Imperial College London
Department of Computing

Inductive Logic Programming with Normal Programs

Haodan Li

September 6, 2013

Supervised by Krysia Broda & Timothy Kimber

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

Acknowledgement

I am greatly indebted to the following to their contribution to this work:

- Krysia Broda, who supervised this work, for her helpful criticism and indispensable guidance on how to approach this research, answering my endless awkward questions, and always have time for me despite her busy schedule - makes of a great superior.
- Timothy Kimber, who supervised me as well. Without his time, patience, valuable advices and intriguing point of views, I would face many more difficulties while understanding the topic and doing the this research.
- Zhiyong Sun, a friend of mine, for his time on helping me check examples and debug my implementation.
- My parents, without them I will not have the ability to support my study in Imperial College this year.

Abstract

Inductive logic programming (ILP) is a field of machine learning and Artificial Intelligence concerned with generalisation of positive and negative examples with respect to the background knowledge. Most existing ILP systems handle with classical clausal programs, especially Horn logic programs, and has limited applications on ILP problems with non-monotonic logic programs. A recent Horn theory learning procedure Induction on Failure (IoF) realizes that using bridge formulas with multiple hierarchically related clauses in ILP can result in better performance over using simple Inverse Entailment (IE) bridge formulas. This report provides a new ILP proof procedure Induction with Completion and Connected Theories (ICC) which is an extension of IoF and can be applied on ILP tasks with normal program settings. The ICC procedure is designed based on a new operational ILP approach α-Connected Theory Generalisation which is defined and proved in this report. A preliminary implementation *icc.pl* for ICC procedure is provided and tested. The result shows the system works well on ILP tasks with small problem size.

Contents

List of Figures

List of Algorithms

CHAPTER 1

Introduction

Inductive Logic Programming (ILP) [20] is an important machine learning technique concerned with learning first order (or relational) rules [17, 20] that explain the given examples relative to some background knowledge. Different from other forms of machine learning, ILP delivers an approach to use expressive logical languages to represent the background knowledge. This feature of ILP has benefited many knowledge discovery areas that have plenty of sources of background knowledge [22].

1.1 Motivation and Objectives

Many learning systems use the Inverse Entailment (IE) approaches to learn Horn theories [11, 17, 21, 25]. These systems compute the solution in a bottom-up manner which firstly compute a most specific hypothesis then search through formulas subsume it. Instead of assuming every clause in a hypothesis H must individually explain at least one example E [21] or each clause must be responsible for at least one abducible Δ from the example E [25], the recent ILP proof procedures Induction on Failure (IoF) [11] allows multiple connected clauses to explain one example. Connected means body conditions in some clauses are explained by some other clauses. The IoF systems, therefore, allows the computation of hypotheses with in a larger search space.

The IoF procedure has four main steps. Given the background knowledge B which is a definite program, and sets of definite clause examples E^+ and E^-, the first step for IoF is to select a seed example e^+ from E^-. Then, it computes a set of ground definite (connected) clauses T such that $B \cup E$ explains e^+. The third step is to perform a subsumption search on T to generate a more general hypothesis T from T. Lastly, it removes the examples in $E+$ that are explained by H and start the procedure with remainders in E^+ again. The loop stops when all the examples in the original E^+ are explained.

This monotonic IoF procedure is designed to learn from definite logic programs and may fail to learn hypotheses accurately on normal programs. Consider the following example:

Example 1.1. [15]

$$B = \left\{ \begin{array}{l} obeys(X,Y) \leftarrow \neg officer(X), officer(Y) \\ wears\_hat(price) \\ wears\_hat(osbourne) \\ has\_stripes(osbourne) \end{array} \right\}$$

$$U = \{officer\}$$

$$E = \left\{ \; obeys(price, osbourne) \; \right\}$$

$$H_{\perp} = \{ \; officer(osbourne) \leftarrow wears\_hat(osbourne), has\_stripes(osbourne) \; \}$$

$$H' = \{ \; officer(X) \leftarrow wears\_hat(X) \; \}$$

B is a normal logic program represents the background knowledge. U is a set of open predicates which indicates what predicates are not completely defined by the background B and therefore can be learned in the hypothesis. E is an example need to explain by the hypothesis. $H_{\perp}$ is the most specific ground hypothesis can be learned from IoF. H' is a possible generalisation of $H_{\perp}$ and can possibly returned by the IoF procedure. However, the clause in H' failed to explain $obeys(price, osbourne)$ in E since it makes both $officer(osburne)$ and $officer(price)$ true.

This is mainly because IE is based on deduction theorem, but it is known that the deduction theorem does not hold in non-monotonic logics in general [29]. The incremental learning in IoF procedure and generalisation technique used in IoF will not be sound in normal program settings. The subsumption search step in IoF system assumes the most specific ground hypotheses $H_{\perp}$ can be freely extended with additional assumptions. This makes the system unsuccessful to find the best hypotheses H for the non-monotonic problem if the hypothesis H generalised from $H_{\perp}$ does not capture the relative negative information from examples.

There are several ILP systems which allows background theories and hypotheses to be nomal logic programs. For example, Corapi et al.[4] provides a top-down ILP system Top-directed Abductive Learning (TAL) that maps the ILP into an equivalent ALP (Abductive Logic Programming) programming and can learn correct hypotheses with appropriate top theory. Induction from Answer Set (IAS) [28] builds a bottom-up theory of induction from non-monotonic logic programs. It introduces an algorithm to construct hypotheses from the answer sets of an extended logic program. The eXtended Hybrid Abductive Inductive Learning (XHAIL) [26] also uses the stable model semantics to make the system works on normal programs. However, none of these systems or procedures systematically find a solution by us-

ing a bottom-up search and simultaneously allows clauses in the hypotheses to be connected. This project aims to provide a bottom-up procedure that is generalised from IoF procedure but which can learn *Connected Theories* [11] in a normal program setting.

1.2 Achievement

This report provides a new inductive proof procedure that can work on ILP problems with normal program settings and new operational ILP approach called α-*Connected Theory Generalisation*.

The clauses in the hypotheses that computed by the procedure provided in this report are allowed to be connected. Connected means the body conditions in some clauses in a hypothesis are explained by some other clauses in that hypothesis. For example, Given the following background knowledge B which is a normal program, a set of literal examples E, and a set of predicates to be learned in U,

Example 1.2.

$$B = \left\{ \begin{array}{l} obeys(X,Y) \leftarrow \neg officer(X), officer(Y) \\ wears\_hat(price) \\ wears\_hat(osbourne) \\ has\_merit(osbourne) \\ has\_stripes(X) \leftarrow has\_badge(X) \end{array} \right\},$$

$$E = \left\{ \; obeys(price, osbourne) \; \right\},$$

$$U = \left\{ \; officer, has\_badge \; \right\}.$$

Assume predicate *has_merit* and *has_stripes* are allowed to be in the bodies of clauses in the hypotheses, then the following H' and H are possible ground and flat hypotheses that can be computed by the procedure provided in this report. Note the hypothesis H in above example is subsumed by $officer(X) \leftarrow has\_merit(X)$, it may be preferred in some particular situation (e.g. when a hypothesis with more clauses but less number of body literals in each clause is preferred).

$$H' = \left\{ \begin{array}{l} officer(osbourne) \leftarrow has\_merit(osbourne), has\_stripes(osbourne) \\ has\_badge(osbourne) \leftarrow has\_merit(osbourne) \end{array} \right\}$$

$$H = \left\{ \begin{array}{l} officer(X) \leftarrow has\_stripes(X) \\ has\_badge(X) \leftarrow has\_merit(X) \end{array} \right\}$$

Here, in both H' and H, the second clause is the explanation for the body condition with predicate *has_stripe* in the first clause since there is a clause $has\_stripes(X) \leftarrow has\_badge(X)$ in the background B.

The new operational ILP approach α-*Connected Theory Generalisation* which is based on the notion of α-*Connected Theory* [15] is defined. The soundness of this approach is proved. It is a similar approach to *Connected Theory Generalisation* [15] and has been proved that

the hypothesis space of α-*Connected Theory Generalisation* is at least as large as *Connected Theory Generalisation*.

An preliminary implementation *icc.pl* is provided and is evaluated based on several small to medium ILP tasks with both definite and normal program settings. It shows the current procedure is able to solve small ILP tasks but the scalablity, generating floundered hypotheses and incompleteness are issues remained to solve.

1.3 Overview

This rest part of this report is organised as follows:

Chapter 2 provides the necessary background information on logic programming including definite programs, normal programs and abductive logic programming.

Chapter 3 gives an overview of Inductive Logic Programming with respect to its formal framework and terminologies, Inverse Entailment and two related works - XHAIL and Induction from Answer Set.

Chapter 4 gives the formal definition of a Connected Theory and Connected Theory Generalisation, describes the IOF proof procedure and provides definitions of a Normal Connected Theory and Normal Connected Theory Generalisation.

Chapter 5 gives an overview of the new proof procedure ICC.

Chapter 6 gives the formal definition and characterisation of a α-Connected Theory and α-Connected Theory Generalisation, and contains proofs of the soundness and a bottom line of its completeness.

Chapter 7 gives detailed algorithms of the new proof procedure ICC.

Chapter 8 gives a implementation *icc.pl* of ICC procedure and provides a evaluation on the procedure based on the implementation.

Chapter 9 concludes the work and gives the possible future work.

CHAPTER 2

Common background

This chapter introduces the logical terminologies that will be used throughout this report and describes how automated deductive and abductive reasoning is performed.

2.1 Definite Programs

Before giving the definition of the definite program, the following definitions for term, formula, literal and clause is firstly given. These are basic terminologies for logic programs and most of them can be found in Lloyd's book [16].

Definition 2.1. (First-Order Term [16]). *A term is defined as follows:*

- *A constant is a term.*
- *A variable is a term.*
- *If f is an n-ary function symbol and $t_1, \cdots, t_n$ are terms, then $f(t_1, \cdots, t_n)$ is a term.*

A ground term is a term which does not contain any variables.

Definition 2.2. (First-Order Formula [16]). *A (well-formed) formula is defined as follows:*

- *If p is an n-ary predicate symbol and $t_1, \cdots, t_n$ are terms, then $p(t_1, \cdots, t_n)$ is a formula, called an atom. A ground atom is an atom which does not contain any variables.*
- *If φ and φ are formulas, then $\neg\varphi$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \rightarrow \psi)$ and $(\varphi \leftrightarrow \psi)$ are formulas.*
- *If φ is a formula, and X is a variable, then $\exists X\varphi$ and $\forall X\varphi$ are formulas.*

Definition 2.3. (Literal [16]). *A literal is an atom or the negation of an atom. If A is an atom, then A is a positive literal and A is a negative literal.*

For example

Example 2.1.

$$p(1, a, X, s) \text{ and } q(f(c), 2, X) \text{ are atoms}$$

and

$$p(1, a, X, s),\ \neg p(1, a, X, s),\ q(f(c), 2, X) \text{ and } \neg q(f(c), 2, X) \text{ are literals.}$$

Definition 2.4. (Clause [16]). *A clause is a finite disjunction of zero or more literals. The clause with zero literals is called the empty clause and denoted* $\square$.

For example

Example 2.2.

$$\neg p(1, a, X, c) \vee q(f(c), 2, X) \vee t(2, b) \text{ is a clause.}$$

Definition 2.5. (Theory [16]). *A theory is a finite set of clauses.*

A definite program is set of definite clauses where a definite clause is a special from of clause, defines below:

Definition 2.6. (Definite Clause [16]). *A definite clause is a clause containing exactly one positive literal.*

For example.

Example 2.3.

$$\neg p(1) \vee \neg q(2) \vee t(f(b)) \text{ is a definite clause.}$$

Definition 2.7. (Definite Program [16]). *A definite program is a finite set of definite clauses.*

Definition 2.8. (Definition of a Predicate [16]). *Let p be a predicate symbol. The definition of p in a definite program* Π *is the set* Π_p *of all clauses in* Π *with p in the head. Then p is defined by* Π_p.

Definition 2.9. (Definite Goal [16]). *A definite goal is a clause of the form*

$$\leftarrow B_1, \cdots, B_k,$$

with an empty consequent. Each atom B_i $(1 \leq i \leq n)$ *is a subgoal of this goal.*

Definition 2.10. (Horn clause [16]). *A Horn clause is a clause that is either a definite clause or a definite goal.*

Selective Linear Definite clause (SLD) Resolution

SLD Resolution is a refinement of resolution, introduced by Kowalski [10]. The definitions for resolution are given below:

Definition 2.11. (Substitution [24]). *A substitution θ is a finite set of the form*

$$\{X_1/t_1, \cdots, X_n/t_n\},$$

where $\{X_1, \cdots, X_n\}$ are distinct variables and $\{t_1, \cdots, t_n\}$ are terms. Each t_i is substituted for X_i, and each X_i/t_i is a binding for X_i. The substitution θ is a ground substitution if every t_i is ground.

