Imperial College London
Department of Computing

# Efficient, Complete and Declarative Search in Inductive Logic Programming

Dianhuan LIN

*Supervisor:* Prof. Stephen MUGGLETON

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing Science of Imperial College London

September 2009

# Acknowledgements

First, I would like to thank my supervisor Prof. Stephen Muggleton, who guides me in the study of ILP (Inductive Logic Programming). Looking back each meeting, none of them finished within one hour; skimming through the notes I have made during and after the meeting, it is amazing to see myself gradually pick up a lot. I know this would be impossible without my supervisor, so I am really grateful for his patience and his time for the meeting. I can't say too much "Thank you" to him!

I am also grateful to Jose Santos for his initial work in TDHD, on which my project extends. I also appreciate the discussions with Jose about the implementation in TopLog.

I also would like to thank Dr. Krysia Broda for supervising my group project, from which I learned Automatic Reasoning and gained the practical skills in writing theorem prover in Prolog.

I also would like to thank the following people who I have asked questions about their expertise:
Prof. Murray Shanahan, Tim Kimber, Prof. Inoue, Dr. Alireza Tamaddoni-Nezhad. Dr. Oliver Ray, Dr. Fariba Sadri,

Finally, I would like thank my parents and my twin sister Dianmin for their unfailing support. This dissertation is dedicated to them!

**Abstract**

Techniques in Machine Learning have been applied to automate the process of scientific discovery. Inductive Logic Programming(ILP), as one of the machine learning techniques, has particular advantages in relational learning and has successful applications in various areas, such as system biology. Learning in ILP comes down to search, while huge search space is a problem. The huge search space problem becomes more significant in multi-clauses setting due to the dependency among clauses.

The method presented in this report addresses this problem by effectively bounding the search space using Top Directed Theory Derivation (TDTD), in which the logic program $\top$ theory is used as declarative bias that defines the search space. Different from the methods based on Inverse Entailment (IE), TDTD derives the hypothesized theory deductively. It is using clauses in $\top$ theory to replace the hypothesized theory itself in the refutation for individual examples that make this forward/deductive computation feasible. Benefitting from the completeness of deduction, TDTD no longer suffers from the incompleteness maybe encountered in IE-based methods, but delegates the completeness issue to $\top$ theory. Since it has been proved in this report that the given $\top$ theory is complete with respect to the hypothesis language, the method presented in this report is guaranteed to be complete.

TDTD also naturally accommodates non-OPL (Observation Predicate Learning) setting. Abduction and Induction are integrated into the same phase in TDTD, so that abductive facts and inductive rules can be learned at the same time. Thus learning problems, which are halted in cycle integration due to the inapplicability of both abductive and inductive procedure, can be solved in TDTD.

An new ILP system is implemented in this project for TDTD. Experimental evaluation on artificial data sets demonstrates its ability in learning multi-clauses problems correctly and efficiently.

# Contents

# Chapter 1

# Introduction

## 1.1 Multi-clauses Problems

"Observation Predicate Learning" (OPL)[17] is a usual Machine Learning setting, in which only hypotheses that define the same predicate with examples are learnable. However, this assumption strictly constraints the range of solvable problems. For example, in the grammar learning problem given in [17], only predicate $s$ is observable, while all the others like *noun* and *np* are not. Here comes the need of integrating abduction and induction. Specifically, if given a problem setting as fig 1.1[1], there is a learning cycle alternates between two separate processes, abduction and induction, to incrementally recover the incomplete knowledge.

**Background Knowledge**
C1: s(S1,S2) :- np(S1,S3), vp(S3,S4), np(S4,S2).
C2: vp(S1,S2) :- verb(S1,S2).

F1: det([a|S],S).
F2: det([the|S],S).
F3: noun([dog|S],S).
F4: noun([ball|S],S).
F5: verb([hits|S],S).
F6: verb([walks|S],S).
F7: prep([to|S],S).

**Positive Example E**
s([a, dog, hits, a, ball],[]).
s([the,man,walks,a, dog],[]).
....

**Target Hypothesis:**
H1: np(S1,S2) :- det(S1,S3), noun(S3,S2).

Figure 1.1: Problem setting for Grammar Learning Problem

However, problems arise if the clause C1 is also missing, then the abductive procedure becomes inapplicable due to the absence of inductive rules. This is a multi-clauses problem involving two inductive rules C1 and H1 missing. This multi-clauses problem can be made more difficult by having the facts $det([a|S], S)$ left-out as well. Then the real challenge lies in learning both abductive facts and inductive rules to explain one individual example. In this case, cycle integration will be halted due to the inapplicability of both abductive and inductive procedures. The only system that can solve this problem should be the one both supporting multi-clauses and integrating abduction and

[1]Different from that given in [17], there is no clause like np(S1,S2):- det(S1,S3), ajd(S3,S4), noun(S4,S2) in the background knowledge. This is to avoid the problem to be solved by the cycle integration which is explained in section 2.5.2

induction into the same phase.

Although the above example is a simple toy example, the similar multi-clauses problems are natural in automatic scientific discovery. For example, in modeling the inhibitory effects of toxins in metabolic networks[26][27], as shown in fig 1.2. The up and down arrows in the figure denote the change of concentration in metabolite which is the only observable. The target hypothesis is about which metabolic reactions are adversely affected in the presence of the toxin. Since there will be multiple such reactions unknown in one pathway, system capable of suggesting hypothesis of multi-clauses is necessary for this task.

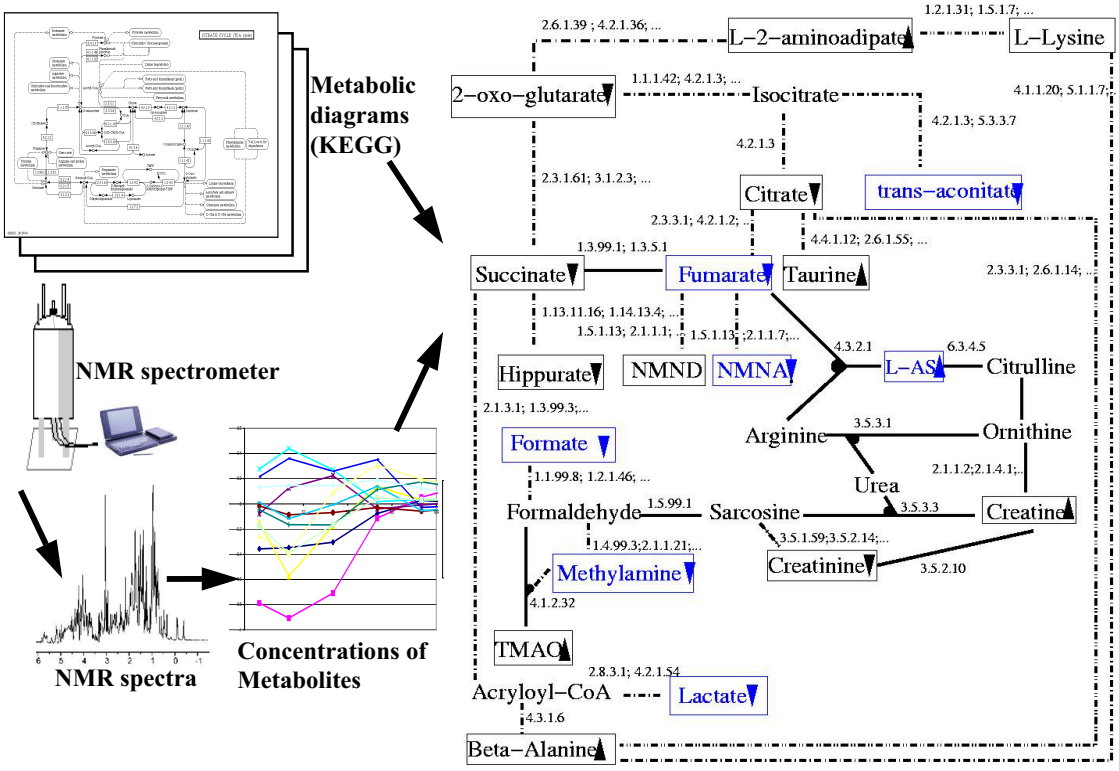


Figure 1.2: A metabolic subnetwork involving metabolites affected by hydrazine[26]

The challenge in multi-clauses learning is its much larger search space results from the cross product operation which accounts for the dependency among clauses. The multi-clauses problem will become more difficult if it is the one mixed with both abductive facts and inductive rules, since it requires a system learning abductive facts and inductive rules at the same time.

## 1.2 Overview of TDTD

Learning in Inductive Logic Programming (ILP) is essentially a search problem[18][14]. However, due to the expressive power of First Order Logic (FOL), the hypothesis space is usually huge, especially in the multi-clauses setting. Therefore, there are primarily two parts in ILP systems to make search tractable. One is bounding the search space, which is role of The Top Directed Theory Derivation (TDTD) presented in this report; the other is controlling the search, which is done by greedy search algorithm.

As an example-driven method, TDTD bounds the search space effectively without trade-off for completeness. In TDTD, $\top$ theory,[19] which is a logic program, is used as declarative bias that defines the hypothesis space. With $\top$ theory, the hypothesized theory can be derived deductively, which is just the opposite to the IE-based methods. Thus benefitting from the completeness of deduction, TDTD no longer suffers from the incompleteness encountered in IE-based methods, but delegates the completeness issue to $\top$ theory. Since it has been proved in this report that the given $\top$ theory is complete with respect to the hypothesis language, the method presented in this report is guaranteed to be complete. The redundancy in upward theory refinement is also analyzed in this

report, thus confirm advantages of TDTD compared to IE-based method.

Learning both abductive and inductive hypotheses at the same time using TDTD will encounter subgoals that succeed with non-ground substitution. While it requires all ground term in IE-based methods, this is not a problem in TDTD thanks to its top-directed search. Therefore TDTD not only supports multi-clauses learning, but also naturally integrates abduction and induction into the same phase.

The following goes through the previous grammar example, in which both C1, H1, and F1 are missing from background knowledge to give an general idea about the whole algorithm. The aim of the first two steps is to bound the search space by exclusively deriving the theories that entail the examples together with background knowledge. The 3rd step is doing greedy search to construct the final theory.

1. First, provide the declarative bias about hypothesis language before learning. In TDTD, this declarative bias is represented by $\top$ theory.
Suppose the alphabet for this example include these predicates: $[s, np, vp, det, adj, noun, verb]$ Then $\top$ theory for this example is in fig 1.3. [2].

```
T10: s(X,Y):-   $body(X,Y).
T20: np(X,Y):- $body(X,Y).
...

T11: $body(X,FinalOut):- np(X,Y), $body(Y,FinalOut).
T13: $body(X,FinalOut):- det(X,Y), $body(Y,FinalOut).
T14: $body(X,FinalOut):- adj(X,Y), $body(Y,FinalOut).
...
T17: $body(X,FinalOut):- prep(X,Y), $body(Y,FinalOut).
Tnt0: $body(FinalOut,FinalOut).

Ta1: det([X|S],S)
Ta2: adj([X|S],S).
...
Ta5: prep([X|S],S).
```

Figure 1.3: Top Theories for Grammar Example

2. Then input the $\top$ theory together with background knowledge B and examples E into the system to derive candidate hypothesized theories.

This involves the following 3 steps:
(1) Search for all possible refutations for the given example s([a, dog, hits, a, ball], []) with clauses in $\top$ theory and B.
(2) Extract out multiple sequences from the refutation sequences.
(3) Deriving candidate theories by SLD-resolution and subsumption according to the derivation sequence obtained in the last step.

Three of the derived theories are in fig 1.4

3. Amongst all the derived candidates, the one that is most compressive will be chosen by greedy search according to the heuristic of Minimum Description Length (MDL). Although all the t1, t2 and t3 above will explain the example s([a, dog, hits, a, ball], []) with background knowledge, but t1 covers other examples like s([the, man,walks, a, dog],[]), thus it will outperform t2 and t3 according to MDL.

The greedy search will continue until all examples are covered or no compression is achieved.

[2]This is just part of the $\top$ theory, the complete one can be found in appendix B

```
     C1: s(S1,S2) :- np(S1,S3), vp(S3,S4), np(S4,S2).
t1   C2: np(S1,S2) :- det(S1,S3), noun(S3,S2).
     C3: det([a|S],S).

     C1: s(S1,S2) :- np(S1,S3), vp(S3,S4), np(S4,S2).
t2   C2: np(S1,S2) :- adj(S1,S3), noun(S3,S2).
     C3: adj([a|S],S).

     C1: s(S1,S2) :- np(S1,S3), vp(S3,S4), np(S4,S2).
t3   C2: np(S1,S2) :- prep(S1,S3), noun(S3,S2).
     C3: prep([a|S],S).
```

Figure 1.4: Three of the derived theories for s([a, small, dog, hits, a, ball], [])

## 1.3 Report Map

Actually, the above example is a snapshot for the whole project. Each step has its corresponding chapter as follows:
1. At step one, you will find in chapter 3 about how to specify $\top$ theory for the declarative bias and ensure that the top theory specified can compose all the hypothesis language within the declarative bias.
2. At step two, if you are skeptical about whether the theory t derived in TDTD will explain the example that generate it, or doubt that all theories that entail the seed example can be derived in TDTD, go to chapter 4 to check the soundness and completeness proof for TDTD. You can also find all the details about TDTD algorithm in that chapter.
3. At step three, when making greedy choice, you may refer to chapter 5 to see what problems may be encountered at this step.
4. Chapter 6 gives you more details about implementation.
5. There are experimental results for this grammar learning example in Chapter 7.

## 1.4 Contribution

1. (a) Prove the completeness of $\top$ theory with respect to the hypothesis language;
   (b) Prove the soundness and completeness of TDTD, thus prove the completeness of the whole method.

2. Extension on the framework of Top Directed Hypothesis Derivation (TDHD) [19]
   (a) Accommodate multi-clauses setting
      i. Remove the constraint that clauses in B can not call hypothesized clauses and extraction algorithm;
      ii. Design an algorithm for extracting multiple derivation sequences and prove its correctness.
      iii. Analyze the redundancy in upward theory refinement under the multi-clauses setting, thus confirm the smaller search space in TDTD
   (b) Integrate abduction and induction into the same phase so that both abductive and inductive hypothesis can be learned at the same time
      i. Augment the SLD-derivation with subsumption for deriving abductive hypothesis in TDTD;
      ii. Allow subgoal to succeed with non-ground substitution

3. A new ILP system is implemented for the TDTD described in this report. Program transformation is used to improve the efficiency.

4. Experimental results show that multi-clauses learning problems can be solved in TDTD correctly and efficiently.

# Chapter 2

# Background

## 2.1 Machine Learning

[15]

**Definition** A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

### 2.1.1 Bias Learning

Learning without bias is not only a matter of intractable search space, but also turns out to be futile. Because the learning will simply memorize all the training data and unable to make inductive leap, thus it will fail to make any predication on unseen data. Two types of bias are usually used, one is declarative bias, the other is search bias.

#### Declarative Bias

Declarative Bias specifies the alphabet for hypothesis language, thus defines the search space. It can be explicitly specified by mode declaration as in Progol, or antecedent language in [5] . In this project, it is represented by $\top$ theory. It can also be implicit incorporated into the learning algorithms, such as considering only conjunctive hypotheses while ignoring disjunctive ones.

#### Search Bias

Search Bias means preferring one candidates hypothesis to another during the search. In greedy search, the search bias is a preference for small sets of maximally general Horn Clauses. However, it is only a heuristic approximation. Thus the result may not be optimal, that is, not the truly shortest set of maximally general Horn Clauses.

### 2.1.2 Cross Validation

As definition 2.1, machine learning improves with experience, therefore it is empirical. Thus its reliability needs to be validated. This is can be done by testing the learning results on unseen data, that is, measured by predictive accuracy. Predictive accuracy is calculated as the percentage of correct predictions on the test data.

If the data available is abundant, then it is feasible to reserve a small part of data for testing. However, if not, then cross validation is necessary.

Cross validation divides the data into N folds. It iterates on the N folds, at each iteration, one fold is preserved for test, while the rest folds are used for training. Predictive accuracy is calculated for each fold, and their results are averaged to give the final predictive accuracy.

Leave-one-out test strategy is a special case of cross validation, in which each example is assigned for one fold. Therefore, at each iteration, only one example is reserved for test, while all the rest are used for training.

## 2.2 Logic Programming

The following are referenced from [12] and [20],

### 2.2.1 Basic Concept and Notation

**Definition** A term is a constant, variable, or the application of a function symbol to the appropriate number of terms. A ground term is a term not containing variables.
An atom is the application of a predicate symbol to the appropriate number of terms.
A literal is an atom or the negation of an atom.

**Definition** A definite goal is a clause of the form

$$\leftarrow B_1, ..., B_n,$$

where $n > 0$ and each $B_i$ is an atom.
Each $B_i$ is called a subgoal of the goal.

**Definition** A definite clause is a clause of the form

$$A \leftarrow B_1, ..., B_n$$

which contains precisely one positive literal A.
A is called the head and $B_1, ..., B_n$ is called the body of the clause.
A Horn clause is either a definite clause or a definite goal.
A unit clause consists of a single literal.

**Definition** A logic program is a finite set of clauses representing their conjunction.

### 2.2.2 SLD-Resolution

**Definition Substitution**
A substitution $\theta$ is a finite set of the form $\{v_1/t_1, .., v_n/t_n\}$, where each $v_i$ is a variable, each $t_i$ is a term distinct from $v_i$ and the variables $v_1, .., v_n$ are distinct. Each element $v_i/t_i$ is called a binding for $v_i$. $\theta$ is called a ground substitution if the $t_i$ are all ground terms.

**Definition** An expression is either a term, a literal, or a conjunction or disjunction of literals. A simple expression is a term or a literal

**Definition Unification**
Let $\Sigma$ be a finite set of expressions. A substitution $\theta$ is called a unifier for $\Sigma$ if $\Sigma\theta$ is a singleton (a set containing exactly one element). A unifier $\theta$ for $\Sigma$ is called a most general unifier (mgu) for $\Sigma$ if, for each unifier $\theta$ of $\Sigma$ , there exists a substitution$\Gamma$ such that $\theta = \Sigma\Gamma$

Details about unification algorithm can be found in [12]

**Definition** Let G be $\leftarrow A_1, ..., A_m, ..., A_k and C be A \leftarrow B_1, ..., Bq$. Then G' is derived from G and C using mgu $\theta$ if the following conditions hold:
(1) $A_m$ is an atom, called the selected atom, in G.
(2) $\theta$ is an mgu of $A_m$ and A. (3) G' is the goal $\leftarrow (A_1, ..., A_{m-1}, B1, ..., B_q, A_{m+1}, ..., A_k)\theta$ G' is called a resolvent of G and C.

