of arguments $S = \{ba, cda, a, aa, faga\}$ as in Table 3.1.

In first iteration, *Xa*, which is the specific generalization of *ba, cda, a*; and *XaYa*, which is the specific generalization of *aa, faga* are added to GEN(S) as in Table 3.2.

GEN(S)	Xa
Exs	$\{1, 2, 3\}$
EG	$\{1, 2, 3, 4, 5\}$

Table 3.3: String generalization, final result

Since EG(ba), EG(cda), EG(a), EG(aa), EG($faga$) and EG($XaYa$) are all subsets of EG(Xa), they are removed from the generalization set and generalized clause set will consist of a single clause in the end, which is p(Xa) as in Table 3.3.

The predicates with multiple string arguments can be generalized in the same way with a little modification. The argument sequence can be treated as a single string separated with a special symbol such as ':', which must not occur as any part of the input. For instance, two example clauses such as, *p(a,bac)* and *p(d,fde)* can be treated as *p(a:bac)* and *p(d:fde)* and the resulting generalization is *p(X:YXZ)*, which corresponds to *p(X,YXZ)*. Therefore the methodology also finds the interdependencies between arguments of a single predicate.

Chapter 4

Inductive Generalization

4.1 Introduction

The heuristic described in [4] is a successfull method for string generalization with potential application areas. But with its current status, it remains a stub as an Inductive Logic System. First of all, only class of background predicates handled by the framework is those denoting the type of the variables. It does not have the ability to process background predicates having arbitrary number of arguments. Second, although the heuristic eliminates some generalizations using specification heuristic, it does not specify the exact methodology to select the hypotheses set, that covers the examples. The last point is that, there is not specific treatment for numbers, which may be necessary for learning in the domains having continuous data, such as learning mutagenecity.

As pointed out in Chapter 1, the aim of the research documented in this thesis is to develop an inductive learning system for domains with positive-only data, using the idea of string generalization proposed in [4]. For this purpose, initially, we define the concept language that our system will work with. The second point that has been worked is to extend the technique to consider arbitrary first-order background predicates. Next issue was to define a sound methodology for selecting the clauses to construct the hypotheses. In the end, we developed a

heuristic to handle numeric arguments. As the result of this effort, we hope to invent an efficient ILP learner particularly for positive-only domains.

4.2 Language

Studies on attribute-value learning paradigms suffer from the lack of a standard language and notation. Inductive learning systems take their power from the declarativeness of the language they use, and Prolog is accepted almost the standard for these systems. The methodology described in this section also takes the input in Prolog notation, but the language is restricted form of Prolog. The example set and background knowledge consist of function-free ground literals without bodies, which correspond to real-life facts. All the examples in the given set must have the same predicate as we aim to build an empirical single predicate learner, but background knowledge may include several types of predicates. In this context, a sample example set may be:
$\{daughter(sibel, ahmet),\ daughter(ceren, mehmet),\ daughter(sibel, zehra)\}$
Background knowledge may be:
$\{sister(sibel, bora),\ parent(mehmet, ceren),\ father(ahmet, sibel)\}$
Functional terms such as
$pair(card(clubs, five))$.
and variables
$prent(ayse, X)$.
are disallowed.

The output hypotheses consits of function-free Horn clauses, which can have variable in their bodies, such as:
$parent(X, cengiz) \leftarrow daughter(cengiz, X)$.

As mentioned in Chapter 2, inductive learners construct output hypotheses that consist of generalized clauses. In our system, two kinds of generalized clauses may appear in the output hypotheses: a general Horn clause with a nonempty body or a unit clause with empty body. Instances of general Horn clauses are:

$p(X,Y) \leftarrow q(X),\ \ p(X,Y) \leftarrow q(X),\ r(Y).$
Instances of unit clause are:
$p(a,X,Y),\ \ p(X,Y)\ \ p(X,Y,X)$
We do not introduce a heuristic to invent new clauses, therefore clauses having nonempty bodies can only appear only in the cases where some background knowledge is specified.

In our framework, symbolic and numeric arguments are generalized differently. Symbolic arguments are generalized based on unique match sequences and numeric arguments are generalized by computing intervals. If i^{th} argument is symbolic in some example clauses and numeric in the others, it is considered as symbolic in all of them.

4.3 Symbolic Generalization

We generalize input examples by considering all symbolic arguments as a single list, where argument boundaries are specified by the special symbol ':'. Therefore usage of this symbol as a seperate token is not allowed in the input and background knowledge set. This rule does not restrict the language, since a midlevel input can be generated by another token that does not occur in the input and post-process the output to reverse the replacement. So, the impact of preprocess is as follows:
$p([a],[b],[a,c])$ is converted into $p([a,:,b,:,a,c])$.
$p([d],[b],[d,e])$ is converted into $p([d,:,b,:,d,e])$.
From this point on, we treat each token in our system as a single symbol. That is, tokens correspond to characters, and lists correspond to strings in string generalization framework proposed in [4].

Having the argument of each example converted into a single list, we investigate the existence of unique match sequence for each pair of these lists. For a pair, if there is not a unique match sequence, we say that there is not any possible generalization for this pair. Otherwise, we compute the unique match

repeat
  $ums := uniqueMatchSequence(arg_1, arg_2)$
  $SIofUMS := specificInstance(ums)$
  for all tokens t in *SIofUMS* **do**
    if t is a difference **then**
      for all background predicates p **do**
        if p covers left and right constituents of the t **then**
          $newSpecInstance := SIofUMS + p(t_{left}, t_{right})$
          Split $newSpecInstance$ to its constituents $nsiLeft, nsiRight$
          $arg_1 = nsiLeft$
          $arg_2 = nsiRight$
        end if
      end for
    end if
  end for
until There is no such background predicate or no UMS

Figure 4.1: Extended specific generalization algorithm of InGen

sequence and search for its most useful seperation difference as defined in Section 3.2. If there is not such a difference, the specific instance is the match sequence itself. If there is a most useful seperation difference, the most specific instance is computed using the specific instance algorithm in Figure 3.1 in Section 3.3.

The symbolic generalization algorithm can be summarized as Figure 4.1. The generated clauses are specialized by appending body literals with the statement: $newSpecInstance := SIofUMS + p(t_{left}, t_{right})$.

The unique match sequence of the arguments of initial two examples given above is $([a], [d])\,[:, b]\,([a, c], [d, e])$. The most useful seperation difference for this unique match sequence is $([a], [d])$ with seperation factor 2. Therefore we seperate the UMS with this seperation difference and get $([a], [d])\,[:, b]\,([a], [d])([c], [e])$ as the result. Since there is not a useful seperation difference for this match sequence, we conclude that it is the most specific instance and can be generalized for this example pair.

Background predicates are handled at this point, after computing the specific instance. For each difference in the most specific instance, we search a background

predicate that contains both left and right constituents in one of its arguments, by using a breadth-first algorithm. If there is such a predicate, we stop the search and reconstruct the initial example pair and extend both of them using the clause that provides coverage. In this step, we also use the Prolog implication symbol ':-' to seperate the target predicate from background clauses. We also treat name of the background predicates as special string tokens. If there is unique match sequence of the new pair, the same procedure is repeated with the unique match sequence of the new pair. The procedure continues until no new unique match sequence is found or there is not a suitable background predicate for extension. In subsequent iterations, the special symbol '$--$' is used as a seperator between body clauses.

For instance, for the example clauses given above, assume the background clauses are also provided as follows:
$q([a, f])$.
$q([d, g])$.
$q([b, b])$.
$r([f], [k, m])$.
$r([g], [k, m])$.
$r([h], [h, k])$.
In the background knowledge, we see that predicate q covers both constituents of the difference $([a], [d])$, therefore we seperate the most specific instance and extend both examples as follows:
$[a, :, b, :, a, c, :-, q, a, :, f]$.
$[d, :, b, :, d, e, :-, q, d, :, g]$.
Unique match sequence of this pair is:
$([a], [d])\ [:, b, :]\ ([a, d], [c, e])\ [:-, q]\ ([a], [d])\ [:]\ ([f], [g])$
and the sepecific instance is
$([a], [d])\ [:, b, :]\ ([a], [d])([c], [e])\ [:-, q]\ ([a], [d])\ [:]\ ([f], [g])$
When repeat the same procedure for the new specific instance and discover that difference $([f], [g])$ is covered by predicate r, and the new pairs are then as follows:
$[a, :, b, :, a, c, :-, q, a, :, f, --, r, f, :, k, m]$.

$[d, :, b, :, d, e, : -, q, d, :, g, --, r, g, :, k, m]$.
having the unique match sequence:
$([a], [d])\ [:, b, :]\ ([a, d], [c, e])\ [: -, q]\ ([a], [d])\ [:]\ ([f], [g])\ [--, r]\ ([f], [g])\ [:, k, m]$
The new specific instance is:
$([a], [d])\ [:, b, :]\ ([a], [d])([c], [e])\ [: -, q]\ ([a], [d])\ [:]\ ([f], [g])\ [--, r]\ ([f], [g])\ [:, k, m]$
Next, we see that there is not any generalization covering any difference, therefore we stop and exit from this step.

Next issue is the generalization of lastly found specific instance, namely specific generalization step. In this step, similarities are kept as constants, as performed in the specific generalization algorithm listed in Figure 3.2 in Section 3.3. Note that, if there are several arguments, all ':' symbols must occur as similarities, since they occur same number of times in each input list. Last process in this step is to replace the differences with variables, respecting the rule that same differences are replaced with same variables. Therefore the generalization that will be extracted for the example pair given above is:
$[X, :, b, :, X, Y, : -, q, X, :, Z, --, r, Z, :, k, m]$.
A fter simple parsing and construction we get the generalized clause:
$p(X, b, [X, Y]) : - q(X, Z), r(Z, [k, m])$.
that correspons to the following Prolog clause:
$p(X, b, [X, Y], Z) : - q(X, Z), r(Z, [k, m]), append(X, Y, Z)$.

Following the same methodology in Section 3.3, the most general clause X is not admitted as a generalization.

4.4 Numeric Generalization

For generalization of numeric input arguments, we follow a different approach. We treat numbers as points in the set of real numbers and generalizations are intervals.

For generalization of numeric arguments, we used a modified version of hierarchical clustering [15] since we believe that it is the most similar way to pairwise

generalization. In this case, closeness in terms of distance corresponds to similarity and intervals correspond to generalizations.

Hierarchical clustering is a technique that builds a hierarchical tree of grouping among input examples. Initially every data point (example) constitutes a single cluster. Then, clusters are merged pairwise until a single cluster that contains all the data points is left. The distance between two clusters is determined by the linkage method, an there are several alternatives:

- take the minimum of all pairwise distances between elements of two clusters
- take the maximum of all pairwise distances between elements of two clusters
- take the distance between the mean of two clusters (centroid method)

We take the minimum pairwise distance to make generalizations.

Suppose that the examples set consists of seven points : $\{1.2, 1.8, 2.5, 6.0, 6.5, 8.4, 8.6\}$.
Initially every point is a single cluster:
$C_1 = \{1.2\}, C_2 = \{1.8\}, C_3 = \{2.5\}, C_4 = \{6.0\}$, $C_5 = \{6.5\}, C_6 = \{8.4\}, C_7 = \{8.6\}$
The closest cluster pair is C_6 and C_7 with distance 0.2, therefore we merge them making a single cluster:
$C_1 = \{1.2\}, C_2 = \{1.8\}, C_3 = \{2.5\}, C_4 = \{6.0\}, C_5 = \{6.5\}, C_{67} = \{8.4, 8.6\}$
The next closest pair C_4 and C_5 with minimum pairwise distance 0.5, are merged.
$C_1 = \{1.2\}, C_2 = \{1.8\}, C_3 = \{2.5\}, C_{45} = \{6.0, 6.5\}, C_{67} = \{8.4, 8.6\}$.
The next closest pair C_1 and C_2 with minimum pairwise distance 0.6, are merged.
$C_{12} = \{1.2, 1.8\}, C_3 = \{2.5\}, C_{45} = \{6.0, 6.5\}, C_{67} = \{8.4, 8.6\}$.
The next closest pair C_{12} and C_3 with minimum pairwise distance 0.7, are merged.
$C_{13} = \{1.2, 1.8, 2.5\}, C_{45} = \{6.0, 6.5\}, C_{67} = \{8.4, 8.6\}$.
The next closest pair C_{45} and C_{67} with minimum pairwise distance 1.9, are merged.
$C_{13} = \{1.2, 1.8, 2.5\}, C_{47} = \{6.0, 6.5, 8.4, 8.6\}$.
The only remaining pair C_{13} and C_{47} are merged in the end and we have a single large cluster

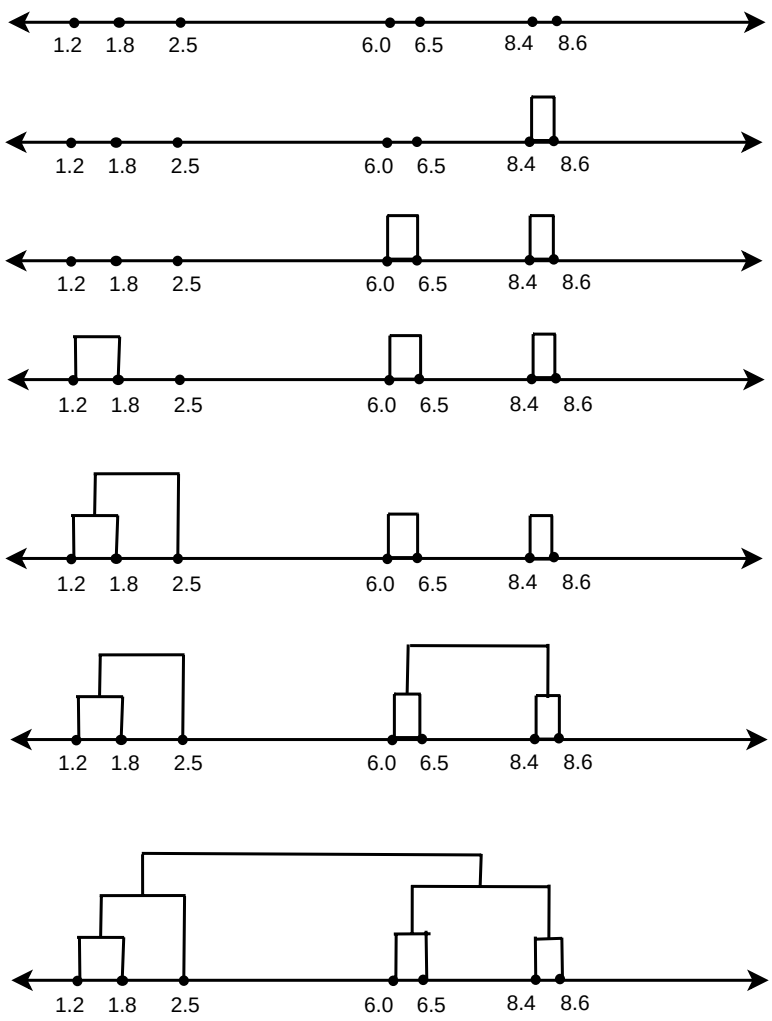

Figure 4.2: Example of hierarchical clustering

$C_{17} = \{1.2, 1.8, 2.5, 6.0, 6.5, 8.4, 8.6\}$,
which is the initial example set.
The whole procedure is illustrated in the Figure 4.2.

It is obvious that the final result of the algorithm does not provide useful information for us. The information that hierarchical clustering gives us is the clustering tree itself. Hence, extra procedure is needed to extract clusters from the the tree. For this purpose, we specify a cut-off point that indicates to stop merging process. We rely on the average distance among the input data points. That is, if the minimum pairwise exceeds the average distance among the data points which is computed at the initial step, two clusters are not merged. Since

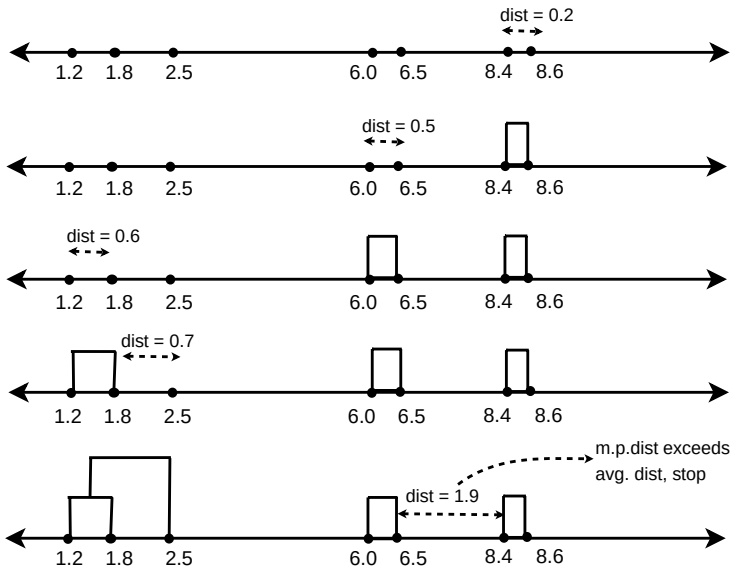

Figure 4.3: Generalization of Numbers

pairwise distance increases at each step, it is appearent that if this condition is violated at certain step, it can never be satisfied in further steps, so we stop merging. The average distance can be calculated as:
$D_{avg} = (max - min) \div (n - 1)$
where n is the set size.

For the example set given in the previous step,
$D_{avg} = (8.6 - 1.2) \div (7 - 1) = 1.23$
indicating that cluster merging process continues until the minimum pairwise distance exceeds 1.23. When merging C_{45} and C_{67}, the pairwise distance is 1.9, which is larger than 1.23, hence merging is stopped at this point and we get three clusters:
$C_{13} = 1.2, 1.8, 2.5, C_{45} = 6.0, 6.5, C_{67} = 8.4, 8.6.$
The procedure is illustrated is in Figure 4.3.