Definition 2.12. (Expression [24]). *An expression is a term, or a conjunction of literals, or disjunction of literals.*

Definition 2.13. (Instance [24]). *Let φ be an expression and θ be a substitution. Then $\varphi\theta$, the instance of φ by θ, is the expression obtained by applying θ to the variables of φ. If $\varphi\theta$ is ground, then $\varphi\theta$ is a ground instance and θ is a ground substitution for φ.*

Definition 2.14. (Unifier [24]). *A unifier for the set of expressions $\varphi_1, \cdots, \varphi_n$ is a substitution θ such that $\varphi_1\theta = \varphi_2\theta = \cdots = \varphi_n\theta$.*

Definition 2.15. (Most General Unifier [24]). *If θ is a unifier for a set of expressions Σ, and, for any unifier σ for Σ, there exists a substitution γ such that $\sigma = \theta\gamma$, then θ is called a most general unifier or mgu for Σ.*

Definition 2.16. (Factor [24]). *Let C be a clause, let $L_1, \cdots, L_n$ $(n \geq 1)$ be some unifiable literals in C and let θ be an mgu for the set $\{L_1, \cdots, L_n\}$. The clause obtained by deleting $\{L_2\theta, \cdots, L_n\theta\}$ from $C\theta$ is a factor of C.*

Definition 2.17. (Binary Resolvent [24]). *Let $C_1 = L_1 \vee \cdots \vee L_i \vee \cdots \vee L_m$, and let $C_2 = M_1 \vee \cdots \vee M_j \vee \cdots \vee M_n$ be two clauses which are standardized apart. If θ is an mgu for the set $\{L_i, M_j\}$, then the clause*

$$(L_1 \vee \cdots \vee L_{i-1} \vee L_{i+1} \vee \cdots \vee L_m \vee M_1 \vee \cdots \vee M_{j-1} \vee M_{j+1} \vee \cdots \vee M_n)\theta$$

is a binary resolvent of C_1 and C_2. The literals L_i and M_j are the literals resolved upon.

Definition 2.18. (Resolvent [24]). *Let C_1 and C_2 be two clauses. A resolvent C of C_1 and C_2 is a binary resolvent of a factor of C_1 and factor of C_2, where the literals resolved upon are the literals unified by the respective factors. C_1 and C_2 are called the parent clauses of C.*

The definition for SLD Derivation is given below:

Definition 2.19. (SLD Derivation [16]). *Let Π be a definite program and let G_0 and G_k be definite goals. An SLD derivation of G_k from $\cup\{G_k\}$ is a sequence $G_0 = G, G_1, \cdots, G_k$ of definite goals such that each $G_i (i > 0)$ is a binary resolvent of G_{i-1} and a definite clause $C_i \in \Pi$, using the head of C_i and a selected atom in G_i as the literals resolved upon.*

The clauses C_i are the input clauses of this SLD derivation. An SLD derivation of $\square$ from $\Pi \cup \{G\}$ is an SLD refutation of $\Pi \cup \{G\}$.

Definition 2.20. (SLD Derivation [16]). *Let* Π *be a definite program and let* G *be a definite goal. An SLD tree for* $\Pi \cup \{G\}$ *is a tree satisfying the following:*

1. *Each node of the tree is a (possibly empty) definite goal.*
2. *The root node is* G*.*
3. *Let* $N = \leftarrow A_1, \cdots, A_i, \cdots, A_k$ *be a node in the tree, where* A_i *is the selected atom. Then, for each clause* $A \leftarrow B_1, \cdots, B_l$ *in* Pi *such that* A_i *and* A *have mgu* θ*,* N *has a child*
$$\leftarrow (A_1, \cdots, A_{i-1}, B_1, \cdots, B_l, A_{i+1}, . \cdots, A_k)\theta.$$
4. *Empty nodes have no children.*

Branches of the tree ending in an empty node are success branches, branches ending in a non-empty node are failure branches and branches that do not terminate are infinite branches.

2.2 Normal Logic Programs

This section introduces the inference rule *Negation as Failure* (NAF) for negative literals, the *Normal Program*, the automated reasoning procedure *SLDNF Resolution Procedure* for normal programs and the *Stable Model Semantics*.

Negation as Failure

Closed World Assumption (CWA) is an inference rule introduced by Reiter [27], which allows us to draw negative conclusions based on the lack of positive informations. However, for a definite program Π and a definite goal $\leftarrow A$, the derivation from $\Pi \cup \{\leftarrow A\}$ may be infinite. For example

Example 2.4.

$$\Pi = \left\{ \begin{array}{l} f(a) \\ f(X) \leftarrow f(X) \end{array} . \right\}$$

$$A = f(c)$$

Thus to show that $\Pi \not\models A$ may take infinite number of steps. *Negation as Failure* (NAF) [3] is a weaker notion of CWA in which $\neg A$ can be inferred if there is a finitely failed SLD tree for $\leftarrow A$. NAF results in a non-classical semantics, since $\Pi \not\models A$ is not equivalent to $\Pi \models \neg A$. However. this can be easily implemented and allows negated atoms to be inferred.

Normal Programs

Since NAF allows $\neg A$ to be inferred, it is possible to include negated atoms in the body of a clause. Programs allowed to include such clauses are called *normal programs* and thus are more expressive than definite programs.

Definition 2.21. (Program Clause [16]). *A program clause is a formula of the form*

$$A \leftarrow L_1, \cdots, L_n,$$

where A is an atom and $L_1, \cdots, L_n$ are literals.

Definition 2.22. (Normal Program [16]). *A normal program Pi is a finite set of program clauses.*

Definition 2.23. (Normal Goal [16]). *A normal goal is a formula of the form*

$$\leftarrow L_1, \cdots, L_n,$$

where $L_1, \cdots, L_n$ are literals.

Although NAF would result in a non-classical semantic, the semantics of normal programs is correct with respect to the *completion* of the logic programs [3]. The semantics is captured by completing the definition of predicates in the program. The *completed definition*,

$compdef(p, \Pi)$ of a predicate p in the program Π is obtained by transforming the set of 'if' formulas into a single 'if and only if' formula.

Definition 2.24. (Clark's Equality Theory [12]). *The equality theory EQ comprises the following set of axioms:*

- $c \neq d$ *where* c *and* d *are distinct constants*
- $\forall(f(X_1, \cdots, X_n) \neq c)$ *where* f *is a function symbol and* c *is a constant*
- $\forall(f(X_1, \cdots, X_n) \neq g(Y_1, \cdots, Y_n))$ *where* f *and* g *are distinct function symbols*
- $\forall(t[X] \neq X)$ *where t[X] is any term containing* X *that is not* X
- $\forall(X = X)$
- $\forall((X_1 \neq Y_1) \vee \cdots \vee (X_n \neq Y_n) \rightarrow f(X_1, \cdots, X_n) \neq f(Y_1, \cdots, Y_n))$ *where f is a function symbol*
- $\forall((X_1 = Y_1) \wedge \cdots \wedge (X_n = Y_n) \rightarrow f(X_1, \cdots, X_n) = f(Y_1, \cdots, Y_n))$ *where f is a function symbol*
- $\forall((X_1 = Y_1) \wedge \cdots \wedge (X_n = Y_n) \rightarrow (p(X_1, \cdots, X_n) \rightarrow p(Y_1, \cdots, Y_n)))$ *where* p *is a predicate symbol*

Definition 2.25. (Completed Definition [3]). *Let* p *be an m-ary predicate, defined in the program* Π *by* k *clauses as follows:*

$$\begin{aligned} &p(t_{11}, ..., t_{1m}) \leftarrow B_1 \\ &\vdots \\ &p(t_{k1}, ..., t_{km}) \leftarrow B_k \end{aligned}$$

and for each B_i*, let the set of variables appearing in* B_i *that are not members of* $\{t_{i1}, ..., t_{im}\}$ *be* $\{Y_{11}, ..., Y_{1m}\}$*. The variables* $t_{11}, ..., t_{1m}$ *are local to* B_i*. The completed definition of* p *in* Π*, written* $compdef(p, \Pi)$*, is defined as follows:*

- *If* $k = 0$*, then*
$$compdef(p, \Pi) = \forall X_1 ... \forall X_m \neg p(X_1, ..., X_m).$$
- *If* $k > 0$*, then*
$$compdef(p, \Pi) = \forall X_1 ... \forall X_m (p(X_1, ..., X_m) \leftrightarrow (E_1 \vee ... \vee E_k)).$$

where $X_1, ..., X_m$ *are variables not appearing in any clause in the definition of* p*, and* E_i *is*

$$\exists Y_{i1} ... Y_{ij} (X_1 = t_{i1} \wedge ... \wedge X_m = t_{im} \wedge B_i).$$

Definition 2.26. (Set of Completed Definitions, [12]). *Let* Π *be a normal program, and let* P *be the set* $\{p_1, ..., p_n\}$ *of predicates. The set* $\{compdef(p_1, \Pi), ...,$ $compdef(p_n, \Pi)\}$ *of completed definitions of predicates in* P *is denoted by* $compdefs(P, \Pi)$.

Definition 2.27. (Completion [3]). *Let* Π *be a normal program. The completion of* Π, *denoted by* $comp(\Pi)$, *is the set of completed definitions of the predicate symbols in* Π, *together with the equality theory.*

For example, the *completion* of background knowledge B in Example 1.1 is

Example 2.5.

$$comp(B) = \left\{ \begin{array}{l} obeys(X,Y) \leftrightarrow \neg officer(X) \wedge fficer(Y) \\ wears\_hat(X) \leftrightarrow X = price \vee X = osburne \\ has\_stripes(X) \leftrightarrow X = osbourne \end{array} \right\} \cup \ EQ$$

Therefore, the purpose of logic programming with a normal program Π is to determine the formulas that are logical consequences of $comp(\Pi)$.

The procedure provided in this report is going to work on the ILP problems normal program settings and therefore can also apply on ILP problems with definite program settings.

SLDNF Resolution

SLDNF resolution is the SLD resolution with NAF. If a negative literal $\neg A$ is selected in a SLD derivation, then a new derivation for goal $\neg A$ is triggered. If there is a finitely failed SLDNF tree for $\neg A$, the original resolution procedure will remove $\neg A$ as a proved literal and continue. The detailed definitions are provided below.

Definition 2.28. (SLDNF Derivation [16]). *Let* Π *be a normal program and* G *be a normal goal. An SLDNF derivation of* $\Pi \cup \{G\}$ *consists of a (possibly infinite) sequence* $G_0 = G, G_1, \cdots$ *of normal goals, a sequence* $C_1, C_2, \cdots$ *of variants of clauses in* Π *or ground negative literals, and a sequence* $\theta_1, \theta_2, \cdots$ *of substitutions such that, for each* i, *either*

1. $G_i = \square$ *and* G_i *is the last goal in the sequence, or*
2. G_i *is* $\leftarrow L_1, \cdots, L_i, \cdots, L_m$, L_i *is selected,* L_i *is an atom, and either:*
   a) G_{i+1} *is derived from* G_i *and* C_{i+1} *using* $_{i+1}$, *or*
   b) *there is no clause in* Π *whose head unifies with* L_i *and* G_i *is the last goal in the sequence, or*
3. G_i *is* $\leftarrow L_1, \cdots, L_i, \cdots, L_m$, L_i *is selected,* L_i *is a negative literal, and either:*
   a) $L_i = \neg A_i$ *is ground and there is a finitely failed SLDNF tree for* $\Pi \cup \{\leftarrow A_i\}$. *In this case,* G_{i+1} *is* $\leftarrow L_1, \cdots, L_{i-1}, L_{i+1}, \cdots, L_m$, $_{i+1} = \varnothing$ *and* C_{i+1} *is* $\neg A_i$. *Or there is no clause in* Π *whose head unifies with* L_i *and* G_i *is the last goal in the sequence, or*

b) L_i is not ground, or $L_i = \neg A_i$ is ground and there is no finitely failed SLDNF tree for A_i. In this case, G_i is the last goal in the sequence.

Definition 2.29. (SLDNF Tree [16]). *Let Π be a normal program and G be a normal goal. An SLDNF tree for $\Pi \cup \{G\}$ is a tree satisfying the following:*

1. *Each node of the tree is a (possibly empty) normal goal.*
2. *The root node is G.*
3. *If a node $N = \square$, then N has no children.*
4. *Let $N = L1, \cdots, Li, \cdots, Lm$ be a non-empty node in the tree, with L_i selected. Then the child nodes of N are defined as follows.*
   a) If L_i is an atom, then for each clause $C = A \leftarrow M_1, \cdots, M_q$ in Π such that L_i and A have mgu θ, N has a child derived from N and C using θ. In this case, N has no other children.
   b) If L_i is a ground negative literal $\neg A_i$ and there is a finitely failed SLDNF tree for $\Pi \cup \{\neg A_i\}$, then N has a child $\leftarrow L_1, \cdots, L_{i-1}, L_{i+1}, \cdots, L_m$. In this case, N has no other children.
   c) If L_i is not an atom or a ground negative literal $\neg A_i$ such that there is a finitely failed SLDNF tree for $\Pi \cup \{\neg A_i\}$, then N has no children.