SLD-resolution stands for SL-resolution for Definite clauses. SL stands for Linear resolution with Selected function.

**Definition SLD-derivation**
Let $\Sigma$ be a set of clauses and C a clause. A derivation of C from $\Sigma$ is a finite sequence of clauses $R_1, .., R_k = C$, such that each $R_i$ is either in $\Sigma$ , or a resolvent of two clauses in $\{R_1, ..., R_{i-1}\}$. If such a derivation exists, $\Sigma \vdash_r C$. Thus C can be derived from $\Sigma$.

SLD-refutation is a special case of SLD-derivation which derives empty clause $\square$.

## 2.3 Inductive Logic Programming

Inductive Logic Programming lies in the intersection of machine learning and logic programming. As an machine learning technique, it has particular advantage in relational or structure learning. Its representation scheme as logic program makes its learning results easy to understand. [18]

The general problem setting for ILP is as follows [18] . Given is background knowledge B and examples E which comprises positive $E^+$ and negative $E^-$. The aim of ILP systems is to find the hypothesis H that the following conditions hold.

$PriorSatisfiability.$ $B \wedge E^- \not\models \square$
$PosteriorSatisfiability.$ $B \wedge H \wedge E^- \not\models \square$
$PriorNecessity.$ $B \not\models E^+$
$PosteriorSufficiency.$ $B \wedge H \models E^+$

Prior Satisfiability requires the consistency in the given B and E; while Prior Necessity ensure positive examples can be explained by background knowledge, otherwise, there is no need for learning. The two posterior condition requires the learned hypothesis to cover all positive examples while cover no negative examples.

Learning in ILP is essentially a search problem [18]. The aim of ILP systems is to search through hypothesis space to find the one holds for the two posterior conditions, while the huge search space is a problem. The methods based on Inverse Entailment (IE) [16] is one way to bound the search space.

### 2.3.1 Inverse Entailment

IE[16] treat induction as inverse of deduction, and compute hypothesis inversely as follows:

$$B \wedge \neg E \models \neg\bot \models \neg HH \models \bot \tag{2.1}$$

In Inverse Entailment, the search space is bounded by one most specific theory, denote it by $\bot$.The IE-operator which computing this $\bot$ varies amongst different IE-based ILP systems, but they all play the role of bounding the search space. Incompleteness is a potential problem in the IE-operator. The ILP systems based on IE are reviewed in later part.

After $\bot$ is computed, the search space is bound by considering only the theories or clauses that are more general than $\bot$. This is done by refinement operator in the following subsection.

### 2.3.2 Refinement Operator

**Clause refinement – Subsumption Lattice**

**Definition $\theta$-subsumption**
clause C $\theta$-subsumes clause D if there exists a substitution $\theta$ such that $D \sqsupseteq C\theta$

Refinement operator was introduced in [24] for search a lattice of clauses ordered by $\theta$-subsumption. The clause refinement was also defined by relative subsumption [21] and generalized subsumption [4] .

**Theory Refinement**

As clause refinement can either be bottom-up or top-down, theory refinement can also be done from either direction. Take the upward theory refinement [20] as example.

There are vary version of upward refinement operator defined for theory, but there are 3 main possible operations. The combination of these 3 will lead to a refinement operator based on entailment:
1. Inverse resolution between two clauses in the theory
2. Subsumption for each clause in the theory: $\theta - subsumption$, relative subsumption, generalized

subsumption
3. Clause addition

At each refinement step, there is also cross product which greatly expand the search space.

**Ordered Search Space**

The search space generated by refinement operator is ordered either from general-to-specific or specific-to-general[15]. The advantage of such ordering is the applicable of efficient pruning[16]. When one candidate hypothesis $H_0$ is evaluated to have no compression, then those more specific than $H_0$ are certain not to get higher compression. Because they won't get higher coverage of examples since they are more specific than $H_0$. Thus the whole subspace of hypotheses that are more specific than $H_0$ can be pruned,

## 2.4 Multi-Clauses Learning

### 2.4.1 Multi-Clauses Learning Problems

**Definition Multi-clauses learning problems** are those in whose refutation sequence for individual example, more than one are missing and need to be learned.

One thing has to be noticed is that "multi" does not refer to the number of clauses in the final learned theory, but the number of missing clauses in the refutation sequence for individual example. There are two situations that these two are different.
(1) Single clause in the final learned theory but used more than once in the refutation. This is common in problems involving recursion. Such as the odd-even example, the target hypothesis is actually only one clause, but it is used more than once in refutation for examples except odd(s(0)).
(2) Multiple clauses in the final theory but only one of them is used in each refutation. This happens because the relation among these clauses are disjunctive, and not every clause in the final learned theory is used in the refutation for individual examples. Such as in the grammar example which will be discussed in detail later.

### 2.4.2 Limitation for Single-Clause ILP systems

Single clause ILP systems like Progol[16] and TopLog[19] are unable to solve these multi-clauses learning problems.

In Progol, this is due to its algorithm for building bottom clause. Bottom clause is conjunction of ground literals, each of them is instantiated by calling themselves as a goal, that is, proved from clauses in B. Therefore, in cases when another hypothesized clause or even more are needed to succeed the calling goal, it will fail due to the non-exitence of hypothesized clauses during learning.

In Toplog, it is the constraint that clauses in B cannot call clauses in $\top$ that prevent it from solving multi-clauseslearning problems. .

The ability of Progol and TopLog to learn recursive concept may be confusing. Recursive hypothesis are composed of at least two clauses, one for base, another for recursion. Also, both of these two clauses are used in the refutation, so isn't it multi-clauses learning problem? Yes, it is. Then how can it be solved in single-clause systems? Actually, it is done by the trick of reducing multi-clauses problems into single clause problems. Base clause is learned first from examples about base case, that is, do not need recursive clause in refutation. After this newly learned base clause is added to the background knowledge, the original multi-clauses problem is reduced to a single-clause problem.

However, this reduce method does not work all the time. In cases when examples about base case are not provided, learning will be halted since it is not until the base clause is learned that the recursive clause can be recovered. More importantly, not all multi-clauses problem can be reduced to single-clause problem in this way, therefore it is crucial to have a generic algorithm capable to solve multi-clauses learning problems.

### 2.4.3 Multi-Clauses ILP systems

#### CF-induction

CF-induction[9] is sound and theoretically complete for finding hypotheses. It is based on Inverse Entailment and support full clausal logic.

$$B \wedge \neg E \models CC(B, E) \tag{2.2}$$
$$H \models \neg CC(B, E) \tag{2.3}$$
$$\tag{2.4}$$

Similar to bottom-clause in Progol, there is bridge-formula $CC(B, E)$ in CF-induction to play the role of bounding the search space. While in Progol, only ground unit consequences are considered in building bottom clause, thus the hypotheses considered are limited to single clause.

In case of pure abduction, the bridge-formula $CC(B, E)$ is directly derived from the whole set of $\overline{N}ewCarc(B, \neg E, P)$. $NewCarc(B, \neg E, P)$ denote the new characteristic clauses which are newly derived when the seed example (new information) is added to the background knowledge B, according to the production field specified by P. The search space of abduction is much smaller than that of induction. Since abductive hypothesis are unit-clause, no generalization is needed, thus SOLAR[7] alone is enough for abduction. SOLAR is one component called within CF-induction system for consequence finding. It is a theorem prover based on SOL-resolution[8] .

$$Carc(B \wedge \neg E) \models \neg CC(B, E) \tag{2.5}$$

For induction, the bridge-formula $CC(B, E)$ can be derived from characteristic clauses $Carc(B \wedge \neg E)$, which is consequences of $B \wedge \neg E$. Unlike the bridge-formula in abduction which is the whole set of $\overline{N}ewCarc(B, \neg E, P)$, clauses in $CC(B, E)$ are only ground instances of a subset of $Carc(B \wedge \neg E)$, thus further selection which not only select a subset of clauses, but also choose the instances are needed after computing $Carc(B \wedge \neg E)$ using SOL-resolution. Unfortunately, this selection process needs user-interaction. This problem has recently been studied in [29]. However, full automation of this selection process is not achieved yet.

In addition, in order to avoid excluding correct hypotheses, the selection process have to take case of too strong bias. For example, in the odd-even example given in [28] [9], if $CC(B, E)$ is chosen as:
$even(0) \wedge (\neg odd(s(0)) \vee even(s(s(0)))) \wedge (\neg odd(s(s(s(0)))))$
the learned hypothesis is H1 = $odd(s(X)) \leftarrow even(X)$. However, another interesting hypothesis H2 will be ignored if only consider the above selection.

$$H2 \quad = \quad \begin{cases} odd(s(X)) \leftarrow even(X) \\ odd(X) \leftarrow even(s(X)) \end{cases}$$

H2 can be obtained by making a different selection as follows: $even(0) \wedge (\neg odd(s(0)) \vee even(s(s(0)))) \wedge (\neg odd(s(s(s(0))))) \vee even(s(s(s(s(0)))))) \wedge (\neg odd(s(s(s(0)))))$
Note that there are two instances of the clause $odd(X) \vee even(s(X))$ in the above selection

After negation, the bottom formula becomes:
$\neg even(0) \vee odd(s(0)) \vee odd(s(s(s(0))))$
$\neg even(0) \vee \neg even(s(s(0))) \vee odd(s(s(s(0))))$
$\neg even(0) \vee odd(s(0)) \vee \neg even(s(s(s(s(s(0)))))) \vee odd(s(s(s(0))))$
$\neg even(0) \vee \neg even(s(s(0))) \vee \neg even(s(s(s(s(s(0)))))) \vee odd(s(s(s(0))))$

Then the first two clauses above will be generalized to the 1st clause in H2 as before, while the 3rd and 4th clauses above will be generalized to the 2nd clause in H2 . H2 is a correct theory which expresses the relation of predecessor.

The implementation of CF-induction also have difficulty in the generalization step. The search space results from upward refinement operator is so huge that it is usually not completely implemented, such as, clause addition is not considered. The undecidability of inverse resolution also

add complexity. Although it is proposed in [30] to replace the complex generalization with single deductive operator, it has nothing to do with the huge size of search space, which is more problematic. Actually, there are redundancy in the search space results from upward refinement operator, which is analyzed in the subsection 2.4.3

**Imparo and HAIL**

Imparo[25] and HAIL[23] [22] are IE-based ILP systems. Imparo is the successor to HAIL.

The most specific clauses that bound the search space are called kernel set in HAIL. Abductive Procedure is applied first to compute a set of ground atoms $\Delta = \alpha_1 ... \alpha_n$, which explain the seed example and form the heads of clauses in the kernel set. Then saturate procedure is applied to each member of $\Delta$ to obtain the kernel set.

$$\mathrm{K} = \begin{cases} \alpha_1 \leftarrow \delta_1^1, ..., \delta_{m_1}^1 \\ . \\ . \\ . \\ \alpha_1 \leftarrow \delta_1^n, ..., \delta_{m_n}^n \end{cases}$$

Figure 2.1: Kernel Set in HAIL

It is the incorporation of abudction that make the kernel set not limit to single clause as that in Progol. However, the saturate procedure in HAIL is still purely deductive which is the same as that in Progol. Thus HAIL will fail on the problems that need another or more hypothesized clauses to explain the body atoms. Such as the example 1 given in [iof]

Imparo extends on HAIL via a recursive inductive procedure. Induction on Failure (IoF) is the proof procedure used in Imparo. The concept of secondary example is defined for the body atoms that fail in the pure deductive procedure, that is, not direct consequences of the background knowledge. For these secondary examples, IoF will recursively start a new inductive procedure which has abduction followed by saturation. The recursive procedure continues until the base case is reached, that is, the one can be explained by background knowledge alone, which is the same as the pure deductive procedure in HAIL and Progol. Intuitively, for the atoms that can not be directed explained by background knowledge, at least one more hypothesized clause is needed. These recursively called clauses form the most specific connected theory denoted by $T_\perp$ that bound the search space.

Although Imparo improves on HAIL with enlarged range of solvable problems, it may encounter problems when learning abductive facts and inductive rules at the same time. For example, learning the recursive concept of path-edge:

path(X,Y):- edge(X,Y).
path(X,Y):- edge(X,Z), path(Z,Y).

Suppose background knowledge B= $\{edge(c,d)\}$,
M=$\{modeh(*, path(+Node, -Node)), modeh(*, edge(+Node, -Node)),$
$modeb(*, edge(+Node, -Node)), modeb(*, path(+Node, -Node)\}$, and the target hypothesis includes both abdutive facts about *edge* and the inductive rules as above.

Given example path(a,b), $\Delta = \{path(a,b)\}$ after applying the abductive procedure. Then applying the saturate procedure, suppose trying modeb(*, edge(+Node,-Node)) first, then the input variable of predicate *edge* will be instantiated with 'a'. Since there is no edge in background knowledge start with node 'a' like edge(a,c), the atom edge(a,Y) will fail with the deductive procedure, thus IoF will recursively call edge(a,Y) as secondary example and start a new hypothesized clause for it. Since the subgoal (secondary example) edge(a,Y) is non-ground, the hypothesized unit clause edge(a,Y) is non-ground either, while ground term is required in this IE-based method. Similar unground output variable will be encountered if trying modeb(*, path(+Node,-Node) first.

However, theories as following are candidate explanations for the example path(a,b).

$t_1$

edge(a,c).
edge(d,b).
path(X,Y):- edge(X,Y).
path(X,Y):- edge(X,Z), path(Z,Y).

$t_2$

edge(a,x).
edge(x,b).
path(X,Y):- edge(X,Y).
path(X,Y):- edge(X,Z), path(Z,Y).

Note that the x in $t_2$ is a skolemised term which invent a missing node [10]

Since it is the ground terms in the abductive facts that substitute the logical variable in the subgoal, the absence of abductive facts lead to subgoal to succeed with non-ground substitution. Therefore if it is a multi-clauses problem with pure inductive rules, Imparo will be applicable; or if the target hypothesis is purely abductive, the abductive procedure in Imparo will handle the binding of variables. While if learning both at the same time, it has to be a problem with only unary predicates or a proposition case as example 2 in [25], otherwise it will encounter the problem of non-ground output variable.

Similar to CF-induction, the generalization in Imparo and HAIL is incompletely implemented, which consider only subsumption. Also, as an IE-based method, Imparo and HAIL suffer from the redundancy problem in Upward Theory Refinement, which is analyzed in the next subsection 2.4.3

**Redundant in Upward Theory Refinement**

**Example** $B = \begin{cases} m(1). \\ b(1). \\ b(2). \end{cases}$ $E = \begin{cases} e(1). \\ e(2). \end{cases}$

$$t_1 = \begin{cases} C_1 : \mathrm{e}(X) \leftarrow \mathrm{m}(X), \mathrm{n}(X). \\ C_2 : \mathrm{n}(X) \leftarrow \mathrm{u}(X), \mathrm{v}(X). \\ C_3 : \mathrm{u}(X) \leftarrow \mathrm{b}(X). \\ C_4 : \mathrm{v}(X) \leftarrow \mathrm{b}(X). \end{cases} \qquad t'_1 = \begin{cases} C'_1 : \mathrm{e}(X) \leftarrow \mathrm{m}(X). \\ C_2 : \mathrm{n}(X) \leftarrow \mathrm{u}(X), \mathrm{v}(X). \\ C_3 : \mathrm{u}(X) \leftarrow \mathrm{b}(X). \\ C_4 : \mathrm{v}(X) \leftarrow \mathrm{b}(X). \end{cases}$$

$t_2 = \mathrm{e}(X) \leftarrow \mathrm{m}(X).$

The theory $t_1$ on the left is one of the candidate theories in the search space. When the first clause $C_1$ is refined to $C'_1$ according to $\theta$-subsumption, the other 3 clauses $C_2$, $C_3$ and $C_4$ remained in the theory $t'_1$ become unnecessary. Because the clause $C'_1$ alone is enough to explain the example e(1) with background knowledge. In this case, $t'_1$ is essentially redundant with t2, since they will cover the same number of examples while $t'_1$ has more numbers of literals.

In addition, clause deletion which specializes the theory is not considered in the upward theory refinement, thus $t'_1$ will not have $C_2$, $C_3$, $C_4$ deleted to become t2. On the contrary, the upward theory refinement will keep on refining on $C_2$, $C_3$, $C_4$ to derive theories like the following $t"_1$ which are also redundant with t2. The cross product operation will further expand this redundancy.

$$t"_1 = \begin{cases} C"_1 : \mathrm{e}(X) \leftarrow \mathrm{m}(X). \\ C"_2 : \mathrm{n}(X) \leftarrow \mathrm{u}(X). \\ C_3 : \mathrm{u}(X) \leftarrow \mathrm{b}(X). \\ C_4 : \mathrm{v}(X) \leftarrow \mathrm{b}(X). \end{cases}$$

Even if $C2$ is in the background knowledge, that is, B becomes B', and one of the derived theories is $t_3$ . It appears that the refinement of $C_1$ i to $C'_1$ will not make the existence of C3 and

C4 unnecessary as previous example. However, since $C_1'$ is no longer related to the clause $C_2$ in the background knowledge, it also disconnects with C3 and C4, thus still redundant with $t_2$.

**Example** $B' = \begin{cases} m(1). \\ b(1). \\ b(2). \\ C_2 : \mathrm{n}(X) \leftarrow \mathrm{u}(X), \mathrm{v}(X). \end{cases}$ $E = \begin{cases} e(1). \\ e(2). \end{cases}$

$t_3 = \begin{cases} C_1 : \mathrm{e}(X) \leftarrow \mathrm{m}(X), \mathrm{n}(X). \\ C_3 : \mathrm{u}(X) \leftarrow \mathrm{b}(X). \\ C_4 : \mathrm{v}(X) \leftarrow \mathrm{b}(X). \end{cases}$ $t_3' = \begin{cases} C_1' : \mathrm{e}(X) \leftarrow \mathrm{m}(X). \\ C_3 : \mathrm{u}(X) \leftarrow \mathrm{b}(X). \\ C_4 : \mathrm{v}(X) \leftarrow \mathrm{b}(X). \end{cases}$

The redundancy discussed above will not be small due to the cross product operation which dramatically expand the search space.