Although we have clusters at this point, our system needs generalizations (intervals in o.w). A simple choice may be to determine the borders with the smallest and largest elements of each cluster, but a bettter alternative may be adding a smoothing factor. We select the smoothing factor as the average distance computed in the initial step, that is we compute the intervals by choosing the left

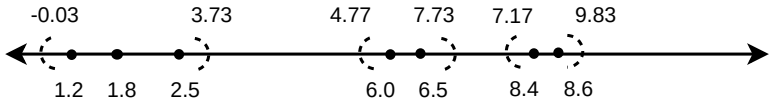

Figure 4.4: Interval computation

most and right most elements, and extending it both ends with average distance. In the example given, three intervals can be constructed from three clusters as follows

$C_{13} = \{1.2, 1.8, 2.5\}, C_{45} = \{6.0, 6.5\}, C_{67} = \{8.4, 8.6\}$.

$I_1 = [1.2 - 1.23, 2.5 + 1.23] = [-0.03, 3.73]$

$I_2 = [6.0 - 1.23, 6.5 + 1.23] = [4.77, 7.73]$

$I_3 = [8.4 - 1.23, 8.6 + 1.23] = [7.17, 9.83]$

The result is illustrated in Figure 4.4.

Note that intervals I_2 and I_3 intersect at $[7.17, 7.73]$. But this is not a problem for our system since what we are trying to do is to make a generalization, not actual clustering. It is not important which generalization covers an example, what is important is that whether an example is covered by some interval or not. An example can be covered by two generalizations, as in the case both generalizations aX and Xb covers string ab.

Although our generalization method is developed for domains with positive only data, it can easily be adapted to benefit from negative examples when there is such data. For this purpose, an additional step is necessary to (possibly) shrink the generalization intervals to make the output hypotheses consistent.

After the intervals are extracted, if a negative example is not covered by any generalizations, that is the data point does not fall in any of the intervals, no extra process is needed. If it falls in some interval, there may be two cases. First possibility is that, it may fall in left or right smoothing extensions. In this case, we shrink the interval from the side where the point falls by dividing from the middle point between the negative example and the closest positive example. If the negative example falls between two positive examples in an interval, that interval is divided into two by cutting from the middle points between the negative

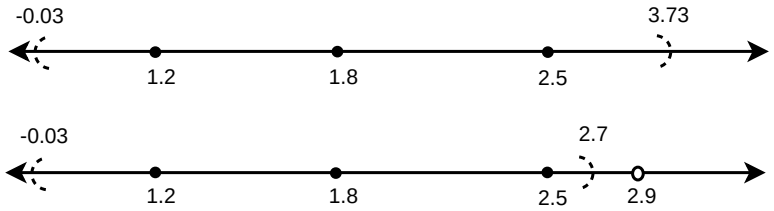

Figure 4.5: Adaption to negative examples, first case

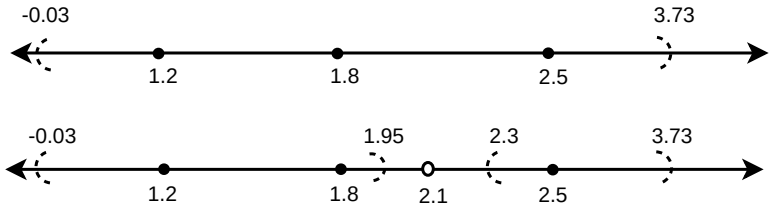

Figure 4.6: Adaption to negative examples, second case

example and the neighbouring positive examples.

For the example set given, suppose we have an extra negative example 2.9. The data point corresponding to this example falls into the right smoothing extension of first interval I1 = [-0.03, 3.73]. In this case, that interval is shrinked from right and the new upper bound is (2.5 + 2.9) / 2 = 2.7. The state of the interval before and after the nagative example is processed, is displayed in Figure 4.5.

Consider that the negative example is 2.1, which falls between two positive examples, 1.8 and 2.5, included in the first interval. In this case, the interval is divided into two, from point 2.1. The left end of the left subinterval and the right end of the right subinterval remain same. The right end of the left subinterval becomes (1.8 + 2.1) / 2 = 1.95. The left end of the right subinterval becomes (2.1 + 2.5) / 2 = 2.3. The resulting intervals are [-0.03, 1.95], and [2.3, 2.73] as shown in Figure 4.6.

4.5 Construction of the Hypotheses

In our system, the hypotheses set consists of the general clauses computed by extended specific generalization algorithm and the examples themselves. We follow the coverage algorithm similar to the one applied in FOIL. From a set of generalized clauses, we select the best clause with respect to our selection criteria, put it into the hypotheses and remove the examples covered by this clause. In the next step, the clauses that do not cover any example are removed from the clause set and the best clause from the remaining clause is chosen. Disjunction of selected clauses build the hypotheses. The iteration procedure continues until all the examples are covered by the hypotheses.

For learning predicates having mixed arguments (symbolic and numeric), all the symbolic arguments are considered as a single list as described in Section 4.3, and each numeric argument is considered as an individual group. Initally, numeric intervals are produced. Then, in the specific generalization step, for each argument, we investigate whether there is an interval that covers numeric arguments of each example. If there is one, we generalize the examples and associate the clause with the body literals *real(N), between(N,a,b)* denoting N is a real number between two real numbers a and b. If there is not such an interval, numeric argument is directly generalized as *real(N)* .

Consider a simple generalization task of generalizing three examples such as:
p(a,b,2.1).
p(c,b,2.7).
p(d,b,6.9).
p(e,b,8.1).
Generalization of numeric arguments produces two intervals $I_1 = [0.1, 4.7]$, $I_2 = [4.9, 10.1]$. In the specific generalization of *(a,b,2.1)* and *(c,b,2.7)*, we observe that most speicific instance of *(a,b)* and *(c,b)* is *(a,c)b* and both numbers are elements of the interval I_1, therefore the generalization of these examples are XbI_1. In the same manner, specific generalization of *(d,b,6.9)* and *(e,b,8.1)* is XbI_2. There is not any interval covering the numeric arguments of the first and third examples, therefore their generalization is XbI.

For this example set InGen extracts three generalized clauses:
p(X,b,N) :- real(N), between(N,0.1,4.7).
p(X,b,N) :- real(N), between(N,4.9,10.1).
p(X,b,N) :- real(N).
but only puts the first two of them into the hypotheses as they cover the whole set together and are preferred over the last one because of the selection criteria described below.

For selecting the best clause to add the hypothesis, various criteria are applied for selection of best clause in the following order:

1. Number of free variables, which are the variables that appear only once in the body of a generalized clause. The clauses having fewer free variables are preferred over the others, since we believe that greater number of free variables lead to irrelevant clauses.

2. Number of body literals. The clauses having greater number of background literals are favored over the others, since we believe that each background literal introduces further specialization and more specific generalizations.

3. Number of examples that are covered by the clause. The clauses that cover more examples are favored over the others.

For the generalized clauses generated in the case described above, we see that none of them contains free variables, and first two clauses contain greater number of body literals and cover the whole set together, therefore this pair is chosen as the hypotheses.

As another sample case, we recall the daughter example again as in Table 2.1. With this input at hand, The initial clauses generated by InGen are:
$c_1 = daughter(senay, X_0) : -parent(X_0, senay).$
$c_2 = daughter(nese, X_0) : -parent(X_0, nese).$
$c_3 = daughter(X_0, X_1) : -parent(X_1, X_0), female(X_0).$
$c_4 = daughter(X_0, X_1) : -parent(X_2, X_0), female(X_0), parent(X_1, X_0).$
The values of selection criteria for each of the clauses are as in Table 2.1

Clause	Free variables	Background literals	Covered examples
c_1	0	1	2
c_2	0	1	2
c_3	0	2	4
c_4	1	3	4

Table 4.1: Input clauses for learning daughter relation

Among the generated caluses, we can observe that each of c_1, c_2, c_3 does not include free varibales but c_4 does, therefore first three clauses are preferred over the fourth clause. Among the first three clauses, we observe that c_3 has the maximum number of background literals, therefore it is selected as the best clause for this stage and add to the hypotheses set. Since clause c_3 covers all of the examples, the remaining uncovered set is empty, and outer loop algorithm stops. Therefore the output hypoteses is:
$H = \{c_3\} = \{daughter(X_0, X_1) :- parent(X_1, X_0), female(X_0).\}$
which is the correct description of the concept.

There may be cases where some examples are covered by none of the generalized clauses. In such cases, the examples which are not covered by any of the clauses are lastly added to the hypotheses set to make the hypotheses complete with respect to given example set.

Chapter 5

Implementation

Our system InGen bascially consists of six modules. These are:

1. Parser
2. Matcher
3. Specializer
4. Interval Generator
5. Generalizer
6. Clause Selector

Figure 5 illustrates the general architecture of our system.

5.1 Parser

Parser processes the input and background knowledge files and parses the literals. Both the example set and background knowledge must consist of function-free ground literals. Since our system is designed as an empirical single predicate

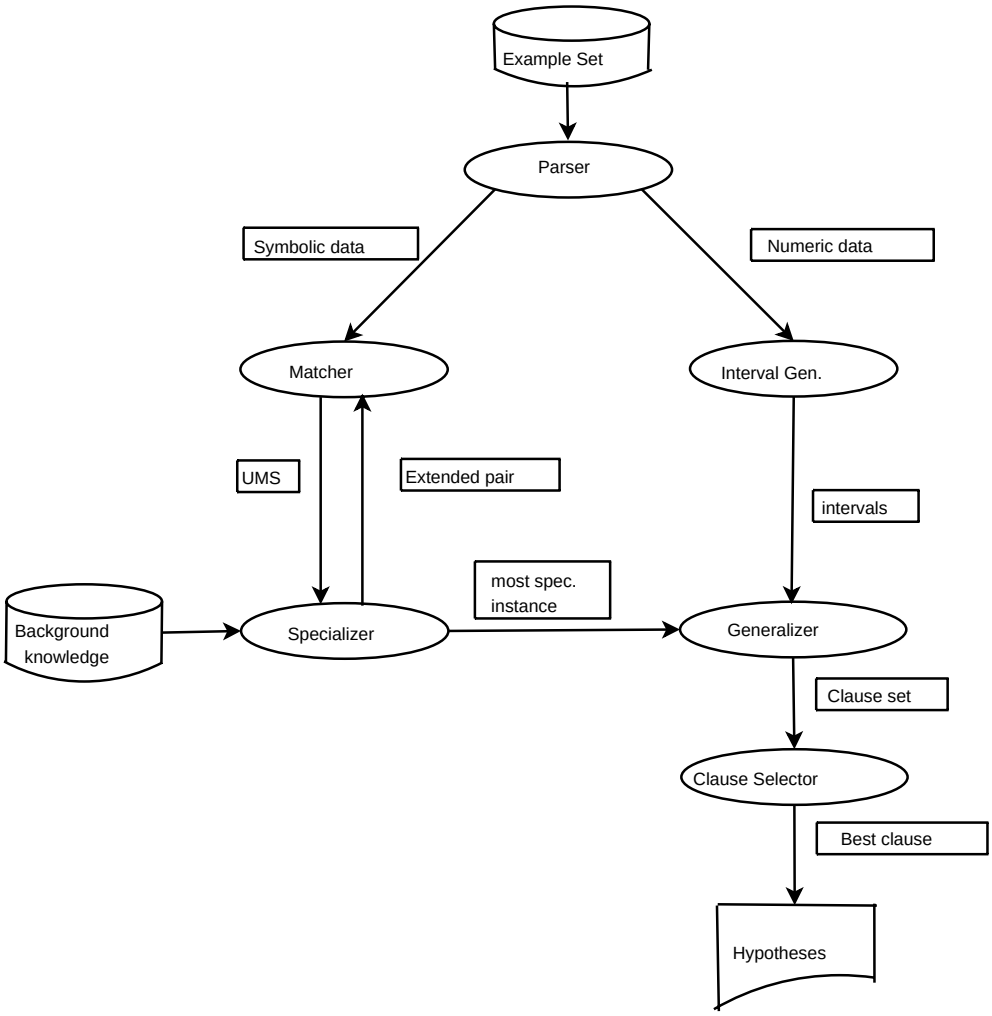

Figure 5.1: Architecture of InGen

learner, all the input examples must have same predicate name and same number of arguments. Firstly, it is investigated whether there is an example or background clause that violates the restrictions, and a message describing the error is asserted in such situations. If there is no problem in input format, the module searches whether there is any numeric input argument among the examples. Argument list is passed to Matcher for computing unique match sequence. Numeric arguments are also passed to Interval Generator.

Consider the case of learning from following example set:

```
p([a],[b],[a,f],2.1).
p([c],[b],[c,g],2.7).
p([d],[b],[h],6.9).
p([e],[b],[k],8.1).
```

associated with the background knowledge:

```
q([a]).  r([c])
q([c]).  r([d])
q([d]).  r([e])
```

All the example and background clauses are ground, have the same predicate name p and there are no functional terms, therefore the input is suitable for learning process. First two arguments of each clause are symbolic and the last is numeric, therefore the literals are passed in two parts to the Matcher. The input passed to Matcher is:

```
[[a,:,b,:,a,f],[2.1]]
[[c,:,b,:,c,g],[2.7]]
[[d,:,b,:,h],[6.9]]
[[e,:,b,:,k],[8.1]]
```

5.2 Matcher

The function of Matcher is to compute the unique match sequences of the symbolic arguments and associate it with the numeric arguments. The argument list contains sublists of symbolic and numeric arguments. For each pair, if there is

a unique match sequence for the symbolic arguments, Matcher computes it and appends the numeric arguments in the end of the match sequence as differences.

Firstly, two restrictions of having a unique match sequence are tested. These restrictions require that common symbols (strings) must occur same number of times and occur in the same order in both arrays. To test this situation, firstly common symbols in both lists are extracted as a set. Using this list, the sequence of common symbols are extracted retaining duplicates and the order for each list. If these sequences are exactly same, there is a unique match sequence, else , null sequence is returned indicating there is no UMS.

If the test result is positive, unique match sequence procedure starts to work. To find unique match sequence, similarites are extracted using the common symbols. Similarities are used as seperators and the symbols between similarites are constituents of differences.

Unique match sequence extraction procedure might be summarized by the algorithm in Figure 5.2. In the initial step, each common symbol is treated as a single similarity. This may be lead to empty differences and consequent similarities. In the end, empty differences are removed and consequent similarities are concatenated to obey the rule of unique match sequence.

The unique match sequence of the symbolic argument pairs of the given input are:
(a,c)b(af,cg), (a,d)b(af,h), (a,e)b(af,k), (c,d)b(cg,h), (c,e)b(cg,k), (d,e)b(h,k),
hence the output of Matcher passed to the Specializer is:

```
[([a],[c]),[:,b,:],([a,f],[c,g])],[(2.1,2.7)]]
[([a],[d]),[:,b,:],([a,f],[h])],[(2.1,6.9)]]
[([a],[e]),[:,b,:],([a,f],[k])],[(2.1,8.1)]]
[([c],[d]),[:,b,:],([c,g],[h])],[(2.7,6.9)]]
[([c],[e]),[:,b,:],([c,g],[k])],[(2.7,8.1)]]
[([d],[e]),[:,b,:],([h],[k])],[(6.9,8.1)]]
```

$commonSymbols \leftarrow common(list_1, list_2)$
$commonSymbols_1 \leftarrow projection(commonSymbols, list_1)$
$commonSymbols_2 \leftarrow projection(commonSymbols, list_2)$
if $commonSymbols1 \neq commonSymbols2$ **then**
  No UMS, return
end if
$commonIndexes_1 \leftarrow indexes(list_1, commonSymbols)$
$commonIndexes_2 \leftarrow indexes(list_2, commonSymbols)$
$commonIndexes_1[0] \leftarrow 0$
$commonIndexes_2[0] \leftarrow 0$
$i \leftarrow 1$
while $i \leq commonIndexes.length - 1$ **do**
  $sublist_1 \leftarrow sublist(list_1, commonIndexes_1[i-1], commonIndexes_1[i])$
  $sublist_2 \leftarrow sublist(list_2, commonIndexes_2[i-1], commonIndexes_2[i])$
  $ums[i-1] \leftarrow New\ difference(sublist_1, sublist_2)$
  $ums[i] \leftarrow commonSymbols1(i-1)$
  $i \leftarrow i + 1$
end while
$sublist_1 = sublist(list_1, commonIndexes_1[i-1], commonIndexes_1[i])$
$sublist_2 = sublist(list_2, commonIndexes_2[i-1], commonIndexes_2[i])$
$ums[i+1] = difference(sublist_1, sublist_2)$
Remove Empty Differences in the ums
unite Neighbouring Similarities in ums
for all Numeric argument pair (n_1, n_2) **do**
  if $n_1 = n_2$ **then**
    Append n_1 as a similarity.
  else
    Append (n_1, n_2) as a difference.
  end if
end for

Figure 5.2: Finding Unique Match Sequence

5.3 Specializer

Specializer module takes the unique match sequence extended with the numerical arguments and outputs the most specific instance of the sequence.

Initially, the unique match sequence is processed and the useful seperation differences for this match sequence is computed. Among the useful seperation differences, if there is the most useful one, the sequence is seperated with that difference. The same process is repeated for the resulting sequence iteratively until no MUSD is found. For the example case, such a seperation difference only exists for the first match sequence and it is: *(a,c)*. The seperation of the first sequence with difference *(a,c)* results the sequence *(a,c)b(a,c)(f,g)*.

Next, specializer performs a breadth-first search to find a background predicate that covers the both constituents of a difference. If there is such a predicate, the specific instance is splitted to the examples that it were built and literals are appended to the examples with a special seperator symbol. Then, these sequences are passed to Matcher to recompute the unique match sequence. If there is not such a background predicate or Matcher returns the null sequence indicating that there is not a UMS of the extended sequences, Specializer produces output and passes it to Generalizer.