Definition 2.30. (Floundering [16]). *Let be a normal program and G be a normal goal. A derivation of $\Pi \cup \{G\}$ flounders if a goal is reached which contains only non-ground negative literals.*

Definition 2.31. (Safe Computation Rule [16]). *A safe computation rule is a function from a set G of normal goals, such that no goal in G consists entirely of non-ground negative literals, to a set of literals, such that the value of the function for each goal is either a positive literal or a ground negative literal, called the selected literal, in that goal.*

2.3 Stable Model Semantics

Stable Model Semantics is one of the most successful declarative semantics for logic programs with negations. Since the semantic is defined based on Herbrand model, the following definitions for Herbrand model is given first.

Definition 2.32. (Herbrand Universe [16]). *Let $\mathcal{L}$ be a first-order language. The Herbrand universe $U_{\mathcal{L}}$ for $\mathcal{L}$ is the set of all ground terms that can be formed from the constants and function symbols appearing in $\mathcal{L}$. In the case that $\mathcal{L}$ has no constants, then a constant is added in order to form ground terms.*

Definition 2.33. (Herbrand Base [16]). *Let $\mathcal{L}$ be a first-order language. The Herbrand base $B_{\mathcal{L}}$ for $\mathcal{L}$ is the set of all ground atoms that can be formed from the predicate symbols appearing in $\mathcal{L}$ and the terms in the Herbrand universe for $\mathcal{L}$.*

Definition 2.34. (Herbrand Pre-Interpretation [16]). *Let $\mathcal{L}$ be a first-order language. The Herbrand pre-interpretation for L is the pre-interpretation where:*

1. *the domain is the Herbrand universe $U^{\mathcal{L}}$;*
2. *all constants in $\mathcal{L}$ are mapped to themselves;*
3. *if f is an n-ary function symbol in $\mathcal{L}$, then f is assigned a mapping J_f from $U_{\mathcal{L}}^n$ to $U_{\mathcal{L}}$ such that $J_f(t) = f(t)$.*

Definition 2.35. (Herbrand Interpretation [16]). *Let $\mathcal{L}$ be a first-order language. A Herbrand interpretation of $\mathcal{L}$ is any interpretation based on the Herbrand pre-interpretation for $\mathcal{L}$.*

Definition 2.36. (Herbrand Model [16]). *Let $\mathcal{L}$ be a first-order language, Σ be a set of formulas in $\mathcal{L}$, and I be a Herbrand interpretation of $\mathcal{L}$. If I is a model of Σ, then I is a Herbrand model of Σ.*

Definitions for *Stable Model* and the implication under *Stable Model Semantics* is given bellow.

Definition 2.37. (Reduct of Program [6]). *Let Π be a normal program and let $M \subseteq U_\Pi$ be a set of atoms. The reduct of Π with respect to M is the ground definite program Π^M, obtained from the set of all ground instances of clauses in by deleting*

1. *each clause containing a negative literal $\neg A$ in its body where $A \in M$, and*
2. *all negative literals in the bodies of the remaining clauses.*

Definition 2.38. (Stable Model [6]). *Let Π be a normal program and let $M \subseteq U_\Pi$ be a set of atoms. Then M is a stable model of if and only if M is the least Herbrand model of Π^M.*

Definition 2.39. (Implication Under Stable Model Semantics [17]). *Let Π be a normal program such that Π has a unique stable model M, and let ϕ be a closed formula. If M satisfies ϕ then is implied by Π under the stable model semantics, denoted $\phi \models_{st} \phi$.*

2.4 Abductive Logic Program

Abductive Logic Program (ALP) extends normal logic program by allowing some predicates to be incompletely defined, namely abducible predicates. Given the knowledge base, some observations and constraints, the task of ALP is to derive some hypotheses on these abducible predicates such that, together with the knowledge base, these hypotheses explains those observations and satisfy the constraints. The formal framework of ALP is given in the Section 2.4. The Section 2.4 introduces the KakasMancarella ALP procedure which is used in the new ILP procedure proposed in this report.

Open Programs

Abduction is implemented in logic programming through an open logic program. An *open program* has three components $\langle \Pi, U, I \rangle$. Π is a logic program. U is a set of predicates that are not completely defined in Π. I is known as integrity constraints that constrain which fact can be abduced.

Definition 2.40. (Open Program [5]). *An open program is a triple $\langle \Pi, U, I \rangle$, where Π is a program, U is a set of predicates called undefined or abducible, and I is a set of first-order axioms. A ground literal with predicate $p \in U$ is called an abducible literal.*

The semantic of an open program P is given by the completion $comp(P)$, defined as following:

Definition 2.41. (Open Program Completion [1]). *Let $\mathcal{L}$ be a first-order language, let $Pred$ be the set of predicates in $\mathcal{L}$ and let $P = \langle \Pi, U, I \rangle$ be an open program in $\mathcal{L}$. The completion, $comp(P)$, of P is*

$$EQ \cup \{p(t_1, \cdots, t_n) \leftarrow \phi \in \Pi \mid p \in U\} \cup \{compdef(q, \Pi) \mid q \in Pred \wedge q \notin U\}.$$

In practice, it is often simpler for ALP task if none of the predicates in U are defined in Π. Such an open program is called *disjoint open program*.

Definition 2.42. (Disjoint Open Program [12]). *Let $P = \langle \Pi, U, I \rangle$ be an open program. If Π does not contain a definition of any predicate in U, then P is a disjoint open program.*

Moreover, An *open program* can be transformed into a *disjoint open program* by introducing auxiliary clauses and predicates [9]. For Example,

Example 2.6. *the open program P_1*

$$P_1 = \{\{p(1) \leftarrow q(1)\}, \{p\}, \varnothing\}$$

can be transformed into the disjoint open program P_2

$$P_2 = \big\{\{p(1) \leftarrow q(1),\ p(X) \leftarrow p'(X)\}, \{p'\}, \varnothing\big\}$$

by introducing a new predicate p'.

The Kakas-Mancarella Proof Procedure

The KakasMancarella (KM) proof procedure [24] implements abductive logic programming. This procedure interleaves two derivation phases - abductive derivation phase and consistency derivation phase that are modified from SLD resolution. The details of the procedure is not given here but can be found in [8].

Given an disjoint open program $P = \langle B, U, I \rangle$. The KM procedure starts with a list of goals G and returns a list of abductive explanations Δ for G. The Δ contains a set of positive literals with predicates in U and all negative literals that are assumed to be false. For example,

Example 2.7. *Given the disjoint open program $P = \langle B, \{q, r\}, \varnothing \rangle$, and goal set $G = \{p\}$*

$$B = \left\{ \begin{array}{l} p \leftarrow q, \neg r \\ r \leftarrow t \end{array} \right\}.$$

The result computed by the KM procedure would be

$$\Delta = \{q, \neg r, \neg t\}.$$

Although t is not an abducible predicate, it is also returned by the KM procedure.

The KM procedure has several constraints, called KM Safe Computation Rule, on the goals list at each iteration when running. Since the procedure in the Section 7 is designed and implemented based on the KM procedure, the definition of these constraints are give below:

Definition 2.43. (KM Safe Computation Rule [8]). *A KM safe computation rule is a function from a set G of normal goals, such that every goal in G contains either a positive non-abducible literal or a ground literal, to a set of literals, such that the value of the function for each goal is either a positive non-abducible literal or a ground literal, called the selected literal, in that goal.*

2.5 Summary

This chapter has presented background information on definite and normal programs and abductive logic programming. The common proof procedures for definite and normal programs have also been given.

The next section introduces the inductive logic programming and the related work on the topic of this project.

CHAPTER 3

Inductive Logic Programming (ILP) and Related Works

Inductive Logic Programming (ILP) is a particular form of declarative machine learning which uses logic programs to represent examples, background knowledge and hypotheses. It is a sub-field of artificial intelligence that attempts to perform the induction of the hypothesised results based on the examples and background knowledge [17, 23]. ILP is different from other forms of machine learning as it delivers an approach to use expressive logical languages to represent the background knowledge. This feature of ILP has benefited many knowledge discovery areas that have plenty of sources of background knowledge [22]. This chapter will firstly give an introduction to ILP and its formal framework. Then, a brief description of *Inverse Entailment* (IE) with IoF system will be presented. Lastly, this chapter will give a short discussion about the related works on ILP problems with normal program settings.

3.1 A formal Framework of ILP

Given the logically encoded background knowledge B and a set of logically represented examples E, ILP systems (procedures) will construct a hypothesis H that entails all the positive examples E^+ within E and none of the negative ones E^- in terms of B.

In order to construct the hypothesis H in an ILP system, the following four logical requirements should be satisfied: (where $\models$ is logical entailment and $\wedge$ is logical and)

- **Necessity**: $B \not\models E^+$
- **Sufficiency**: $B \wedge H \models E^+$
- **Weak Consistency**: $B \wedge H \not\models false$
- **Strong Consistency**: $B \wedge H \wedge E^- \not\models false$

The requirement *necessity* indicates that there is no redundant example or we only consider those examples E^+ that are not provable according to the background knowledge B. The

requirement *sufficiency* meets the aim of constructing a hypothesis H that can explain all the positive examples E^+ with the background knowledge B. [20, 23, 24]. The *weak (strong) consistency* simply means the derived hypothesis H should be satisfiable with the background knowledge B (and negative examples E^-).

Since the hypotheses space is always large in ILP problems, a *mode declaration* is always defined as a part of the ILP problem. A *mode declaration* is a language restriction and can effectively restrict the search space of the hypotheses.

Definition 3.1. (Mode Declaration [4]) *A mode declaration is either a head or body declaration, respectively modeh(s) and modeb(s) where s is a ground literal called schema. schema can contain placemarker. A placemarker is either '+type', '−type' or '#type', which stand, respectively, for ground terms, input termss and output terms of a predicate type. The* mode declaration *restricts the predicates can be involved in the head and body of the hypothesis.*

Since this project works on the ILP problems with disjoint open program settings, the terminologies for this kind of ILP problems are defined by Kimber [13] and are given below:

Definition 3.2. (Hypothesis [13]). *Let $P = \langle B, U, I\rangle$ be an open program, let E be a set of clauses, and let $\mathcal{L}_H$ be a language. A set of clauses $H \subseteq \mathcal{L}_H$ is a hypothesis for P and E only if H only defines predicates in U.*

Definition 3.3. (Complete Hypothesis [13]). *Let $P = \langle B, U, I\rangle$ be an open program, let E^+ and E^- be sets of clauses and let H be a hypothesis for P, E^+ and E^-. Then H is complete with respect to E^+ if and only if $B \cup H \models E^+$.*

Definition 3.4. (Consistent Hypothesis [13]). *Let $P = \langle B, U, I\rangle$ be an open program, let E^+ and E^- be sets of clauses and let H be a hypothesis for P, E^+ and E^-. Then H is consistent with respect to E^- if and only if $B \cup H \cup I \cup \overline{E^-}$ is satisfiable.*

Definition 3.5. (Correct Hypothesis [13]). *Let Let $P = \langle B, U, I\rangle$ be an open program, let E^+ and E^- be sets of clauses and let H be a hypothesis for P, E^+ and E^-. H is correct with respect to E^+ and E^- if and only if H is complete with respect to E^+, and H is consistent with respect to E^-. A correct hypothesis is also called an inductive solution.*

3.2 Inverse Entailment

Inverse Entailment (IE), which is firstly introduced by Muggleton [21], has been widely applied in ILP systems.

Theorem 3.1. (Inverse Entailment [21]). *Let B be a Horn program, and let H and E be Horn clauses. Then $B \wedge H \models E$ if and only if $B \wedge \neg E \models \neg H$.*

Most IE based ILP systems first construct a *most specific* hypothesis $H_\perp$ from a seed example E such that

$$B \wedge \neg E \models \neg H_\perp,$$

then, use subsumption search to find a better hypothesis H such that

$$H \models H_\perp$$

Definition 3.6. (Clause Subsumption [16]) *Let C and D be first-order clauses. C subsumes D, write $C \succeq D$, if there is a substitution θ such that $C\theta \subseteq D$ (every literal in $C\theta$ appears in D).*

However, the relationship between the hypothesis H and the most hypothesis $H_\perp$ is not always going to be $H \models H_\perp$. For example, given the background knowledge $B = \{(p \leftarrow q),\ a\}$ and example $\{x\}$, the most specific hypothesis generated by IoF [11] is possibly going to be $H_\perp = \{(x \leftarrow a, p),\ (q \leftarrow a)\}$. And a possible hypothesis H can be $\{x \leftarrow a\}$. It is obvious H subsumes $H_\perp$ with respect to the background knowledge B since $x \leftarrow a$ subsumes $x \leftarrow a, p$ and $q \leftarrow a$ in $H_\perp$ is an auxiliary clause for the atom p. In this case, however, without considering the background knowledge B, H does not necessarily entails $H_\perp$.