#### Top-down ILP systems

Contrary to the above IE-based methods, the two ILP systems reviewed in this section both start with a overly general theory and do downward theory refinement. Another difference is that they are based on generate-and-test, rather than example-driven as IE-based methods.

HYPER[2] starts with a overly general theory which can have multiple clauses. However, the downward refinement operator in HYPER considers only subsumption, thus the numbers of clauses in the learning theory is fixed.

The numbers of clauses in the learning theory is not fixed in SPECTRE[1]. In addition, the theory refinement in SPECTRE is based on entailment, clause deletion is also considered. Starting with a overly general theory, SPECTRE keep on unfolding clauses in the initial theory until no negative examples is covered while all positive examples are covered. As a generate-and-test method, SPECTRE use information gain as heuristic, which is similar to FOIL, to decide which literal to unfold. The disadvantage of SPECTRE are:
(1) It needs to be provided a overly general theory to start with.
(2 ) It makes no key distinctions between $\top$ and background knowledge, which leads to problems in learning recursive programs in the early version of SPECTER.

#### Explanation-Based Generalisation (EBG)

TDTD share several similarities with EBG[11]:
(1) Search for refutation for the individual example during learning;
(2) Having terminal and non-terminal predicates (it is called operational and non-operational predicates in EBG)

However, the key difference is that EBG is essentially a deductive learning, which is essentially knowledge compilation, which aim at speeding up program performance; while TDTD is inductive learning.

## 2.5 Abduction, Induction and their integration

$B = hasFeather(a)$
$H = bird(X) \leftarrow hasFeather(X)$ E = bird(a)

Figure 2.2: Bird example

### 2.5.1 Deduction, Abduction and Induction

Deduction is based on the sound inference rule. In fig 2.2, E = bird(a) can be derived from B and H according to Modus Ponens or Resolution.

Unlike deduction that has sound inference rule, abduction and induction are empirical. Both of Abduction and Induction can be viewed as inverse of Deduction, while abductive hypothesis is about ground or existentially quantified facts which requires minimal answer, and inductive hypothesis is about general rules. For example, in fig 2.2, B=hasFeather(a) can be abduced from given example bird(a) with the general rule H. The general rule $H = bird(X) \leftarrow hasFeather(X)$ can be induced from the given example bird(a) with the ground fact B = hasFeather(a).

### 2.5.2 Integration of Abduction and Induction

[6]

#### Necessity for Integration

"Observation Predicate Learning" (OPL)[17] is a usual Machine Learning setting, in which only hypotheses that define the same predicate with examples are learnable. Therefore, learning hypothesis defining non-observable predicate requires the integration of abduction. In addition, theory completion[27] is a task of recovering the incomplete theory T after observing examples E, in which learning both abductive and inductive hypothesis is necessary.

#### Cycle Integration

$$B = \begin{cases} b1 : s(S1,S2) : -np(S1,S3), vp(S3,S4), np(S4,S2). \\ b2 : np(S1,S2) : -det(S1,S3), ajd(S3,S4), noun(S4,S2). \\ b3 : vp(S1,S2) : -verb(S1,S2). \\ ... \end{cases}$$

$$E = \begin{cases} e1 : s([the, small, dog, hits, a, big, ball], []). \\ e2 : s([a, dog, hits, a, ball], []). \end{cases}$$

$$H = \begin{cases} H1 : np(S1,S2) : -det(S1,S3), noun(S3,S2). \\ H2 : noun([dog|R], R). \end{cases}$$

In cycle Integration, learning alternates between abduction and induction, which forms a cycle, so that the incomplete theory can be recovered incrementally. Here is one example.

Suppose H is the target hypothesis which comprises both abductive fact and inductive rule. Learning this problem need applying abduction first for example e1. It is not until H2 is learned that H1 can be learned from given example e2. In addition, learning process will be halted if clause b2 is not in the background knowledge, since abductive procedure has no rule to carry out abduction.

# Chapter 3

# Top Theory as Declarative Bias

Declarative bias defines the hypothesis space. In TDTD, it is explicitly specified by $\top$theory, which is a logic program. The advantages of using $\top$ theory to represent the declarative bias are[19]:
(1) Having all the expressive power of a logic program
(2) Can be efficiently reasoned with using standard logic programming techniques.
(3) Make the declarative bias itself learnable[3] due to its form of logic program

This chapter gives details about $\top$ theory in TDTD. It compares $\top$ theory in TDTD to a context-free grammar, and explain how to derive each hypothesized clause with the grammar-like $\top$ theory. Also, it is proved that this grammar ($\top$ theory ) is complete with respect to the hypothesis language.

## 3.1 Grammar like $\top$ theory

There are terminal and non-terminal symbols in a context-free grammar, correspondingly, there are terminal and non-terminal literals in $\top$. Terminal literals are of which the hypothesized clause is composed; while the non-terminal literals will not appear in the hypothesized clauses, but they are used for control purposes within $\top$, such as, connecting terminal literals and binding variables.

Each head clause or body clause has one terminal literal and is responsible for the predicates appeared in the derived clause. If no function symbols, the head clauses and body clauses are enough for composing the hypothesized language; while other clauses with purely non-terminals are responsible for binding the variable with function term or pervious appear substitutions.

There are three requirements on the specification of $\top$ theory:

1. Non-terminal literals can not appear in clauses in B or E, this ensure the clear distinction between the $\top$ and B.

2. At least one non-terminal literals appear in each clause of $\top$, otherwise it can not resolve with other clauses in $\top$ to compose hypothesis language. One exception is those for abducible predicates, they are unit clause, so only one terminal predicate.

3. At most one terminal literal is allowed in each clause of $\top$, so that each clause is a basic unit for composing hypothesis language.

Note here is no constraint that any predicate appearing in the head of some clause in $\top$ must not occur in the body of any clause in B, which essentially avoid clauses in B to call clauses in $\top$. Actually, it is this constraint that prevent TopLog from learning multi-clauses problems.

### 3.1.1 Head Clause and Body Clause

These clauses with one terminal literal are directly relevant to its composing language, since it contains the terminal literals that will appear in the language. According to whether they are head and body of derived, they can be further distinguished as Head clause and Body clause.

**Definition Head Clause**
**Head Clauses in $\top$ are those having terminal literal as head.**

Head clause is similar to the modeh in mode declaration. It corresponds to the head of derived clauses. It is the only type of clauses in $\top$ that have terminal literal at head.

**Definition BodyClause**
**Body Clauses in $\top$ are those having terminal literal as one of its body literals.**

Body clause is similar to the modeb in mode declaration. The terminal literal in the body clause will become the body of derived clauses. Body clauses always have non-terminal literal as head, denote it by the predicate of $body.

The following figure 3.1 is an example given in [19].

$$\begin{array}{l}\text{modeh(mammal(+animal)).}\\ \text{modeb(has\_milk(+animal)).}\\ \text{modeb(has\_eggs(+animal)).}\end{array} \qquad \top \;=\; \begin{cases} \top_1 : \text{mammal}(X) \leftarrow \text{\$body}(X) \\ \top_2 : \text{\$body}(X) \leftarrow \text{has\_milk}(X), \text{\$body}(X) \\ \top_3 : \text{\$body}(X) \leftarrow \text{has\_eggs}(X), \text{\$body}(X) \\ \top_{nt0} : \text{\$body}(X) \leftarrow \end{cases}$$

Figure 3.1: Mode declarations and corresponding $\top$ theory

$\top_1$ is a head clause, while $\top_2$ and $\top_3$ are body clauses. There is no function symbols in this simple example and only unary predicates, thus this simple $\top$ theory is enough to derive clauses as following. Details about deriving clauses will be given in next section 3.2

$C_1 : \text{mammal}(X) \leftarrow \text{has\_milk}(X), \text{has\_eggs}(X).$
$C_2 : \text{mammal}(X) \leftarrow \text{has\_milk}(X).$
$C_3 : \text{mammal}(X) \leftarrow \text{has\_eggs}(X).$

### 3.1.2 Non-Terminal Literals

This section introduce each non-terminal literal in $\top$ according to their roles in composing language. Although these non-terminal literals will not appear in the final composed language, they are actually important for binding the variables in the derived clause.

#### Connecting Terminal Literals: $body

Let $\$body$ denote the non-terminal literal responsible for connecting terminal literals.

By resolving the pair of $\$body$ predicates, two terminal literals are connected into the same clause. Such as resolving $\top_1$ and $\top_2$ in fig 3.1 will result in concatenating two terminal literals into the same clause as follows
$\text{mammal}(X) \leftarrow \text{has\_milk}(X), \text{\$body}(X)$

Since it is responsible for connection, the information shared among terminal literals are propagated through this predicate, such as variable binding.In case of unary terminal predicates, it can simply propagate that single variable like $body(X). While in non-unary case, variables or constants appeared need to be collected so that input variable of later predicate can be bound to them. Therefore, the variables in the predicate $body are lists that store all the terms appeared, such as $\text{\$body}(InputSoFar, BodyVars)$. Here are two variables, $InputSoFar$ as input variable store those appear so far, while$BodyVars$ is output variable that will return all the terms appear in the body of clause.

There is one nullified clause composed by this predicate alone $\top_{nt0} : \text{\$body}(X)$. It has no effect for terminal literals in the derived clause, but it is still required for resolving away the remaining predicate $\$body$ in the deriving clause, so that the non-terminal predicate $body will not appear in the final derived clause.

**Binding with function term: $bind**

The $\top$ theory for binding the variable with function term is specified in fig 3.2. Lemmas 3.1.1 and 3.1.2 prove that this $\top$ theory is complete with respect to binding function term with any depth, respectively for ground and non-ground term. Let $s$ denotes arbitrary function symbol.

**Lemma 3.1.1** *The ground function terms with any depth can be derived by* $\top_{nt1}$ *and* $\top_{nt3}$.

**Proof** (1) Base case: $\top_{nt1}$ : $\$bind(0, 0)$.

(2) Assume ground term with depth of k can be derived, that is $\$bind(s^k(0),0)$ can be derived. Then ground term with depth of (k+1), that is $\$bind( s^{k+1}(0),0)$ can be derived by resolving $\$bind(s^k(0),0)$ and $\top_{nt3}$. ∎

**Lemma 3.1.2** *The non-grounded function terms with any depth can be derived by* $\top_{nt2}$ *and* $\top_{nt3}$*:*

**Proof** (1)Base case: $\top_{nt2}$ : $\$bind(X, X)$.
(2)Assume non-ground term with depth of k can be derived, that is $\$bind(s^k(X),X)$ can be derived. Then non-ground term with depth of (k+1), that is $\$bind( s^{k+1}(X),X)$ can be derived by resolving $\$bind(s^k(X),X)$ and $\top_{nt3}$. ∎

$\top_{nt1}$ : $\$bind(0, 0)$.
$\top_{nt2}$ : $\$bind(X, X)$.
$\top_{nt3}$ : $\$bind(X, Z) \leftarrow X{=}s(Y), \$bind(Y, Z)$.

Figure 3.2: $\top$ theory for binding variable with function term

$$\top \;=\; \begin{cases} \top_1 : \text{odd}(X) \leftarrow \$\text{bind}(X, Y), \$\text{body}(Y) \\ \top_2 : \$\text{body}(X) \leftarrow \text{even}(X), \$\text{bind}(X, Y), \$\text{body}(Y) \\ \top_{nt0} : \$\text{body}(X) \leftarrow \end{cases}$$

Figure 3.3: $\top$ theory for odd-even example

Here is one example using $\$bind$. Following the SLD-derivation sequence of $[\top_1, \top_{nt3}, \top_{nt2}, \top_2, \top_{nt2}, \top_{nt0}]$, the clause $\text{odd}(s(X)) \leftarrow \text{even}(X)$ will be derived.

**Binding for Input Variables: memberBind and inputUpdate**

In IE-based method, the input variables of each predicate in modeB are instantiated with previous appeared ground term which is of the same type, then the list of previous appeared is updated with its output variable after it succeeds with ground substitution. It is similar here since the terminal predicate in the body clause corresponding to the predicates in modeB. Therefore, before calling the terminal predicate in the body clause, it is unified with previous appeared term by logic program of *memberBind*; after it succeeds, its output variables are used to update the list of $InputSoFar$ by logic program of *inputUpdate*. Logic programs about *memberBind* and *inputUpdate* are specified in fig 3.4.

Here is one example 3.5 using *memberBind* and *inputUpdate*. In body clause $\top_3$, before calling the terminal predicate *edge*, its input variable is bound by *memberBind*. After the subgoal of predicate *edge* succeeds, *inputUpdate* will update the list of InputSoFar with the substitution in the output variable of *edge*. In head clause $\top_1$, the output variable of *path* is bound to one of the variables in the Body which is returned in the list of $BodyVars$. Then the variables in the clause are connected together.

Note that the difference here with IE-based method is that the substitution is not necessarily ground. *memberBind* and *inputUpdate* only bind the variables in the clause together, while no requirement of ground term. For example, when the facts about predicate *edge* are missing, it will be replaced in the refutation for example path(a,b) by non-ground clause $\top_{a1}$ to denote one abductive hypothesized clause. Then the subgoal of *edge* will succeed with non-ground substitution

$\top_{nt4}$ : memberBind$(X, [X|_])$.
$\top_{nt5}$ : memberBind$(X, [Y|List]) : -$
$X \neq Y$,
memberBind$(X, List)$.

$\top_{nt6}$ : inputUpdate$(X, [], [X]) : -!$.
$\top_{nt7}$ : inputUpdate$(X, [Y|InList], [X|InList]) : -$
$X == Y, !$.
$\top_{nt8}$ : inputUpdate$(X, [Y|InList], [Y|NewInList]) : -$
inputUpdate$(X, InList, NewInList)$.

Figure 3.4: $\top$ theory for binding variable with previous appeared term

$\top_1$ : path$(X, Y) : -$
$body([X], BodyVars)$,
$memberBind(Y, BodyVars)$
$\top_2$ : $body([InputSoFar, BodyVars]) : -$
$memberBind([X], InputSoFar)$,
$path(X, Y)$,
$inputUpdate(Y, InputSoFar, NInput)$,
$body([NInput, BodyVars])$.
$\top_3$ : $body([InputSoFar, BodyVars]) : -$
$memberBind([X], BodyVars)$,
$edge(X, Y)$,
$inputUpdate(Y, InputSoFar, NInput)$,
$body([NInput, BodyVars])$.
$\top_{nt0}$ : $body(BodyVars, BodyVars)$.
$\top_{a1}$ : $edge(X, Y)$.

Figure 3.5: $\top$ theory for path-edge example

as edge(a,Y), and update the list of InputSoFar with non-ground Y. Although it is non-ground, it is bound with other variables through *memberBind* and *inputUpdate*. It is this difference make it possible for TDTD to learn both abductive and inductive hypothesis at the same time.

**Lemma 3.1.3** *The input variable of body predicates can be bound to any term appeared before it by* $\top_{nt4}$ *and* $\top_{nt5}$ [1]

**Proof** (1) Base case: $\top_{nt4} : memberBind(X, [X|_])$ Input variable can be bound to the first element in the list.

(2) Assume input variable can be bound to kth element in the list, then there will a sequence as [nt5, nt5, nt5, nt5,...nt5, nt4 ] to answer the goal memberBind(X,[...Xk— Rest]).

By resolving this goal with $\top nt5$, another element is added at the head of the list, thus there is k+1 element before X. Therefore, X is bound to the (k+1) element in the list. ∎

## 3.2 Composing hypothesis language

### 3.2.1 SLD-resolution: non-unit clause

Composing the language by grammar is to connect the terminals while remove the non-terminals. For the grammar like clauses in $\top$ theory, the non-terminals are removed by resolution. The focus of current system is definite clause, thus SLD resolution is used for refutation of positive examples, correspondingly, the SLD resolution is used for deriving hypothesis language.

[1] this predicate has the same effect as the build-in predicate *member*

Let Dc denote the SLD-derivation sequence for clauses. Dc is extracted from SLD-refutation, and it preserves the same order as that in the SLD-refutation sequence. The detail of SLD-derivation can be found in section **??** and corresponding reference.

The following lemmas show and prove the proposition of the SLD derivation of hypothesized clause

**Proposition 3.2.1** *All and only non-terminal literals are resolved in the SLD derivation for non-unit hypothesized clause.*

**Proof** (1) prove all

Suppose there is one-terminal literal that not resolved. Then it will appear in the composed language, thus disobey the requirement that non-terminal predicate can not appear in the derived language.

(2) prove only

Assume terminal literals are resolved, then it will contradict the their property as terminal.

### 3.2.2 Subsumption: unit clause

The hypothesized clauses about abducible predicates are unit. It is also ground or existentially quantified, rather than universally quantified.

The SLD-resolution alone is not enough to derive all the hypothesized clauses, such as those about abducible predicates. This is due to the fact that SLD-resolution is only refutably complete, but not deductively complete. According to the sumption theorem[20], this can be solved by augmenting with subsumption

Since abduced hypothesis is ground or existentially quantified. Substitution or skolemization is applied for derivation. It gets its variables bound from the refutaion

No matter whether the hypothesized clause C is derived by SLD-resolution or subsumption, it always holds for $\top \models C$.

## 3.3 Completeness with respect to hypothesis language

Since the $\top$ theory defines the search space, it is important that all the clauses in the hypothesis language can be derived from it, so that it is not at risk of excluding candidate hypothesis.

The following proves first that every clause with any length can be obtained by the $\top$ theory in fig 3.6, then by adding non-terminal literals about binding variables to where it is needed, all the binding of variables within the clause can be achieved. Thus all the non-unit clauses in the hypothesis language can be derived from the $\top$ theory in fig 3.7.