For the first instance, we observe that the difference *(a,c)* is covered by background predicate *q*. Therefore the arguments are splitted and extended as follows:
`[[a,:,b,:,a,f],[2.1],[:-,q,a]`
`[[c,:,b,:,c,g],[2.7],[:-,q,c]]`
These examples are returned to Matcher to recompute the unique match sequences and then turn back to the specializer as follows:
`[([a],[c]),[:,b],([a],[c]),([f],[g]),(2.1,2.7),[:-,q],([a],[c])]`
It can be observed there is no further specialization for these examples, therefore it is the final output of the Specializer for the first instance.

There is no most useful seperation difference for other instances, but background specialization takes place and the instance set generated by the Specializer

is:

```
[([a],[d]),[:,b],([a,f],[h]),[:],(2.1,6.9),[:-,q],([a],[c])]
[([a],[e]),[:,b],([a,f],[k]),[:],(2.1,8.1),[:-,q],([a],[e])]
[([c],[d]),[:,b],([c,g],[h]),[:],(2.7,6.9),[:-,q],([c],[d])]
[([c],[e]),[:,b],([c,g],[k]),[:],(2.7,8.1),[:-,r],([c],[e])]
[([d],[e]),[:,b],([h],[k]),[:],(6.9,8.1)],[:-,r],([d],[e])
```

5.4 Interval Generator

Real number intervals are generated using the methodology described in Section 4.4. During generation process, we represent intervals as lists, where first element corresponds to left endpoint and last element correspond to right endpoint. Cluster elements are ordered between two endpoints and are removed from the set at the end of the process.

For the example we follow, Interval Generator takes the set $\{2.1, 2.7, 6.9, 8.1\}$ as input. The average distance is $(8.1 - 2.1)/3 = 2.0$, indicating that for each element of a cluster, there must be another element from the same cluster where the distance between these two elements is less than or equal to 2.0. The output of Interval Generator for this input consists of two closed intervals:
$I_1 = [0.1, 4.7]$
$I_2 = [4.9, 10.1]$

5.5 Generalizer

Generalizer computes generalized clauses using the specific instances instantiated by the Specializer together with the real number intervals generated by the Interval Generator.

As discussed in previous two chapters, in generalization step, differences are replaced with variables. The important point was to give the same name to

same differences. To achieve this purpose, we index variables with integers and use the index of the variables in the variable name. When generalizing real number arguments, it is investigated whether there is an interval that covers both constituents of the difference and those intervals are appended as literals to the body.

Generalizer forms the set of generalized clauses for construction of the hypotheses. For the instances passed from the Generalizer, the following generalized clauses are built by the Generalizer:

```
p(X0,b,[X0, X1],N) :- q(X0), real(N), between(N,0.1,4.7)
p(X0,b,X1,N) :- q(X0), real(N).
p(X0,b,X1,N) :- r(X0), real(N), between(N,4.9,10.1).
```

5.6 Clause Selector

Clause Selector constructs the resulting hypotheses as disjunction of the clauses generated by Generalizer. Resulting disjunction is complete with respect to example set. The selection criteria are applied in sequence as discussed in Section 4.5.

For the example followed, none of the general clauses include free variables. But first and third clauses have three body literals while the second has two, therefore they are preferred over the second. First and third clauses both have the same number of arguments (3) and cover same number of examples (2) therefore has equal priority. Using breadth-first search, first clause is selected and put it into the hypotheses. Then, it is removed from the generalized set and the examples it covers are removed from the example set.

At this stage, there are two uncovered examples,

```
[[d,:,b],[6.9]]
[[e,:,b],[8.1]]
```

and two generalized clauses left:

```
p(X0,b,X1,N) :- q(X0), real(N).
p(X0,b,X1,N) :- r(X0), real(N), between(N,4.9,10.1).
```

We see that neither of the clauses contain free variables, first clause contains two body literals while the second contains three and each of clauses cover the remaining two examples. Since InGen prefers the clauses having grater number of body literals over the others, Clause Selector selects the second clause and puts it into the hypotheses. At this step, hypotheses is:

```
p(X0,b,[X0, X1],N) :- q(X0), real(N), between(N,0.1,4.7).
p(X0,b,X1,N) :- r(X0), real(N), between(N,4.9,10.1).
```

The hypotheses covers all the examples given in the set, therefore selection procedure stops and the system gives this hypotheses as output.

Chapter 6

Experimentation

We evaluated the performance of our system with nominal datasets belonging to family relations, grammar learning, the card game Pisti and the continuous Mutagenesis dataset and compared the generated results with two concurrent ILP learners, Progol and FOIL. The results show that InGen is competitive with these state-of-art systems.

6.1 Experiments with Symbolic Arguments

6.1.1 Family relations

6.1.1.1 daughter relation

Consider a student studying for a Calculus final. For him, main factors that can help to predict his success may be, number of the example problems he solves and their closeness to final questions, the understanding of the concept explained in lectures and finally (arguably) Mathematical intelligence of the student. With an analogy, the success of an inductive learner on the broadness of the example set, the quality of background knowledge and efficiency of the learner in the problem domain.

Our first experiment is learning daughter relation which describes the concept of a person being daughter of another person. The example set and background knowledge are as in Table 2.1.

In order to start learning, Progol system needs mode and type declarations. That is, in addition to presentation of inputs in Prolog notation, Progol requires:

```
modeh(1,daughter(+person,+person))?
modeb(*,parent(-person,+person))?
modeb(*,parent(+person,-person))?
modeb(*,female(+person))?
```

as mode declatration and

```
person(nese).
person(ali).
person(senay).
...
```

as type declaration. Hovewer, even with this additional information, the most specific clause produced by Progol for this input set is:

```
daughter(A,B) :- parent(B,A), parent(C,A), female(A).
```

and the output hypotheses produced is a single clause:

```
daughter(A,B) :- parent(B,A).
```

discarding the condition that A must be female, and is inconsistent. For the same input examples presented in a similar way, FOIL gives the same result as output.

For the same example, InGen does not need any type or mode declaration. As mentioned in the previoues chapter, the output hypotheses generated by InGen for the daughter example set is:

```
daughter(X0, X1) :- parent(X1, X0), female(X0)
```

which is the correct description of the concept and shows that InGen is able to find the correct concept description using only examples and background clauses.

Example set (All positive)	Background Clauses
granddaughter(nese, ali).	parent(mehmet, senay). male(mehmet).
granddaughter(nese, gul).	female(zehra). parent(gul, aylin).
granddaughter(senay, ahmet).	parent(fatma, senay). parent(ahmet, mehmet).
granddaughter(senay, zehra).	parent(halil, nese). parent(zehra, fatma).
	father(halil, nese). female(nese).
	parent(ali, halil). female(senay).

Table 6.1: Input clauses for learning granddaughter relation

6.1.1.2 granddaughter relation

The example set for this relation consists of four clauses that informs a person is grandaughter of another person as in Table 6.1. The examples are instances of nonrecursive relation, that is, the cases like if X is granddaughter of Y and Y is granddaguhter Z, then X is granddaughter of Z are not present. The background knowledge includes 15 instances of relevant and irrelevant clauses.

Progol needs the following mode declaration for granddaughter predicate.

```
:- modeh(1,granddaughter(+person,+person))?
```

and extracts the following most specific clause

```
granddaughter(A,B) :- parent(B,C), parent(C,A).
```

and gives it as the resulting hypotheses. *female(A)* is discarded again, leading to inconsistency. FOIL was unable to answer for this simple input in more than 30 minutes on a work station with 400Mhz SUN-SPARC processor and we gave up waiting for the output.

InGen computes the following generalizations in initial step:

```
granddaughter(nese, X0) :- parent(X0, X1)
granddaughter(senay, X0) :- parent(X0, X1)
granddaughter(X0, X1) :- female(X0), parent(X2, X0), parent(X1, X3)
granddaughter(X0, X1) :- female(X0), parent(X2, X0), parent(X1, X2)
```

First clause is generated from the generalization of first two examples and only covers those two. The second clause is generated from the last two examples and covers only those two. Each of last two generalizations cover the whole set. The hypotheses produced by the learner is:

Example set (All positive)	Background Clauses
aunt(jane,henry).	father(sam,henry). sister(jane,sam).
aunt(sally,jim).	parent(sam,henry). sister(sally,sarah).
aunt(judy,jim).	mother(sarah,jim). sister(judy,sarah).
	parent(sarah,jim)

Table 6.2: Input clauses for learning aunt relation

`granddaughter(X0, X1) - female(X0), parent(X2, X0), parent(X1, X2).`
which is complete and correctly describes the concept. The last clause is preferred over the third, because all of its variables are bound, where the third clause contains the unbound variable X_3.

6.1.1.3 aunt relation

The aunt relation is generated and used by Muggleton to test his system Progol. The example set consists of three positive examples of the concept a person being aunt of another one, and a background knowledge set that includes father, mother, sister and parent relations that are related with the example set. This example is more complex and has features that can mislead an ILP system because a new variable (standing for the parent) must be introduced to correctly describe the concept.

The example and background knowledge set are as in Table 6.2. For this input, Progol can learn the relation:
`aunt(A,B) :- parent(C,B), sister(A,C).`
successfully. Furthermore Progol is capable of learning the same fact using the non-ground background clauses:
`parent(Parent,Child) :- father(Parent,Child).`
`parent(Parent,Child) :- mother(Parent,Child).`
instead of:
`parent(henry,sam).`
`parent(jim,sarah).`
which Ingen is incapable of. FOIL was unable to answer for this input.

Example set (All positive)	Background Clauses
s([a man, sleeps])	np(a man). iverb(sleeps).
s([the boy, sleeps]).	np(the man). tverb(walks).
s([the dog,walks]).	np(a dog). tverb(hits)
s([a boy,walks]).	np(the dog). tverb(takes)
s([a man,walks,a dog]).	np(a boy)
s([the boy,walks,the cat]).	np(the boy).
s([the man,hits,the ball]).	np(the ball).
s([a boy,hits,a dog]).	np(the house).
s([the man,hits,the ball,at,the house]).	np(a picnic).
s([a boy,hits,a dog,at,a picnic]).	np(a room).
s([the man,takes,the ball,to,the house]).	np(the cat)
s([a boy,takes,a dog,to,a room]).	iverb(walks).

Table 6.3: Input clauses for grammar learning

InGen produces the following clauses:

```
aunt(X0, X1) :- sister(X0, X2), parent(X2, X1).
aunt(X0, jim) :- sister(X0, sarah).
```

The first clause takes its root from the generalization of first and second examples and the second clause takes its root from the generalization of last two. The output is:

```
aunt(X0, X1) :- sister(X0, X2), parent(X2, X1).
```

which is complete and correctly describes the concept. The first clause is preferred over the second since it covers all three examples while the second clause covers the last two.

6.1.2 Grammar Learning

A popular application area for ILP is grammar learning, and it is defined as a seperate research field as Learning Language in Logic in [7]. In our example, there are twelve sentences, and each sentence is considered to be argument of the sentence predicate as a list. Each phrase is considered as a constant symbol. Background knowledge consists of the predicates denoting the phrase types.

The example and background knowledge set are as in Figure 6.3. Among

many clauses generated, the learner hypotheses includes the following generalizations:

```
s([X0, sleeps]) :- np(X0).
s([X0, walks]) :- np(X0).
s([X0, walks, X1]) :- np(X0), np(X1).
s([X0, hits, X1]) :- np(X0), np(X1).
s([X0, hits, X1, at, X2]) :- np(X0), np(X1), np(X2).
s([X0, takes, X1, to, X2]) :- np(X0), np(X1), np(X2).
```

Considering that the variables with same type are same tokens, InGen produces second level generalization and the result of this generalization is:

```
s([X0, Y0]) :- np(X0), iverb(Y0).
s([X0, Y0, X1]) :- np(X0), np(X1), tverb(Y0).
s([X0, Y0, X1, Y1, X2]) :- np(X0), np(X1), np(X2), tverb(Y0), prep(Y1).
```

which defines the structures of three regular English sentences. Note that the first clause in the second pass correspond to the generalization of first two clauses in the first pass, the second to third and fourth, the third to fifth and sixth in the same manner.

The resulting hypotheses of Progol is same as the one produced by InGen for this example.

6.2 Learning Pisti Game

Pisti is a card game played by two players, where each player holds one to four cards at a time. The players put out their cards in turn. A player collects the open cards if he puts out the same card that his opponent put in previous turn, or he throws a *jack*. Some examples of collection conditions are:
$previous\ card = (clubs, six),\ current\ card = (spades, six).$
$previous\ card = (spades, queen),\ current\ card = (hearts, queen).$
$previous\ card = (diamonds, seven),\ current\ card = (clubs, jack).$
In this notation, we conclude that collection event occurs when curent card is a *jack* or face of current card is same as the previous one. To represent this concept,

we can consider the collection condition as a four-argument predicate, where first two argument represent the suit and face of the previous card and the last two represent those of the current card.

In this context, our example set is:

```
collect(clubs,queen,spades,queen).
collect(hearts,ace,diamonds,ace).
collect(spades,six,hearts,jack).
collect(clubs,eight,diamonds,jack).
collect(clubs,seven,spades,jack).
```

and we do not need any background knowledge.

For Progol, mode declaration can be one of two cases:

```
modeh(1,collect(+suit,+face,+suit,#face))?
modeh(1,collect(+suit,+face,+suit,+face))?
```

In first case, the hypotheses induced by Progol is:

```
collect(clubs,queen,spades,queen).
collect(A,B,C,ace).
collect(A,B,C,jack).
```

The hypotheses, produced by the second case is:

```
collect(spades,six,hearts,jack).
collect(clubs,eight,diamonds,jack).
collect(clubs,seven,spades,jack).
collect(A,B,C,B).
```

So, Progol is not able to describe the concept correctly in either case. Note the impossibility for Progol to construct correct hypotheses for this concept, since the last argument can be a constant or a variable. Although FOIL constructed the clause `collect(A,B,C,D) :- B=D`, its controller did not accept this clause because it is regarded as too inaccurate and a null hypotheses is returned indicating no generalization can be made.

Hovewer, InGen learns the concept as follows:

```
collect(X0, X1, X2, jack).
collect(X0, X1, X2, X1).
```

which is complete and consistent with the game rules.

6.3 Experiments with Numeric Arguments

6.3.1 Mutagenesis

Predicting mutagenecity of an organic compound is important, since it is related with the prediction of carcinogenesis. Mutagenesis dataset is originated by Oxford University, and it contains features of 230 compounds [9]. 138 of these examples are labeled as positive, meaning that they have mutagenecity and 92 of them are labeled as negative, meaning no mutagenecity. The dataset is divided into two categories. 188 examples are regression-friendly compounds and 42 are regression-unfriendly. Predicting the label of regression-unfriendly compounds are harder than predicting the regression-unfriendly ones. The regression-friendly example set is called 188-dataset and consists of 125 positive and 63 negative examples.

Each compound is associated with four attributes in this dataset. These are:

- A numeric attribute correspoding to hydorphobicity of the compound termed as `logP`.
- A numeric attribute correspoding to energy level of the lowest unoccupied molecular orbital noted as `lumo`.
- A boolean attribute indicating whether the compound contains three or more benzyl rings.
- A boolean attribute indicating whether the compound is element of a subclass acenthryles.

The example set is distributed into ten files as ground caluses, where clauses starting with ":-" represent negative examples, and remaining ones represent positive examples.

Test Group	Success rate with benzyl, acenthryles and hiydrophobicity features	Success rate with benzyl, acenthryles and energy level features
1	63	47
2	79	74
3	74	74
4	68	58
5	84	52
6	68	42
7	84	68
8	75	79
9	74	53
10	77	53
Overall	76	60

Table 6.4: Experiment results of InGen for Mutagenecity dataset

```
active(d8).
active(d16).
:- active(d1).
:- active(d22).
```

The attributes are presented as background literals such as:

```
logp(d8, 3.46).
logp(d16, 4.44).
logp(d21, 3.52).
logp(d22, 5.09).
```

For experimentation, we combine these files to a single file such as:

```
active(3.46).
active(4.44).
:- active(3.52).
:- active(5.09).
```

and use it as the input to InGen test.

Different ILP learners such as Progol are tested on this dataset [9]. We also tested our system with 188-dataset using the first two numeric attributes with ten-fold cross validation and the results are as in Table 6.4.

Success rate of Progol for mutagenecity dataset is %89 using all four features. InGen classsifies the examples with %76 correctness using three features and we that it needs to be enhanced with predicate invention to produce better results.

Chapter 7

Conclusion

Inductive Logic Programming is a relatively new research area that induces logic programs from given example set and background knowledge. String generalization is an application area of ILP. The Specific Generalization technique described in [4] generates generalized strings that cover a given example set. The heuristic is for the generalization of strings only, and does not consider the generic background clauses.

The research outlined in this thesis is an initial attempt to build an ILP system based on the specific generalization of strings proposed in [4]. Firstly, generalization with using arbitrary first-order background knowledge is integrated. This is performed by considering the background clauses as sequences of strings, performing the generalization in a loop, and parsing the generalization to generate clauses. Secondly, the system is extended to generalize predicates with numeric arguments. To perform numeric generalization, a modified version of hierarchical clustering is used.

The experiments we performed demonstrate that the system has potential usage. It could successfully learn the family relations such as daughter, aunt and granddaughter, a card game, and possible grammatical structure of English sentences. In the numeric case, we tested the system using the famous Mutagenesis dataset and observed maximum success rate is %76. This shows that our system

needs to be enhanced to invent new predicates for more efficient learning.