One typical ILP system using IE method is the Progol system [21]. Give the background knowledge B and a set of examples E, the system firstly picks one positive example e^+ and negates it into a set of ground literals $\overline{e^+}$ and if $\overline{e^+}$ is not ground then a skolem substitution is applied.

Definition 3.7. (Skolem Substitution [24]) *Let Σ be a set of clauses and C be a clause. Let $X_1, \cdots, X_n$ be the variables appearing in C and $a_1, \cdots, a_n$ be distinct constants not appearing in Σ or C. Then the substitution $\{X_1/a_1, \cdots, X_n/a_n\}$ is a Skolem substitution for C with respect to Σ. The symbols $a_1, \cdots, a_n$ are Skolem constants.*

Then, the system constructs a disjunction of the ground literals $B \cup \overline{e^+}$ and search if there is a better clause H that implies $\overline{B \cup \overline{e^+}}$.

The HAIL [25] system widens the search space of Progol as it allows to generate a set of clauses (a theory) in response to a seed example. It defines a Kernel Set K as its most specific hypothesis.

Definition 3.8. (Kernel Set [25]) *Let B be a Horn theory and E be a Horn clause such that $B \not\models E$. Then a ground Horn theory $K = \{C_1, \cdots, C_k\} (K \geq 1)$ is a Kernel Set of B and E if and only if each clause C_i, $1 \leq i \leq k$ is given by $A_0^i \leftarrow A_1^i, \cdots, A_n^i$, where $B \cup \{A_0^1, \cdots, A_0^k\} \models E^+$, and $B \cup E^- \models \{A_1^1, \cdots, A_{n_k}^k\}$*

To compute the Kernel, the system firstly computes the ground abductive explanations $\Delta = \{\alpha_0, ..., \alpha_n\}$ for e^+ (e^+ is Skolemised if it is not a ground example). Then the system finds sets of atoms β_i $(1 \leq i \leq n)$ can be directly deduced from the background knowledge B and the body of e^+ to construct the Kernel Set:

$$K = \left\{ \begin{array}{l} \beta_0 \rightarrow \alpha_0 \\ \vdots \\ \beta_n \rightarrow \alpha_n \end{array} \right\}.$$

Finally, the system tries to find a better hypothesis H such that $H \succeq K$. Here, $H \succeq K$ means that every clause in K is subsumed by at least one clause in S.

Definition 3.9. (Kernel Set Subsumption [25]) *Let B be a Horn theory and E be a Horn clause such that $B \not\models E$. A set of Horn clauses H is said to be derivable from B and E by Kernel Set Subsumption , denoted $B, E \vdash_{KSS} H$, if and only if there is a K such that K is a Kernel Set of B and E, and $H \succeq E$.*

3.3 Induction on Normal Programs

This section presents and gives short discussions on two related works on ILP problems [26, 28] with normal program settings by using bridge formulas.

XHAIL

Extended Hybrid Abductive Inductive Learning (XHAIL) introduced by Ray [26] is an ILP system that is generalised from XHAIL and can solve ILP problems with normal program settings. Given a normal program B and a set of literal examples E, the system firstly generates a XHAIL Kernel set K then search through hypotheses H such that $H \succeq K$. The XHAIL Kernel set is a set of ground clauses

$$K = \left\{ \begin{array}{l} \alpha_1 \leftarrow l_1^1, \cdots, l_1^{m_1} \\ \vdots \\ \alpha_n \leftarrow l_n^1, \cdots, l_n^{m_n} \end{array} \right\},$$

where $\{\alpha_1, \cdots, \alpha_n\}$ is an abductive explanation for E with respect to B such that $B \cup \{\alpha_1, \cdots, \alpha_n\} \models_{st} E$, and $\{l_1^1, \cdots, l_n^{m_n}\}$ is a set of ground literals such that for each $l_i^{j_i}$ $(1 \leq i \leq n,\ 1 \leq j \leq m)$, $B \cup \{\alpha_1, \cdots, \alpha_n\} \models_{st} l_i^{j_i}$. for exmaple, given the following background knowledge B, examples E and mode declaration M,

Example 3.1. [26]

$$B = \left\{ \begin{array}{l} bird(X) \leftarrow penguin(X) \\ bird(a) \\ bird(b) \\ bird(c) \\ penguin(d) \end{array} \right\},$$

$$B = \left\{ \begin{array}{l} flies(a) \\ flies(b) \\ flies(c) \\ \neg flies(d) \end{array} \right\},$$

$$M = \left\{ \begin{array}{l} modeh(flies(+bird)) \\ modeb(penguin(+bird)) \\ modeb(\neg penguin(+bird)) \end{array} \right\}$$

the corresponding XHAIL Kernel set K is

$$K = \left\{ \begin{array}{l} flies(a) \leftarrow \neg penguin(a) \\ flies(b) \leftarrow \neg penguin(b) \\ flies(c) \leftarrow \neg penguin(c) \end{array} \right\}.$$

and the hypothesis H satisfies $H \succeq K$ and $B \cup H \models_{st} E$ is

$$H = \left\{ \; flies(X) \leftarrow \neg penguin(X) \; \right\}.$$

There are several similarities between XHAIL and the procedure ICC procedure which is the new ILP procedure proposed in this report. First, both XHAIL and ICC process all the examples in one go. Second, both XHAIL and ICC uses a ground set of clauses as a bridge formula between the original examples and the derived hypothesis.

However, the Kernel set K in XHAIL is quite different from the bridge formula T (details in Sections 6) used in ICC. Given a normal program B and a set of literal examples E. In XHAIL, the Kernel set K is not necessarily an explanation for E with respect to B. However, the relation $B \cup T \models E$ is always hold in ICC. For example [26], if

$$B = \{ \; b \leftarrow e \; \}$$
$$E = \{ \; e \; \},$$

then $K' = \{e \leftarrow b\}$ is a Kernel Set of B and e. However, $B \cup K' \not\models E$. Furthermore, The XHAIL system, as the author himself notes in [26], has an intrinsic drawbacks. That is the system may generate some redundant clauses in its Kernel Set since the abductive phase of XHAIL allows the abductive explanation of the example not to be the minimal. Given background knowledge B and examples E, if an abductive explanation Δ for B and E is said to be minimal, then there is no strict subset of Δ is a abductive explanation with respect to B and E. Consider the Yamamoto's example [30],

Example 3.2. [30]

$$B = \left\{ \begin{array}{l} even(s(X)) \leftarrow odd(X) \\ even(0) \end{array} \right\}$$

$$E = \left\{ \begin{array}{l} odd(s^3(0)), odd(s^5(0)), \\ \neg even(s^1(0)), \neg even(s^3(0)) \end{array} \right\}$$

$$Modeh = \{ \; odd(+any) \; \}$$

$$Modeb = \{ \; even(+any), +any = s(-any) \; \}.$$

A possible non-minimal abductive explanation Δ' for B and E is

$$\Delta' = \{odd(s^1(0)), odd(s^3(0)), odd(s^4(0)), odd(s^5(0)), \}.$$

It is obvious that the clause formed from $odd(s^4(0))$ will be a redundant clause. However,

ICC will use the minimal abductive result Δ for B and E

$$\Delta = \{odd(s^3(0)), odd(s^5(0))\}$$

and gradually generalise it to a larger theory T relative to B. Therefore, there would be no redundant clause in T.

Induction From Answer Set

Sakama [28] has given a set of related methods *Induction From Answer Set* (IAS) for learning inductive hypotheses from normal programs (and extended programs). Since this project works only on the ILP problems with normal program settings, we only discuss the application of Sakama's methods on normal programs.

Given the background knowledge B and a literal example L, The key part of Sakama's methods IAS^{pos} is to construct a bridge formula R which is related to literal L such that $B \not\models_{st} R$. The Proposition 3.1 and Proposition 3.2 show below imply that $B \not\models_{st} R$ is a necessary condition for R, together with B,to be an explanation for L.

Proposition 3.1. [28] *Let B be a program and R a rule such that $B \cup \{R\}$ is consistent. For any ground literal L, $B \cup \{R\} \models_{st} L$ and $B \not\models_{st} R$ imply $B \not\models_{st} L$.*

Proposition 3.2. [28] *Let B be a program and R a rule such that $B \cup \{R\}$ is consistent. For any ground literal L, $B \cup \{R\} \not\models_{st} L$ and $B \models_{st} R$ imply $B \not\models_{st} L$.*

If L is a positive example, the corresponding procedure is IAS^{pos}. The first step of IAS^{pos} procedure is to compute a set of literals S^+ such that

$$S^+ = S \cup \{\neg A \mid A \in Atoms \text{ and } A \notin S\}.$$

where $Atoms$ is the set of all ground Atoms in the language of the program and S is a stable model (it is 'answer set' here in the original paper, but it is been proved that when an extended logic program is an normal program, answer sets coincides with stable models [6]) of B. Then, it computes a set of literals Γ such that $\Gamma \subseteq S^+$ and each element in Γ is related to L. Thus, the following relation holds:

$$B \not\models_{st} \leftarrow \Gamma.$$

Since $B \not\models_{st} L$, $\neg L$ is included in Γ. Thus a ground bridge formula R is formed:

$$R \equiv \ L \leftarrow \Gamma'$$

where $\Gamma' = \Gamma - \{\neg L\}$. The clause C involved in the hypothesis H can is generalised from H' such that $C\theta \equiv R$ for some substitution θ. An example is illustrated below:

Example 3.3. [28]

$$B = \left\{ \begin{array}{l} bird(X) \leftarrow penguin(X) \\ bird(tweety) \\ penguin(polly) \end{array} \right\},$$

$$L = flies(tweety).$$

The S^+ is

$$S^+ = \left\{ \begin{array}{l} bird(tweety), bird(polly), penguin(polly), \\ \neg penguin(tweety), \neg flies(tweety), \neg flies(polly) \end{array} \right\}.$$

The literals related to L in S^+ are

$$\Gamma = \left\{ \begin{array}{l} bird(tweety), \\ \neg penguin(tweety), \neg flies(tweety), \end{array} \right\}.$$

And

$$\leftarrow \Gamma \equiv \leftarrow bird(tweety), \neg penguin(tweety), \neg flies(tweety).$$

Therefore, by shifting $flies(tweety)$ to the head, the ground R is

$$R \equiv flies(tweety) \leftarrow bird(tweety), \neg penguin(tweety).$$

And the generalised H is

$$H = \{flies(X) \leftarrow bird(X), \neg penguin(X)\}.$$

where $\theta = \{X/tweety\}$.

If L is a negative example, the corresponding learning procedure IAS^{neg} is slightly changed from IAS^{pos}. Since the following purpose for R remains unchanged

$$B \not\models_{st} R.$$

the clause $\leftarrow \Gamma$ is generated as in IAS^{pos}. However, the way of constructing ground bridge formula R from $\leftarrow \Gamma$ is different from IAS^{pos}. The procedure IAS^{neg} shift a literal K ($\neg K \in \Gamma$) to the head of the clause where K is a literal on which L negatively depends. Then the clause C in hypothesis H is computed by anti-instantiation of R and dropping every body literal with a predicate which strongly and negatively depends on the predicate of K. An example is illustrated below,

Example 3.4. [28]

$$B = \left\{ \begin{array}{l} flies(X) \leftarrow bird(X), \neg ab(X) \\ bird(X) \leftarrow penguin(X) \\ bird(tweety) \\ penguin(polly) \end{array} \right\},$$

$$L = flies(polly).$$

The set of literals Γ that satisfies the stable model of B and is related to L is

$$\Gamma = \{bird(polly), penguin(polly), flies(polly), \neg ab(polly)\}$$

and the clause $\leftarrow \Gamma$ and the ground bridge formula R are

$$\leftarrow \Gamma \equiv \leftarrow bird(polly), penguin(polly), flies(polly), \neg ab(polly)$$

$$R \equiv ab(polly) \leftarrow bird(polly), penguin(polly), flies(polly).$$

The literal $ab(polly)$ is shifted because it is strongly and negatively depended by the negative example $flies(polly)$. Then the following H generalised from R is a possible hypothesis for the problem:

$$H = \{ab(X) \leftarrow bird(X), penguin(X)\}$$

The literal $flies(X)$ is dropped since the predicate $flies$ is strongly and negatively depend on the predicate ab.

The Sakama's method has two main limitations. The first one is the incompleteness caused by the loop checking on negative literals. Since IAS procedure requires that each hypothesis should be negative-loop-free on the predicate level, it will treat some correct hypotheses as incorrect hypotheses. For example,

Example 3.5.

$$B = \left\{ \; father(peter, chris) \; \right\},$$

$$L = parent(peter, chris).$$

The bridge formula R computed by IAS would be

$$R \equiv parent(peter, chris) \leftarrow father(peter, chris), \neg father(chris, peter), \neg parent(chris, peter)$$

which would generalised to a flat clause C that satisfies $C\theta \equiv R,\ \theta = \{X/peter, Y/chris\}$:

$$C \equiv parent(X, Y) \leftarrow father(X, Y), \neg father(Y, X), \neg parent(Y, X).$$

Since in clause C the predicate *parent* in body is strongly and negatively depended on the

predicate *parent* in head, IAS will not return H. However, H is a correct hypothesis for the problem. This incompleteness problem also happens in the procedure proposed in this report.