Assume the problem setting of arbitrary numbers of predicates with arbitrary numbers of arguments. The numbers of input and output variable in each predicate is also arbitrary. All variables are standardized apart and no binding among any of them. Assume the $\top$ theory for this general case is as that in fig 3.6. Let predicate *head* denote the head of derived clause, and $bp_i$ denote the body predicates.

$$\top = \begin{cases} \top_0 : \text{head}(Xh_1, Xh_2, ..., Xh_{N_h}) \leftarrow \$\text{body}(List) \\ \top_1 : \$\text{body}(List) \leftarrow \text{bp1}(X^1_{in1}, ..., X^1_{inN1}, X^1_{out1}, ..., X^1_{outN1}), \$\text{body}(NList) \\ \top_2 : \$\text{body}(List) \leftarrow \text{bp2}(X^2_{in1}, ..., X^2_{inN2}, X^2_{out1}, ..., X^2_{outN2}), \$\text{body}(NList) \\ ... \\ \top_M : \$\text{body}(List) \leftarrow \text{bpM}(X^M_{in1}, ..., X^M_{inNM}, X^M_{out1}, ..., X^M_{outNM}), \$\text{body}(NList) \\ \top_{nt0} : \$\text{body}(List). \end{cases}$$

Figure 3.6: One generic $\top$ theory without variable binding

**Corollary 3.3.1** *There is at least one unit clause in the SLD refutation.*

$$
\top \;=\; \left\{
\begin{array}{l}
\top_0 : \text{head}(Xh_{in1}, ..., Xh_{inNh}, Xh_{out1}, ...., Xh_{outNh}) \leftarrow \\
\qquad \$\text{body}([Xh_{in1}, ..., Xh_{inNh}], BodyVars), \\
\qquad \text{memberBind}(Xh_{out1'}, BodyVars), \$bind(Xh_{out1'}, Xh_{out1}), ..., \\
\qquad memberBind(Xh_{outNh}, BodyVars), \$bind(Xh_{outNh'}, Xh_{outNh}). \\
\top_1 : \$\text{body}(InputSoFar, BodyVars) \leftarrow \\
\qquad \text{memberBind}(X^1_{in1'}, InputSoFar), \$bind(X^1_{in1'}, X^1_{in1}), ..., \\
\qquad \text{memberBind}(X^1_{inN1'}, InputSoFar), \$bind(X^1_{inN1'}, X^1_{inN1}), \\
\qquad \text{bp1}(X^1_{in1}, ..., X^1_{inN1}, X^1_{out1}, ..., X^1_{outN1}), \\
\qquad \text{inputUpdate}(X^1_{out1}, InputSoFar, NInputs_1), ..., \text{inputUpdate}(X^1_{outN1}, NInputs_{outN1-} \\
\qquad \$\text{body}(NInputs, BodyVars). \\
... \\
\top_M : \$\text{body}(InputSoFar, BodyVars) \leftarrow \\
\qquad \text{memberBind}(X^M_{in1'}, InputSoFar), \$bind(X^M_{in1'}, X^M_{in1}), ..., \\
\qquad \text{memberBind}(X^M_{inN1'}, InputSoFar), \$bind(X^M_{inN1'}, X^M_{inN1}), \\
\qquad \text{bpM}(X^M_{in1}, ..., X^M_{inNM}, X^M_{out1}, ..., X^M_{outNM}), \\
\qquad \text{inputUpdate}(X^M_{out1}, InputSoFar, NInputs_1), ..., \\
\qquad \text{inputUpdate}(X^M_{outNM}, NInputs_{outNM-1}, NInputs), \\
\qquad \$\text{body}(NInputs, BodyVars). \\
\top_{nt0} : \$\text{body}(BodyVars, BodyVars).
\end{array}
\right.
$$

Figure 3.7: One generic $\top$ theory with Binding

**Proof** Suppose there is no unit clause in the refutation. The length of derived clause by resolution never decrease, thus empty is not derivable, which contradict that empty is derived at the end of refutation. ∎

**Lemma 3.3.2** *All the clauses composed of Predicate=[head, bp1, bp2, ... , bpm] without variable binding can be derived from the $\top$ theory in the fig 3.6 .*

**Proof** proof by induction on the length of clause

1. Base case: All unit clause can be derived. By resolving $\top_1$ and $\top_{nt0}$, non-ground unit clause $\text{head}(Xh_1, Xh_2, ..., Xh_N)$. can be derived.

2. Suppose theorem holds for clause with length k, and there is a corresponding derivation sequence $D_{hk}$ for it.

According to Lemma 3.3.1, the only unit clause $\top_{nt0}$ must be in the derivation sequence for any clause. Replace it with $\top_i (2 \leq i \leq M)$ and $\top_{nt0}$, then another terminal literal in $\top_i (2 \leq i \leq M)$ is added to the derived clause, therefore, its length is increased to k+1.

Therefore, all the clauses composed of Predicate=[head, bp1, bp2, ... , bpm] can be derived

**Theorem 3.3.3 Complete $\top$ theory with respect to hypothesis language**
*All the non-unit clauses composed of Predicate=[head, bp1, bp2, ... , bpm] with any binding can be derived from the $\top$ theory in the fig 3.7, together with those in fig 3.2 and ?efvBind.*

**Proof.** *By lemma 3.3.2, clauses like head : bp1, bp2, ..$bp_M$ can be obtained, but there is no binding in its variables.*

*Add non-terminal literals about binding variables to where it is needed as in fig 3.7. In head clause, the list that store terms appear so far are initialized with the substitution of input variables in the head; when all the terms appear in the body of clause is returned in BodyVars, the output variables in the head are bound to them by memberBind. In body clause, the input variables of terminal literal $bp_i$ are unified with previously appeared term, function term can also be added if necessary ($ bind(X,X) will be used if no function symbol).*

*According to lemma 3.1.1, 3.1.2 and 3.1.3, they are complete for binding variables together and also with function terms, therefore, non-unary clauses with variable binding together can be derived from fig 3.7 .*

*Therefore, the $\top$ theory specified in fig 3.7 is complete with respect to the given hypothesis language.*

Note that $\top$ theory in fig 3.7 is a general case applicable for all. It can be simplified by removing $\$bind$ if it is a problems setting without function symbols. Similarly, $memberBind$ and $inputUpdate$ are not necessary in case that variable binding is fixed, such as unary predicates or continuous connected like that in grammar learning example.

# Chapter 4

# Top Directed Theory Derivation

This chapter gives the algorithm for addressing the issue of bounding search space. Example-driven algorithm is ideal for bounding search space, since the efficiency is gained without the trade-off for completeness. In example-driven algorithm, only hypotheses hold for the equation(4.1) [1] are generated, thus the search space is bounded. But how to derive only these candidate hypothesese?

$$B \wedge H \models E \tag{4.1}$$

One way is through Inverse Entailment (IE). IE treat induction as inverse of deduction, thus inversely compute bottom formula.Then all the hypothesese in the subsumption lattice bound by bottom formula, that is, those are more general than bottom formula, hold for equation(4.1). But there is a dangerous of incompleteness in the defined IE operator. Such as in the one implemented in Progol, only hypothesese with single clause are derivable. In addition, even if the IE operator defined is complete, the search space bound by bottom formula is still so huge that only incomplete search at generalization step is performed.

On the contrary to IE, candidate hypotheses are derived deductively in TDTD. They are extracted from the refutation for positive examples. It sounds impossible at first instance. Since H is unknown and on the left hand side of equation(4.1), then how can refutation for E be possible without knowing H beforehand? However, it is realized in TDTD by replacing the hypothesized theory with its grammar version in the refutation, and then derive itself from this grammar version.

Benefitting from the completeness of deduction, which is proved in subsumption theorem[20], TDTD no longer suffers from the incompleteness issue in IE, but delegates the completeness issue to the $\top$ theory. Also, under the multi-clauses learning setting the search space bound by TDTD is smaller than that by bottom formula, as explained in section 4.5.

The following diagram on the left of fig 4.1 illustrates the role of TDTD in the whole ILP system, that is, bound the search space before starting search. The diagram on the right of fig 4.1 gives the framework for the TDTD algorithm, and the table 4.1 [2]about TDTD algorithm gives more detail. It is similar to the framework introduced in [19] on which this project is extended.

In the following sections, details about the TDTD algorithm are given first, then the soundness and completeness of TDTD are proved.

## 4.1 SLD refutation for positive example

### 4.1.1 Replaceability of Grammar Version

As discussed in section 2.1.1, bias free learning is futile. Therefore TDTD start with the declarative bias which assume that the target hypothesis H exists in the language specified by $\top$. Since in ILP,

[1]Hypothesis H is a theory which can be multi-clauses or single clause, it is the union of theories t that account for each individual example. E denote all the examples, which is contrary to e which denote individual example

[2]Step 0 in the table does not belong to TDTD, but input to it

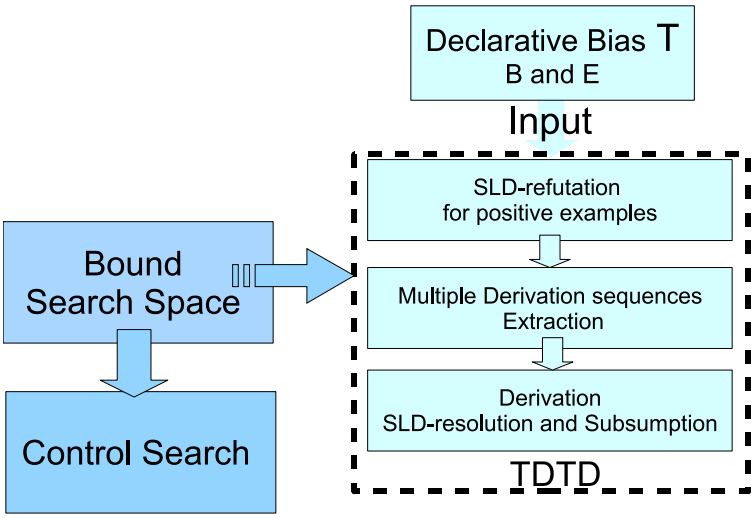


Figure 4.1: TDTDframework

| | |
|---|---|
| 0. | Provide declarative bias $\top$, as well as B and E to the TDTD system |
| 1. | Search for SLD-refutation of $\neg$e with $\top$ and B, and obtain refutation sequence R |
| 2. | Reorder and Extract multiple derivation sequences $D_h$ from R |
| 3. | Derive candidate hypothesis according to the $D_h$ obtained in step(2) using SLD-resolution and subsumption. |

Table 4.1: TDTD algorithm

the target hypothesis H holds for equation(4.2)[3] and all the clauses derived from $\top$ meet equation(4.3), TDTD start with the assumptions of equation(4.2) and equation(4.3). The lemma 4.1.1 proves that given these two assumptions, refutation for each positive example exists. From the perspective of grammar interpretation, this lemma justify that, the grammar version of hypothesized clause can take up the place where it is needed in the refutation for positive example.

$$B \wedge t \models e \tag{4.2}$$

$$\top \models t \tag{4.3}$$

**Lemma 4.1.1 Replaceability of Grammar Version.** *[19]* [4]
*Assumptions (4.2) and (4.3) hold only if for each positive example $e \in E$ there exists an SLD refutation $R$ of $\neg e$ from $\top, B$.*

**Proof.** *Assuming there is no SLD refutation of $\neg e$ from $\top and B$.*
*According to (4.2), the following (4.4)holds.*

$$\top \wedge B \models B \wedge t. \tag{4.4}$$

*Then from (4.4) and (4.3) it follows that for each positive example $e \in E$*

$$\top, B \models e. \tag{4.5}$$

*According to (4.5) and that SLD-resolution is refutably complete [12] it follows that there exists an SLD refutation $R$ of $\neg e$ from $\top, B$. This contradicts the initial assumption, thus completes the proof.* ∎

[3] t denote a subset of final theory H and e denote individual example
[4] this lemma is called Example derivability in [19]

### 4.1.2 Subgoal succeed with non-ground substitution

Different with the IE-based method that require all ground terms in building $\perp$, TDTD allow subgoal to succeed with non-ground substitution during the refutation. There are 2 situations as follows.

1. The non-ground term will finally be grounded by the binding propagated back from later resolving. When unifying with the ground substitution in the output variable of the seed example, such as 2*100+(30+2) in the given example wordnum([two,hundred,and,thirty,two],[],2*100+(30+2))[5]. The unground variable D in the subgoal hundred([two,hundred,and,thirty,two], [and,thirty,two], D*100) and digit(two, D) will finally be grounded to 2 in the expression 2*100+(30+2) by the binding propagated back. Thus the abducible fact digit(two, 2) is obtained.

2. The non-ground term is still left ungrounded and requires skolemisation. Such as the path-edge example used before. $edge(a, X) \wedge edge(X, b)$ with X universally quantified is a suggested explanation, while minimal answer is required in abductive hypothesis, therefore, further skolemisation is needed in order to derive abductive hypothesis.

## 4.2 Multiple Derivation Sequences Extraction

### 4.2.1 Problems in Multi-clauses Extraction

After refutation is found, the next question is how to extract multiple derivation sequences from the refutation sequence R. Unfortunately, it is not as easy as that in single-clause learning. Under the assumption of single-clause learning, all clauses in $\top$ that appear in R belong to the single hypothesized clause. However, in case of multiple sequences, how to isolate them from R become a problem. Such as in fig 4.2, the refutation sequence for odd(s(s(0)))) is
$[T1, Tnt3, Tnt2, T2, B2, T1, Tnt3, Tnt2, T2, B1, Tnt2, Tnt0, Tnt2, Tnt0]$,
from which we can see
(1) Each Dci $[T1, Tnt3, Tnt2, T2, Tnt2, Tnt0]$ is not continuous but scatter in R.
(2) No obvious separation. Clauses in B seems to be a choice, but actually they are not ideal for separation.
$[T1, Tnt3, Tnt2, T2, -, T1, Tnt3, Tnt2, T2, -, Tnt2, Tnt0, Tnt2, Tnt0]$
Neither $[T1, Tnt3, Tnt2, T2]$ nor $[Tnt2, Tnt0, Tnt2, Tnt0]$ will derive a complete clause.

Then how to gather and correctly separate sequences? Let's go through one example in fig 4.2 to see how does the extraction strategy proposed in the project work, and then justify the correctness of this extraction algorithm.

#### Example Section

Now let's start from the negation of odd(s(s(s(0)))). First, it calls the head clause T1 in $\top$, which is a start point for a new hypothesized clause, thus initiate a new list Dc1 to record it. Next subgoal has non-terminal predicate, so collect the clause Tnt3 matches that subgoal. Continue until terminal predicate *even* is encountered. This is going to call either clauses in B or another new hypothesized clause, in a word, it won't belong to the current collecting sequence. Due to the left-first computation rule in Prolog, query for that terminal subgoal even(s(s(0))) is executed first. Thus collection for Dc1 is temporarily paused here.

However collection for other Dci continues. Now it is a clause in B that is called. Since we are only interested in extracting clauses in $\top$, B2 which is labeled in black is not recorded. The next subgoal is one terminal literal again, but it call one head clause, thus initiate another new list Dc2 for it. Then collection carries on as before until the subgoal calling head clause is succeed (END sign in the fig 4.2).

Finally, two sequences are extracted. They are

[5]This is number example given in [17]

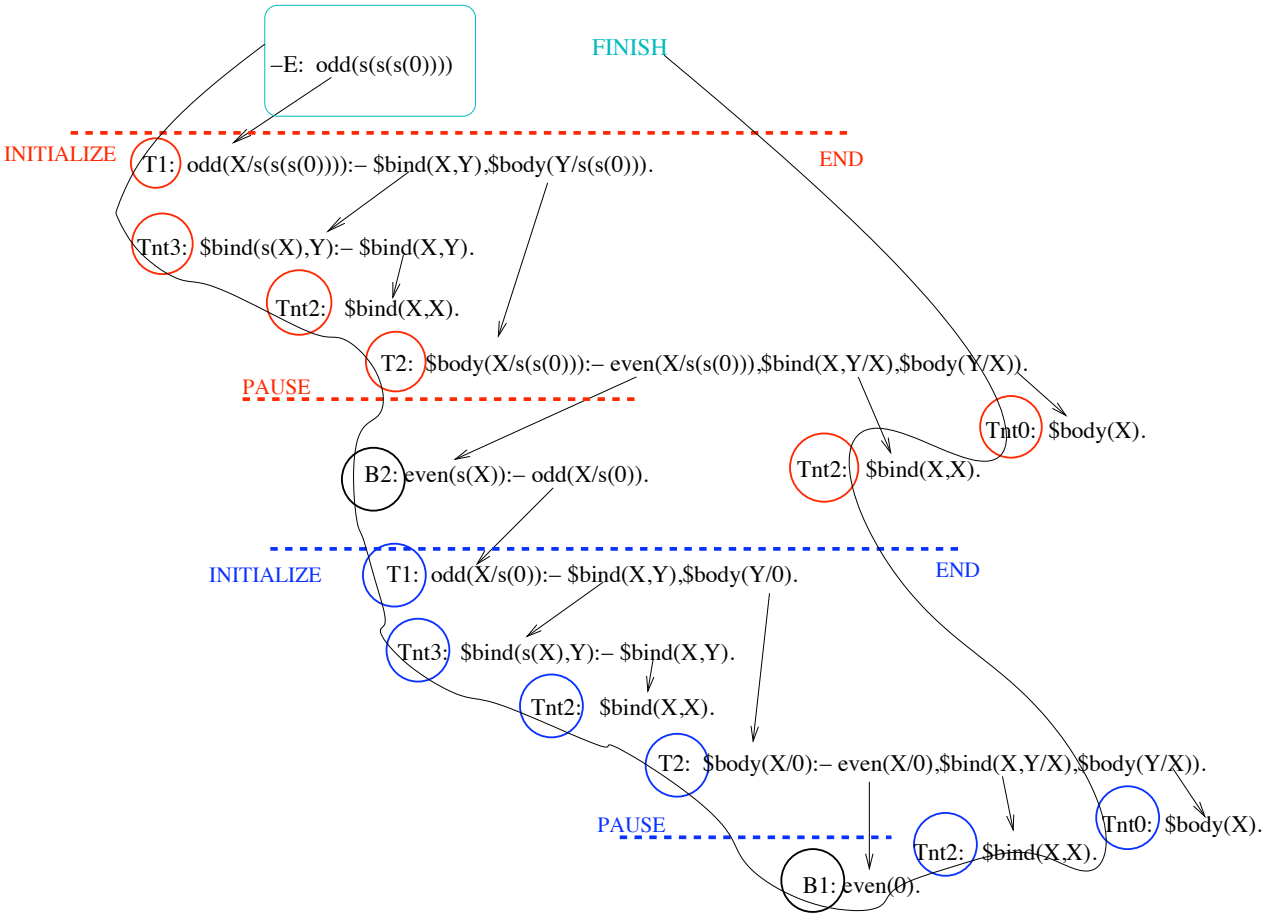


Figure 4.2: Applying Multiple Sequence Extraction Algorithm for Odd-Even Example

$[T1, Tnt3, Tnt2, T2, Tnt2, Tnt0] and [T1, Tnt3, Tnt2, T2, Tnt2, Tnt0]$[6].
Each of them still preserve the same order as that in the refutation sequence R. Actually, they are nested relation, rather than random scatter
$[T1, Tnt3, Tnt2, T2, -[T1, Tnt3, Tnt2, T2, -, Tnt2, Tnt0], Tnt2, Tnt0]$

### 4.2.2 Extraction Algorithm and its Correctness Prove

| Collect along the refutation: | |
|---|---|
| 1. | Start Point: head clause is sign for a new hypothesized clause. |
| 2. | Middle: collect only if it is clause in $\top$ with non-terminal predicate as head. |
| 3. | End: the same as the subgoal that calls head clause |

Table 4.2: Multiple Sequence Extraction Algorithm

The SLD derivation sequence $D_c i$ preserve the same order as that in the refutation R, so the collection for each $D_h$ follow along the refutation. In order to collect for each $D_h$, its start and end point should be identified. Also during the collection, it should be able to identify which one belongs to it while which one should be ignored. The following 3 rules solve these problems and compose the algorithm for multiple $D_h$ extraction.