Although our system is capable of learning these concepts successfully, it is not a perfect ILP system. There are several potential directions to study to make our system compatible with concurrent ILP systems:

- We generalized numeric arguments only the less than and distance background knowledge. Several other mathematical relations may be used such as power, root, exponent, etc.
- The system is capable of learning function-free ground clauses only. The concept decription language may be enlarged to cover functional and non-ground clauses.
- Our system is not capable of inventing new predicates, which may be necessary for learning some concepts such as sorting.
- The system should be tested with dataset having several thousands of example to see if it can work in real world cases.

As a conclusion, this work is an initial attempt for building an ILP system using specific generalization based on LGG schema and it can be serve as a basis for future research to construct an effective learner using the same notion.

Bibliography

[1] I. Bratko. Applications of machine learning: Towards knowledge synthesis. In *Proc. International Conference on Fifth Generation Computer Systems*, pages 1207–1218, Tokyo, 1992.

[2] R. B. C. Sammut. Learning concepts by asking questions. In T. M. R. Michalski, J. Carnobell, editor, *Machine Learning: An Artificial Intelligence Approach*, pages 167–192. Morgan Kauffman, 1986.

[3] J. Carnobell. Introduction: Paradigms for machine learning. *Artificial Intelligence*, 40(1-3):1–9, 1989.

[4] I. Cicekli and N. K. Cicekli. Generalizing predicates with string arguments. *Applied Intelligence*, 2005.

[5] L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Thirten International Joint Conference on Artifical Intelligence*. Morgan Kauffman, 1993.

[6] B. Dolsak. Constructing finite element meshes using artificial intelligence methods. Master's thesis, University of Maribor, 1991.

[7] S. Džeroski, J. Cussens, and S. Manandhar. An introduction to inductive logic programming and learning language in logic. In J. Cussens and S. Džeroski, editors, *Learning Language in Logic*, volume 1925, pages 3–35. Springer-Verlag, 2000.

[8] S. Dzeroski. Handling noise in inductive logic programming. Master's thesis, University of Ljubljana, 1991.

[9] A. S. et. al. Mutagenesis: Ilp experiments in a non-determinate biological domain. In *Proceedings of the Inductive Logic Programming Workshop*, 1994.

[10] D. G. F. Bergadano. *Inductive Logic Programming, From Machine Learning to Software Engineering*. MIT Press, 1995.

[11] C. Feng. Inducing temporal fault diagnostic rules from a qualitative model. In *Proc. of the Eighth International Workshop on Machine Learning*, pages 403–406, Evanston, IL, 1991.

[12] K. Godel. Uber formal unentscheidbare. *Satse der Principia und verwandter System*, 1(32):173–198, 1931.

[13] M. Grobelnik. An optimized model inference system. In *Proceedings of Workshop on Logical Approaches to Machine Learning, Tenth European Conference on Artificial Intelligence*, 1992.

[14] J. L. Hopson. Fetal psychology. *Psychology Today*, 31(5):44, 1998.

[15] S. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32:241–254, 1967.

[16] N. Lavrac and S. Dzeroski. *Inductive Logic Programming*. Ellis Horwood, 1994.

[17] J. Lloyd, editor. *Computational Logic*. Springer, 1990.

[18] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1984.

[19] R. Michalski. A theory and methodology of inductive learning. *Artificial Intelligence*, 20:11–116, 1983.

[20] D. Michie. Machine learning in the next five years. In *Proc. Third European Working Session on Learning*, pages 107–122, 1988.

[21] T. M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[22] T. M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997.

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

[24] K. Morik. Balanced cooperative modelling. In *Proc. First International Workshop on Multistrategy Learning*, pages 65–80, Fairfax, VA, 1991. George Mason University.

[25] S. Muggleton. http://wwwhomes.doc.ic.ac.uk/shm/software/progol5.0/.

[26] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[27] S. Muggleton. Inverse entailment and progol. *New Generation Comput.*, 13(3):245–286, 1995.

[28] S. Muggleton. Learning from positive data. *Machine Learning*, 2001.

[29] S. Muggleton, R. King, and M. Sternberg. Protein secondary structure prediction using logic. In *Proc. Second International Workshop on Inductive Logic Programming*, Tokyo, Japan, 1992.

[30] S. Muggleton, A. Srinivasan, and M. Bain. Compression, significance and accuracy. In D. Sleeman and P. Edwards, editors, *Proceedings of the 9th International Workshop on Machine Learning*, pages 338–347. Morgan Kaufmann, 1992.

[31] M. G. N. Lavrac, S. Dzeroski. Learning nonrecursive definitions of relations with linus. In *Proceedings of Fifth European Working Session on Learning*, 1991.

[32] M. Pazzani and D. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9(1):57–94, 1992.

[33] G. D. Plotkin. *Automatic methods of inductive inference*. PhD thesis, Edinburgh University, 1971.

[34] J. R. Quinlan. Discovering rules from large collections of examples: a case study. In D. Michie, editor, *Expert Systems in the Micro-electronic Age*, pages 168–201. Edinburgh University Press, 1979.

[35] J. R. Quinlan. Learning relations: comparison of a symbolic and a connectionist approach. Technical Report 346, University of Sydney, 1989.

[36] G. T. R. Michalski, editor. *Proc. First International Workshop on Multistrategy Learning*, 1991.

[37] L. D. Raedt and M. Bruynooghe. Towards friendly concept-learners. In *Proc. of the 11th IJCAI*, pages 849–854, 1989.

[38] L. D. Raedt, N. Lavrač, and S. Džeroski. Multiple predicate learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1993.

[39] B. L. Richards and R. J. Mooney. Learning relations by pathfinding. In *National Conference on Artificial Intelligence*, pages 50–55, 1992.

[40] J. A. Robinson. A machine-oriented logic based on the resolution principle. *jacm*, 12(1):23–41, January 1965.

[41] C. Rouveirol. Extension of inversion of resolution to theory completion. In S. Muggleton, editor, *Inductive Logic Programming*. Academic Press, London, 1992.

[42] C. .Rouveirol and J. F. Puget. Beyond inversion of resolution. In B. Porter and R. Mooney, editors, *Proc. of the Int. conf. on Machine Learning*, pages 122–131. Morgan Kauffman, 1990.

[43] C. F. S. Muggleton. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, 1990.

[44] W. B. S. Muggleton. Machine invention of first order predicates by invertig resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352, 1988.

[45] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[46] G. Tecuci and Y. Kodratoff. Apprenticeship learning in non-homogenuous domain theories. In Y. Kodratoff and R. S. Michalski, editors, *Machine Learning: An Artificial Intelligence Approach*, volume III, pages 514–551, San Mateo, CA, 1990. Morgan Kaufmann.

[47] A. Turing. Systems of logic based on ordinals. In *Proc. of the London Mathematical Society*, pages 161–228, 1939.

[48] A. Turing. The automatic computing engine. *Lecture to the London Mathematical Society*, 1947.

[49] J. M. U. Nilsson. *Logic, Programming and Prolog.* John Wiley and Sons Ltd., 2000.

[50] C. M. W. Frawley, G. Piatetsky-Shapiro. Knowledge discovery in databases: an overview. In G. P.-S. W. Frawley, editor, *Knowledge discovery in databases*, pages 1–27. AAAI Press/MIT Press, 1991.

Appendix A

Test Input and Output Files

A.1 Daughter example

A.1.1 Progol

Input:

```
% Settings
:-set(posonly)?
% Mode declarations
:- modeh(1,daughter(+person,+person))?
:- modeb(*,parent(-person,+person))?
:- modeb(*,parent(+person,-person))?
:- modeb(*,female(+person))?
%Types
person(nese).
person(ali).
person(senay).
person(ahmet).
person(zehra).
person(halil).
```

```
person(mehmet).
person(fatma).
person(ayse).
person(gul).
%Background knowledge
parent(mehmet,senay).
parent(fatma,senay).
parent(halil,nese).
parent(aylin,nese).
female(senay).
female(nese).
%Examples
daughter(senay,mehmet).
daughter(senay,fatma).
daughter(nese,halil).
daughter(nese,aylin).
```

Output:

```
Noise has been set to 100%
Example inflation has been set to 400%
The posonly flag has been turned ON
:- set(posonly)?  - Time taken 0.00s
:- modeh(1,daughter(+person,+person))?  - Time taken 0.00s
:- modeb(100,parent(-person,+person))?  - Time taken 0.00s
:- modeb(100,parent(+person,-person))?  - Time taken 0.00s
:- modeb(100,female(+person))?  - Time taken 0.00s
Testing for contradictions
No contradictions found
Generalising daughter(senay,mehmet).
daughter(A,B) :- parent(B,A), parent(C,A), female(A).
Most-specific clause reduced by 1 literals
Most specific clause is
daughter(A,B) :- parent(B,A), female(A).
```

```
Learning daughter/2 from positive examples
C:-0,16,15,0 daughter(A,B).
C:13,16,1,0 daughter(A,B) :- parent(B,A).
C:8,16,6,0 daughter(A,B) :- parent(C,A).
C:12,16,1,0 daughter(A,B) :- parent(B,A), female(A).
C:7,16,6,0 daughter(A,B) :- parent(C,A), female(A).
C:8,16,6,0 daughter(A,B) :- female(A).
6 explored search nodes
f=13,p=16,n=1,h=0
Result of search is
daughter(A,B) :- parent(B,A).
4 redundant clauses retracted
daughter(A,B) :- parent(B,A).
Total number of clauses = 1
Time taken 0.01s
```

A.1.2 FOIL

```
Input:
X: nese, senay.
Y: mehmet, fatma, halil, aylin.

daughter(X,Y)
senay,mehmet
senay,fatma
nese,halil
nese,aylin
.
female(X)
nese
senay
```

```
.
parent(Y,X)
mehmet,senay
fatma,senay
halil,nese
aylin,nese
.
```

Output:

```
FOIL 6.4 [January 1996]
--------
Options:
no negated literals
verbosity level 3
Types X and Y are not compatible
Relation daughter
Relation *female
Relation *parent
Ordering constants of type X
Checking arguments of daughter
Checking arguments of parent
unordered

Ordering constants of type Y
Checking arguments of daughter
Checking arguments of parent
unordered
----------
daughter:
State (4/8,
10.0 bits available)
female(A) 4[4/8] gain 0.0
female tried 1/1] 0.0 secs
```

```
parent(B,C) 4[4/8] A=+C XDet #
parent(B,A) 4[4/4] gain 3.4
parent tried 2/3] 0.0 secs
Save clause ending with parent(B,A) (cover 4, accuracy 100%)
Best literal parent(B,A) (2.6 bits)
Initial clause (0 errs):  daughter(A,B) :- parent(B,A).
parent(B,A) essential
Clause 0:  daughter(A,B) :- parent(B,A).
Clause 0 needed for senay,mehmet
daughter(A,B) :- parent(B,A).
```

A.1.3 InGen

Input is:

```
// Examples
daughter(senay,mehmet).
daughter(senay,fatma).
daughter(nese,halil).
daughter(nese,aylin).
// Background literals
parent(mehmet,senay).
parent(fatma,senay).
parent(halil,nese).
parent(aylin,nese).
female(senay).
female(nese).
```

Output is:

```
Initially:
daughter([senay, :, mehmet])
daughter([senay, :, X0]) :- parent([X0, :, senay])
daughter([X0, :, X1]) :- parent([X1, :, X0]), female(X0)
daughter([X0, :, X1]) :- parent([X2,:,X0]), female(X0), parent([X1,:,X0])
```

```
daughter([senay, :, fatma])
daughter([nese, :, halil])
daughter([nese, :, X0]) :- parent([X0, :, nese])
daughter([nese, :, aylin])
------------------------------------------------------------
Result of first pass:
daughter([X0, :, X1]) :- parent([X1, :, X0]), female(X0)
------------------------------------------------------------
Result of second pass:
daughter([X0, :, X1]) :- parent([X1, :, X0]), female(X0)
------------------------------------------------------------
Resulting Hypotheses:
daughter(X0, X1) :- parent(X1, X0), female(X0).
```

A.2 Granddaughter example

A.2.1 Progol

Input:

```
% Settings :-set(posonly)?
% Mode declarations
:- modeh(1,granddaughter(+person,+person))?
:- modeb(*,parent(-person,+person))?
:- modeb(*,parent(+person,-person))?
:- modeb(*,female(-person))?
%Types
person(nese).
person(ali).
person(senay).
person(ahmet).
person(zehra).
```

```
person(halil).
person(mehmet).
person(fatma).
person(ayse).
person(gul).
%Background knowledge
parent(mehmet,senay).
parent(fatma,senay).
parent(halil,nese).
parent(ayse,nese).
parent(ali,halil).
parent(gul,ayse).
parent(ahmet,mehmet).
parent(zehra,fatma).
female(nese).
female(senay).
%Examples
granddaughter(nese,ali).
granddaughter(nese,gul).
granddaughter(senay,ahmet).
granddaughter(senay,zehra).
```

Output:

```
Noise has been set to 100%
Example inflation has been set to 400%
The posonly flag has been turned ON
:- set(posonly)?  - Time taken 0.00s
:- modeh(1,granddaughter(+person,+person))?  - Time taken 0.00s
:- modeb(100,parent(-person,+person))?  - Time taken 0.00s
:- modeb(100,parent(+person,-person))?  - Time taken 0.00s
:- modeb(100,female(-person))?  - Time taken 0.00s
Testing for contradictions
```

```
No contradictions found
Generalising granddaughter(nese,gul).
granddaughter(A,B) :- parent(B,C), parent(C,A), parent(D,A), parent(E,D).
Most-specific clause reduced by 2 literals
Most specific clause is
granddaughter(A,B) :- parent(B,C), parent(C,A).
Learning granddaughter/2 from positive examples
C:-0,12,11,0 granddaughter(A,B).
C:4,12,6,0 granddaughter(A,B) :- parent(B,C).
C:8,12,1,0 granddaughter(A,B) :- parent(B,C), parent(C,A).
C:6,12,3,0 granddaughter(A,B) :- parent(B,C), parent(D,A).
C:4,12,6,0 granddaughter(A,B) :- parent(C,A).
5 explored search nodes
f=8,p=12,n=1,h=0
Result of search is
granddaughter(A,B) :- parent(B,C), parent(C,A).
3 redundant clauses retracted
granddaughter(A,B) :- parent(B,C), parent(C,A).
Total number of clauses = 1
Time taken 0.01s
```

A.2.2 FOIL

Input:

```
X: nese, senay.
Y: ali, gul, ahmet, zehra.
Z1:  mehmet, fatma, halil, aylin, ali, gul, ahmet, zehra.
Z2:  senay, nese, halil, ayse, mehmet, fatma.

granddaughter(X,Y)
nese,ali
nese,gul
```

```
senay,ahmet
senay,zehra
.
female(X)
nese
senay
.
parent(Z1,Z2)
mehmet,senay
fatma,senay
halil,nese
aylin,nese
ali,halil
gul,ayse
ahmet,mehmet
zehra,fatma
.
```

FOIL failed to give output for this test set.

A.2.3 InGen

Input:

```
//Examples
granddaughter(nese,ali).
granddaughter(nese,gul).
granddaughter(senay,ahmet).
granddaughter(senay,zehra).
//Background clauses
parent(mehmet,senay).
parent(fatma,senay).
parent(halil,nese).
parent(ayse,nese).
```

```
parent(ali,halil).
parent(gul,ayse).
parent(ahmet,mehmet).
parent(zehra,fatma).
female(nese).
female(senay).
```

Output:

```
Initially:
granddaughter([nese, :, ali])
granddaughter([nese, :, X0]) :- parent([X0, :, X1])
granddaughter([X0,:,X1]) :- parent([X2,:,X0]),female(X0),parent([X1,:,X2])
granddaughter([X0,:,X1]) :- parent([X2,:,X0]),female(X0),parent([X1,:,X3])
granddaughter([nese, :, gul])
granddaughter([senay, :, ahmet])
granddaughter([senay, :, X0]) :- parent([X0, :, X1])
granddaughter([senay, :, zehra])
------------------------------------------------------------
Result of first pass:
granddaughter([X0,:,X1]) :- parent([X2,:,X0]),female(X0),parent([X1,:,X2])
------------------------------------------------------------
Result of second pass:
granddaughter([X0,:,X1]) :- parent([X2,:,X0]),female(X0),parent([X1,:,X2])
------------------------------------------------------------
Resulting Hypotheses:
granddaughter(X0,X1) :- parent(X2,X0),female(X0),parent(X1,X2).
------------------------------------------------------------
```

A.3 Aunt example

A.3.1 Progol

Input:

```
% Settings
:- set(posonly)?
% Mode declarations :- modeh(1,aunt(+person,+person))?
:- modeb(*,parent(-person,+person))?
:- modeb(*,parent(+person,-person))?
:- modeb(*,sister(+person,-person))?
% Types
person(jane).
person(henry).
person(sally).
person(jim).
person(sam).
person(sarah).
person(judy).
% Background knowledge
parent(Parent,Child) :- father(Parent,Child).
parent(Parent,Child) :- mother(Parent,Child).
father(sam,henry).
mother(sarah,jim).
sister(jane,sam).
sister(sally,sarah).
sister(judy,sarah).
% Examples
aunt(jane,henry).
aunt(sally,jim).
aunt(judy,jim).
```

Output:

```
Noise has been set to 100%
Example inflation has been set to 400%
The posonly flag has been turned ON
:- set(posonly)?  - Time taken 0.00s
:- modeh(1,aunt(+person,+person))?
:- modeb(100,parent(-person,+person))?
:- modeb(100,parent(+person,-person))?
:- modeb(100,sister(+person,-person))?
Testing for contradictions
No contradictions found
Generalising aunt(jane,henry).
Most specific clause is
aunt(A,B) :- parent(C,B), sister(A,C).
Learning aunt/2 from positive examples
C:-0,12,11,0 aunt(A,B).
C:5,12,5,0 aunt(A,B) :- parent(C,B).
C:8,12,1,0 aunt(A,B) :- parent(C,B), sister(A,C).
C:6,12,3,0 aunt(A,B) :- parent(C,B), sister(A,D).
C:5,12,5,0 aunt(A,B) :- sister(A,C).
5 explored search nodes
f=8,p=12,n=1,h=0
Result of search is
aunt(A,B) :- parent(C,B), sister(A,C).
3 redundant clauses retracted
aunt(A,B) :- parent(C,B), sister(A,C).
Total number of clauses = 1
Time taken 0.01s
```

A.3.2 FOIL

Input:

```
X: jane, sally, judy.
Y: henry, jim.
```

```
Z: sam, sarah.

aunt(X,Y)
jane,henry
sally,jim
judy,jim
.
female(X)
jane
sally
judy
.
parent(Z,Y)
sam,henry
sarah,jim
.
sister(X,Z)
jane,sam
sally,sarah
judy,sarah
.
```

FOIL failed to give output for this test set.