The second main problem of IAS, is that when IAS is applied on ILP problems with multiple examples, it performs a incremental learning, as noted by the author, it is not guaranteed that the hypotheses learned will correctly explain the examples [28]. However, the procedure propose in this report does not have this problem. Since the procedure proposed in this report uses α-*Connected Theory Generalisation* (see Section 6) and solves all the examples as a whole, the hypothesis computed must can explain all the examples.

3.4 Summary

This chapter briefly describes the inductive logic programming in terms of its formal frame work, inverse entailment and gives short discussion on two related works - Ray's XHAIL system and Sakama's IAS procedure. The next chapter will introduce Kimber's works on ILP problems with definite program and normal program settings.

CHAPTER 4

Connected Theory

The HAIL system allows a theory to be derived from a single seed example in case the example has multiple abducibles from the background knowledge. However, it is possible to use a theory to explain one abducible in the sense that some clauses in the theory are used to explain other clauses. Kimber et al. [11] introduces an proof procedure called Induction on Failure (IoF) to achieve this based on the notion of Connected Theory Generalisation. This chapter will firstly introduce this Connected Theory Generalisation and the Induction on Failure Procedure. Then, a short discussion on Kimber's Normal Connected Theory Generalisation which is closely related to the ILP approach used in ICC procedure will be provided.

In the rest part of this report including this chapter, we will use T^h and T^b to denote head and body literals of clauses in a given theory T. And we will use T^{pos} and T^{neg} to denote the clauses with positive and negative head literals in theory T.

4.1 Connected Theory Generalisation (CTG)

A *Connected Theory* (CT) is a set of ground definite clauses such that a clause may depend on some other clauses.

Definition 4.1. (Connected Theory [11]). *Let $P = \langle B, U, I\rangle$ be a definite open program, and let E be a ground atom. Let $T_1, \cdots, T_n$ be n disjoint sets of ground definite clauses defining only predicates in U. $T = T_1 \cup \cdots \cup T_n$ $(n \geq 1)$ is a Connected Theory for P and E if and only if the following conditions are satisfied:*

1. $B \models T_n^b$,
2. $B \cup T_n^h \cup \cdots \cup T_{i+1}^h \models T_i^b$, *for all* i $(1 \leq i < n)$,
3. $B \cup T_1^h \models E$, *and*

4. $B \cup T \cup I$ is consistent.

A CT T is viewed as n sub-theories and each sub-theory T_i $(1 < i \leq n)$ is 'connected' to some body literals of clauses in its adjacent sub-theory T_{i-1}. For example, given the following open program $P = \langle B, \{x, q, s\}, \varnothing \rangle$ and example E

Example 4.1.

$$B = \left\{ \begin{array}{l} p \leftarrow q \\ s \leftarrow t \\ a \end{array} \right\}$$

$$E = \left\{ \; x \; \right\}$$

a possible CT T for P and E is

$$T = \left\{ \begin{array}{l} x \leftarrow a, p \\ q \leftarrow a, s \\ s \leftarrow a \end{array} \right\}.$$

There are three layers in T:

$$T_1 = \{x \leftarrow a, p\},$$
$$T_2 = \{q \leftarrow a, s\},$$
$$T_3 = \{s \leftarrow a\}.$$

It is obvious that $T_2^h \cup B$ explains p in T_1^b and $T_3^h \cup B$ explains s in T_2^b.

A set of Horn clauses which entails a CT is said to be derivable by *Connected Theory Generalisation* (CTG), defines below

Definition 4.2. *Let $P = \langle B, U, I \rangle$ be a definit open program and E be a ground Horn clause such that $B \not\models E$. A set H of Horn clauses is said to be derivable from B and E by Connected Theory Generalisation, denoted $B, E \vdash_{CTG} H$, if and only if there is a T such that T is a Connected Theory for B and E and $H \models T$.*

4.2 Induction on Failure (IoF)

Induction on Failure (IoF) [11] is an ILP system that implements the *Connected Theory Generalisation* (CTG). The procedure firstly defines a ground *Connected Theory* (CT) *CT* as its most specific hypothesis and then searchs for hypotheses subsumes it. The main work flow of IoF procedure is shown in Figure 4.1

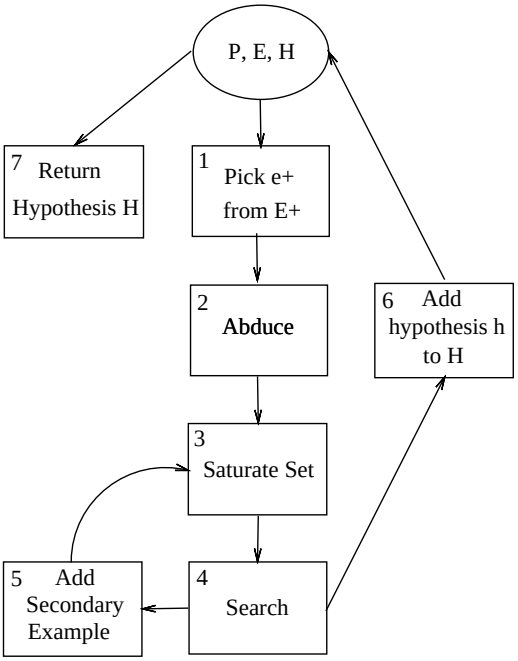

Figure 4.1: Graph for IoF procedure

For the better illustration, this section will descibe the IoF procedure using the following Example 4.2

Example 4.2.

$$B = \left\{ \begin{array}{l} p \leftarrow q \\ s \leftarrow t \\ a \end{array} \right\}$$

$$U = \left\{ \; q, t \; \right\}$$

$$I = \varnothing$$

$$E = \left\{ \; p \; \right\}$$

The procedure starts with P, E and an empty list H. The first step is to pick one seed example from E^+ (box 1). Here, the seed example picked is p. The next step is to abduce the seed example p and get $\{q\}$ (box 2). Then, the process goes to the box 3 which is to add as many as body literals to the clauses in the anductble set of the seed example (box 4).

This produces a first ground theory T_0

$$T_0 = \{q \leftarrow a\}$$

Next, a subsumption search procedure is applied on the theory (T_0) produced from the last step to see if there is a theory h such that it subsumes T_0, has a better structure than T_0 and do not prove any negative examples in E (box 4). Suppose we define the 'better structure' of a theory to be larger number of clauses and larger number of body literals in each clause. Then the best hypothesis h searched from T_0 would be $h = \{p \leftarrow a\}$. After this search step, the procedure has two options, it can either add a secondary example to T_0 (box 5) or add h to H as a part of the final hypothesis (box 6). In this case, there is a secondary example s can be added, the procedure will go to box 5. By adding the secondary example s and its abductive explaination $\{t\}$ to the theory t_0 and saturating it again (box 5 then box 3), a new ground CT T_1 is formed:

$$T_1 = \left\{ \begin{array}{l} q \leftarrow a, s \\ t \leftarrow a \end{array} \right\}.$$

The search procedure is called again on T_1 and h is updated (box 4). Since we assume a larger hypothesis is more preferable, the hypothesis h becomes

$$h = \left\{ \begin{array}{l} q \leftarrow a, s \\ t \leftarrow a \end{array} \right\}.$$

And h is added to the list H because there is no secondary example can be added into T_1. Thus, the cover loop for the seed example p is finished. If there are other positive examples in E, the procedure will go to box 1 again and start another cover loop. However, in this case, there is no other examples in E, the procedure end up with returning H

$$H = \left\{ \begin{array}{l} q \leftarrow a, s \\ t \leftarrow a \end{array} \right\}.$$

4.3 Normal Connected Theory

The main reason for the existing Horn theory based ILP system fail to be successfully applied on normal programs is that those system assume the logic is monotonic. Monotonic means the consequence of any set of formulas are preserved when more formulas are added. Formally, let Γ, Δ and Φ be sets of formulas, the logic is monotonic is to say if $\Delta \models \Phi$ then $\Delta \cup \Gamma \models \Phi$. This property of definite programs allows these system to perform incremental learning.

However, the program completion is non-monotonic. Incremental learning can hardly applied on ILP problems with normal program settings. Consider the following example of an ILP problem with a normal disjoint program $P = \langle B, U, \varnothing \rangle$ and examples E,

Example 4.3.

$$B = \left\{ \begin{array}{l} w(1), w(2), w(3), \\ s(2), s(3), \\ t(2), \\ p(X) \leftarrow q(X), \neg r(X) \end{array} \right\}$$

$$E = \left\{ \; p(1), r(2), r(3) \; \right\}$$

$$Modeh \; (U) = \{ \; q(+any), r(+any) \; \}$$

$$Modeb = \{ \; w(+any), s(+any), t(+any) \; \}$$

If an incremental learning is performed, then following partial hypotheses h_1, h_2 and h_3 can be learned from seed examples e_1, e_2 and e_3:

$$\begin{array}{l} e_1 = p(1) \;\Rightarrow\; h_1' = \{q(1) \leftarrow w(1)\} \;\Rightarrow\; h_1 = \{q(X) \leftarrow w(X)\} \\ e_2 = r(2) \;\Rightarrow\; h_2' = \{r(2) \leftarrow w(2)\} \;\Rightarrow\; h_2 = \{r(X) \leftarrow w(X)\} \\ e_3 = r(3) \;\Rightarrow\; h_3' = \{r(3) \leftarrow w(3)\} \;\Rightarrow\; h_3 = \{r(X) \leftarrow w(X)\}. \end{array}$$

It satisfies

$$\begin{array}{l} comp(B \cup h_1) \models \{e_1\} \\ comp(B \cup h_2) \models \{e_2\} \\ comp(B \cup h_3) \models \{e_3\}. \end{array}$$

However, it does *not* satisfy

$$comp(B \cup h_1 \cup h_2 \cup h_3) \models \{e_1, e_2, e_3\}$$

since $comp(B \cup h_2 \cup h_3)$ proves $r(1)$ and makes $comp(B \cup h_1 \cup h_2 \cup h_3)$ fail to prove $p(1)$ (e_1).

The undlying problem of this is at the generalisation phase. In ILP problems with definite program, since the logic is monotonic, it is free to replace ground terms in the hypothesis by variables. In above case, however, generalise h_2' to h_2 is equal to add every other instances

of $r(X) \leftarrow w(X)$ including $r(1) \leftarrow w(1)$ to the hypothesis. However, $comp(B \cup h_1 \cup \{r(1) \leftarrow w(1)\} \not\models p(1)$.

The way poroposed by Kimber [15] to overcome this problem is to find a bridge formula T called *Normal Connected Theory* (nCT) for P and E such that the negative infomation are as explicit as the positive in T. The hypothesis H is generalised such that $compdefs(U, H)$ implies this T.

Definition 4.3. (Normal Connected Theory [15]) *Let $P = \langle B, U, I\rangle$ be an open program, and let E be a set of ground literals. Let $T1, ..., Tn$ be n $(n \geq 1)$ disjoint sets of ground clauses of the form $L_0 \leftarrow L_1, ..., L_k$ where L_i is a literal for all i $(0 \leq i \leq k)$, defining only predicates in U. $T = T_1 \cup ... \cup T_n$ is a Normal Connected Theory for P and E if and only if*

1. $comp(P) \models T_n^-$
2. $comp(P) \cup T_n^+ \cup ... \cup T_{i+1}^+ \models T_i^-$, *for all* i $(1 \leq i \leq n)$
3. $comp(P) \cup T_1^+ \models E$
4. $comp(P) \cup T \cup I$ *is consistent.*

for some $n \geq 1$

Since both positive and negative information are explicitly included in a nCT T, it always satisfy

$$comp(P) \cup T \models E.$$

Proposition 4.1. [12] *Let $P = \langle B, U, I\rangle$ be an open program, let E be a set of ground literals. If T is a normal connected theory for P and E, then $comp(P) \cup T \models E$.*

The definition of *Normal Connected Theory Generalisation* (nCTG) is give below.

Definition 4.4. (Normal Connected Theory Generalisation [15]). *Let $P = \langle B, U, I\rangle$ be an open program, let E be a set of ground literals. A set H of normal clauses is derivable from P and E by Normal Connected Theory Generalisation, denoted $P, E \vdash_{nCTG} H$, if and only if there is a T such that T is a Normal Connected Theory for P and E, and $compdefs(U, H) \models T$, and $comp(P) \cup compdefs(U, H) \cup I$ is consistent.*

For the Example 4.3, a possible nCT T is

$$T = \left\{ \begin{array}{l} q(1) \\ r(2) \leftarrow s(2) \\ r(3) \leftarrow s(3) \\ \neg r(1) \leftarrow \neg s(1) \end{array} \right\}.$$

And the hypothesis H satisfies $compdefs(U, H) \models T$ is

$$H = \left\{ \begin{array}{l} q(X) \\ r(Y) \leftarrow s(Y) \end{array} \right\}.$$

4.4 Summary

So far we have introduced all the related knowledge of this project. The rest part of this report introduces a new proof procedure which can solve ILP problems with normal program settings and describes the relationship between the new ILP system and Kimber's *Normal Connected Theory Generalisation*. The next chapter will give a brief view on the new ILP system.