Let $HC_{head}$ denote the head of hypothesized clause. Record each $D_h$ in a list and collect it along with the refutation for positive training example:

(1) Start rule. Whenever a head clause is encountered in the refutation sequence R, according to the following proposition 4.2.2, it marks the start point for a new $D_h$. Therefore, a new list is initialized to record the $D_h$ start with it.

(2) Middle collection rule. During the collection, only clauses in $\top$ with non-terminal predicate as head are recorded; while ignore if its head is terminal predicates, that is, clauses in B and head clauses in $\top$ are ignored. This middle collection rule is based on the proposition that only resolution between non-terminal predicates happens in the SLD derivation of hypothesized clause. Therefore, the collection should skip the refutation sequence for the subgoal of terminal literals.

(3) Stop rule. Collection for $D_h$ with $HC_{head}$ as head finish when the refutation for subgoall $\leftarrow HC_{head}$ finishes.

The following corollary and propositions prove the correctness of above extraction algorithm.

**Corollary 4.2.1** *The clause derived by SLD-resolution has the same head as that of the first clause in the derivation sequence*

**Proof** Assume they have different head, then the head of first clause must be resolved away, otherwise, there will be two positive literals in the resulting clause which contradicts the definition for Horn clause.

Since the head literal is positive, there should be a negative literal in one input clause to resolve with it. However, this contradict the the computational rules of SLD which requires resolving with the head of input clause which is positive.

Therefore, they should have the same head. ∎

**Proposition 4.2.2** *One clause in the refutation sequence R is the start point for the derivation sequence $D_h$ (first clause in $D_h$) if and only if it is a head clause in $\top$*

**Proof** (1) if part. (if head clause, then must be start point) Assume it is not the first clause in the SLD derivation sequence $D_h$, then according to the definition of SLD derivation, as input clause, the head of this head clause should resolve with the previous resolvent. Since the head of head clause is a terminal literal, this contradict to the proposition that only resolution between

[6]It is the case that the same hypothesized clause is used twice in the refutation, therefore the two sequences are the same, so after removing duplicates, only one clause remains in derived theory t: odd(s(X)):- even(X)

non-terminal predicates happens during the derivation. Therefore, head clause can not appear else where except the start point.

(2) only if part. (top clause must be head clause) According to the previous corollary, the derived hypothesized clause has the same head as that of the first clause $D_h$. The head of derived clause is a terminal predicate, therefore the first clause in the derivation must be a clause with terminal predicate as head. Since the only clause in $\top$ that has terminal predicate in head is the head clause, the first clause in the derivation must be head clause. ∎

**Proposition 4.2.3** *Collection for SLD derivation sequence $D_h$ whose derived clause has $HC_{head}$ as head should stop when the refutation for subgoall $\leftarrow HC_{head}$ finishes.*

**Proof** Prove by refutation: suppose this proposition is not correct, then there are two possibilities:

(1) collection for $D_h$ is still unfinished when the end of refutation sequence Rh is reached This means there are elements in $D_h$ that is not in Rh. However, $D_h$ essentially derives from the reordering of Rh, and is a subset of it. Therefore, the assumption that there are elements in $D_h$ that is not in Rh will contradict the fact that $D_h$ is a subset of Rh.

(2) another situation is over finished, that is, the collection over collect that not belong to it, but to other $D_h$s. Suppose that there are elements in collection that not belongs to the collecting $D_h$, but other $D_h$s.

$D_h = D_{h'}D_{over}$, let $D_{over}$ denote the part that is over collected and belong to other $D_h$. According to the proposition 4.2.2, the first clause in $D_{over}$ must be a head clause with terminal predicate as its head, this contradict the middle collection rule that only clauses with non-terminal heads are collected and recorded. Therefore, over collection will not happen. ∎

it is the assumption come from declarative bias, but not arbitrary reorder – it is the specific way of SLD derivation that ensure the correctness whether the derive holds remains to be a question. by reorder and resolving the non-terminals, the derived still

let R denote. reordering theorem. if there is a refuation, then R derive from the

## 4.3 Hypothesized Clause derivation

After the derivation sequence is extracted, the candidate hypothesis can be derived by applying the SLD resolution on non-unit sequence or subsumption on unit ones.

This part is actually composing the hypothesis language with grammar, so the same as that in the section 3.2 of last chapter.

## 4.4 Soundness and Completeness of TDTD

**Theorem 4.4.1 Soundness of TDTD** *The theory t* [7] *holds for equation (4.2) if it is derived by the TDTD algorithm*

**Proof.** *Assume the theory t derived by the TDTD algorithm does not hold for equation (4.2), that is, there is no refutation of e by clauses from t and B.*

*Theory t is derived by TDTD, so at the first step of its derivation, there is one SLD-refutation of $\neg e$ via R.*

*(1) Non-unit hypothesized clause is derived by applying SLD-resolution to the derivation sequence extracted from R.*

*According to the independence of computation rule of SLD-resolution, empty is still derivable after reordering R. Therefore, there are pairs of complement literals among the clauses appeared in R.*

*According to the proposition 3.2.1 that all and only non-terminal literals are resolved during this derivation, only pairs of non-terminal literals in $\top$ are resolved away.*

[7] t is related to individual examples, and it is a subset of final theory H which covers all examples

*Since non-terminal literals do not appear in clauses in B, thus the other pairs of terminal literals are not affected, thus there are still paris of complement literals left if replacing all clauses in $\top$ that appear in R. Therefore, empty is still derivable.*

*(2) Unit clause gets its variables substituted as that in the refutation, so the refutation R is not changed if replace the clause in $\top$ with this grounded version, thus empty is still derivable.*

*Therefore, there is one refutation of e by clauses from t and B. Thus t holds for equation (4.2)*

**Theorem 4.4.2 Completeness of TDTD** *Provided that the given $\top$ theory t is complete with respect to hypothesis language,*
*the theory t holds for equation (4.2) only if it is derived by the TDTD algorithm*

**Proof.** *Assume the theorem is false, that is, there is one theory t holds for equation (4.2) but can not be derived by the TDTD algorithm.*

*given assumption that the given $\top$ theory is complete with respect to the hypothesis language, then the theory holds for (4.3), according to lemma 4.1.1, there is refutation R exist,*

*Then by reordering and multiple sequence extraction, clauses in theory t can be derived by SLD-resolution or subsumption.*

*Therefore, contradict the assumption that theory t can not be derived.*

Since the condition for this theorem is the completeness of $\top$ theory with respect to hypothesis language, the completeness issue is delegated to $\top$ theory.

## 4.5 Search Space Analysis

The search space bound by TDTD is compared with that by IE-based method in this section.

### 4.5.1 Multi-Clauses Setting

The advantage of TDTD is to support multi-clauses setting, but this is trade-off for much bigger search space. However, it is still tractable as shown by experiments in Chapter 7. Although IE-based methods are also example-driven, there is redundancy in its search space due to the upward theory refinement, which is discussed in section 2.4.3.

Take the example in section 2.4.3 again. Clauses $C_2$, $C_3$ and $C_4$ are no longer connected to $C_1'$ via predicate $n$, thus will not involve in the refutation for seed example with $C_1'$. Therefore they will not appear in the theory that has $C_1'$. Thus the redundant ones like $t_1'$ and $t"_1$ will not be derived in TDTD.

$$t_1 = \begin{cases} C_1 : \mathrm{e}(X) \leftarrow \mathrm{m}(X), \mathrm{n}(X). \\ C_2 : \mathrm{n}(X) \leftarrow \mathrm{u}(X), \mathrm{v}(X). \\ C_3 : \mathrm{u}(X) \leftarrow \mathrm{b}(X). \\ C_4 : \mathrm{v}(X) \leftarrow \mathrm{b}(X). \end{cases} \qquad t_1' = \{\ C_1' : \mathrm{e}(X) \leftarrow \mathrm{m}(X).$$

$$t_1' = \begin{cases} C_1' : \mathrm{e}(X) \leftarrow \mathrm{m}(X). \\ C_2 : \mathrm{n}(X) \leftarrow \mathrm{u}(X), \mathrm{v}(X). \\ C_3 : \mathrm{u}(X) \leftarrow \mathrm{b}(X). \\ C_4 : \mathrm{v}(X) \leftarrow \mathrm{b}(X). \end{cases} \qquad t_1'' = \begin{cases} C_1' : \mathrm{e}(X) \leftarrow \mathrm{m}(X). \\ C_2 : \mathrm{n}(X) \leftarrow \mathrm{u}(X). \\ C_3 : \mathrm{u}(X) \leftarrow \mathrm{b}(X). \\ C_4 : \mathrm{v}(X) \leftarrow \mathrm{b}(X). \end{cases}$$

Therefore, the search space bound by TDTD will be smaller by avoiding the redundancy. Different from the incomplete generalization in IE-based method, such as omitting clause addition or inverse resolution which are at risk of excluding correct learning results, the smaller search space in TDTD is not a trade-off for the completeness, because it is redundant ones unconsidered,.

### 4.5.2 Single-clause Setting

In the single-clause setting, provided that the $\top$ theory is complete with respect to the hypothesis language, the search space bound by TDTD[8] are exactly the same as that by bottom clause.

[8]TDTD is actually the same as TopLog under the single-clause setting

**Lemma 4.5.1 Same Space in Single-clause Setting** *The search space bound by TDTD and that by bottom clause are exactly the same in the single-clause setting, provided that the* $\top$ *theory is complete with respect to the hypothesis language.*

**Proof.**

*The IE operator in Progol is complete with respect to relative subsumption[28]. Therefore, the search space bound by bottom clause is complete for single-clause.*

*Assume the search space bound in TDTD is bigger than that of bottom clause, according to the soundness of TDTD, there is one hypothesized clause derived in TDTD while not inside that bound by bottom clause. This is contradict to the completeness of bottom clause in the single-clause setting.*

*Now assume the search space bound in TDTD is smaller. According to theorem 4.4.2, provided that the given* $\top$ *theory t is complete with respect to hypothesis language, TDTD is complete for deriving hypothesized clauses. However, according the soundness of bottom clause, if the space bound by TDTD is smaller, then there are at least one hypothesized clause not included in TDTD, which contradict the completeness of TDTD.*

*Therefore, the two space are exactly the same.*

The same space can also be interpreted from the perspective of treating *top* theory as grammar. For each clause within the space bound by bottom clause, there is a corresponding grammar version denoted by $\top$ theory, thus can be derived in TDTD.

Although the size of two spaces are exactly the same, there is advantage in that bound by bottom clause. Because it is organized in subsumption lattice so that efficient pruning can be applied. Therefore, in the single-clause setting, Progol will outperform TopLog and TDTD.

# Chapter 5

# Greedy Search for Final Theory

## 5.1 Covering Algorithm

Although it is theory that derived for each individual example, covering algorithm is still necessary for constructing the final theory. For example, the given example $s([a, dog, hits, a, ball], [])$ will only derive theories that including the clause np(S1,S2):- det(S1,S3), noun(S3,S2). While theories including the clause np(S1,S2):- det(S1,S3), adj(S3,S4), noun(S4,S2) will only be generated by example like $s([the, small, dog, hits, a, ball], [])$.

The covering algorithm used is in table 5.1, which is the similar to that in [16]. Clearly, this covering algorithm terminates in at most $|E|$ iterations. It may also terminate when there is no compression which will happen when the size of training data is small.

| | |
|---|---|
| 0. | Let $T$ denote the final theory, which is empty when initialized; |
| let ts be the set of theories derived by one seed example; | |
| 1. | if E= $\emptyset$, return T; |
| 2. | Let e be the first example in E, and derive $ts$ using TDTD algorithm 4.1; |
| 3. | Choose from $ts$ the one that maximumly increase the score of T and add to the background knowledge; |
| 4 | Remove from E the examples that coverd after adding the newly chosen theory; |
| 5 | Go to 1. |

Table 5.1: Covering Algorithm

Note that redundant clauses are removed when adding newly chosen theories to the background knowledge. Redundancy is defined as follows: [16]

**Definition Redundant clauses**
Let C be a clause and T be a set of clauses.
C is redundant in $T \bigcup C$ if and only if $T \models C$

The covering algorithm 5.1 makes the greedy choice at local scale, that is, amongst theories generated by one seed example, thus it will make a choice that is locally optimal while not globally. In addition, not only the running time, but also the final learning results will be dependent on the order of given examples. Despite of these disadvantages, it is more efficient than making greedy choice at global scale [19]which requires seeding on all the positive examples.

## 5.2 Minimum Description Length as Heuristic

Minimum Description Length (MDL) is used as heuristic for greedy search. Let Ne denote the numbers of examples covered; '+' and '-' respectively denote positive and negative examples; $|T|$ denote the number of literals in theory T. Then the MDL is defined as follows:

$$Ne_+ - Ne_- - |T| \tag{5.1}$$

When there is no noise in the training data, the target theory should not cover any negative examples, thus MDL can be simplified to

$$Ne_+ - |T| \tag{5.2}$$

The complexity of theory T is measured as that in [16] and [19], that is, the number of literals in T which is denoted by $|T|$. Note that complex score is calculated after removing redundancy.

However, the complexity measured by number of literals is too simple to account for the generality of theory. For example, both even(s(s(0))) and even(s(s(X))) are unit clause, that is, the same length, but the later is more general than the previous one. Although this example can be solved by considering the variables in the clause, here is a situation that generality varies even with same number of variables and literals. In the following 3 theories, t2 is more general than both t1 and t3, although they have the same score of minimum description length.

t1: $odd(s(A)) \leftarrow even(A).even(s(B)) \leftarrow odd(B).odd(s(0))$.
t2: $odd(s(A)) \leftarrow even(A).even(s(B)) \leftarrow odd(B).even(0)$.
t3: $odd(s(A)) \leftarrow even(A).even(s(s(B))) \leftarrow even(B).even(0)$.

# Chapter 6

# Implementation

## 6.1 program transformation

Both $\top$ and B are definite clauses, Prolog interpreter should have been enough. However, there are certain functions not available. One usual way is to manipulate at meta-level, but this will compromise efficiency. Therefore in this system, program transformation[1] is used, which make use of the quick variable binding in Prolog, thus add little overhead.

The following give the details about how to handle the unavailable while necessary functions by program transform.

### 6.1.1 Record SLD-Derivation Sequences

In the implemented system, the derivation sequence are directly extracted out along the refutation, but in order to illustrate how it is implemented, the following start with how to record the refutation sequence with program transform before talking about how to directly extract them.

#### Record Refutation Sequence by Program Transform

Two arguments are preserved in each predicate for this record. RSoFar denotes the input variable which gives the record before calling the goal of this predicate, while the other R as output variable is used for returning the record when the query is answered.

Here is one example for unit clause:

$$\$body(X).$$
$$\$body(R, [nt0|R], X).$$

The following are another examples for non-unit clause, in which number 10 is the ID for this clause. Then after calling the goal odd([], R, s(s(s(0)))), R will return the whole refutation sequence for the goal odd(s(s(s(0)))).

$$\$body(X) : -even(X).$$
$$\$body(RSoFar, R, X) : -even([2|RSoFar], R, X).$$

$$odd(X) : -\$bind(X, Y), \$body(Y).$$
$$odd(RSoFar, R, X) : -bind([10|RSoFar], R1, X, Y), body(R1, R, Y).$$

The previous examples will record all the clauses in the refutation, but what about those not to be recorded, such as clauses in B. Even thought the clause itself is not recorded, it still need be

[1] It is also called partial evaluation

transformed in order to propagate the record. Here are another three examples for the unrecorded clauses.

$$verb([walks|S], S)$$
$$verb(R, R, [walks|S], S)$$

$$vp(S1, S2) : -verb(S1, S2).$$
$$vp(RSoFar, R, S1, S2) : -verb(RSoFar, R, S1, S2).$$

$$vp(S1, S2) : -verb(S1, S3), prep(S3, S2)$$
$$vp(RSoFar, R, S1, S2) : -verb(RSoFar, R1, S1, S3), prep(R1, R, S3, S2).$$

Notice that no ID is put at the head of record list compared to the recorded one.