A.3.3 InGen

Input:

```
//Examples:
aunt(jane,henry).
aunt(sally,jim).
aunt(judy,jim).
//Background Literals:
father(sam,henry).
```

```
parent(sam,henry).
mother(sarah,jim).
parent(sarah,jim)
sister(jane,sam).
sister(sally,sarah).
sister(judy,sarah).
```

Output:

```
Initially:
aunt([jane, :, henry])
aunt([X0, :, X1]) :- sister([X0, :, X2]), parent([X2, :, X1])
aunt([sally, :, jim])
aunt([X0, :, jim]) :- sister([X0, :, sarah])
aunt([judy, :, jim])
------------------------------------------------------------
Result of first pass:
aunt([X0, :, X1]) :- sister([X0, :, X2]), parent([X2, :, X1])
------------------------------------------------------------
Result of second pass:
aunt([X0, :, X1]) :- sister([X0, :, X2]), parent([X2, :, X1])
------------------------------------------------------------
Resulting Hypotheses:
aunt(X0, X1) :- sister(X0, X2), parent(X2, X1).
------------------------------------------------------------
```

A.4 Grammar example

A.4.1 Progol

Input:

```
% Grammar learning problem.  Learns s/2, a simple English language phrase
grammar.
```

```
:- modeh(1,s(+wlist,-wlist1))?
:- modeb(1,det(+wlist,-wlist))?
:- modeb(1,prep(+wlist,-wlist))?
:- modeb(1,noun(+wlist,-wlist))?
:- modeb(1,tverb(+wlist,-wlist))?
:- modeb(1,iverb(+wlist,-wlist))?
:- modeb(*,np(+wlist,-wlist))?
:- modeb(*,vp(+wlist,-wlist))?
:- set(i,5)?
:- set(c,5)?
:- set(h,10000000)?
:- set(posonly)?
% Types wlist([]).
wlist([W|Ws]) :- word(W), wlist(Ws).
wlist1([]).
word(a).  word(at).  word(ball).  word(big).  word(dog).
word(hits).  word(house).  word(in).  word(man).
word(on).  word(small).  word(takes).  word(the).  word(to).  word(walks).
:- [wpair]?
% Background knowledge
np(S1,S2) :- det(S1,S3), noun(S3,S2).
np(S1,S2) :- det(S1,S3), adj(S3,S4), noun(S4,S2).
det([a|S],S).
det([the|S],S).
det([every|S],S).
vp(S1,S2) :- tverb(S1,S2).
vp(S1,S2) :- tverb(S1,S3), prep(S3,S2).
noun([man|S],S).
noun([dog|S],S).
noun([house|S],S).
noun([ball|S],S).
tverb([hits|S],S).
tverb([takes|S],S).
```

```
tverb([walks|S],S).
iverb([barks|S],S).
iverb([hits|S],S).
iverb([takes|S],S).
iverb([walks|S],S). prep([at|S],S).
prep([to|S],S).
prep([on|S],S).
prep([in|S],S).
prep([from|S],S).
% Positive examples
s([a man,sleeps],[]).
s([the boy,sleeps],[]).
s([the dog,walks],[]).
s([a boy,walks],[]).
s([a man,walks,a dog],[]).
s([the boy,walks,the cat],[]).
s([the man,hits,the ball],[]).
s([a boy,hits,a dog],[]).
s([the man,hits,the ball,at,the house],[]).
s([a boy,hits,a dog,at,a picnic],[]).
s([the man,takes,the ball,to,the house],[]).
s([a boy,takes,a dog,to,a room],[]).
```

Overall output for Progol is too long for this experiment. The summary of the output is:

```
:- modeh(1,s(+wlist,-wlist1))?  - Time taken 0.00s
:- modeb(1,det(+wlist,-wlist))?  - Time taken 0.00s
:- modeb(1,prep(+wlist,-wlist))?  - Time taken 0.00s
:- modeb(1,noun(+wlist,-wlist))?  - Time taken 0.00s
:- modeb(1,tverb(+wlist,-wlist))?  - Time taken 0.00s
:- modeb(1,iverb(+wlist,-wlist))?  - Time taken 0.00s
:- modeb(100,np(+wlist,-wlist))?  - Time taken 0.00s
```

```
:- modeb(100,vp(+wlist,-wlist))?  - Time taken 0.00s
:- set(i,5)?  - Time taken 0.00s
:- set(c,5)?  - Time taken 0.00s
:- set(h,10000000)?  - Time taken 0.00s
Noise has been set to 100%
Example inflation has been set to 400%
The posonly flag has been turned ON
:- set(posonly)?  - Time taken 0.00s
Cannot find wpair.pl
:- wpair?  - Time taken 0.00s
Testing for contradictions
No contradictions found
Generalising s(a,man,hits,the,ball,at,the,dog,).
Most specific clause is
s(A,B) :- det(A,C), np(A,D), noun(C,D), tverb(D,E), iverb(D,E),
vp(D,E), det(E,F), np(E,G), prep(G,H), noun(F,G), det(H,I), np(H,B).
...
...
8 redundant clauses retracted
s(A,B) :- np(A,C), tverb(C,D), np(D,E), prep(E,F), np(F,B).
s(A,B) :- det(A,C), np(A,D), vp(D,E), np(E,B).
s(A,B) :- np(A,C), iverb(C,B).
Total number of clauses = 3
Time taken 1.00s
```

A.4.2 InGen

Input:

```
//Examples:
s([a man,sleeps]).
s([the boy,sleeps]).
s([the dog,walks]).
```

```
s([a boy,walks]).
s([a man,walks,a dog]).
s([the boy,walks,the cat]).
s([the man,hits,the ball]).
s([a boy,hits,a dog]).
s([the man,hits,the ball,at,the house]).
s([a boy,hits,a dog,at,a picnic]).
s([the man,takes,the ball,to,the house]).
s([a boy,takes,a dog,to,a room]).
//Background Literals:
tverb(hits).
np(a man).
np(a cat).
tverb(walks).
np(the man).
np(the cat).
tverb(takes).
np(a dog).
np(a boy).
np(the dog).
np(the boy).
iverb(sleeps).
np(a house).
np(a room).
iverb(walks).
np(the house).
np(the room).
np(a ball).
np(a picnic).
prep(at).
np(the ball).
prep(to).
prep(in).
```

Output:

Initially:

```
s([a man, sleeps])
s([X0, sleeps]) :- np(X0)
s([a man, X0])
s([the boy, sleeps])
s([the boy, X0])
s([the dog, walks])
s([X0, walks]) :- np(X0)
s([X0, walks, X1]) :- np(X0)
s([a boy, walks])
s([a boy, X0])
s([a man, walks, a dog])
s([X0, walks, X1]) :- np(X0), np(X1)
s([X0, a dog])
s([X0, a dog, X1])
s([the boy, walks, the cat])
s([the man, hits, the ball])
s([X0, hits, X1]) :- np(X0), np(X1)
s([the man, hits, the ball, X0])
s([X0, hits, X1]) :- np(X0)
s([the man, X0, the ball, X1]) :- tverb(X0)
s([a boy, hits, a dog])
s([a boy, hits, a dog, X0])
s([a boy, X0, a dog, X1]) :- tverb(X0)
s([the man, hits, the ball, at, the house])
s([X0, hits, X1, at, X2]) :- np(X0), np(X1), np(X2)
s([the man, X0, the ball, X1, the house]) :- tverb(X0), prep(X1)
s([a boy, hits, a dog, at, a picnic])
s([the man, takes, the ball, to, the house])
s([X0, takes, X1, to, X2]) :- np(X0), np(X1), np(X2)
s([a boy, takes, a dog, to, a room])
------------------------------------------------------------
```

```
Result of first pass:
s([X0, hits, X1, at, X2]) :- np(X0), np(X1), np(X2)
s([X0, takes, X1, to, X2]) :- np(X0), np(X1), np(X2)
s([X0, walks, X1]) :- np(X0), np(X1)
s([X0, hits, X1]) :- np(X0), np(X1)
s([X0, sleeps]) :- np(X0)
s([X0, walks]) :- np(X0)
------------------------------------------------------------
Result of second pass:
s([X0, hits, X1, at, X2]) :- np(X0), np(X1), np(X2)
s([X0, Y0, X1, Y1, X2]) :- np(X0), np(X1), np(X2), tverb(Y0), prep(Y1)
s([X0, takes, X1, to, X2]) :- np(X0), np(X1), np(X2)
s([X0, walks, X1]) :- np(X0), np(X1)
s([X0, Y0, X1]) :- np(X0), np(X1), tverb(Y0)
s([X0, hits, X1]) :- np(X0), np(X1)
s([X0, sleeps]) :- np(X0)
s([X0, Y0]) :- np(X0), iverb(Y0)
s([X0, walks]) :- np(X0)
------------------------------------------------------------
Resulting Hypotheses:
s([X0, Y0, X1, Y1, X2]) :- np(X0), np(X1), np(X2), tverb(Y0), prep(Y1).
s([X0, Y0, X1]) :- np(X0), np(X1), tverb(Y0).
s([X0, Y0]) :- np(X0), iverb(Y0).
```

A.5 Pisti example

A.5.1 Progol

Input: `% Settings`

```
:-set(posonly)?
% Mode declarations
```

```
:- modeh(1,collect(+suit,+face,+suit,+face))?
%Types
suit(clubs).
suit(spades).
suit(hearts).
suit(diamonds).
face(ace).
face(two).
face(three).
face(four).
face(five).
face(six).
face(seven).
face(eight).
face(nine).
face(ten).
face(jack).
face(queen).
face(king).
%Examples
collect(clubs,queen,spades,queen).
collect(hearts,ace,diamonds,ace).
collect(spades,six,hearts,jack).
collect(clubs,eight,diamonds,jack).
collect(clubs,seven,spades,jack).
```

Output:

```
Noise has been set to 100%
Example inflation has been set to 400%
The posonly flag has been turned ON
:- set(posonly)?  - Time taken 0.00s
:- modeh(1,collect(+suit,+face,+suit,+face))?  - Time taken 0.00s
Testing for contradictions
```

```
No contradictions found
Generalising collect(clubs,queen,spades,queen).
Most specific clause is

collect(A,B,C,B).

Learning collect/4 from positive examples
C:13,8,2,0 collect(A,B,C,B).
C:-0,20,19,0 collect(A,B,C,D).
2 explored search nodes
f=13,p=8,n=2,h=0
Result of search is

collect(A,B,C,B).

2 redundant clauses retracted
Generalising collect(spades,six,hearts,jack).
Most specific clause is

collect(A,B,C,D).
Learning collect/4 from positive examples
C:-4,16,18,0 collect(A,B,C,D).
1 explored search nodes
f=-4,p=16,n=18,h=0
No compression

Generalising collect(clubs,eight,diamonds,jack).
Most specific clause is

collect(A,B,C,D).

Learning collect/4 from positive examples
C:-4,16,18,0 collect(A,B,C,D).
```

```
1 explored search nodes
f=-4,p=16,n=18,h=0
No compression

Generalising collect(clubs,seven,spades,jack).
Most specific clause is

collect(A,B,C,D).

Learning collect/4 from positive examples
C:-4,16,18,0 collect(A,B,C,D).
1 explored search nodes
f=-4,p=16,n=18,h=0
No compression

collect(spades,six,hearts,jack).
collect(clubs,eight,diamonds,jack).
collect(clubs,seven,spades,jack).
collect(A,B,C,B).

Total number of clauses = 4

Time taken 0.01s
```

A.5.2 FOIL

Input:

```
R: clubs, spades, hearts, diamonds.  S: ace, two, three, four, five,
six, seven, eight, nine, ten, jack, queen, king.
collect(R,S,R,S) clubs,queen,spades,queen hearts,ace,diamonds,ace spades,six,hear
clubs,eight,diamonds,jack clubs,seven,spades,jack .
```

Output:

```
Options:
```

```
no negated literals
verbosity level 3

Types R and S are not compatible

Relation collect

Ordering constants of type R
Checking arguments of collect
arguments 1,3 are consistent
Finding maximal consistent set
best so far collect:1>3
Final order:
diamonds hearts spades clubs

Ordering constants of type S
Checking arguments of collect
arguments 2,4 are not consistent
Finding maximal consistent set
Final order:
ace two three four five six seven eight nine ten jack queen king

----------
collect:

State (5/2704, 67.6 bits available)

A=C 0[0/1] [5/5] abandoned(0%)
B=D 2[2/208] gain 5.4
= tried 2/2 0.0 secs

Best literal B=D (2.0 bits)
Note B=D
```

```
State (2/208, 34.5 bits available)

A=C 0[0/1] [2/2] abandoned(1%)
= tried 1/1 0.0 secs

No literals

Initial clause (206 errs):  collect(A,B,C,D) :- B=D.
B=D essential

Clause too inaccurate (2/208)

** Warning:  the following definition
** does not cover 5 tuples in the relation


Time 0.0 secs
```

A.5.3 InGen

Input is:

```
//Examples:
collect(clubs,queen,spades,queen).
collect(hearts,ace,diamonds,ace).
collect(diamonds,three,clubs,three).
collect(spades,six,hearts,jack).
collect(hearts,eight,diamonds,jack).
collect(clubs,seven,spades,jack).
//There are no background literals
```

Output is:

```
Initially:
collect([clubs, :, queen, :, spades, :, queen])
collect([X0, :, X1, :, X2, :, X1])
collect([clubs, :, X0, :, spades, :, X1])
collect([hearts, :, ace, :, diamonds, :, ace])
collect([hearts, :, X0, :, diamonds, :, X1])
collect([diamonds, :, three, :, clubs, :, three])
collect([spades, :, six, :, hearts, :, jack])
collect([hearts, :, eight, :, diamonds, :, jack])
collect([X0, :, X1, :, X2, :, jack])
collect([clubs, :, seven, :, spades, :, jack])
------------------------------------------------------------
Result of first pass:
collect([X0, :, X1, :, X2, :, X1])
collect([X0, :, X1, :, X2, :, jack])
------------------------------------------------------------
Result of second pass:
collect([X0, :, X1, :, X2, :, X1])
collect([X0, :, X1, :, X2, :, jack])
------------------------------------------------------------
Resulting Hypotheses:
collect(X0, X1, X2, X1).
collect(X0, X1, X2, jack).
```

A.6 Tests on Mutagenesis Dataset

We used tewlve input files of Mutagenesis dataset to test our system. Ten of the files include example literals (They are divided by the authors possibly for cross validation). The example file contents are as follows:
File: s1.pl:

```
active(d18).      active(d26).      active(d28).
```

```
active(d51).      active(d63).      active(d67).
active(d107).     active(d127).     active(d137).
active(d151).     active(d174).     active(d178).
active(d85).      active(d92).      :- active(d38).
:- active(d84).   :- active(d100).  :- active(d116).
:- active(d160).
```

File: s2.pl:

```
active(d10).      active(d61).      active(d86).
active(d94).      active(d105).     active(d128).
active(d173).     active(d183).     active(d29).
:- active(d42).   :- active(d147).  :- active(d186).
:- active(d3).    :- active(d5).    :- active(d39).
:- active(d78).   :- active(d142).  :- active(d182).
:- active(d185).
```

File: s3.pl:

```
active(d46).      active(d52).      active(d57).
active(d80).      active(d82).      active(d125).
active(d152).     active(d169).     active(d21).
active(d22).      active(d108).     active(d180).
:- active(d119).  :- active(d34).   :- active(d65).
:- active(d66).   :- active(d143).  :- active(d154).
:- active(d181).
```

File: s4.pl:

```
active(d1).       active(d12).      active(d20).
active(d31).      active(d83).      active(d115).
active(d164).     active(d165).     active(d166).
active(d184).     active(d16).      active(d99).
active(d102).     active(d145).     active(d161).
```

```
active(d170).     :- active(d114).  :- active(d19).
:- active(d133).
```

File: s5.pl:

```
active(d6).       active(d15).      active(d69).
active(d71).      active(d87).      active(d95).
active(d104).     active(d109).     active(d177).
active(d187).     active(d44).      active(d93).
active(d97).      active(d106).     :- active(d132).
:- active(d141).  :- active(d73).   :- active(d76).
:- active(d113).
```

File: s6.pl:

```
active(d25).      active(d45).      active(d54).
active(d58).      active(d75).      active(d101).
active(d140).     active(d149).     active(d159).
active(d171).     active(d8).       active(d47).
active(d121).     active(d134).     :- active(d70).
:- active(d77).   :- active(d111).  :- active(d62).
:- active(d179).
```

File: s7.pl:

```
active(d13).      active(d24).      active(d43).
active(d64).      active(d72).      active(d79).
active(d90).      active(d122).     active(d126).
active(d163).     active(d172).     active(d30).
active(d37).      :- active(d2).    :- active(d17).
:- active(d120).  :- active(d130).  :- active(d131).
:- active(d129).
```