CHAPTER 5

Introduction to Induction with Completion and Connected Theories (ICC) Procedure

This chapter gives a brief informal description about the Induction with Completion and Connected Theories (ICC) procedure. The procedure is a generalisation of IoF, which, same as IoF, interleaves top-down subsumption-based search with bottom-up generalisation of a ground normal connected theory T. The main steps in ICC procedure is shown in Figure 5.1

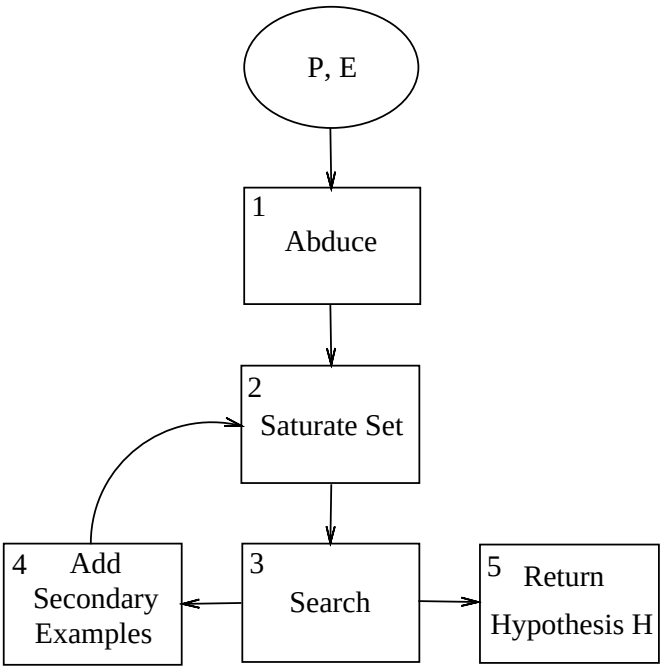

Figure 5.1: Graph for ICC procedure

The procedure starts with a disjoint open program P and a set of literal examples E. The example set E is firstly abuduced according to P (box 1). And a set of abductive facts Δ is generated. The second step is to saturate the set Δ (box 2). Saturation means adding

all the possible literals into the bodies of clauses in Δ under the predefined language bias. The saturation is different here from the saturation in IoF. In ICC, only clauses with positive heads are saturated. More details will come in Chapter 7. Thus, the first ground theory T for P and E is formed. The next step is to use a subsumption-based search to derive a most successful hypothesis H from T (box 3). The H is derived by generalising the positive part of T and check it against the negative literals in T (for details, see Section 7.2). If the current T can be generalised by adding a secondary example, then a secondary example and its abductive explanations are added in to T (box 4) and the new theory T is saturated again (box 2). The hypothesis H is than updated by searching the new theory T (box 3). If there is no secondary example can be added into T, then the current H is returned (box 5).

There are two main differences between ICC and IoF. First, IoF performs an incremental learning on E, whereas ICC processes all examples together. That is to say the ground normal connected theory T in ICC is produced by all the examples in E rather than one seed example e in E. Second, the "subsumption" search step in ICC uses a new operational way called *α-Normal Connected Theory Generalisation* (α-nCTG) (defined in Section 6). It is a similar approach as *Normal Connected Theory Generalisation* (nCTG) in [15], and can derive all the hypotheses that are derivable from nCTG given the normal open program P and examples E (see Proposition 6.1). By using α-nCTG, If hypothesis H can be derived from P, E, then it is guaranteed that all the relevant negative information in T will be retained in this generalised hypothesis H, where T is a *α-Normal Connected Theory* for P and E. The next chapter provides more details on this *α-Normal Connected Theory Generalisation*.

CHAPTER 6

α-Normal Connected Theory Generalisation

This section proposes a new approach to ILP, called *α-Normal Connected Theory Generalisation* (α-nCTG). This approach is based on the notion of a *α-Normal Connected Theory* (α-nCT), which is a special form of normal connected theory and is defined in the Section 4.3. It is proved that the hypotheses that are derivable from *Normal Connected Theory Generalisation* (nCTG) are also derivable from *α-Normal Connected Theory Generalisation* given the normal disjoint open program P and literal examples E.

Given a normal open program $P = \langle B, U, I \rangle$ and a set of literal examples E, if the hypothesis H is derivable from nCTG, then, by definition, it must satisfy

$$compdef(U, H) \models T, \tag{6.1}$$

where T is a *Normal Connected Theory* and satisfy also

$$comp(P) \cup T \models E. \tag{6.2}$$

This means for each predicate p appearing in T, there should be one or more completed definitions for p that implies all the clauses in T related to p.

Returning to Example 1.1, the completion of $P = \langle B, U, \varnothing \rangle$ is

$$B = \left\{ \begin{array}{l} \forall X \forall Y (obeys(X, Y) \leftrightarrow \neg officer(X) \wedge officer(Y)) \\ \forall X (wears\_hat(X) \leftrightarrow X = price \vee X = osbourne) \\ \forall X (has\_stripes(X) \leftrightarrow X = osbourne)) \end{array} \right\}$$

and both bridge formulas

$$T_1 = \left\{ \begin{array}{l} officer(osbourne) \leftarrow wears\_hat(osbourne) \\ \neg officer(price) \leftarrow wears\_hat(price) \end{array} \right\}$$

and

$$T_2 = \left\{ \begin{array}{l} officer(osbourne) \leftarrow has\_stripes(osbourne) \\ \neg officer(price) \leftarrow \neg has\_stripes(price) \end{array} \right\}$$

are *Normal Connected Theories.*

However, there is no non-ground hypothesis H that satisfies 6.1 can be generalised from T_1

In the case of T_1, the most specific ground hypothesis h^g is

$$h^g = \{office(osbourne) \leftarrow wears\_hat(osbourne)\}.$$

Two flat hypotheses generalised from h^g are

$$H_1 = \{office(X) \leftarrow wears\_hat(X)\},$$

$$H_2 = \{office(X)\}.$$

However, both

$$compdef(\{officer\}, H_1) = \{\forall X(office(X) \leftrightarrow wears\_hat(X))\} \text{ and}$$

$$copmdef(\{officer\}, H_2) = \{\forall X(office(X))\}$$

do not imply the second clause $\neg officer(price) \leftarrow wears\_hat(price)$ in T_1.

In the case of T_2, a non-ground hypotheses $\{office(X) \leftarrow has\_stripe(X)\}$ that satisfies 6.1 can be generalised from the first clause in T_1.

This difference between T_1 and T_2 implies that if a procedure is designed based on nCTG, then it is not trivial to find a way that can generate the ground theory at the completed level.

The proof procedure proposed in this report uses a new operational approach of nCTG called α-nCTG, defines below:

Definition 6.1. (α-Connected Theory) *Let $P = \langle B, U, I\rangle$ be a disjoint normal open program and E be a set of ground literals such that $comp(P) \not\models E$. A set of normal clauses T is said to an α-Connected Theory for P and E if and only if*

- *T is a Normal Connected Theory for P and E*

- *T is in the form of*

$$T = \left\{ \begin{array}{l} A^1 \leftarrow L^1_1, \cdots, L^1_{m_1} \\ \vdots \\ A^n \leftarrow L^n_1, \cdots, L^n_{m_n} \\ nL^1 \\ \vdots \\ nL^k \end{array} \right\},$$

where A^i for $0 \leq i \leq n$ are ground atoms, $L^i_{j_i}$ for $1 \leq i \leq n$ and $1 \leq j \leq m_i$ are ground literals, nL^u for $1 \leq u \leq k$ are ground negative literals.

The definition of α-nCTG is very similar to nCTG. The only different part is that the hypotheses H generated by α-nCTG should satisfy $comp(P) \cup compdefs(U, H) \models T$ where T^{pos}. This is because in an α-nCT, clauses with negative literal in heads has no body literals. Therefore, these clauses cannot be simply entailed by $compdefs(U, H)$.

Definition 6.2. (α-Connected Theory Generalisation) *Let $P = \langle B, U, I \rangle$ be a disjoint normal open program and E be a set of ground literals such that $comp(P) \not\models E$. A set H of normal clauses is said to be deriable from P and E by α-Connected Theory Generalisation, denoted $P, E \vdash_{\alpha-nCTG} H$, if and only if there is a T such that T is an α-Normal Connected Theory for P and E, and $comp(P) \cup compdefs(U, H) \models T$, and $compdefs(U, H) \models T^{pos}$ where T^{pos} is the set of clauses with positive heads in T, and $comp(P) \cup compdefs(U, H) \cup I$ is consistent.*

For example, a possible *α-Normal Connected Theory* T for Example 1.1 is

Example 6.1.

$$T = \left\{ \begin{array}{l} officer(osbourn) \leftarrow wears\_hat(osbourne), has\_stripe(osbourne) \\ \neg officer(price) \end{array} \right\}.$$

The soundness of α-nCTG is proved below:

Lemma 6.1. [12] *If $P = \langle B, U, I \rangle$ be a disjoint open program, and H is a set of clauses defining only predicates in U, then, $comp(B \cup H) = comp(P) \cup compdefs(U, H)$.*

Theorem 6.1. (Soundness of α-Connected Theory Generalisation) *Let $P = \langle B, U, I \rangle$ be a disjoint open program, let E be a set of ground literals such that $B \not\models E$, and let H be an inductive hypothesis for P and E. if $P, E \vdash_{\alpha-nCTG} H$ then H is a correct hypothesis for P and E.*

Proof. By assuming that H is an inductive hypothesis for P and E, and by Definition 6.2, $comp(P) \cup compdefs(U, H) \cup I$ is consistent. So, by Lemma 6.1 $comp(B \cup H) \cup I$ is consistent. Thus, by Definitions 3.5 to show that H is a correct hypothesis for P and E it is sufficient to show that $comp(B \cup H) \models E$. By Definition 6.2 there is a T such that T is a *α-Normal Connected Theory* and therefore a *Normal Connected Theory* for P

and E, and $comp(P) \cup compdefs(U,H) \models T$. Therefore, by monotonicity and reflexivity of $\models$, $comp(P) \cup compdefs(U,H) \models comp(P) \cup T$. Furthermore, by Proposition 4.1, $comp(P) \cup T \models E$, and so $comp(P) \cup compdefs(U,H) \models E$ by transitivity of $\models$. Thus, by Lemma 6.1, $comp(B \cup H) \models E$. □

Thus α-nCTG is a sound inductive learning method for disjoint normal programs. Proposition 6.1, below, shows that the hypotheses space of α-nCTG is at least as large as the hypotheses space of nCTG.

Proposition 6.1. *Let $P = \langle B, U, I\rangle$ be a disjoint open program, let E be a set of ground literals such that $B \not\models E$. The set of hypotheses derivable from P and E by Normal Connected Theory Generalisation is also derivable by α-Normal Connected Theory Generalisation.*

Proof. Assume $P, E \vdash_{nCTG} H$ where H is a set of normal clauses. Then, by definition of nCTG, there are some *Normal Connected Theories* T such that $compdefs(U,H) \models T$. let T be in the following form

$$T = \left\{ \begin{array}{l} A^1 \leftarrow L^1_1, \cdots, L^1_{m_1} \\ \vdots \\ A^n \leftarrow L^n_1, \cdots, L^n_{m_n} \\ NL^1 \leftarrow L^{n+1}_1, \cdots, L^{n+1}_{m_{n+1}} \\ \vdots \\ NL^k \leftarrow L^{n+k}_1, \cdots, L^{n+k}_{m_{n+k+1}} \end{array} \right\},$$

and T' be

$$T' = \left\{ \begin{array}{l} A^1 \leftarrow L^1_1, \cdots, L^1_{m_1} \\ \vdots \\ A^n \leftarrow L^n_1, \cdots, L^n_{m_n} \\ NL^1 \\ \vdots \\ NL^k \end{array} \right\},$$

where A^i for $0 \leq i \leq p$ are ground atoms, $L^i_{j_u}$ for $0 \leq i \leq p+k$ and $1 \leq j \leq m$ and $1 \leq u \leq n+k$ are ground literals, nL^u for $1 \leq u \leq k$ are ground negative literals. We show $comp(P) \cup comdefs(U,H) \models T'$ and $compdefs(U,H) \models T'^{pos}$.