You may notice this way of collection, which always put new record at the head of list will result in a reverse order. Actually, this is not a problem since this record can be used backwardly to derive the hypothesized clause. Details about derivation will be given later

**Direct extraction**

The record above only give a linear sequence in which multiple clauses are mixed, thus need further work in order to extract each derivation sequence directly. Here two pairs of variables are needed. One pair ClaRSoFar and ClaR for single clause which is a list, the other pair TrSoFar and Tr for theory which is a list of lists. ClaRSoFar will record the derivation sequence for one clause according to the multiple extraction algorithm explained in section 4.2, it is directly added into TrSoFar once finish, so that they are directly extracted out. Examples for this are in appendix.

### 6.1.2 Control the SLD-refutation Search

**Numerical Constraint**

Since arithmetic execution is expensive in Prolog, while variable matching is much quicker, piano number is used for decreasing on counter. {s(Limit), Limit} is comparable to NLimit is Limit-1. For all the following control that related to counter, another argument in predicate is preserved for storing the counter which is in the form of piano number.

1. Depth Limit

The left-first computational rule in SLD-refutation has the effect of depth-first search. However, this depth-first search may run into infinite loop, like the following path-edge example, thus need depth limit. Iterative Deepening Search is a choice here, but for most of the problems encountered so far, the search tree is not deep, thus the overhead in Iterative Deepening Search will outweigh its benefits, so in the current system only a maximum depth limit is imposed.

path(X,Z):- path(X, Y), edge(Y,Z).
path(X,Y):- edge(X,Y).

Transformed version with Depth Limit

path(s(DL), X,Z):- path(DL, X,Y), edge(DL, Y,Z).
path(s(DL), X,Y):- edge(DL, X,Y).

The only drawback is the heavily nested function term, such as s(s(s(0)))). Since there is build-in mechanism in YAP to constrain the maximum depth, the depth limit is imposed as *depth_bound_call*(user:Goal, DepthLimit) in the current system.

2. Limit on the times appeared in the clause

Clauses like $mammal(X) : -has\_milk(X), has\_milk(X), has\_milk(X), has\_milk(X).$ will be derived if no constrain on the times that certain literal is allowed in the clause. Therefore there are a series of arguments specifying the maximum times that each predicate is allowed. There seems to be an order, but just an order of specification, and has nothing to do with the order these predicates

appeared in the clause. Here is one example.
$\top_1 : \text{mammal}(X) : -\$\text{body}([\mathbf{s(0)}, \mathbf{s(0)}], \mathbf{X})$
$\top_{\mathbf{2}} : \$\text{body}([\mathbf{s(N1)}, \mathbf{N2}], \mathbf{X}) : -\text{has\_milk}(\mathbf{X}), \$\text{body}([\mathbf{N1}, \mathbf{N2}], \mathbf{X})$
$\top_{\mathbf{3}} : \$\text{body}([\mathbf{N1}, \mathbf{s(N2)}], \mathbf{X}) : -\text{has\_eggs}(\mathbf{X}), \$\text{body}([\mathbf{N1}, \mathbf{N2}], \mathbf{X})$
$\top_{\mathbf{nt0}} : \$\text{body}(\mathbf{X})$.

3. Limit on clause length The length of clauses in First Order Logic can be infinite, so a constraint is needed. It can be controlled by constraint on the number of body clauses 3.1.1. This is handled in the non-terminal predicate $body, since it is responsible for connecting terminal literals.

In the following example, the maximum length is specified in the head clause as s(s(0))=2, while actually it allows only one body literal. This is because after resolving with two body clauses, such as
$odd(s(s(X))) : -even(s(X)), odd(X), \$body(\mathbf{0}, \mathbf{X})$.
It decreases to 0, while still need at least s(0) to match with $\top nt0$, thus it will fail in deriving the clause with length 2.

$\top 10 : odd(X) : -bind(X, Y), body(\mathbf{s(s(0))}, \mathbf{Y})$.
$\top 20 : even(X) : -bind(X, Y), body(\mathbf{s(s(0))}, \mathbf{Y})$.
$\top 1 : body(s(LenLimit), X) : -even([X), bind(X, Y), body(R2, R, Y)$.
$\top 2 : body(s(LenLimit), X) : -odd([X), bind(X, Y), body(R2, R, Y)$.
$\top nt0 : body(s(LenLimit), X)$.

This length limit is somewhat redundant with the previous Limit on the times appeared in the clause, because the length of clause will be under control with constrain on the times appeared. However, it is still necessary in some situation. Such as in the grammar learning example, to avoid directly parse the sentence to words by words like s:- det, adj, noun, verb, prep, det, noun, by specifying the Length Limit to 4, such clauses will be discarded.

**Integrate Constraint**

Integrate constraint can be implemented within the transformed program by disallow those violate it. This way, the theories including the clause that violate the integrate constraint will be discarded before fully derived, thus it is more efficient than the generate-and-test. For example, the $Hr1\backslash == [nt2, 10]$ in the following program will discard the theories include clauses odd(X):- even(X) which contradict the integrate constraint that a natural number can not be odd and even at the same time; it also discard the theories including tautology odd(X):- odd(X).

```
odd(TrSoFar, [Hr|Tr,X):-
$ bind([10], Hr1,X,Y),
Hr1\ == [nt2, 10],
$body(TrSoFar,Tr,Hr1,Hr,[s(0),s(0)],Y).
```

## 6.2 Theory Derivation

### 6.2.1 SLD-derivation

Compare the following two logic programs, hInterpreter/2 and hInterpreter/3. Both are recursive, and they have the same efficiency. But the first one is bottom-up, while the second one is top-down. Since the derivation sequence extracted is in a reversed order, so it is the first one used.

```
 hInterpreter([HeadI],Cla0):-
    topCla(HeadI,Cla0).
hInterpreter([I |Indexes],NewH):-
```

```
    hInterpreter(Indexes,H),
    topCla(I,Cla),
    resolve(H,Cla,NewH).

hInterpreter([],H,H).
hInterpreter([I|Indexes],HSoFar,H):-
    topCla(I,Cla),
    resolve(HSoFar,Cla,UpdatedH),
    hInterpreter(Indexes,UpdatedH,H).
```

The derivation follows the left-most computation rule in Prolog, therefore, the head of input clause resolve with the most left literal that match with it. Building control predicate *once* is used to avoid backtrack which may lead to resolving with non-left-most

```
resolve([Head1|Body1],[Head2|Body2],[Head1|NewBody]):-
        once((append(Body1A,[Atom1|Body1B],Body1),Atom1=Head2)),
        concatenateLists([Body1A,Body2,Body1B],NewBody).
```

### 6.2.2 Subsumption

Derivation using subsumption is to substitute the variable in the $\top$ clause with that in refutation. Since the substitution is recorded in the derivation sequence, it is easy to get it. In the following example, a3 is the ID for this $\top$ clause, the associated X record the substitution in the refutation, then the variable X in the predicate noun get the same substitution.
topCla(a3-**X, [noun([X|S],S)]).**

## 6.3 Greedy Search

### 6.3.1 Score

In greedy search, each candidate theory is scored according to the minimum description length, the one with maximum score will be chosen.

Since there is no noise in the artificial data used, by discarding the theories cover negative examples, the MDL is simply $Ne_+ - |T|$ where $Ne_+$ is the numbers of positive examples covered and $|T|$ denote the number of literal in the theory.

While the covering algorithm focuses on the positive examples uncover, so it is uncover list $Uncover$ recorded, thus the score will be $Total - |Uncover| - |T|$. Since the exact value is not interested, but their comparable value, therefore the score can be further simplified as $|Uncover| + |T|$ then the previous maximum one becomes minimum.

### 6.3.2 Removing Redundancy

In order to remove redundant clause, it need a procedure that can identify whether one clause C is redundant with the set of clauses T. According to the definition **??** of redundancy, it can be identified by checking whether C can be entailed by T. Proved by refutation, $T \cup \neg C \models \emptyset$, therefore, it is implemented as follows.

| | |
|---|---|
| 1. | Skolemise clause C |
| 2. | Assert the skolemised atom in the body of clause C |
| 3. | Call the head of clause C as goal |
| 4 | Clause C is redundant to current background knowledge if the goal succeed, otherwise not |

Table 6.1: Redundancy Check

# Chapter 7

# Empirical Evaluation

The purpose of the experiments in this chapter is to evaluate the performance of TDTD system in the multi-clauses setting. These experiments are carried out using artificial data on Linix Machine with Intel Core 2 Duo @ 2.13 GHz with 2Gb RAM.

## 7.1 Mutually dependent concept: odd-even example

### 7.1.1 Materials

The original odd-even example given in [28] is

$$B = \begin{cases} b1 : even(0). \\ b2 : even(s(X)) \leftarrow odd(X). \end{cases} \qquad \text{E} = \text{odd(s(s(s(0))))}$$

Figure 7.1: Yamamoto's odd-even example

It is unsolvable for single-clause ILP systems like Progol and TopLog, because the single hypothesized clause is used more than once in the refutation, thus a special case of multi-clauses learning problem.

In this experiment, the previous odd-even example is modified further to be more difficult. Suppose both of the clauses b1 and b2 are unknown, then nothing remains in the background knowledge. Therefore, the challenge here is to learn a mutually dependent concept: in order to learn predicate *odd*, knowledge about *even* is needed while it is unknown; similarly, learning concept about *even* can not be carried out when nothing is known about *odd*.

The $\top$ theory and its transformed version for this odd-even example are given in appendix A.

The integrity constraints for this example are as follows.
1. No tautalogy like even(X)← even(X) and odd(X)←odd(X).
2. A natural number can not be even and odd at the same time. thus discard theories include the clauses like even(X)←odd(X) and odd(X) ←even(X).
3. Non-unit clauses is not allowed to be ground, in other words, among all the ground clauses, only unit ones are allowed. Thus theories including clauses like odd(s(0))← even(0) are discarded.

Another constraint is about the length of clause. By restricting it to be no more than 2, clauses like odd(s(s(X))) ←even(s(X)), odd(X) is not considered although they are consistent with examples. Actually, even if they are considered, they will be removed during redundancy check. Because they are subsumed by either odd(s(X))← even(X). or odd(s(s(X))) ← odd(X).

All the above constraints are incorporated into transformed version of $\top$ theory as shown in appendix A.1.

### 7.1.2 Methods

Two experiments are carried out on this odd-even example:

(1) Measure the predictive accuracy.
Because not only the size of training data, but also the variety within it will affect the learning results, thus randomly sample 10 times for same size of training data, and their results are averaged. Leave-one-out cross validation was used.

(2) Measure running time.
Due to the covering algorithm used, the running time will vary with the seed examples, so run this experiment with different seed examples to see how the search space varies with different seed example.

The training data used for experiments are natural numbers from 2 to 9 which is also given in appendix. There are 8 positive and 8 negative, so 16 in total.

### 7.1.3 Results and Analysis

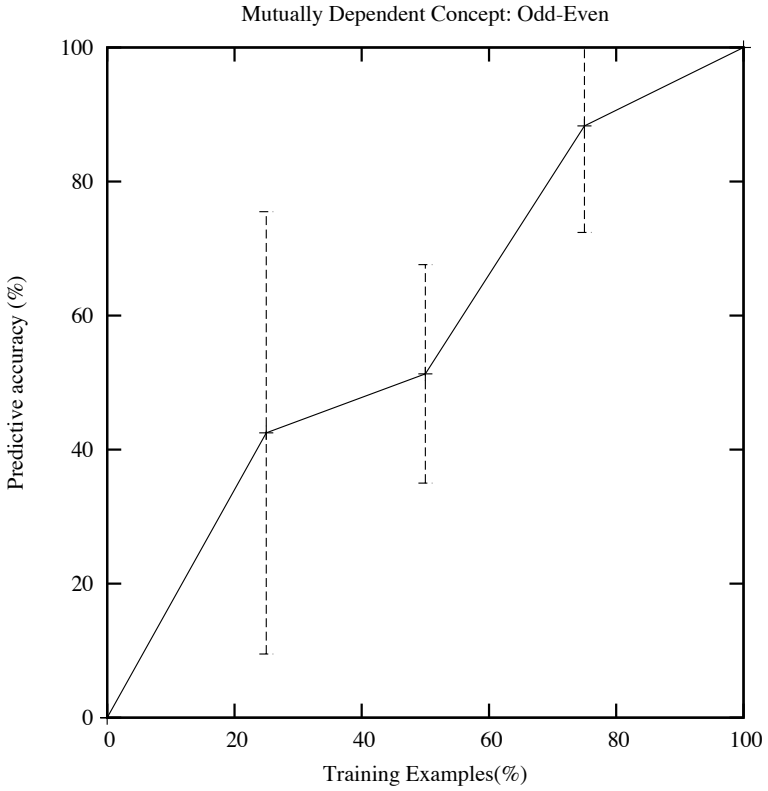


Figure 7.2: Predictive Accuracy for odd-even Example

#### Predictive Accuracy

As shown in fig 7.2, (1) It achieves 100% when there is sufficient data. The learning result is also correct. Actually, it suggests 3 hypotheses as follows.
t1: $odd(s(A)) \leftarrow even(A).$ $even(s(B)) \leftarrow odd(B).$ $odd(s(0)).$
t2: $odd(s(A)) \leftarrow even(A).$ $even(s(B)) \leftarrow odd(B).$ $even(0).$
t3: $odd(s(A)) \leftarrow even(A).$ $even(s(s(B))) \leftarrow even(B).$ $even(0).$

Although these 3 got the same score according to the MDL, t2 is more general than both t1 and t3 .

(2) As there is less training data, not only the predictive accuracy decreases, but also more incorrect[1] results are produced, such as:

$$t_{incorrect} = \begin{cases} even(s(s(X))) \leftarrow even(X) \\ even(s(s(0))). \\ odd(s(X)). \end{cases}$$

Examining the output of derived theories, the real correct theories are still derived and exist in the search space, but they are all more complex than the incorrect ones. When there is no negative examples to rule out odd(s(X)), also not enough positive examples to make the real correct theory get much higher coverage, the complexity of theory will dominate the MDL score, which make the incorrect ones outperform instead.

**Running Time**

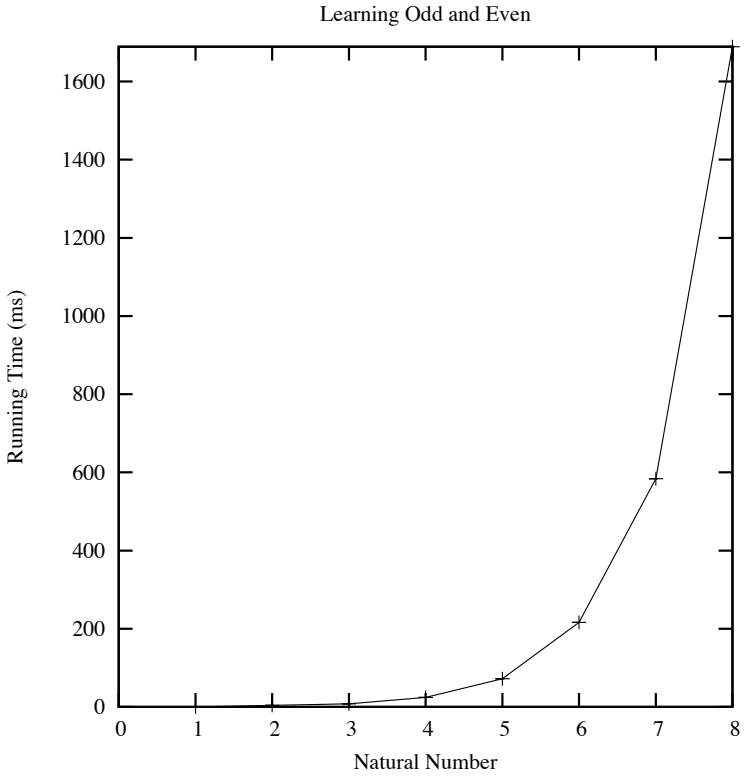


Figure 7.3: Running Time for odd-even Example

The running time increases exponentially, as shown in fig 7.3. This is because as the seed example is further away from 0, the search space expand quickly. For example, for odd(5) theories like the following will be derived:

$$t_{odd(5)} = \begin{cases} odd(s(A)) \leftarrow even(X) \\ \mathbf{even(s(s(s(X))))} \leftarrow \mathbf{odd(X)} \\ odd(s(0)). \end{cases}$$

Actually, this problem can be solved by using multiple examples as seed, that is, generalizing on more than one example at the same time, so that only their common ones are derived. Thus the theories which are only generated for particular example like that for odd(5) will not be generated.

[1]Incorrect means not the same as our target hypothesis, but it still covers all the training examples with the remained clauses in background knowledge.

### 7.1.4 Further discussion

1. Accommodate non-OPL setting. Actually, single example of odd is enough to learn both concepts about *odd* and *even*, that is, the whole theory can be derived by only one example. But if there are only examples about predicate *odd*, the following theory will turn out to be more compressive. Therefore, examples about *even* is still given.

$$t_{odd_{alone}} = \begin{cases} odd(s(s(X))) \leftarrow odd(X) \\ odd(s(0)) \end{cases}$$

2. Advantage of example-driven. If add the clause $\$bind(X, s(Y)) \leftarrow \$bind(X, Y)$ to the $\top$, one interesting theory will be derived:

$$t_{pre_suc} = \begin{cases} odd(X) \leftarrow even(s(X)). \\ odd(s(X)) \leftarrow even(X). \end{cases}$$

Although it is term expansion and has risk of non-termination, benefit from the example-driven, it still derive theories without additional control.

## 7.2 Grammar Learning example: Integrating abduction and induction

### 7.2.1 Materials

There is a hierarchical structure in parsing grammar, as shown in fig 7.4. Each line labeled with $C_i$ indicates a relation between them. The complete theory is given in fig 7.5. There are 23 clauses[2] in total, 6 of them is general rules, while 17 are ground facts. For all the predicates in this theory, except the predicate S, all the others are non-observable. Thus Progol5 need applying abduction before learning general rules about the non-observable predicates like NP. However, in situation that multi-clauses are missing, such as both C1 and C3 in fig are both missing, abduction is inapplicable for Progol5 when given the example $s([a, small, dog, hits, a, ball], [])$.