File: s8.pl:

```
active(d11).      active(d27).      active(d53).
active(d56).      active(d68).      active(d146).
active(d33).      active(d74).      :- active(d188).
:- active(d7).    :- active(d9).    :- active(d14).
:- active(d36).   :- active(d40).   :- active(d123).
:- active(d135).  :- active(d150).  :- active(d155).
:- active(d175).
```

File: s9.pl:

```
active(d4).       active(d41).      active(d49).
active(d50).      active(d59).      active(d96).
active(d117).     active(d136).     active(d153).
active(d23).      active(d32).      active(d158).
active(d167).     active(d176).     :- active(d98).
:- active(d88).   :- active(d124).  :- active(d139).
:- active(d168).
```

File: s10.pl:

```
active(d48).      active(d60).      active(d112).
active(d148).     active(d157).     active(d35).
active(d81).      active(d91).      active(d103).
active(d118).     active(d162).     :- active(d89).
:- active(d138).  :- active(d144).  :- active(d55).
:- active(d110).  :- active(d156).
```

The background knowledge files we used are:

File: logp.pl:

```
logp(d8, 3.46).     logp(d16, 4.44).    logp(d21, 3.52).
logp(d22, 5.09).    logp(d23, 5.07).    logp(d29, 2.42).
logp(d30, 5.07).    logp(d32, 3).       logp(d33, 4.23).
logp(d35, 3.06).    logp(d37, 4.18).    logp(d44, 4.53).
```

```
logp(d47, 5.02).    logp(d74, 5.87).    logp(d81, 1.49).
logp(d85, 5.07).    logp(d91, 3.01).    logp(d92, 5.28).
logp(d93, 5.87).    logp(d97, 3.95).    logp(d99, 2.72).
logp(d102, 2.4).    logp(d103, 4.69).   logp(d106, 4.34).
logp(d108, 4.69).   logp(d118, 6.57).   logp(d121, 4.18).
logp(d134, 4.73).   logp(d145, 2.68).   logp(d158, 3.08).
logp(d161, 5.61).   logp(d162, 3).      logp(d167, 4.44).
logp(d170, 2.52).   logp(d176, 2.74).   logp(d180, 6.07).
logp(f4, -0.47).    logp(f5, 0.95).     logp(d2, 1.44).
logp(d17, 0.87).    logp(d42, 1.77).    logp(d70, 0.47).
logp(d77, 2.68).    logp(d89, 1.77).    logp(d98, 1.65).
logp(d111, 1.56).   logp(d114, 2.68).   logp(d119, 3.26).
logp(d120, 1.77).   logp(d130, 1.73).   logp(d131, 2.83).
logp(d132, 1.74).   logp(d138, 1.77).   logp(d141, 3.05).
logp(d144, 2.9).    logp(d147, 1.94).   logp(d186, 2.07).
logp(d188, 3.51).   logp(d189, 2.29).   logp(d196, 3.51).
logp(e4, 2.5).      logp(e5, 1.85).     logp(e8, 2.03).
logp(e12, 1.58).    logp(e14, 2.39).    logp(e18, 3.12).
logp(e22, 2.55).    logp(e25, 4.78).    logp(e26, 5.06).
logp(d1, 4.23).     logp(d4, 4.69).     logp(d6, 3.92).
logp(d10, 4.62).    logp(d11, 4.23).    logp(d12, 3.63).
logp(d13, 4.44).    logp(d15, 4.69).    logp(d18, 3.06).
logp(d20, 3.4).     logp(d24, 6.79).    logp(d25, 3.43).
logp(d26, 2.17).    logp(d27, 5.87).    logp(d28, 4.11).
logp(d31, 4.44).    logp(d41, 4.83).    logp(d43, 4.69).
logp(d45, 1.46).    logp(d46, 6.57).    logp(d48, 4.23).
logp(d49, 2.29).    logp(d50, 2.58).    logp(d51, 6.01).
logp(d52, 6.01).    logp(d53, 3.35).    logp(d54, 6.24).
logp(d56, 2.52).    logp(d57, 3.26).    logp(d58, 3.01).
logp(d59, 5.07).    logp(d60, 2.3).     logp(d61, 4.68).
logp(d63, 2.79).    logp(d64, 2.84).    logp(d67, 4.66).
logp(d68, 5.87).    logp(d69, 3.71).    logp(d71, 6.01).
logp(d72, 3.37).    logp(d75, 3.52).    logp(d79, 4.73).
```

```
logp(d80, 4.69).    logp(d82, 4.18).    logp(d83, 3.51).
logp(d86, 5.07).    logp(d87, 3.83).    logp(d90, 5.87).
logp(d94, 4.69).    logp(d95, 2.55).    logp(d96, 4.18).
logp(d101, 6.26).   logp(d104, 6.26).   logp(d105, 1.84).
logp(d107, 4.18).   logp(d109, 6.07).   logp(d112, 3.81).
logp(d115, 2.74).   logp(d117, 3.26).   logp(d122, 5.41).
logp(d125, 4.99).   logp(d126, 2.29).   logp(d127, 3.26).
logp(d128, 3.36).   logp(d136, 4.66).   logp(d137, 4.44).
logp(d140, 2.52).   logp(d146, 2.68).   logp(d148, 3.36).
logp(d149, 2.29).   logp(d151, 1.75).   logp(d152, 4.44).
logp(d153, 3.85).   logp(d157, 4.19).   logp(d159, 4.19).
logp(d163, 4.44).   logp(d164, 4.44).   logp(d165, 4.69).
logp(d166, 4.42).   logp(d169, 1.49).   logp(d171, 3).
logp(d172, 2.06).   logp(d173, 3.26).   logp(d174, 4.44).
logp(d177, 2.29).   logp(d178, 1.77).   logp(d182, 1.99).
logp(d183, 4.44).   logp(d184, 4.44).   logp(d187, 5.41).
logp(d190, 2.13).   logp(d191, 4.35).   logp(d194, 0.88).
logp(d197, 1.58).   logp(e1, 5.87).     logp(e2, 6.16).
logp(e27, 5.87).    logp(f1, 1.01).     logp(f2, 0.96).
logp(f3, 0.23).     logp(f6, -0.04).    logp(d3, 1.86).
logp(d5, 1.89).     logp(d7, 3.99).     logp(d9, 1.64).
logp(d14, 1.77).    logp(d19, 1.84).    logp(d34, 3.24).
logp(d36, 3.43).    logp(d38, 3.77).    logp(d39, 1.87).
logp(d40, 3.19).    logp(d55, 3.43).    logp(d62, 1.36).
logp(d65, 2.83).    logp(d66, 2.68).    logp(d73, 2.68).
logp(d76, 2.24).    logp(d78, 4.27).    logp(d84, 2.61).
logp(d88, 1.53).    logp(d100, 2.68).   logp(d110, 3).
logp(d113, 1.84).   logp(d116, 1.8).    logp(d123, 4.49).
logp(d124, 1.89).   logp(d129, 1.46).   logp(d133, 7.13).
logp(d135, 2.74).   logp(d139, 2.72).   logp(d142, 6.68).
logp(d143, 2.35).   logp(d150, -0.02).  logp(d154, 1.59).
logp(d155, 1.72).   logp(d156, 1.92).   logp(d160, 2.78).
logp(d168, 2.03).   logp(d175, 7.84).   logp(d179, 2.73).
```

```
logp(d181, 0.53).   logp(d185, 1.89).   logp(d192, 5.41).
logp(d193, 5.41).   logp(d195, 3.61).   logp(e3, 2.86).
logp(e6, 2.5).      logp(e7, 2.13).     logp(e9, 1.69).
logp(e10, 2.13).    logp(e11, 1.9).     logp(e13, 2.41).
logp(e15, 5.41).    logp(e16, 3.09).    logp(e17, 4.78).
logp(e19, 1.38).    logp(e20, 1.38).    logp(e21, 6.57).
logp(e23, 2.55).    logp(e24, 5.41).
```

File: lumo.pl:

```
lumo(d8, -1.437).     lumo(d16, -2.172).    lumo(d21, -1.665).
lumo(d22, -1.602).    lumo(d23, -2.164).    lumo(d29, -2.837).
lumo(d30, -2.005).    lumo(d32, -2.562).    lumo(d33, -1.591).
lumo(d35, -1.176).    lumo(d37, -1.428).    lumo(d44, -1.265).
lumo(d47, -1.88).     lumo(d74, -1.689).    lumo(d81, -1.937).
lumo(d85, -2.113).    lumo(d91, -2.032).    lumo(d92, -1.208).
lumo(d93, -1.729).    lumo(d97, -1.361).    lumo(d99, -2.159).
lumo(d102, -3.172).   lumo(d103, -1.487).   lumo(d106, -1.607).
lumo(d108, -1.676).   lumo(d118, -1.8).     lumo(d121, -2.68).
lumo(d134, -1.951).   lumo(d145, -1.178).   lumo(d158, -1.34).
lumo(d161, -2.221).   lumo(d162, -2.687).   lumo(d167, -2.31).
lumo(d170, -2.113).   lumo(d176, -1.304).   lumo(d180, -2.182).
lumo(f4, -1.645).     lumo(f5, -1.526).     lumo(d2, -1.429).
lumo(d17, -0.529).    lumo(d42, -1.19).     lumo(d70, -1.786).
lumo(d77, -1.029).    lumo(d89, -1.028).    lumo(d98, -1.598).
lumo(d111, -1.687).   lumo(d114, -1.148).   lumo(d119, -1.995).
lumo(d120, -0.937).   lumo(d130, -0.93).    lumo(d131, -1.538).
lumo(d132, -1.499).   lumo(d138, -1.157).   lumo(d141, -1.228).
lumo(d144, -1.288).   lumo(d147, -0.937).   lumo(d186, -0.574).
lumo(d188, -0.872).   lumo(d189, -3.025).   lumo(d196, -1.092).
lumo(e4, -0.746).     lumo(e5, -1.089).     lumo(e8, -1.117).
lumo(e12, -1.834).    lumo(e14, -1.36).     lumo(e18, -1.538).
lumo(e22, -1.636).    lumo(e25, -1.748).    lumo(e26, -1.82).
```

```
lumo(d1, -1.246).     lumo(d4, -1.591).     lumo(d6, -3.406).
lumo(d10, -1.387).    lumo(d11, -1.254).    lumo(d12, -1.627).
lumo(d13, -2.292).    lumo(d15, -1.698).    lumo(d18, -1.861).
lumo(d20, -1.764).    lumo(d24, -1.728).    lumo(d25, -1.398).
lumo(d26, -2.072).    lumo(d27, -1.801).    lumo(d28, -1.558).
lumo(d31, -2.055).    lumo(d41, -1.6).      lumo(d43, -1.57).
lumo(d45, -2.227).    lumo(d46, -1.804).    lumo(d48, -1.616).
lumo(d49, -2.808).    lumo(d50, -1.932).    lumo(d51, -2.184).
lumo(d52, -2.189).    lumo(d53, -2.155).    lumo(d54, -1.464).
lumo(d56, -2.234).    lumo(d57, -2.242).    lumo(d58, -1.991).
lumo(d59, -2.14).     lumo(d60, -2.468).    lumo(d61, -1.556).
lumo(d63, -3.768).    lumo(d64, -2.338).    lumo(d67, -1.536).
lumo(d68, -1.766).    lumo(d69, -1.929).    lumo(d71, -2.095).
lumo(d72, -1.448).    lumo(d75, -1.87).     lumo(d79, -1.26).
lumo(d80, -1.329).    lumo(d82, -2.71).     lumo(d83, -1.145).
lumo(d86, -1.918).    lumo(d87, -1.488).    lumo(d90, -1.62).
lumo(d94, -1.585).    lumo(d95, -2.434).    lumo(d96, -2.871).
lumo(d101, -1.598).   lumo(d104, -1.546).   lumo(d105, -1.749).
lumo(d107, -2.791).   lumo(d109, -2.284).   lumo(d112, -1.208).
lumo(d115, -1.161).   lumo(d117, -2.142).   lumo(d122, -1.61).
lumo(d125, -1.256).   lumo(d126, -2.718).   lumo(d127, -2.196).
lumo(d128, -2.149).   lumo(d136, -1.685).   lumo(d137, -2.263).
lumo(d140, -1.751).   lumo(d146, -1.102).   lumo(d148, -2.158).
lumo(d149, -2.87).    lumo(d151, -1.411).   lumo(d152, -2.191).
lumo(d153, -1.151).   lumo(d157, -1.623).   lumo(d159, -1.742).
lumo(d163, -1.974).   lumo(d164, -2.306).   lumo(d165, -1.522).
lumo(d166, -1.709).   lumo(d169, -2.17).    lumo(d171, -2.508).
lumo(d172, -1.487).   lumo(d173, -2.328).   lumo(d174, -2.209).
lumo(d177, -2.614).   lumo(d178, -1.213).   lumo(d182, -1.366).
lumo(d183, -2.294).   lumo(d184, -2.074).   lumo(d187, -1.276).
lumo(d190, -0.798).   lumo(d191, -2.138).   lumo(d194, -0.857).
lumo(d197, -1.293).   lumo(e1, -1.767).     lumo(e2, -1.35).
lumo(e27, -1.845).    lumo(f1, -1.785).     lumo(f2, -1.851).
```

```
lumo(f3, -1.412).     lumo(f6, -1.503).     lumo(d3, -1.456).
lumo(d5, -1.59).      lumo(d7, -1.144).     lumo(d9, -0.982).
lumo(d14, -1.289).    lumo(d19, -1.478).    lumo(d34, -1.451).
lumo(d36, -1.24).     lumo(d38, -1.228).    lumo(d39, -1.443).
lumo(d40, -1.266).    lumo(d55, -1.177).    lumo(d62, -0.923).
lumo(d65, -1.952).    lumo(d66, -0.959).    lumo(d73, -0.648).
lumo(d76, -1.069).    lumo(d78, -1.276).    lumo(d84, -1.256).
lumo(d88, -1.605).    lumo(d100, -1.034).   lumo(d110, -2.14).
lumo(d113, -1.491).   lumo(d116, -1.37).    lumo(d123, -1.056).
lumo(d124, -1.596).   lumo(d129, -1.592).   lumo(d133, -1.492).
lumo(d135, -1.562).   lumo(d139, -1.019).   lumo(d142, -1.474).
lumo(d143, -1.046).   lumo(d150, -0.995).   lumo(d154, -1.362).
lumo(d155, -1.737).   lumo(d156, -0.854).   lumo(d160, -1.691).
lumo(d168, -1.112).   lumo(d175, -1.616).   lumo(d179, -1.889).
lumo(d181, -0.727).   lumo(d185, -2.09).    lumo(d192, -1.429).
lumo(d193, -1.478).   lumo(d195, -1.465).   lumo(e3, -0.56).
lumo(e6, -0.868).     lumo(e7, -1.05).      lumo(e9, -1.321).
lumo(e10, -1.125).    lumo(e11, -1.358).    lumo(e13, -1.306).
lumo(e15, -1.723).    lumo(e16, -1.351).    lumo(e17, -1.755).
lumo(e19, -1.392).    lumo(e20, -1.447).    lumo(e21, -1.717).
lumo(e23, -1.808).    lumo(e24, -1.113).
```