First, we show that $comp(P) \cup T \models T'$. By reflexivity of $\models$, we have $comp(P) \cup T \models \{A^1 \leftarrow L^1_1, \cdots, L^1_m, \cdots, A^p \leftarrow L^p_1, \cdots, L^p_m\}$. By Lemma 6.2, we have $comp(P) \cup T \models \{A^1, \cdots, A^p, nL^1, \cdots, nL^k\}$. By reflexivity of $\models$, $\{A^1 \cdots, A^p, nL^1, \cdots, nL^k\} \models \{nL^1, \cdots, nL^k\}$. By transitivity of $\models$, $comp(P) \cup T \models \{nL^1, \cdots, nL^k\}$. Therefore, $comp(P) \cup T \models \{A^1 \leftarrow L^1_1, \cdots, L^1_m, \cdots, A^p \leftarrow L^p_1, \cdots, L^p_m\} \cup \{nL^1, \cdots, nL^k\}$ which is same as $comp(P) \cup T \models T'$

Now, we show $comp(P) \cup comdefs(U,H) \models T'$. To proof $comp(P) \cup comdefs(U,H) \models T'$, it is sufficient to show $compdefs(U,H) \models comp(P) \rightarrow T'$ by definition of $\models$. Furthermore,

by $comp(P) \cup T \models T'$ and definition of $\models$, we have $T \models comp(P) \rightarrow T'$. Therefore, by transitivity of $\models$ and $compdefs(U, H) \models T$, we have $compdefs(U, H) \models comp(P) \rightarrow T'$.

Second, we show $compdefs(U, H) \models T'^{pos}$. This is simple, by reflexivity of $\models$, we have $T \models T'^{pos}$. Therefore, by transitivity of $\models$ and $compdefs(U, H) \models T$, we have $compdefs(U, H) \models T'^{pos}$.

Hence, $P, E \vdash_{nCTG} H$ only if $P, E \vdash_{\alpha-nCTG} H$ □

Lemma 6.2. [15] *Let $P = \langle B, U, I \rangle$ be a disjoint open program, let E be a set of ground literals such that $B \not\models E$, let T be an n-layered normal connected theory for P and E, T comprises n disjoint subsets $T_1, \cdots, T_n$, such that $comp(P) \models T_n$, and $comp(P) \cup T_n^h \cup \cdots \cup T_{i+1}^h \models T_i^b$ $(1 \leq i < n)$. Then, $comp(P) \cup T \models T_n^h \cup \cdots \cup T_1^h$ for all T_i^h $(n \geq i \geq 1)$.*

The proof for the Lemma 6.2 shown below is a part of the proof for Proposition 4.1 in [15]. To complete the proof of Proposition 6.1, it is also presented here.

Proof. This is shown by induction on i. The base case is to show that $comp(P) \cup T_n \models T_n^h$. Since $comp(P) \models T_n^b$, and $Tn \cup T_n^b \models T_n^h$, then $comp(P) \cup Tn \models T_n^h$. Assume the inductive hypothesis that: $comp(P) \cup T \models T_n^h \cup \cdots \cup T_j^h$ for all j $(n \geq j > i)$. Therefore, $comp(P) \cup T \models T_n^h \cup \cdots \cup T_{i+1}^h$ by the inductive hypothesis. So, by reflexivity of $\models$, $comp(P) \cup T \models comp(P) \cup T_n^h \cup \cdots \cup T_{i+1}^h$. Then, since $comp(P) \cup T_n^h \cup \cdots \cup T_{i+1}^h \models T_i^b$, by transitivity of $\models$, $comp(P) \cup T \models T_i^b$. So, since $T_i \subset T$, then $comp(P) \cup T \models T_i \cup T_i^b$ by reflexivity, and since $T_i \cup T_i^b| \models T_i^h$, by transitivity $comp(P) \cup T \models T_n^h \cup \cdots \cup T_i^h$. Therefore, since $comp(P) \cup T \cup T_n \models T_n^h$, by induction on i, $comp(P) \cup T \models T_n^h \cup \cdots \cup T_1^h$. □

Thus, this section has defined *α-Normal Connected Theory Generalisation* and has proved its soundness and the bottom line of its hypotheses space. The next chapter will provide details of ICC that applies *α-Normal Connected Theory Generalisation*. The procedure firstly compute a most specific *α-Normal Connected Theory* T, then search through all possible hypotheses H such that $comp(P) \cup compdef(U, H) \models T$.

CHAPTER 7

Proof Procedure

This chapter describes a proof procedure called Induction with Completion and Connected Theories (ICC). Given a normal open program $P = \langle B, U, I \rangle$ and sets of literal examples E^+ and E^-, ICC computes a hypothesis H that is a normal program, by generalising an α-*Normal Connected Theories* for P and E. The chapter is separated into three sections. The first two sections introduce two key features in ICC. The last section gives detail algorithms for ICC procedure.

7.1 Loop Checking

This section introduces a new loop checking method used in ICC and can successfully applied on ILP problems with normal program setting setting.

In IoF, the approach used to prevent loops is based on the notion of a *dependencies list*. If a theory is built, each clause in that theory has a *dependencies list* which is a collection of atoms which are depend on the head tom of that clause. For example, given the following open program $P = \langle B, \{x, u\}, \varnothing \rangle$

Example 7.1.

$$B = \left\{ \; u \leftarrow a, \; \right\}$$

$$E = \{x\}$$

A possible *Connected Theory* T for P and E is

$$T = \left\{ \begin{array}{ll} x \leftarrow u, & (\{\}) \\ a & (\{x, u\}) \end{array} \right\}$$

The list in the bracket at the end of each clause is the *dependencies list*. Since to prove u

and x we need a, both u and x are in the dependencies list of the second clause second. So, if a literal l is going to be added into the body of the second clause in T, both l and the abductive explanations for l should not be x or u. This prevent the theory like T' shown in Example 7.2 to be learned.

Example 7.2.

$$T' = \left\{ \begin{array}{l} x \leftarrow u, \\ a \leftarrow x \end{array} \right\}$$

In the ILP problem with the normal program setting, however, the *dependencies list* cannot solve all the loop problems. This is because, in an ILP problem with normal program setting, the loop can be caused by not only the positive atoms but also the negative literals. For example, given the ILP program in Example 7.3

Example 7.3.

$$B = \left\{ \begin{array}{l} p(X) \leftarrow \neg r(X), \\ r(X) \leftarrow s(X), t(X) \end{array} \right\}$$

$$E = \left\{ \begin{array}{l} a(1), s(2) \end{array} \right\}$$

$$modeh = \left\{ \begin{array}{l} a(+any), s(+any), t(+any) \end{array} \right\}$$

$$modeb = \left\{ \begin{array}{l} p(+any) \end{array} \right\}$$

A possible α-nCT T^{loop} and a possible hypothesis H^{loop} is given below.

$$T^{loop} = \left\{ \begin{array}{l} a(1) \leftarrow p(1) \\ s(2) \leftarrow p(2) \\ \neg t(1), \neg t(2) \end{array} \right\},$$

$$H^{loop} = \left\{ \begin{array}{l} a(X) \leftarrow p(X) \\ s(X) \leftarrow p(X) \end{array} \right\}.$$

Note, we assume the Kakas-Mancarella's abductive procedure [8] is used here. Thus one possible abductive result for $p(1)$ would be $\{\neg t(1), \neg r(1)\}$. Some other abductive system may generate different abductive result for $p(1)$. For example, A-System would generate $\{\neg s(1)\}$ and $\{s(1), \neg t(1)\}$ for $p(1)$.

Here, H^{loop} has a loop is because, if we query $?a(1)$ in Sicstus Prolog, by the first clause in H^{loop}, the Prolog will then query $?p(1)$. By $p(X) \leftarrow \neg r(X)$ in B, the next query would be $?fail\ r(1)$. To fail $r(1)$ then Prolog will query $r(1)$ to prove that $r(1)$ is not provable, by

$r(X) \leftarrow s(X), t(X)$ in B, the Prolog will firstly query $?s(1)$. However, $?s(1)$ lead to an query $p(1)$ already exists in the previous queries, then it goes into an infinite loop:

$$?a(1) \rightarrow \textbf{?p(1)} \rightarrow ?fail\ r(1) \rightarrow ?r(1) \rightarrow ?s(1) \rightarrow \textbf{?p(1)} \rightarrow \cdots$$

The question is how can we detect a looped hypothesis H^{loop} can be learned from the ground α-nCT T^{loop}. The answer proposed in this report is to assign indices to the predicates in the head modes. The details of the loop checking is present in the Algorithm 7.1. And the procedure can also applied for literal that is a secondary example.

Algorithm 7.1 The NoLoop(l, C, T) Procedure

Require: l is a ground literal, C is a normal clause
Ensure: b is either $true$ or $false$

```
1:  b := true
2:  for all (∅, Δ_l) := Abduce(l) do
3:     if Δ_l^+ ∩ depend(C) ≠ ∅ then
4:        b := false
5:        return b
6:     else if index(Δ_l^-) ≥ index(C^h) then
7:        b := false
8:        return b
9:     end if
10: end for
11: return b
```

The loops caused by positive literals are checked the same way as the loop checking in IoF (lines 3-5). The loops caused by negative literals are detected in the way shown in lines 6-8. The basic idea is for this loop checking procedure is that for any clause

$$HA \leftarrow Ts$$

learned from the system, where HA is the head atom and Ts is set of body literalfor the clause, there should not exist a literal $l \in Ts$ which is explained by some $\neg l'$ where the predicate of l' has a equal or higher index than index of the predicate of HA.

This idea is very similar to the idea of stratification in logic programs.

Definition 7.1. (Stratified Program [16]) *A normal program* Π *is stratified if there is a level mapping of* Π *such that, for every clause* $p(t_1, \cdots, t_n) \leftarrow L_1, \cdots, L_m$ *in* Π*, the level of the predicate symbol of every positive literal in* $L_1, \cdots, L_m$ *is less than or equal to the level of* p*, and the level of the predicate symbol of every negative literal in* $L_1, \cdots, L_m$ *is less than the level of* p*.*

The difference are that the levels are only mapped on those abduciable predicates and the stratification only happens when a literal has negative abduciables.

Returning to the example 7.3. The ground α-nCT T^{loop} will never be generated if we apply *NoLoop* procedure when building the theory. For example, if we give the following indices to the predicates in *modeh*

$$Indices = \left\{ \begin{array}{l} index(a, 3), \\ index(s, 2), \\ index(t, 2) \end{array} \right\}.$$

Then, the only α-nCT can be found are

$$T_1 = \left\{ \begin{array}{l} a(1) \leftarrow p(1) \\ s(2) \\ \neg t(1) \end{array} \right\}, T_2 = \left\{ \begin{array}{l} a(1) \leftarrow p(1) \\ s(2) \\ \neg s(1) \end{array} \right\}.$$

The reason of T^{loop} cannot be learned is that the second clause $s(2) \leftarrow p(2)$ in T^{loop} can never be formed since a possible adductive explanation for $p(2)$ is $\{\neg s(2)\}$ and $\neg s(2)$ has the same predicate as the head atom $s(2)$. Although $t(2)$ is assumed to be false and T^{loop} is a sound α-nCT, it will not be generated since the possible loop cause by its generalisations.

7.2 Subsumption Search

Given a open disjoint program $P = \langle B, U, I \rangle$, a set of literal examples E and an α-nCT T for P and E. This section provides a procedure that can generate a hypothesis H that subsumes the positive part of T and guaranteed $comp(P) \cup H$ also proves the negative part in T.
The details of *Search* procedure is provided in Algorithm 7.2. It performs an incremental learning on each predicate appears in T.

Algorithm 7.2 The SEARCH(T) Procedure

Require: ground α-normal connected theory T
Ensure: H is a normal program

```
1: H = ∅
2: while T ≠ ∅ do
3:    pick a set of clauses T^sub ∈ T that has same predicate in heads
4:    T := T - T^sub
5:    h := best(SEARCHDEFS(T^sub))
6:    if h ≠ ∅ then
7:       H := H ∪ h
8:    else
9:       return false
10:   end if
11: end while
12: return H
```

Returning to Example 4.3, given ILP probem with a disjoint open program $P = \langle B, U, \varnothing \rangle$ and examples E,

$$B = \left\{ \begin{array}{l} w(1), w(2), w(3), \\ s(2), s(3), \\ t(2), \\ p(X) \leftarrow q(X), \neg r(X) \end{array} \right\}$$

$$E = \left\{ \; p(1), r(2), r(3) \; \right\}$$

$$Modeh(U) = \{ \; q(+any), r(+any) \; \}$$

$$Modeb = \{ \; w(+any), s(+any), t(+any) \; \}$$

A possible α-nCT T would be

$$T = \left\{ \begin{array}{l} q(1) \leftarrow w(1) \\ r(2) \leftarrow w(2), s(2), t(2) \\ r(3) \leftarrow w(3), s(3) \\ \neg r(1) \end{array} \right\}.$$

If the *Search* Procedure is applied on the T, a group of clauses T^{sub} which is the subsect of T and has same head predicate will be first chose (line 3).

$$T^{sub} = \left\{ \; q(1) \leftarrow w(1) \; \right\}.$$

And the rest of T (line 4) is

$$T^{rest} = \left\{ \begin{array}{l} r(2) \leftarrow w(2), s(2), t(2) \\ r(3) \leftarrow w(3), s(3) \\ \neg r(1) \end{array} \right\}.$$

After picking the T^{sub}, a new non-deterministic procedure *SearchDefs* will be run on T^{sub}. The *SearchDefs* performs a top down subsumption search and searches for sets of possible clauses that subsumes the positive part in T^{sub} and ensure that those clauses will not prove the negative literals in T^{sub}. By this way, it implicitly compute a nCT that is implied by the $comp(U, H)$. The procedure *SearchDefs* is shown in the Algorithm 7.3.