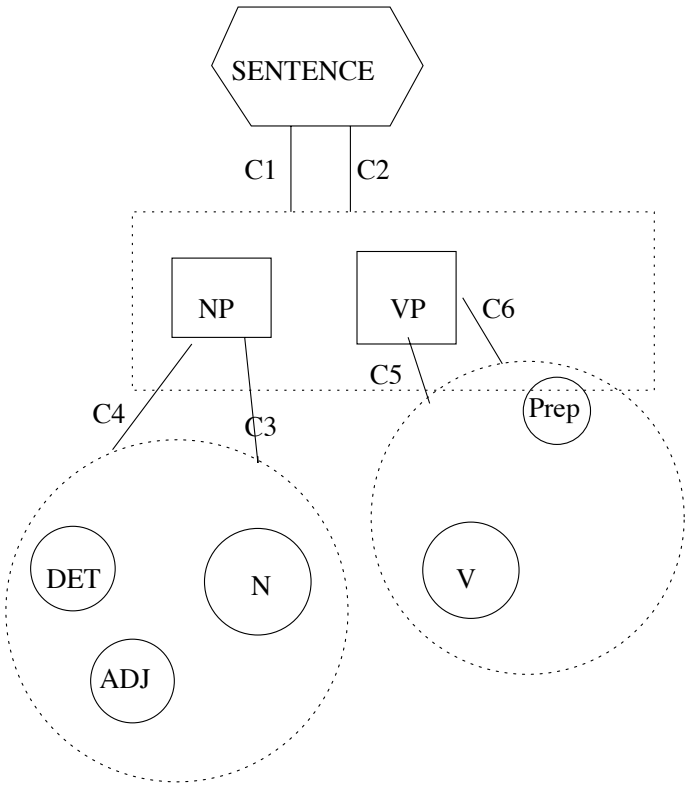


Figure 7.4: Hierarchical structure in Grammar Learning

[2]The facts conj([and|S], S) is not included since it does not appear in the given example. Therefore the complete theory here is one clause less than that in Progol5

```
C1  s(S1,S2) :- np(S1,S3), vp(S3,S4), np(S4,S2).
C2  s(S1,S2) :- np(S1,S3), vp(S3,S4), np(S4,S5), prep(S5,S6), np(S6,S2).

C3  np(S1,S2) :- det(S1,S3), noun(S3,S2).
C4  np(S1,S2) :- det(S1,S3), adj(S3,S4), noun(S4,S2).

C5  vp(S1,S2) :- verb(S1,S2).
C6  vp(S1,S2) :- verb(S1,S3), prep(S3,S2).

det([a|S],S).
det([the|S],S).

adj([big|S],S).
adj([small|S],S).
adj([nasty|S],S).

noun([man|S],S).
noun([dog|S],S).
noun([house|S],S).
noun([ball|S],S).

verb([takes|S],S).
verb([walks|S],S).
verb([hits|S],S).

prep([at|S],S).
prep([to|S],S).
prep([on|S],S).
prep([in|S],S).
prep([into|S],S).
```

Figure 7.5: Complete Theory for Grammar Learning Example

The $\top$ theory and its transformed version for this example is given in appendix B.

In this $\top$ theory, there is no input list but a single variable. Because sentence is continuously[3] parsed. Therefore, binding variables is much simplified, thus no need to record all the variables or constants appeared before, but only the output variable of preceding atom.

Integrity Constraints:

1. Suffix requirement. For each predicate, its output should be suffix of its input.
2. The noun phrase must be related to noun, similarly, verb phrase should have relation with verb.
3. Predicate 's' is not directly related to noun and verb, since they are primitive and incorporated in noun phrases and verb phrases

Another constraint is about the length of hypothesized clause. In order to avoid directly parse the sentence words by words, its length is restricted to one less than that of original sentence.

### 7.2.2 Methods

This experiment is to test the ability of TDTD system to recover complete theory in fig 3.6 after a randomly chosen subset of it is left-out.

Performance was compared when randomly chosen subset of size 3, 6, 9, 12 were left out. For each size 10 randomly chosen left-out subsets were sampled and the results were averaged. Performance was measured on all the 33 examples[4] given in Progol5 which are randomly chosen. Leave-one-out cross validation was used to measure the predictive accuracy.

### 7.2.3 Results and Analysis

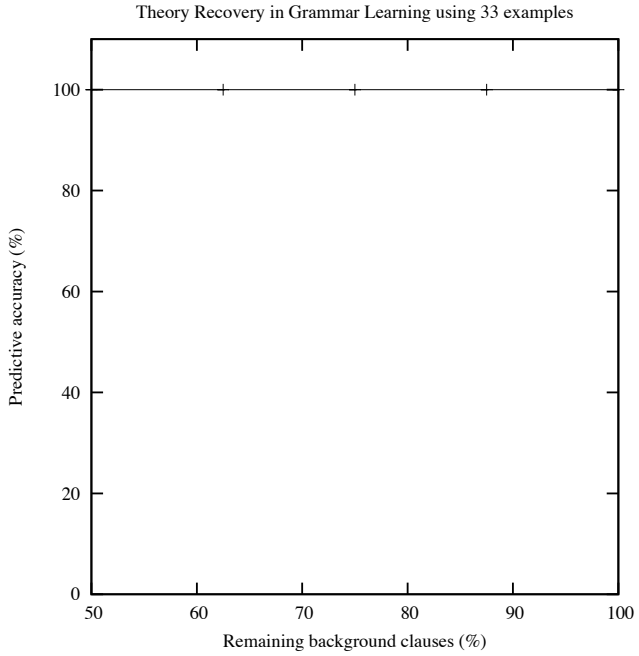


Figure 7.6: Predictive Accuracy for Grammar Learning Example

#### Predictive Accuracy

Fig7.6 gives the predictive accuracy. It is uniformly 100% for different sizes of left-out samples. It indicates that for all the samples, even when half is left-out, the missing theory can still be completely reconstructed. The 100% is due to sufficient training data. Although 33 is not a big

[3]For the atoms in the body of clause, its output is the input for the next one

[4]Examples in this experiment exclude the sentences that have word of plural form

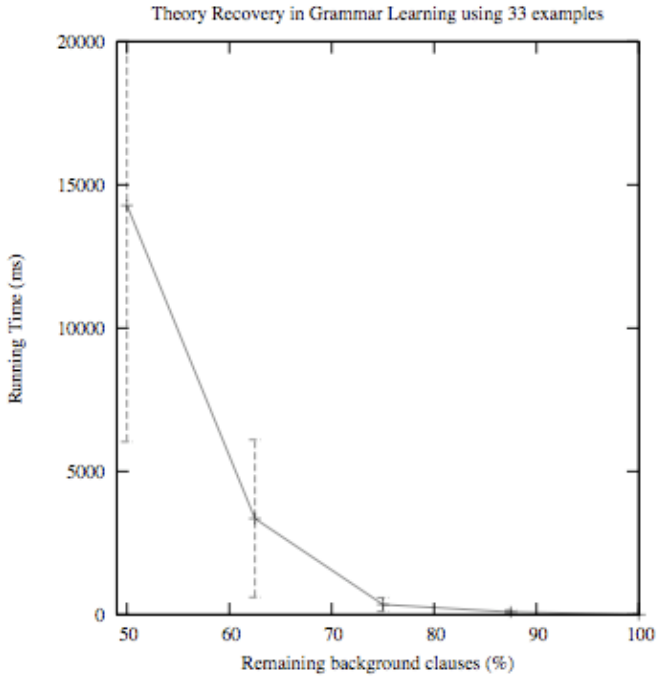


Figure 7.7: Running Time for Grammar Learning Example

number, but in terms of this simple language grammar, it is adequate. If the size of training data decreases, the predictive accuracy will drop down as well.

**Running Time**

Although the predictive accuracy does not vary among different sizes of left-out, the learning time has big difference. As shown in fig 7.7, the running time increases dramatically as more clauses were left-out. This results from the increasing search space. Even for the same size of left-out, the search space also vary greatly, as shown by the standard deviation in the fig 7.7.

Despite of the significant increase in running time, for all the samples in the experiment, the maximum one takes 36895ms (less than 40s). Therefore, this experiment demonstrates the ability of TDTD system to reconstruct the incomplete theory even when half is missing.

### 7.2.4 Further Discussion

Due to its 100% recovery even when half is left-out, you may wonder its performance when 3/4 are left-out or even all are left-out. As expected, when further more clauses are left out, the search space explodes. At the extreme case that all clauses are left-out, it led to out of stack in YAP interpreter, thus nearly incomputable.

The exploded space is not the only problem. It also tends to learn incorrect results when further more are left-out. Fig 7.8 give one example of incorrect learning results.

Among all the 3468 derived theories for the seed example in fig 7.8, there are $t_1$ and $t_2$. By explaining the phrase "walks to" as noun phrase, $t_1$ which is one clause less than the real correct one $t_2$, thus $t_1$ outperform $t_2$ according to MDL.

Although it remains to characterize the exact left-out boundary that TDTD is able to recover within reasonable running time, as well as give correct results, the fig 7.6 and fig 7.7 demonstrate its ability in reconstruct the whole theory when less than half is left-out.

**Original Incomplete Theory (5 clauses remained):**
det([a|S],S).
adj([big|S],S).
noun([man|S],S).
verb([takes|S],S).
prep([at|S],S).

**Seeded Example:** s([the,dog,walks,to,the,man],[]).

**t1**
s(A,B):- np(A,C),**np**(C,D),np(D,B).
np(E,F):- det(E,G),noun(G,F).
det([the|H],H).
**noun([to|I],I).**
**det([walks|J],J).**
noun([dog|K],K).

**t2**
s(A,B):- np(A,C),**vp**(C,D),np(D,B).
np(E,F):- det(E,G),noun(G,F).
**vp(I,J):- verb(I,K),prep(K,J).**
det([the|H],H).
**prep([to|L],L).**
**verb([walks|M],M).**
noun([dog|N],N)]].

Figure 7.8: Incorrect learning results

# Chapter 8

# Future Work and Conclusion

#### Generalization on multiple examples simultaneously

As shown in the experiment of odd-even example, the search space varies with different seed examples. Therefore, if generalizing on multiple examples simultaneously, then only their common ones are derivable, thus the search space will be further constrained.

#### Ordering in Search Space

Although the search space is effectively bound in TDTD, candidate theories are derived all at once and there is no ordering like subsumption lattice in the search space. Then efficient pruning discussed in section 2.3.2 is not applicable in TDTD.

Different from refinement operator whose operational result is decidable, the way that theories are derived in TDTD make it difficult to predict the effect of using certain clause in $\top$. In single-clause case, the length of clause is related to the depth of refutation, thus iterative deepening search seems to be a solution. However, in the setting of multi-clauses, it tends to derive more general theories when the refutation search goes deeper, which contradicts that in single-clause setting. In addition, extra control on refutation search will run at the risk of excluding target hypotheses, but considering the significant efficiency gained from pruning, finding a way to organize the search space in TDTD so that pruning can be applied is well worth exploration.

#### Beam Search

Greedy Search greedily chooses the candidate theory with maximum score, but problems arise when more than one candidate theories get the same score that is maximum. Current implementation simply chooses one of them, which will ignore other correct ones. One solution is to start beam search for the bunch of candidate theories with maximum score. Due to the huge search space in multi-clause, this will be quite expensive. However, to preserve the completeness of the whole system, this is indispensable.

#### Automatic Construction of $\top$ Theory

In TDTD, $\top$ theory is input to the system as background knowledge and training examples. Currently, it is specified by the user, but this is error-prone. While the $\top$ theory defines the search space, it is important to make sure that the $\top$ theory input to the system is correct. Thus an automatic construction of $\top$ theory is necessary. It will also take the burden off users.

#### Extension to full clausal logic

The learning problems dealt with so far are all definite clauses, thus the current system only support definite clause. In addition, efficiency can be gained for dealing with definite clauses compared to full clauses, accommodating full clausal logic is not considered in the current system. However, the

whole framework of TDTD is not restricted to definite clauses. It can be extended to full clauses by replacing SLD-resolution[8] with Model Elimination (ME)[13].

**Conclusion**

As shown by the experiments, multi-clauses problems can be learned in TDTD correctly and efficiently. As an example-driven method, it effectively bounds the search space. Also, benefiting from its top-directed framework, it does not suffer from the redundancy that results from upward theory refinement. In addition, its efficiency is not a trade-off for completeness. As proved in the report, TDTD is complete provided the given $\top$ theory is correct.

Its ability to learn multi-clauses problems makes it naturally integrates abduction and induction into the same phase, so that both abductive hypothesis and inductive hypothesis can be learned at that same time.

# Bibliography


[1] H. Boström and P. Idestam-Almquist. Specialisation of logic programs by pruning SLD-trees. In S. Wrobel, editor, *Proceedings of the Fourth Inductive Logic Programming Workshop (ILP94)*, pages 31–48, Bonn, 1994. GDM-studien Nr. 237.

[2] I. Bratko. Refining complete hypotheses in ILP. In *Proc. of the 9th International Workshop on Inductive Logic Programming (ILP 99)*, pages 44–55, Berlin, 1999. Springer-Verlag.

[3] W. Bridewell and L. Todorovski. Learning declarative bias. In *ILP07*, pages 63–77, 2007.

[4] W. Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36:375–399, 1988.

[5] W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, 1994.

[6] P.A. Flach and A.C. Kakas. *Abduction and Induction: Essays on their relation and integration*. Kluwer Academic Publishers, 2000.

[7] K. Iwanuma H. Nabeshima and K. Inoue. Solar: A consequence nding system for advanced reasoning. In *Proceedings of Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX 2003)*, page 257263, 2003.

[8] K INOUE. Linear resolution for consequence finding. *Artificial Intelligence*, 56:301–353, 1992.

[9] K Inoue. Induction as consequence finding. *Machine Learning*, 55:109–135, 2004.

[10] K. Furukawa K. Inoue1 and I. Kobayashi. Abducing rules with predicate invention. In *ILP09*, 2009.

[11] S.T. Kedar-Cabelli and L.T. McCarty. Explanation-based generalization as resolution theorem proving. In P. Langley, editor, *Proceedings of the Fourth International Workshop on Machine Learning*, pages 383–389, Los Altos, 1987. Morgan Kaufmann.

[12] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[13] D. W. Loveland. Mechanical theorem proving by model elimination. *Journal of the Association for Computing Machinery*, 15(2):236–251, 1968.

[14] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982/3.

[15] Tom M. Mitchell. *Machine Learning*. Mcgraw-Hill, 1993.

[16] S.H. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[17] S.H. Muggleton and C.H. Bryant. Theory completion using inverse entailment. In *Proc. of the 10th International Workshop on Inductive Logic Programming (ILP-00)*, pages 130–146, Berlin, 2000. Springer-Verlag.

[18] S.H. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[19] S.H. Muggleton, J. Santos, and A. Tamaddoni-Nezhad. Toplog: ILP using a logic program declarative bias. In *Proc. of the 24th International Conference on Logic Programming (ICLP)*, 2008.

[20] S-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag, Berlin, 1997. LNAI 1228.

[21] G.D. Plotkin. A further note on inductive generalization. In *Machine Intelligence*, volume 6, pages 101–124. Edinburgh University Press, 1971.

[22] Oliver Ray. Hail: Hybrid abductive inductive learning. Technical report, Department of Computing, Imperial College London, July 2003.

[23] Oliver Ray, Krysia Broda, and Alessandra Russo. Hybrid abductive inductive learning: A generalisation of progol. In T. Horvath and A. Yamamoto, editors, *13th International Conference on Inductive Logic Programming*, pages 311–328. Springer, October 2003.

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

[25] K. Broda T. Kimber and A. Russo. Induction on failure: Learning connected horn theories. In *Induction on Failure: Learning Connected Horn Theories*, pages 169–181, Berlin, 2009. Springer-Verlag.

[26] A. Tamaddoni-Nezhad, R. Chaleil, A. Kakas, M. Sternberg, J. Nicholson, and S.H. Muggleton. Modeling the effects of toxins in metabolic networks. *IEEE Engineering in Medicine and Biology*, 26:37–46, 2007.

[27] A. Tamaddoni-Nezhad, A. Kakas, S.H. Muggleton, and F. Pazos. Modelling inhibition in metabolic pathways through abduction and induction. In *Proceedings of the 14th International Conference on Inductive Logic Programming*, LNAI 3194, pages 305–322. Springer-Verlag, 2006.

[28] A. Yamamoto. Which hypotheses can be found with inverse entailment? In N. Lavrač and S. Džeroski, editors, *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, pages 296–308. Springer-Verlag, Berlin, 1997. LNAI 1297.

[29] Y. Yamamoto, K. Inoue, and K. Iwanuma. Hypothesis enumeration by CF-induction. In *Proceedings of Sixth Workshop on Learning with Logics and Logics for Learning (LLLL2009)*, 2009.

[30] Y. Yamamoto, O. Ray, and K. Inoue. Towards a logical reconstruction of CF-induction. In K Satoh, A Inokuchi, K Nagao, and T Kawamura, editors, *NEW FRONTIERS IN ARTIFICIAL INTELLIGENCE*, volume 4914 of *LECTURE NOTES IN ARTIFICIAL INTELLIGENCE*, pages 330–343. SPRINGER-VERLAG BERLIN, 2008.