File: ind1.pl:

```
ind1(d1,ind1).      ind1(d2,noind1).    ind1(d3,noind1).
ind1(d4,ind1).      ind1(d5,noind1).    ind1(d6,ind1).
ind1(d7,noind1).    ind1(d8,ind1).      ind1(d9,noind1).
ind1(d10,ind1).     ind1(d11,ind1).     ind1(d12,ind1).
ind1(d13,ind1).     ind1(d14,noind1).   ind1(d15,ind1).
ind1(d16,ind1).     ind1(d17,noind1).   ind1(d18,ind1).
ind1(d19,noind1).   ind1(d20,ind1).     ind1(d21,noind1).
ind1(d22,ind1).     ind1(d23,ind1).     ind1(d24,ind1).
ind1(d25,ind1).     ind1(d26,noind1).   ind1(d27,ind1).
```

```
ind1(d28,ind1).     ind1(d29,ind1).     ind1(d30,ind1).
ind1(d31,ind1).     ind1(d32,noind1).   ind1(d33,ind1).
ind1(d34,noind1).   ind1(d35,ind1).     ind1(d36,noind1).
ind1(d37,noind1).   ind1(d38,noind1).   ind1(d39,noind1).
ind1(d40,noind1).   ind1(d41,ind1).     ind1(d42,noind1).
ind1(d43,ind1).     ind1(d44,ind1).     ind1(d45,noind1).
ind1(d46,ind1).     ind1(d47,ind1).     ind1(d48,ind1).
ind1(d49,ind1).     ind1(d50,noind1).   ind1(d51,ind1).
ind1(d52,ind1).     ind1(d53,ind1).     ind1(d54,ind1).
ind1(d55,noind1).   ind1(d56,ind1).     ind1(d57,noind1).
ind1(d58,ind1).     ind1(d59,ind1).     ind1(d60,noind1).
ind1(d61,ind1).     ind1(d62,noind1).   ind1(d63,noind1).
ind1(d64,ind1).     ind1(d65,noind1).   ind1(d66,noind1).
ind1(d67,ind1).     ind1(d68,ind1).     ind1(d69,ind1).
ind1(d70,noind1).   ind1(d71,ind1).     ind1(d72,ind1).
ind1(d73,noind1).   ind1(d74,ind1).     ind1(d75,noind1).
ind1(d76,noind1).   ind1(d77,noind1).   ind1(d78,noind1).
ind1(d79,ind1).     ind1(d80,ind1).     ind1(d81,noind1).
ind1(d82,ind1).     ind1(d83,ind1).     ind1(d84,noind1).
ind1(d85,ind1).     ind1(d86,ind1).     ind1(d87,ind1).
ind1(d88,noind1).   ind1(d89,noind1).   ind1(d90,ind1).
ind1(d91,ind1).     ind1(d92,ind1).     ind1(d93,ind1).
ind1(d94,ind1).     ind1(d95,noind1).   ind1(d96,ind1).
ind1(d97,ind1).     ind1(d98,noind1).   ind1(d99,ind1).
ind1(d100,noind1).  ind1(d101,ind1).    ind1(d102,ind1).
ind1(d103,ind1).    ind1(d104,ind1).    ind1(d105,ind1).
ind1(d106,ind1).    ind1(d107,ind1).    ind1(d108,ind1).
ind1(d109,ind1).    ind1(d110,noind1).  ind1(d111,noind1).
ind1(d112,ind1).    ind1(d113,noind1).  ind1(d114,noind1).
ind1(d115,noind1).  ind1(d116,noind1).  ind1(d117,noind1).
ind1(d118,ind1).    ind1(d119,noind1).  ind1(d120,noind1).
ind1(d121,ind1).    ind1(d122,ind1).    ind1(d123,noind1).
ind1(d124,noind1).  ind1(d125,ind1).    ind1(d126,ind1).
```

```
ind1(d127,noind1).  ind1(d128,ind1).    ind1(d129,noind1).
ind1(d130,noind1).  ind1(d131,noind1).  ind1(d132,noind1).
ind1(d133,ind1).    ind1(d134,ind1).    ind1(d135,noind1).
ind1(d136,ind1).    ind1(d137,ind1).    ind1(d138,noind1).
ind1(d139,noind1).  ind1(d140,noind1).  ind1(d141,noind1).
ind1(d142,ind1).    ind1(d143,noind1).  ind1(d144,noind1).
ind1(d145,noind1).  ind1(d146,noind1).  ind1(d147,noind1).
ind1(d148,ind1).    ind1(d149,ind1).    ind1(d150,noind1).
ind1(d151,noind1).  ind1(d152,ind1).    ind1(d153,ind1).
ind1(d154,noind1).  ind1(d155,noind1).  ind1(d156,noind1).
ind1(d157,ind1).    ind1(d158,ind1).    ind1(d159,ind1).
ind1(d160,noind1).  ind1(d161,ind1).    ind1(d162,noind1).
ind1(d163,ind1).    ind1(d164,ind1).    ind1(d165,ind1).
ind1(d166,ind1).    ind1(d167,ind1).    ind1(d168,noind1).
ind1(d169,noind1).  ind1(d170,ind1).    ind1(d171,noind1).
ind1(d172,noind1).  ind1(d173,noind1).  ind1(d174,ind1).
ind1(d175,ind1).    ind1(d176,noind1).  ind1(d177,ind1).
ind1(d178,noind1).  ind1(d179,noind1).  ind1(d180,ind1).
ind1(d181,noind1).  ind1(d182,noind1).  ind1(d183,ind1).
ind1(d184,ind1).    ind1(d185,noind1).  ind1(d186,noind1).
ind1(d187,ind1).    ind1(d188,ind1).    ind1(d189,noind1).
ind1(d190,noind1).  ind1(d191,ind1).    ind1(d192,ind1).
ind1(d193,ind1).    ind1(d194,noind1).  ind1(d195,noind1).
ind1(d196,ind1).    ind1(d197,noind1).  ind1(e1,ind1).
ind1(e2,ind1).      ind1(e3,noind1).    ind1(e4,noind1).
ind1(e5,noind1).    ind1(e6,noind1).    ind1(e7,noind1).
ind1(e8,noind1).    ind1(e9,noind1).    ind1(e10,noind1).
ind1(e11,noind1).   ind1(e12,noind1).   ind1(e13,noind1).
ind1(e14,noind1).   ind1(e15,ind1).     ind1(e16,noind1).
ind1(e17,ind1).     ind1(e18,noind1).   ind1(e19,ind1).
ind1(e20,ind1).     ind1(e21,ind1).     ind1(e22,ind1).
ind1(e23,ind1).     ind1(e24,ind1).     ind1(e25,ind1).
ind1(e26,ind1).     ind1(e27,ind1).     ind1(f1,noind1).
```

```
ind1(f2,noind1).    ind1(f3,noind1).    ind1(f4,noind1).
ind1(f5,noind1).    ind1(f6,noind1).
```

File: inda.pl

```
inda(d1,noinda).    inda(d2,noinda).    inda(d3,noinda).
inda(d4,noinda).    inda(d5,noinda).    inda(d6,noinda).
inda(d7,noinda).    inda(d8,noinda).    inda(d9,noinda).
inda(d10,noinda).   inda(d11,noinda).   inda(d12,noinda).
inda(d13,noinda).   inda(d14,noinda).   inda(d15,noinda).
inda(d16,noinda).   inda(d17,noinda).   inda(d18,noinda).
inda(d19,noinda).   inda(d20,noinda).   inda(d21,noinda).
inda(d22,noinda).   inda(d23,inda).     inda(d24,noinda).
inda(d25,noinda).   inda(d26,noinda).   inda(d27,noinda).
inda(d28,noinda).   inda(d29,noinda).   inda(d30,inda).
inda(d31,noinda).   inda(d32,noinda).   inda(d33,noinda).
inda(d34,noinda).   inda(d35,noinda).   inda(d36,noinda).
inda(d37,noinda).   inda(d38,noinda).   inda(d39,noinda).
inda(d40,noinda).   inda(d41,noinda).   inda(d42,noinda).
inda(d43,noinda).   inda(d44,noinda).   inda(d45,noinda).
inda(d46,noinda).   inda(d47,noinda).   inda(d48,noinda).
inda(d49,noinda).   inda(d50,noinda).   inda(d51,noinda).
inda(d52,noinda).   inda(d53,noinda).   inda(d54,noinda).
inda(d55,noinda).   inda(d56,noinda).   inda(d57,noinda).
inda(d58,noinda).   inda(d59,inda).     inda(d60,noinda).
inda(d61,noinda).   inda(d62,noinda).   inda(d63,noinda).
inda(d64,noinda).   inda(d65,noinda).   inda(d66,noinda).
inda(d67,noinda).   inda(d68,noinda).   inda(d69,noinda).
inda(d70,noinda).   inda(d71,noinda).   inda(d72,noinda).
inda(d73,noinda).   inda(d74,noinda).   inda(d75,noinda).
inda(d76,noinda).   inda(d77,noinda).   inda(d78,noinda).
inda(d79,noinda).   inda(d80,noinda).   inda(d81,noinda).
inda(d82,noinda).   inda(d83,noinda).   inda(d84,noinda).
inda(d85,inda).     inda(d86,inda).     inda(d87,noinda).
```

```
inda(d88,noinda).   inda(d89,noinda).   inda(d90,noinda).
inda(d91,noinda).   inda(d92,noinda).   inda(d93,noinda).
inda(d94,noinda).   inda(d95,noinda).   inda(d96,noinda).
inda(d97,noinda).   inda(d98,noinda).   inda(d99,noinda).
inda(d100,noinda).  inda(d101,noinda).  inda(d102,noinda).
inda(d103,noinda).  inda(d104,noinda).  inda(d105,noinda).
inda(d106,noinda).  inda(d107,noinda).  inda(d108,noinda).
inda(d109,noinda).  inda(d110,noinda).  inda(d111,noinda).
inda(d112,noinda).  inda(d113,noinda).  inda(d114,noinda).
inda(d115,noinda).  inda(d116,noinda).  inda(d117,noinda).
inda(d118,noinda).  inda(d119,noinda).  inda(d120,noinda).
inda(d121,noinda).  inda(d122,noinda).  inda(d123,noinda).
inda(d124,noinda).  inda(d125,noinda).  inda(d126,noinda).
inda(d127,noinda).  inda(d128,noinda).  inda(d129,noinda).
inda(d130,noinda).  inda(d131,noinda).  inda(d132,noinda).
inda(d133,noinda).  inda(d134,noinda).  inda(d135,noinda).
inda(d136,noinda).  inda(d137,noinda).  inda(d138,noinda).
inda(d139,noinda).  inda(d140,noinda).  inda(d141,noinda).
inda(d142,noinda).  inda(d143,noinda).  inda(d144,noinda).
inda(d145,noinda).  inda(d146,noinda).  inda(d147,noinda).
inda(d148,noinda).  inda(d149,noinda).  inda(d150,noinda).
inda(d151,noinda).  inda(d152,noinda).  inda(d153,noinda).
inda(d154,noinda).  inda(d155,noinda).  inda(d156,noinda).
inda(d157,noinda).  inda(d158,noinda).  inda(d159,noinda).
inda(d160,noinda).  inda(d161,noinda).  inda(d162,noinda).
inda(d163,noinda).  inda(d164,noinda).  inda(d165,noinda).
inda(d166,noinda).  inda(d167,noinda).  inda(d168,noinda).
inda(d169,noinda).  inda(d170,noinda).  inda(d171,noinda).
inda(d172,noinda).  inda(d173,noinda).  inda(d174,noinda).
inda(d175,noinda).  inda(d176,noinda).  inda(d177,noinda).
inda(d178,noinda).  inda(d179,noinda).  inda(d180,noinda).
inda(d181,noinda).  inda(d182,noinda).  inda(d183,noinda).
inda(d184,noinda).  inda(d185,noinda).  inda(d186,noinda).
```

```
inda(d187,noinda).  inda(d188,noinda).  inda(d189,noinda).
inda(d190,noinda).  inda(d191,noinda).  inda(d192,noinda).
inda(d193,noinda).  inda(d194,noinda).  inda(d195,noinda).
inda(d196,noinda).  inda(d197,noinda).  inda(e1,noinda).
inda(e2,noinda).    inda(e3,noinda).    inda(e4,noinda).
inda(e5,noinda).    inda(e6,noinda).    inda(e7,noinda).
inda(e8,noinda).    inda(e9,noinda).    inda(e10,noinda).
inda(e11,noinda).   inda(e12,noinda).   inda(e13,noinda).
inda(e14,noinda).   inda(e15,noinda).   inda(e16,noinda).
inda(e17,noinda).   inda(e18,noinda).   inda(e19,noinda).
inda(e20,noinda).   inda(e21,noinda).   inda(e22,noinda).
inda(e23,noinda).   inda(e24,noinda).   inda(e25,noinda).
inda(e26,noinda).   inda(e27,noinda).   inda(f1,noinda).
inda(f2,noinda).    inda(f3,noinda).    inda(f4,noinda).
inda(f5,noinda).    inda(f6,noinda).
```

We merged example files and background files and produced 20 files as listed below:

File: logp1.pl

```
active(true,noinda,3.06).     active(false,noinda,2.17).
active(true,noinda,4.11).     active(true,noinda,6.01).
active(false,noinda,2.79).    active(true,noinda,4.66).
active(true,noinda,4.18).     active(false,noinda,3.26).
active(true,noinda,4.44).     active(false,noinda,1.75).
active(true,noinda,4.44).     active(false,noinda,1.77).
active(true,inda,5.07).       active(true,noinda,5.28).
:-active(false,noinda,3.77).  :-active(false,noinda,2.61).
:-active(false,noinda,2.68).  :-active(false,noinda,1.8).
:-active(false,noinda,2.78).
```

File: logp2.pl

```
active(true,noinda,4.62).     active(true,noinda,4.68).
```

```
active(true,inda,5.07).          active(true,noinda,4.69).
active(true,noinda,1.84).        active(true,noinda,3.36).
active(false,noinda,3.26).       active(true,noinda,4.44).
active(true,noinda,2.42).        :-active(false,noinda,1.77).
:-active(false,noinda,1.94).     :-active(false,noinda,2.07).
:-active(false,noinda,1.86).     :-active(false,noinda,1.89).
:-active(false,noinda,1.87).     :-active(false,noinda,4.27).
:-active(true,noinda,6.68).      :-active(false,noinda,1.99).
:-active(false,noinda,1.89).
```

File: logp3.pl

```
active(true,noinda,6.57).        active(true,noinda,6.01).
active(false,noinda,3.26).       active(true,noinda,4.69).
active(true,noinda,4.18).        active(true,noinda,4.99).
active(true,noinda,4.44).        active(false,noinda,1.49).
active(false,noinda,3.52).       active(true,noinda,5.09).
active(true,noinda,4.69).        active(true,noinda,6.07).
:-active(false,noinda,3.26).     :-active(false,noinda,3.24).
:-active(false,noinda,2.83).     :-active(false,noinda,2.68).
:-active(false,noinda,2.35).     :-active(false,noinda,1.59).
:-active(false,noinda,0.53).
```

File: logp4.pl

```
active(true,noinda,4.23).        active(true,noinda,3.63).
active(true,noinda,3.4).         active(true,noinda,4.44).
active(true,noinda,3.51).        active(false,noinda,2.74).
active(true,noinda,4.44).        active(true,noinda,4.69).
active(true,noinda,4.42).        active(true,noinda,4.44).
active(true,noinda,4.44).        active(true,noinda,2.72).
active(true,noinda,2.4).         active(false,noinda,2.68).
active(true,noinda,5.61).        active(true,noinda,2.52).
:-active(false,noinda,2.68).     :-active(false,noinda,1.84).
:-active(true,noinda,7.13).
```

File: logp5.pl

```
active(true,noinda,3.92).      active(true,noinda,4.69).
active(true,noinda,3.71).      active(true,noinda,6.01).
active(true,noinda,3.83).      active(false,noinda,2.55).
active(true,noinda,6.26).      active(true,noinda,6.07).
active(true,noinda,2.29).      active(true,noinda,5.41).
active(true,noinda,4.53).      active(true,noinda,5.87).
active(true,noinda,3.95).      active(true,noinda,4.34).
:-active(false,noinda,1.74).   :-active(false,noinda,3.05).
:-active(false,noinda,2.68).   :-active(false,noinda,2.24).
:-active(false,noinda,1.84).
```

File: logp6.pl

```
active(true,noinda,3.43).      active(false,noinda,1.46).
active(true,noinda,6.24).      active(true,noinda,3.01).
active(false,noinda,3.52).     active(true,noinda,6.26).
active(false,noinda,2.52).     active(true,noinda,2.29).
active(true,noinda,4.19).      active(false,noinda,3.0).
active(true,noinda,3.46).      active(true,noinda,5.02).
active(true,noinda,4.18).      active(true,noinda,4.73).
:-active(false,noinda,0.47).   :-active(false,noinda,2.68).
:-active(false,noinda,1.56).   :-active(false,noinda,1.36).
:-active(false,noinda,2.73).
```

File: logp7.pl

```
active(true,noinda,4.44).      active(true,noinda,6.79).
active(true,noinda,4.69).      active(true,noinda,2.84).
active(true,noinda,3.37).      active(true,noinda,4.73).
active(true,noinda,5.87).      active(true,noinda,5.41).
active(true,noinda,2.29).      active(true,noinda,4.44).
active(false,noinda,2.06).     active(true,inda,5.07).
active(false,noinda,4.18).     :-active(false,noinda,1.44).
:-active(false,noinda,0.87).   :-active(false,noinda,1.77).
:-active(false,noinda,1.73).   :-active(false,noinda,2.83).
```

```
:-active(false,noinda,1.46).
```

File: logp8.pl

```
active(true,noinda,4.23).      active(true,noinda,5.87).
active(true,noinda,3.35).      active(true,noinda,2.52).
active(true,noinda,5.87).      active(false,noinda,2.68).
active(true,noinda,4.23).      active(true,noinda,5.87).
:-active(true,noinda,3.51).    :-active(false,noinda,3.99).
:-active(false,noinda,1.64).   :-active(false,noinda,1.77).
:-active(false,noinda,3.43).   :-active(false,noinda,3.19).
:-active(false,noinda,4.49).   :-active(false,noinda,2.74).
:-active(false,noinda,-0.02). :-active(false,noinda,1.72).
:-active(true,noinda,7.84).
```

File: logp9.pl

```
active(true,noinda,4.69).      active(true,noinda,4.83).
active(true,noinda,2.29).      active(false,noinda,2.58).
active(true,inda,5.07).        active(true,noinda,4.18).
active(false,noinda,3.26).     active(true,noinda,4.66).
active(true,noinda,3.85).      active(true,inda,5.07).
active(false,noinda,3.0).      active(true,noinda,3.08).
active(true,noinda,4.44).      active(false,noinda,2.74).
:-active(false,noinda,1.65).   :-active(false,noinda,1.53).
:-active(false,noinda,1.89).   :-active(false,noinda,2.72).
:-active(false,noinda,2.03).
```

File: logp10.pl

```
active(true,noinda,4.23).      active(false,noinda,2.3).
active(true,noinda,3.81).      active(true,noinda,3.36).
active(true,noinda,4.19).      active(true,noinda,3.06).
active(false,noinda,1.49).     active(true,noinda,3.01).
active(true,noinda,4.69).      active(true,noinda,6.57).
active(false,noinda,3.0).      :-active(false,noinda,1.77).
:-active(false,noinda,1.77).   :-active(false,noinda,2.9).
```

```
:-active(false,noinda,3.43).  :-active(false,noinda,3.0).
:-active(false,noinda,1.92).
```