# Appendices

# Appendix A

# Input Files for Odd-Even Example

```
/***************** TOP THEORY ******************/
% for reconstruct H
topCla(10,[odd(X),$bind(X,Y),$body(Y)]).
topCla(20,[even(X),$bind(X,Y),$body(Y)]).
topCla(1,[$body(X),odd(X)]).
topCla(2,[$body(X),even(X)]).
topCla(nt0,[$body(X)]).
topCla(nt1,[$bind(0,0)]).
topCla(nt2,[$bind(X,X)]).
topCla(nt3,[$bind(s(Y),Z), $bind(Y,Z)]).


/*********BACKGROUND KNOWLEDGE *************/
:- dynamic even/1, odd/1.      % declare as dynamic predicate for later assertion
%-----------------  empty  --------------------%


/****************EXAMPLES****************/
ex(1,even(PianoNum),1):- numMap(2,PianoNum).
ex(2,odd(PianoNum),1):- numMap(3,PianoNum).
ex(3,even(PianoNum),1):- numMap(4,PianoNum).
ex(4,odd(PianoNum),1):- numMap(5,PianoNum).
ex(5,even(PianoNum),1):- numMap(6,PianoNum).
ex(6,odd(PianoNum),1):- numMap(7,PianoNum).
ex(7,even(PianoNum),1):- numMap(8,PianoNum).
ex(8,odd(PianoNum),1):- numMap(9,PianoNum).

ex(9,odd(PianoNum),0):- numMap(2,PianoNum).
ex(10,even(PianoNum),0):- numMap(3,PianoNum).
ex(11,odd(PianoNum),0):- numMap(4,PianoNum).
ex(12,even(PianoNum),0):- numMap(5,PianoNum).
ex(13,odd(PianoNum),0):- numMap(6,PianoNum).
ex(14,even(PianoNum),0):- numMap(7,PianoNum).
ex(15,odd(PianoNum),0):- numMap(8,PianoNum).
ex(16,even(PianoNum),0):- numMap(9,PianoNum).


%       PIANO NUMBER MAPPING
%       numMap(integer,pianoNum)
numMap(0,0):- !.       % to avoid -1,-2...
numMap(D,s(P)):-
        D > 0,
        D0 is D-1,
        numMap(D0,P).
```

Figure A.1: Input File for Odd-Even Example

TRANSFORMED VERSION

```
/***************** TOP THEORY ******************/

odd(TrSoFar,[Hr|Tr],X):-
        $bind([10],Hr1,X,Y),
        Hr1\==[nt2,10], % prune odd(X):-odd(X).and odd(X):- even(X).
        $body(TrSoFar,Tr,Hr1,Hr,[s(0),s(0)],Y).

even(TrSoFar,[Hr|Tr],X):-
        $bind([20],Hr1,X,Y),
        Hr1\==[nt2,20], % prune even(X):-even(X).and even(X):- odd(X).
        $body(TrSoFar,Tr,Hr1,Hr,[s(0),s(0)],Y).

$body(TrSoFar,Tr,Hr,[1|Hr],[N1,s(N2)],X):-
        Hr=[Last|_],Last\==nt1,
        odd(TrSoFar,Tr,X).

$body(TrSoFar,Tr,Hr,[2|Hr],[s(N1),N2],X):-
        Hr=[Last|_],Last\==nt1,
        even(TrSoFar,Tr,X).


$body(Tr,Tr,Hr,[nt0|Hr],_,X).

$bind(Hr,[nt1|Hr],0,0).
$bind(Hr,[nt2|Hr],X,X).
$bind(HrSoFar,Hr,s(Y),Z):-
        $bind([nt3|HrSoFar],Hr,Y,Z).


/***************** BACKGROUND KNOWLEDGE ******************/
:- dynamic even/1, odd/1.  % declare as dynamic predicate for later assertion
%------------------  empty  --------------------%


/***************EXAMPLES***************/
gEx(1,even([],Tr,PianoNum)):- numMap(2,PianoNum).
gEx(2,odd([],Tr,PianoNum)):- numMap(3,PianoNum).
gEx(3,even([],Tr,PianoNum)):- numMap(4,PianoNum).
gEx(4,odd([],Tr,PianoNum)):- numMap(5,PianoNum).
gEx(5,even([],Tr,PianoNum)):- numMap(6,PianoNum).
gEx(6,odd([],Tr,PianoNum)):- numMap(7,PianoNum).
gEx(7,even([],Tr,PianoNum)):- numMap(8,PianoNum).
gEx(8,odd([],Tr,PianoNum)):- numMap(9,PianoNum).
```

Figure A.2: Transformed Input File for Odd-Even Example

# Appendix B

# Input Files for Grammar Learning

```
/***************** TOP THEORY ******************/
% for reconstruct H
topCla(10,[s(X,Y),body(X,Y)]).
topCla(20,[np(X,Y),body(X,Y)]).
topCla(30,[vp(X,Y),body(X,Y)]).

topCla(11,[body(X,FinalOut),np(X,Y),body(Y,FinalOut)]).
topCla(12,[body(X,FinalOut),vp(X,Y),body(Y,FinalOut)]).
topCla(13,[body(X,FinalOut),det(X,Y),body(Y,FinalOut)]).
topCla(14,[body(X,FinalOut),adj(X,Y),body(Y,FinalOut)]).
topCla(15,[body(X,FinalOut),noun(X,Y),body(Y,FinalOut)]).
topCla(16,[body(X,FinalOut),verb(X,Y),body(Y,FinalOut)]).
topCla(17,[body(X,FinalOut),prep(X,Y),body(Y,FinalOut)]).
topCla(18,[body(X,FinalOut),conj(X,Y),body(Y,FinalOut)]).

topCla(a1-X,[det([X|S],S)]).
topCla(a2-X,[adj([X|S],S)]).
topCla(a3-X,[noun([X|S],S)]).
topCla(a4-X,[verb([X|S],S)]).
topCla(a5-X,[prep([X|S],S)]).
topCla(a6-X,[conj([X|S],S)]).

topCla(nt0,[body(FinalOut,FinalOut)]).


/***************** BACKGROUND KNOWLEDGE ******************/
:- dynamic s/2, np/2, vp/2, det/2, adj/2, noun/2, verb/2, prep/2, conj/2.

%  23 in total % comment it out if it is left-out in the experiments
s(S1,S2) :- np(S1,S3), vp(S3,S4), np(S4,S2). %[0,11,12,11,10]
s(S1,S2) :- np(S1,S3), vp(S3,S4), np(S4,S5), prep(S5,S6), np(S6,S2).
np(S1,S2) :- det(S1,S3), noun(S3,S2).
np(S1,S2) :- det(S1,S3), adj(S3,S4), noun(S4,S2). % [0,15,14,13,20]
vp(S1,S2) :- verb(S1,S2). %[0,16,30]
vp(S1,S2) :- verb(S1,S3), prep(S3,S2).

det([a|S],S).
det([the|S],S).

adj([big|S],S).
adj([small|S],S).
adj([nasty|S],S).

noun([man|S],S).
noun([dog|S],S).
noun([house|S],S).
noun([ball|S],S).

verb([takes|S],S).
verb([walks|S],S).
verb([hits|S],S).
```

```
prep([at|S],S).
prep([to|S],S).
prep([on|S],S).
prep([in|S],S).
prep([into|S],S).



/***************EXAMPLES***************/
ex(1,s([the,dog,walks,to,the,man],[]),1).
ex(2,s([the,man,walks,the,dog],[]),1).
ex(3,s([a,dog,hits,a,ball],[]),1).
ex(4,s([the,man,walks,in,the,house],[]),1).
ex(5,s([the,man,walks,into,the,house],[]),1).
ex(6,s([the,man,hits,the,dog],[]),1).
ex(7,s([a,ball,hits,the,dog],[]),1).
ex(8,s([the,dog,walks,on,the,house],[]),1).
ex(9,s([the,man,hits,at,the,ball],[]),1).
ex(10,s([the,big,man,hits,at,the,ball],[]),1).
ex(11,s([the,small,dog,walks,on,the,house],[]),1).
ex(12,s([the,small,dog,walks,in,the,house],[]),1).
ex(13,s([the,small,dog,walks,into,the,house],[]),1).
ex(14,s([the,small,man,hits,the,dog],[]),1).
ex(15,s([the,big,man,hits,the,dog],[]),1).
ex(16,s([a,ball,hits,the,small,dog],[]),1).
ex(17,s([the,nasty,man,hits,the,dog],[]),1).
ex(18,s([the,man,hits,the,nasty,dog],[]),1).

% More complex positive examples.
ex(19,s([a,man,hits,the,ball,at,the,dog],[]),1).
ex(20,s([the,man,hits,the,ball,at,the,house],[]),1).
ex(21,s([the,man,takes,the,dog,to,the,ball],[]),1).
ex(22,s([a,man,takes,the,ball,to,the,house],[]),1).
ex(23,s([the,dog,takes,the,ball,to,the,house],[]),1).
ex(24,s([the,dog,takes,the,ball,to,the,man],[]),1).
ex(25,s([the,man,hits,the,ball,to,the,dog],[]),1).
ex(26,s([the,man,walks,the,dog,to,the,house],[]),1).

ex(27,s([a,dog,walks,the],[]),0).
ex(28,s([a,man,walks,the],[]),0).
ex(29,s([a,man,walks,the,walks],[]),0).
ex(30,s([a,man,walks,the,house,a],[]),0).
ex(31,s([a,man,walks,the,dog,at],[]),0).
ex(32,s([the,man,walks,the,dog,to,the],[]),0).
ex(33,s([the,dog],[]),0).
```

TRANSFORMED VERSION

```
/***************** TOP THEORY ******************/

s(TrSoFar,[Hr|Tr],S1,S2):-
        length(S1,K), numMap(K,LengthLimit),     % constraint on clause length
        $body(TrSoFar,Tr,[10],Hr,LengthLimit,[s(s(s(0))),s(s(0)),s(s(0)),s(s(0)),0,0,s(s(0)),s(s(0))],S1,S2).
        % disallow connect to noun,verb

np(TrSoFar,[Hr|Tr],S1,S2):-
        numMap(5,LengthLimit),
        $body(TrSoFar,Tr,[20],Hr,LengthLimit,[0,0,s(0),s(0),s(0),0,s(0),s(0)],S1,S2),
        % disallow connect to np(11),vp(12),verb(16)
        member(15,Hr). % np should include noun

vp(TrSoFar,[Hr|Tr],S1,S2):-
        numMap(5,LengthLimit),
        $body(TrSoFar,Tr,[30],Hr,LengthLimit,[0,0,s(0),s(0),0,s(0),s(0),s(0)],S1,S2),
        %disallow connect to np(11),vp(12),noun(15)
        member(16,Hr). % vp should include verb

$body(TrSoFar,Tr,HrSoFar,Hr,s(LengthLimit),[s(N1),N2,N3,N4,N5,N6,N7,N8],PreOut,FinalOut):-
        np(TrSoFar,Tr1,PreOut,OutVar),
        $body(Tr1,Tr,[11|HrSoFar],Hr,LengthLimit,[N1,N2,N3,N4,N5,N6,N7,N8],OutVar,FinalOut).


$body(TrSoFar,Tr,HrSoFar,Hr,s(LengthLimit),[N1,s(N2),N3,N4,N5,N6,N7,N8],PreOut,FinalOut):-
        vp(TrSoFar,Tr1,PreOut,OutVar),
        $body(Tr1,Tr,[12|HrSoFar],Hr,LengthLimit,[N1,N2,N3,N4,N5,N6,N7,N8],OutVar,FinalOut).

$body(TrSoFar,Tr,HrSoFar,Hr,s(LengthLimit),[N1,N2,s(N3),N4,N5,N6,N7,N8],PreOut,FinalOut):-
        det(TrSoFar,Tr1,PreOut,OutVar),
        $body(Tr1,Tr,[13|HrSoFar],Hr,LengthLimit,[N1,N2,N3,N4,N5,N6,N7,N8],OutVar,FinalOut).

$body(TrSoFar,Tr,HrSoFar,Hr,s(LengthLimit),[N1,N2,N3,s(N4),N5,N6,N7,N8],PreOut,FinalOut):-
        adj(TrSoFar,Tr1,PreOut,OutVar),
        $body(Tr1,Tr,[14|HrSoFar],Hr,LengthLimit,[N1,N2,N3,N4,N5,N6,N7,N8],OutVar,FinalOut).

$body(TrSoFar,Tr,HrSoFar,Hr,s(LengthLimit),[N1,N2,N3,N4,s(N5),N6,N7,N8],PreOut,FinalOut):-
        noun(TrSoFar,Tr1,PreOut,OutVar),
        $body(Tr1,Tr,[15|HrSoFar],Hr,LengthLimit,[N1,N2,N3,N4,N5,N6,N7,N8],OutVar,FinalOut).

$body(TrSoFar,Tr,HrSoFar,Hr,s(LengthLimit),[N1,N2,N3,N4,N5,s(N6),N7,N8],PreOut,FinalOut):-
        verb(TrSoFar,Tr1,PreOut,OutVar),
        $body(Tr1,Tr,[16|HrSoFar],Hr,LengthLimit,[N1,N2,N3,N4,N5,N6,N7,N8],OutVar,FinalOut).

$body(TrSoFar,Tr,HrSoFar,Hr,s(LengthLimit),[N1,N2,N3,N4,N5,N6,s(N7),N8],PreOut,FinalOut):-
        prep(TrSoFar,Tr1,PreOut,OutVar),
        $body(Tr1,Tr,[17|HrSoFar],Hr,LengthLimit,[N1,N2,N3,N4,N5,N6,N7,N8],OutVar,FinalOut).

$body(Tr,Tr,Hr,[nt0|Hr],s(_),_,FinalOut,FinalOut).
```

```
/***************** BACKGROUND KNOWLEDGE ******************/
% comment it out if it is left-out
s(TrSoFar,Tr,S1,S2) :- np(TrSoFar,Tr1,S1,S3), vp(Tr1,Tr2,S3,S4), np(Tr2,Tr,S4,S2).
s(TrSoFar,Tr,S1,S2) :- np(TrSoFar,Tr1,S1,S3), vp(Tr1,Tr2,S3,S4), np(Tr2,Tr3,S4,S5),
                       prep(Tr3,Tr4,S5,S6), np(Tr4,Tr,S6,S2).
np(TrSoFar,Tr,S1,S2) :- det(TrSoFar,Tr1,S1,S3), noun(Tr1,Tr,S3,S2).
np(TrSoFar,Tr,S1,S2) :- det(TrSoFar,Tr1,S1,S3), adj(Tr1,Tr2,S3,S4), noun(Tr2,Tr,S4,S2).
vp(TrSoFar,Tr,S1,S2) :- verb(TrSoFar,Tr,S1,S2).
vp(TrSoFar,Tr,S1,S2) :- verb(TrSoFar,Tr1,S1,S3), prep(Tr1,Tr,S3,S2).


det(Tr,Tr,[a|S],S):- !.
det(Tr,Tr,[the|S],S):- !.
det(Tr,[[a1-X]|Tr],[X|S],S):-
        \+word(X),              % still need for avoid re-parsing known word
        (member([ClaID-X],Tr)->
                ClaID==a1;
                Foo=1           % if not appear before, then no constraint
        ).

adj(Tr,Tr,[big|S],S):- !.
adj(Tr,Tr,[small|S],S):- !.
adj(Tr,Tr,[nasty|S],S):- !.
adj(Tr,[[a2-X]|Tr],[X|S],S):-
        \+word(X),
        (member([ClaID-X],Tr)->
                ClaID==a2;
                Foo=1           % if not appear before, then no constraint
        ).

noun(Tr,Tr,[man|S],S):- !.
noun(Tr,Tr,[dog|S],S):- !.
noun(Tr,Tr,[house|S],S):- !.
noun(Tr,Tr,[ball|S],S):- !.
noun(Tr,[[a3-X]|Tr],[X|S],S):-
        \+word(X),
        (member([ClaID-X],Tr)->
                ClaID==a3;
                Foo=1           % if not appear before, then no constraint
        ).

verb(Tr,Tr,[takes|S],S):- !.
verb(Tr,Tr,[walks|S],S):- !.
verb(Tr,Tr,[hits|S],S):- !.
verb(Tr,[[a4-X]|Tr],[X|S],S):-
        \+word(X),
        (member([ClaID-X],Tr)->
                ClaID==a4;
                Foo=1           % if not appear before, then no constraint
        ).
```

```
prep(Tr,Tr,[at|S],S):- !.
prep(Tr,Tr,[to|S],S):- !.
prep(Tr,Tr,[on|S],S):- !.
prep(Tr,Tr,[in|S],S):- !.
prep(Tr,Tr,[into|S],S):- !.
prep(Tr,[[a5-X]|Tr],[X|S],S):-
        \+word(X),
        (member([ClaID-X],Tr)->
                ClaID==a5;
                Foo=1           % if not appear before, then no constraint
        ).

% known word
word(X):- det([X|S],S).
word(X):- conj([X|S],S).
word(X):- adj([X|S],S).
word(X):- prep([X|S],S).
word(X):- noun([X|S],S).
word(X):- verb([X|S],S).


/****************EXAMPLES***************/
gEx(1,s([],Tr,[the,dog,walks,to,the,man],[])).
gEx(2,s([],Tr,[the,man,walks,the,dog],[])).
gEx(3,s([],Tr,[a,dog,hits,a,ball],[])).
gEx(4,s([],Tr,[the,man,walks,in,the,house],[])).
gEx(5,s([],Tr,[the,man,walks,into,the,house],[])).
gEx(6,s([],Tr,[the,man,hits,the,dog],[])).
gEx(7,s([],Tr,[a,ball,hits,the,dog],[])).
gEx(8,s([],Tr,[the,dog,walks,on,the,house],[])).
gEx(9,s([],Tr,[the,man,hits,at,the,ball],[])).
gEx(10,s([],Tr,[the,big,man,hits,at,the,ball],[])).
gEx(11,s([],Tr,[the,small,dog,walks,on,the,house],[])).
gEx(12,s([],Tr,[the,small,dog,walks,in,the,house],[])).
gEx(13,s([],Tr,[the,small,dog,walks,into,the,house],[])).
gEx(14,s([],Tr,[the,small,man,hits,the,dog],[])).
gEx(15,s([],Tr,[the,big,man,hits,the,dog],[])).
gEx(16,s([],Tr,[a,ball,hits,the,small,dog],[])).
gEx(17,s([],Tr,[the,nasty,man,hits,the,dog],[])).
gEx(18,s([],Tr,[the,man,hits,the,nasty,dog],[])).

% More complex positive examples.
gEx(19,s([],Tr,[a,man,hits,the,ball,at,the,dog],[])).
gEx(20,s([],Tr,[the,man,hits,the,ball,at,the,house],[])).
gEx(21,s([],Tr,[the,man,takes,the,dog,to,the,ball],[])).
gEx(22,s([],Tr,[a,man,takes,the,ball,to,the,house],[])).
gEx(23,s([],Tr,[the,dog,takes,the,ball,to,the,house],[])).
gEx(24,s([],Tr,[the,dog,takes,the,ball,to,the,man],[])).
gEx(25,s([],Tr,[the,man,hits,the,ball,to,the,dog],[])).
gEx(26,s([],Tr,[the,man,walks,the,dog,to,the,house],[])).
```

Figure B.1: Input File for Grammar Learning Example