InGen output for the last test (when the examples in logp10.pl are used for testing) is as follows:

```
Resulting Hypotheses:
active(X0, noinda, N28) :- real(N28), between(N28, 4.04, 4.25).
active(true, X0, N4) :- real(N4), between(N4, 4.9199996, 5.1600003).
active(X0, noinda, N18) :- real(N18), between(N18, 2.45, 2.5949998).
active(X0, noinda, N23) :- real(N23), between(N23, 2.93, 3.03).
active(true, noinda, N3) :- real(N3), between(N3, 4.7599998, 4.9).
active(false, noinda, N28) :- real(N28), between(N28, 4.04, 4.25).
active(false, noinda, N16) :- real(N16), between(N16, 2.045, 2.065).
active(true, noinda, N22) :- real(N22), between(N22, 2.835, 2.9099998).
active(true, noinda, N12) :- real(N12), between(N12, 6.72, 6.86).
active(false, noinda, N23) :- real(N23), between(N23, 2.93, 3.03).
active(true, noinda, N23) :- real(N23), between(N23, 2.93, 3.03).
active(true, noinda, N33) :- real(N33), between(N33, 4.3050003, 4.4100003).
active(true, noinda, N8) :- real(N8), between(N8, 5.7999997, 5.94).
active(true, noinda, N34) :- real(N34), between(N34, 4.51, 4.6000004).
active(true, noinda, N6) :- real(N6), between(N6, 5.3399997, 5.48).
active(true, noinda, N30) :- real(N30), between(N30, 2.2649999, 2.32).
active(true, noinda, N10) :- real(N10), between(N10, 6.1699996, 6.3300004).
active(false, noinda, N18) :- real(N18), between(N18, 2.45, 2.5949998).
active(true, noinda, N32) :- real(N32), between(N32, 3.8, 3.9199998).
active(true, noinda, N26) :- real(N26), between(N26, 3.64, 3.74).
active(true, noinda, N27) :- real(N27), between(N27, 3.8500001, 3.97).
active(true, noinda, N18) :- real(N18), between(N18, 2.45, 2.5949998).
active(true, noinda, N7) :- real(N7), between(N7, 5.54, 5.6800003).
active(true, noinda, N19) :- real(N19), between(N19, 2.6100001, 2.725).
active(true, noinda, N1) :- real(N1), between(N1, 3.5600002, 3.7).
active(true, noinda, N4) :- real(N4), between(N4, 4.9199996, 5.1600003).
```

```
active(true, noinda, N11) :- real(N11), between(N11, 6.5, 6.6400003).
active(true, noinda, N31) :- real(N31), between(N31, 2.375, 2.49).
active(true, noinda, N15) :- real(N15), between(N15, 1.8199999, 1.85).
active(true, noinda, N5) :- real(N5), between(N5, 5.21, 5.3500004).
active(true, noinda, N29) :- real(N29), between(N29, 4.35, 4.465).
active(true, noinda, N2) :- real(N2), between(N2, 4.5499997, 4.8).
active(false, noinda, N21) :- real(N21), between(N21, 2.7849998, 2.81).
active(true, noinda, N9) :- real(N9), between(N9, 5.94, 6.1400003).
active(true, noinda, N28) :- real(N28), between(N28, 4.04, 4.25).
active(false, noinda, N17) :- real(N17), between(N17, 2.1000001, 2.205).
active(true, noinda, N24) :- real(N24), between(N24, 3.0549998, 3.1499999).
active(false, noinda, 3.26).
active(false, noinda, 1.75).
active(false, noinda, 1.77).
active(true, noinda, 3.36).
active(false, noinda, 1.49).
active(false, noinda, 3.52).
active(true, noinda, 3.4).
active(true, noinda, 3.51).
active(false, noinda, 2.74).
active(false, noinda, 2.68).
active(true, noinda, 3.43).
active(false, noinda, 1.46).
active(true, noinda, 3.46).
active(true, noinda, 3.37).
active(true, noinda, 3.35)
```

File lumo1.pl

```
active(true,noinda,-1.861).   active(false,noinda,-2.072).
active(true,noinda,-1.558).   active(true,noinda,-2.184).
active(false,noinda,-3.768).  active(true,noinda,-1.536).
active(true,noinda,-2.791).   active(false,noinda,-2.196).
active(true,noinda,-2.263).   active(false,noinda,-1.411).
```

```
active(true,noinda,-2.209).   active(false,noinda,-1.213).
active(true,inda,-2.113).     active(true,noinda,-1.208).
:-active(false,noinda,-1.228).:-active(false,noinda,-1.256).
:-active(false,noinda,-1.034).:-active(false,noinda,-1.37).
:-active(false,noinda,-1.691).
```

File lumo2.pl

```
active(true,noinda,-1.387).   active(true,noinda,-1.556).
active(true,inda,-1.918).     active(true,noinda,-1.585).
active(true,noinda,-1.749).   active(true,noinda,-2.149).
active(false,noinda,-2.328).  active(true,noinda,-2.294).
active(true,noinda,-2.837).   :-active(false,noinda,-1.19).
:-active(false,noinda,-0.937).:-active(false,noinda,-0.574).
:-active(false,noinda,-1.456).:-active(false,noinda,-1.59).
:-active(false,noinda,-1.443).:-active(false,noinda,-1.276).
:-active(true,noinda,-1.474). :-active(false,noinda,-1.366).
:-active(false,noinda,-2.09).
```

File lumo3.pl

```
active(true,noinda,-1.804).   active(true,noinda,-2.189).
active(false,noinda,-2.242).  active(true,noinda,-1.329).
active(true,noinda,-2.71).    active(true,noinda,-1.256).
active(true,noinda,-2.191).   active(false,noinda,-2.17).
active(false,noinda,-1.665).  active(true,noinda,-1.602).
active(true,noinda,-1.676).   active(true,noinda,-2.182).
:-active(false,noinda,-1.995).:-active(false,noinda,-1.451).
:-active(false,noinda,-1.952).:-active(false,noinda,-0.959).
:-active(false,noinda,-1.046).:-active(false,noinda,-1.362).
:-active(false,noinda,-0.727).
```

File lumo4.pl

```
active(true,noinda,-1.246).   active(true,noinda,-1.627).
active(true,noinda,-1.764).   active(true,noinda,-2.055).
active(true,noinda,-1.145).   active(false,noinda,-1.161).
```

```
active(true,noinda,-2.306).   active(true,noinda,-1.522).
active(true,noinda,-1.709).   active(true,noinda,-2.074).
active(true,noinda,-2.172).   active(true,noinda,-2.159).
active(true,noinda,-3.172).   active(false,noinda,-1.178).
active(true,noinda,-2.221).   active(true,noinda,-2.113).
:-active(false,noinda,-1.148).:-active(false,noinda,-1.478).
:-active(true,noinda,-1.492).
```

File lumo5.pl

```
active(true,noinda,-3.406).   active(true,noinda,-1.698).
active(true,noinda,-1.929).   active(true,noinda,-2.095).
active(true,noinda,-1.488).   active(false,noinda,-2.434).
active(true,noinda,-1.546).   active(true,noinda,-2.284).
active(true,noinda,-2.614).   active(true,noinda,-1.276).
active(true,noinda,-1.265).   active(true,noinda,-1.729).
active(true,noinda,-1.361).   active(true,noinda,-1.607).
:-active(false,noinda,-1.499).:-active(false,noinda,-1.228).
:-active(false,noinda,-0.648).:-active(false,noinda,-1.069).
:-active(false,noinda,-1.491).
```

File lumo6.pl

```
active(true,noinda,-1.398).   active(false,noinda,-2.227).
active(true,noinda,-1.464).   active(true,noinda,-1.991).
active(false,noinda,-1.87).   active(true,noinda,-1.598).
active(false,noinda,-1.751).  active(true,noinda,-2.87).
active(true,noinda,-1.742).   active(false,noinda,-2.508).
active(true,noinda,-1.437).   active(true,noinda,-1.88).
active(true,noinda,-2.68).    active(true,noinda,-1.951).
:-active(false,noinda,-1.786).:-active(false,noinda,-1.029).
:-active(false,noinda,-1.687).:-active(false,noinda,-0.923).
:-active(false,noinda,-1.889).
```

File lumo7.pl

```
active(true,noinda,-2.292).   active(true,noinda,-1.728).
```

```
active(true,noinda,-1.57).    active(true,noinda,-2.338).
active(true,noinda,-1.448).   active(true,noinda,-1.26).
active(true,noinda,-1.62).    active(true,noinda,-1.61).
active(true,noinda,-2.718).   active(true,noinda,-1.974).
active(false,noinda,-1.487).  active(true,inda,-2.005).
active(false,noinda,-1.428).  :-active(false,noinda,-1.429).
:-active(false,noinda,-0.529).:-active(false,noinda,-0.937).
:-active(false,noinda,-0.93). :-active(false,noinda,-1.538).
:-active(false,noinda,-1.592).
```

File lumo8.pl

```
active(true,noinda,-1.254).   active(true,noinda,-1.801).
active(true,noinda,-2.155).   active(true,noinda,-2.234).
active(true,noinda,-1.766).   active(false,noinda,-1.102).
active(true,noinda,-1.591).   active(true,noinda,-1.689).
:-active(true,noinda,-0.872). :-active(false,noinda,-1.144).
:-active(false,noinda,-0.982).:-active(false,noinda,-1.289).
:-active(false,noinda,-1.24). :-active(false,noinda,-1.266).
:-active(false,noinda,-1.056).:-active(false,noinda,-1.562).
:-active(false,noinda,-0.995).:-active(false,noinda,-1.737).
:-active(true,noinda,-1.616).
```

File lumo9.pl

```
active(true,noinda,-1.591).   active(true,noinda,-1.6).
active(true,noinda,-2.808).   active(false,noinda,-1.932).
active(true,inda,-2.14).      active(true,noinda,-2.871).
active(false,noinda,-2.142).  active(true,noinda,-1.685).
active(true,noinda,-1.151).   active(true,inda,-2.164).
active(false,noinda,-2.562).  active(true,noinda,-1.34).
active(true,noinda,-2.31).    active(false,noinda,-1.304).
:-active(false,noinda,-1.598).:-active(false,noinda,-1.605).
:-active(false,noinda,-1.596).:-active(false,noinda,-1.019).
:-active(false,noinda,-1.112).
```

File lumo10.pl

```
active(true,noinda,-1.616).   active(false,noinda,-2.468).
active(true,noinda,-1.208).   active(true,noinda,-2.158).
active(true,noinda,-1.623).   active(true,noinda,-1.176).
active(false,noinda,-1.937).  active(true,noinda,-2.032).
active(true,noinda,-1.487).   active(true,noinda,-1.8).
active(false,noinda,-2.687).  :-active(false,noinda,-1.028).
:-active(false,noinda,-1.157).:-active(false,noinda,-1.288).
:-active(false,noinda,-1.177).:-active(false,noinda,-2.14).
:-active(false,noinda,-0.854).
```

InGen output for the last test (when the examples in lumo10.pl are used for testing) is as follows:

```
Resulting Hypotheses:
active(X0, noinda, N43) :- real(N43), between(N43, -1.8845, -1.851).
active(X0, noinda, N21) :- real(N21), between(N21, -2.0839999, -2.062).
active(X0, noinda, N19) :- real(N19), between(N19, -2.206, -2.13).
active(true, X0, N19) :- real(N19), between(N19, -2.206, -2.13).
active(X0, X1, N19) :- real(N19), between(N19, -2.206, -2.13).
active(X0, noinda, N38) :- real(N38), between(N38, -1.223, -1.198).
active(true, X0, N20) :- real(N20), between(N20, -2.123, -2.103).
active(X0, noinda, N44) :- real(N44), between(N44, -1.761, -1.7395).
active(X0, noinda, N13) :- real(N13), between(N13, -2.348, -2.318).
active(X0, noinda, N17) :- real(N17), between(N17, -2.252, -2.211).
active(X0, noinda, N62) :- real(N62), between(N62, -1.171, -1.1495).
active(X0, noinda, N25) :- real(N25), between(N25, -1.942, -1.919).
active(X0, noinda, N55) :- real(N55), between(N55, -1.49, -1.4825001).
active(false, noinda, N37) :- real(N37), between(N37, -1.314, -1.294).
active(true, noinda, N35) :- real(N35), between(N35, -1.35, -1.33).
active(false, noinda, N10) :- real(N10), between(N10, -2.572, -2.552).
active(true, noinda, N62) :- real(N62), between(N62, -1.171, -1.1495).
active(true, inda, N19) :- real(N19), between(N19, -2.206, -2.13).
active(false, noinda, N25) :- real(N25), between(N25, -1.942, -1.919).
```

```
active(true, noinda, N5) :- real(N5), between(N5, -2.818, -2.798).
active(true, noinda, N46) :- real(N46), between(N46, -1.69, -1.688).
active(true, noinda, N51) :- real(N51), between(N51, -1.5915, -1.5905).
active(false, noinda, N64) :- real(N64), between(N64, -1.107, -1.092).
active(false, noinda, N59) :- real(N59), between(N59, -1.4284999, -1.418).
active(true, inda, N23) :- real(N23), between(N23, -2.015, -1.9950001).
active(false, noinda, N55) :- real(N55), between(N55, -1.49, -1.4825001).
active(true, noinda, N24) :- real(N24), between(N24, -1.984, -1.964).
active(true, noinda, N57) :- real(N57), between(N57, -1.4495, -1.4454999).
active(true, noinda, N13) :- real(N13), between(N13, -2.348, -2.318).
active(true, noinda, N53) :- real(N53), between(N53, -1.58, -1.566).
active(true, noinda, N42) :- real(N42), between(N42, -1.9514999, -1.941).
active(true, noinda, N8) :- real(N8), between(N8, -2.69, -2.67).
active(true, noinda, N58) :- real(N58), between(N58, -1.44, -1.4330001).
active(false, noinda, N11) :- real(N11), between(N11, -2.518, -2.498).
active(true, noinda, N3) :- real(N3), between(N3, -2.881, -2.86).
active(false, noinda, N44) :- real(N44), between(N44, -1.761, -1.7395).
active(false, noinda, N43) :- real(N43), between(N43, -1.8845, -1.851).
active(true, noinda, N41) :- real(N41), between(N41, -1.993, -1.9810001).
active(true, noinda, N56) :- real(N56), between(N56, -1.469, -1.46).
active(true, noinda, N33) :- real(N33), between(N33, -1.408, -1.388).
active(true, noinda, N49) :- real(N49), between(N49, -1.613, -1.606).
active(true, noinda, N60) :- real(N60), between(N60, -1.3615, -1.351).
active(true, noinda, N65) :- real(N65), between(N65, -1.733, -1.718).
active(true, noinda, N61) :- real(N61), between(N61, -1.286, -1.271).
active(true, noinda, N9) :- real(N9), between(N9, -2.624, -2.604).
active(false, noinda, N12) :- real(N12), between(N12, -2.444, -2.424).
active(true, noinda, N55) :- real(N55), between(N55, -1.49, -1.4825001).
active(true, noinda, N40) :- real(N40), between(N40, -2.105, -2.0925).
active(true, noinda, N25) :- real(N25), between(N25, -1.942, -1.919).
active(true, noinda, N45) :- real(N45), between(N45, -1.708, -1.6945).
active(true, noinda, N1) :- real(N1), between(N1, -3.416, -3.396).
active(true, noinda, N20) :- real(N20), between(N20, -2.123, -2.103).
```

```
active(true, noinda, N17) :- real(N17), between(N17, -2.252, -2.211).
active(false, noinda, N39) :- real(N39), between(N39, -1.188, -1.168).
active(true, noinda, N2) :- real(N2), between(N2, -3.182, -3.162).
active(true, noinda, N21) :- real(N21), between(N21, -2.0839999, -2.062).
active(true, noinda, N29) :- real(N29), between(N29, -1.719, -1.699).
active(true, noinda, N31) :- real(N31), between(N31, -1.532, -1.512).
active(true, noinda, N14) :- real(N14), between(N14, -2.32, -2.296).
active(false, noinda, N62) :- real(N62), between(N62, -1.171, -1.1495).
active(true, noinda, N63) :- real(N63), between(N63, -1.1465, -1.1445).
active(true, noinda, N22) :- real(N22), between(N22, -2.065, -2.045).
active(true, noinda, N28) :- real(N28), between(N28, -1.776, -1.7540001).
active(true, noinda, N48) :- real(N48), between(N48, -1.637, -1.618).
active(true, noinda, N47) :- real(N47), between(N47, -1.686, -1.666).
active(true, noinda, N50) :- real(N50), between(N50, -1.6035, -1.597).
active(false, noinda, N30) :- real(N30), between(N30, -1.675, -1.655).
active(true, noinda, N67) :- real(N67), between(N67, -1.2655001, -1.243).
active(true, noinda, N7) :- real(N7), between(N7, -2.728, -2.7).
active(true, noinda, N36) :- real(N36), between(N36, -1.339, -1.319).
active(false, noinda, N17) :- real(N17), between(N17, -2.252, -2.211).
active(true, noinda, N27) :- real(N27), between(N27, -1.814, -1.791).
active(true, noinda, N4) :- real(N4), between(N4, -2.847, -2.827).
active(true, noinda, N15) :- real(N15), between(N15, -2.304, -2.274).
active(false, noinda, N13) :- real(N13), between(N13, -2.348, -2.318).
active(true, noinda, N44) :- real(N44), between(N44, -1.761, -1.7395).
active(true, noinda, N52) :- real(N52), between(N52, -1.5875001, -1.575).
active(true, inda, N26) :- real(N26), between(N26, -1.928, -1.908).
active(true, noinda, N34) :- real(N34), between(N34, -1.397, -1.377).
active(true, noinda, N38) :- real(N38), between(N38, -1.223, -1.198).
active(false, noinda, N38) :- real(N38), between(N38, -1.223, -1.198).
active(true, noinda, N18) :- real(N18), between(N18, -2.219, -2.1990001).
active(false, noinda, N32) :- real(N32), between(N32, -1.421, -1.401).
active(true, noinda, N16) :- real(N16), between(N16, -2.273, -2.253).
active(false, noinda, N19) :- real(N19), between(N19, -2.206, -2.13).
```

```
active(true, noinda, N6) :- real(N6), between(N6, -2.8009999, -2.781).
active(true, noinda, N54) :- real(N54), between(N54, -1.537, -1.526).
active(false, noinda, N0) :- real(N0), between(N0, -3.7779999, -3.758).
active(true, noinda, N19) :- real(N19), between(N19, -2.206, -2.13).
active(true, noinda, N66) :- real(N66), between(N66, -1.56, -1.542).
```