Algorithm 7.3 The SEARCHDEFS(T) Procedure

Require: ground α-normal connected theory T, abductive explanations Δ, Mode declaration M

Ensure: h is a normal program

```
1:  h = ∅
2:  while T^pos ≠ ∅ do
3:    pick a clause C ∈ T^pos
4:    Anti-instantiate C
5:    c := C^h
6:    if the current c is not learnable by M then
7:      return false.
8:    else if the clause c in the current node does explain the original ground literal
        and fails to prove all the literals in T^neg with respect to comp(P) ∪ Δ then
9:      if c is subsumed by some clauses in h then
10:       h := h
11:     else
12:       remove all clauses that are subsumed by c in h
13:       h := h ∪ c
14:     end if
15:   else
16:     add one more body literals from C to c and go back to line 6
17:   end if
18: end while
19: return h
```

By applying *SearchDefs* on T^{sub}, we have the first part of the hypothesis $\{p(X)\}$. The process

of the search is shown in Algorithm 7.3:

$$p(X),\ Succeed$$

The search starts from $p(X)$ (line 5). Since $p(X)$ proves $p(1)$ and there is no negative literal with predicate p, the clause $p(X)$ satisfies the condition in line 8. Therefore, it is returned as a possible definition for predicate p.

Then, the *Search* procedure will be applied on the rest clauses in T (T^{rest}). Since T^{rest} only has clauses with head predicate r, it will be passed to *SearchDefs* procedure. Suppose the clause $r(2) \leftarrow w(2), s(2), t(2)$ is chosen first, then by anti-instantiation we have $r(X) \leftarrow w(X), s(X), t(X)$.

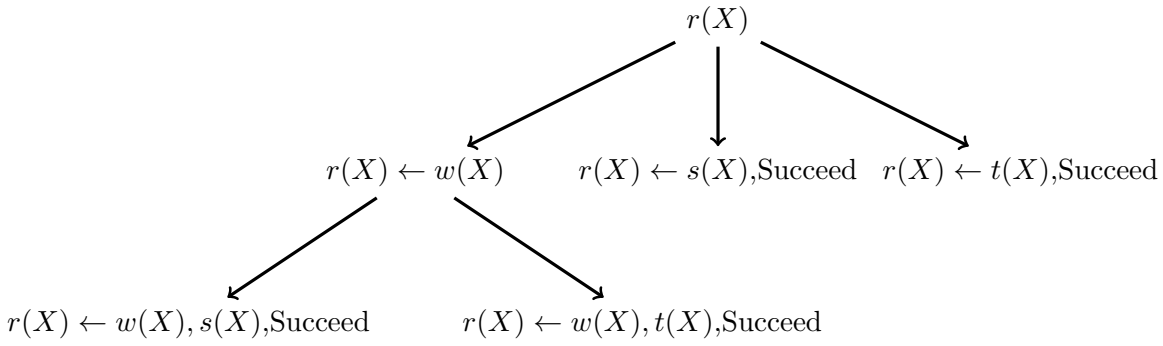

Figure 7.1: Search tree for $r(2) \leftarrow w(2), s(2), t(2)$

The search tree for $r(2) \leftarrow w(2), s(2), t(2)$ is given in Figure 7.2. The search starts from the top node $r(X)$ and expand this node by adding one body literal. Since it dose not satisfy the condition in line 8, the node will be expand by adding one body literal (lines 10-11). In this case, it has three options: $w(X)$, $s(X)$ and $t(X)$. By adding $s(X)$ and $t(X)$ into the body (The middle and right branch in the graph), the procedure stops with succeed since it explains $r(2)$ and fails $r(1)$ (line 9). However, the left branch is not closed since $r(X) \leftarrow w(X)$ proves $r(1)$. Thus this brunch will be expanded by adding one more body literals. And both $r(X) \leftarrow w(X), s(X)$ and $r(X) \leftarrow w(X), t(X)$ will be successfully returned.

For another clause $r(3) \leftarrow w(3), s(3)$, the search tree is shown below

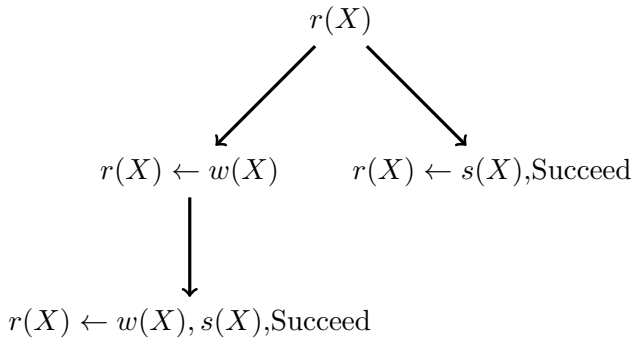

Figure 7.2: Search tree for $r(3) \leftarrow w(3), s(3)$

Thus, we have four possible clauses for $r(2)$ and two possible clauses for $r(3)$

$$Clauses\ for\ r(2) = \left\{ \begin{array}{l} r(X) \leftarrow w(X), s(X) \\ r(X) \leftarrow w(X), t(X) \\ r(X) \leftarrow s(X) \\ r(X) \leftarrow t(X) \end{array} \right\},$$

$$Clauses\ for\ r(3) = \left\{ \begin{array}{l} r(X) \leftarrow w(X), s(X) \\ r(X) \leftarrow s(X) \end{array} \right\}.$$

Combine with $p(X)$ learned previously (lines 9-14), there are five different results computed by the *SearchDefs* procedure:

$$H_1^{possible} = \left\{ \begin{array}{l} p(X) \\ r(X) \leftarrow s(X) \end{array} \right\}, \qquad H_2^{possible} = \left\{ \begin{array}{l} p(X) \\ r(X) \leftarrow t(X) \\ r(X) \leftarrow s(X) \end{array} \right\},$$

$$H_3^{possible} = \left\{ \begin{array}{l} p(X), \\ r(X) \leftarrow t(X) \\ r(X) \leftarrow w(X), s(X) \end{array} \right\}, \; H_4^{possible} = \left\{ \begin{array}{l} p(X) \\ r(X) \leftarrow w(X), s(X) \end{array} \right\},$$

$$H_5^{possible} = \left\{ \begin{array}{l} p(X) \\ r(X) \leftarrow w(X), t(X) \\ r(X) \leftarrow w(X), s(X) \end{array} \right\}.$$

If a hypothesis with less number of clauses and less number of body literals in each clause is more preferable (line 5 in Algorithm 7.2), then the *Search* procedure will return the hypothesis $H_1^{possible}$ as the best hypothesis H for T:

$$H = \left\{ \begin{array}{l} p(X) \\ r(X) \leftarrow s(X) \end{array} \right\}.$$

7.3 Procedure

Given a normal open disjoint program $P = \langle B, U, I \rangle$, and a set of literal examples E, ICC computes one or more than one correct hypotheses H that is a normal program. The top level procedure is shown in Algorithm 7.4.

Algorithm 7.4 The MAIN(P, E, n) Procedure

Require: P is a normal open disjoint program
Require: E is a set of literals
Require: n is positive integer and is larger than $|E^+|$
Ensure: H is a correct inductive hypothesis for P, E

1: $\Delta := \varnothing$
2: $H := E$
3: **for all** consistent Δ such that $\Delta :=$ ABDUCEALL(E) **do**
4: $\quad H' := H$
5: $\quad T :=$ SATURATESET(Δ)
6: $\quad H :=$ NCTSEARCH(H', T, n)
7: **end for**
8: **return** H

Different from IoF in which picks a seed example at the beginning of the procedure, the ICC procedure starts by abducing all the examples in E (line 3). Then, a hypothesis H will be computed (lines 4-8) and $comp(B \cup H)$ will successfully explains all the examples in E. The integer n is a parameter that restricts the maximum number of ground clauses in the T^{pos}. Initially, H is set to be E. Thus, if no hypothesis searched is preferable than E, the procedure will return E^+. The criteria used to determine whether one hypothesis H is preferred over another is simply the structure of the hypothesis: number of clauses, length of each clause. The exact choice is implementation-specific, and does not form part of this procedure. Moreover, the indices of the head mode predicates (see Section 7.1) is also a part of the problem, therefore not included in the this procedure.

Example 7.4.

$$B = \left\{ \begin{array}{l} w(1), w(2), w(3), \\ s(2), s(3), \\ t(2), \\ p(X) \leftarrow q(X), \neg r(X) \end{array} \right\}$$

$$E = \left\{ \; p(1), r(2), r(3) \; \right\}$$

$$Modeh = \{ \; q(+any), r(+any) \; \}$$

$$Modeb = \{ \; w(+any), s(+any), t(+any), \neg q(+any), \neg r(+any) \; \}$$

The above example is slightly modified from Example 4.3 where two new body modes $\neg r(+any)$ and $\neg q(+any)$ are allowed. For the better illustration, we assume the index assigned to q, r are 1 and 2:

$$Indices = \left\{ \begin{array}{l} index(q,1), \\ index(r,2) \end{array} \right\}.$$

The first step is to compute sets of abductive explanations Δ for E (line 3 in Algorithm 7.4). Here, there is only one abductive explanation $\Delta = \{q(1), \neg r(1), r(2), r(3)\}$.

The next step is to saturate $\Delta^+ = \{q(1), r(2), r(3)\}$ (line 5 in Algorithm 7.4). The *SaturateSet* procedure adds all possible body literals to members in $\{q(1), r(2), r(3)\}$ that do not require extra clauses to explain them.

Algorithm 7.5 The SATURATESET(T) Procedure

Require: T is a ground α-normal connected theory
Ensure: T is a α-ground normal connected theory in which the positive part is saturated

```
1: while T has clause atom in head do
2:    Choose a clause C ∈ T with positive head
3:    T := T − C
4:    for all L := POSSIBLEBODYLITERAL(C) do
5:       body(C) := body(C) ∪ {L}
6:    end for
7:    T' := T' ∪ {C}
8: end while
9: T' := T' ∪ T
10: return T'
```

The procedure *PoosibleBodyLiteral* returns those possible body literals for a given clause C. Since the negative literals are allowed to be inferred in the normal program setting, the criteria for adding a literal into the body of a clause in a bit more complicated than criteria in IoF. The loop checking procedure *NoLoop* (details in Section 7.1) will used here. The detailed description of the procedure is shown in Algorithm 7.6.

Generally speaking, the procedure *PossibleBodyLiteral* returns a literal l which can be proved by $comp(P) \cup \Delta$ and cannot cause any loop in both ground theory and generalised hypotheses if it is added into the clause C.

The key part of *PossibleBodyLiteral* is that all positive and negative literals in $\Delta_{mb\theta}$ are included in T^h'. In addition to the saturation phase in IoF, those literals that have some extra negative abducibles that have head mode predicates are treated as secondary examples, and therefore can not be added into the picked clause. For example, the positive literal $p(2)$ does not satisfy this condition since to prove $p(2)$ we need to prove $\neg r(2)$. The predicate r

Algorithm 7.6 The PossibleBodyLiteral(C) Procedure

Require: ground normal clause$C$, normal program B, mode declaration M, abductive explanations Δ

Ensure: T is a ground literal

1: $L := \varnothing$
2: **Choose** a body mode mb in M
3: instantiate the inputs of mb using terms in C
4: $(\theta, \Delta_{mb\theta}) :=$ Abduce(mb)
5: **if** $\Delta_{mb\theta}$ is consistent with Δ
and all positive and negative literals in $\Delta_{mb\theta}$ are included in T^h
and $true :=$ NoLoop(mb, C, T) **then**
6: $L := mb\theta$
7: **end if**
8: **return** L

is an open predicate so we cannot infer $\neg r(2)$. Therefore, in case we want to add $p(2)$ into some clauses' bodies, an auxiliary clause $\neg r(2)$ is needed.

For the Example 7.4, after saturate the set of abductive explanations Δ. The first 1-layered *α-Normal Connected Theory* T_0 is built:

$$T_0 = \left\{ \begin{array}{l} q(1) \leftarrow w(1), \\ r(2) \leftarrow w(2), s(2), t(2) \\ r(3) \leftarrow w(3), s(3) \\ \neg r(1) \end{array} \right\}.$$

Then, the procedure *NCTSearch*, showed in Algorithm 7.7, will be called and searches for all hypotheses with $n = i = 4$ clauses or fewer. Similar to the *CTSearch* procedure in IoF [11]. The *NCTSearch* procedure gradually generalises T_0 to a larger ground α-nCT (line 9) relative to B and simultaneously searches hypotheses from each α-nCT by the procedure *Search* introduced in 7.2 (line 4, 7). The *Compare* procedure simply returns the more preferable hypothesis between two input hypotheses.

The procedure *AddSE* in Algorithm 7.8 adds secondary examples in T. The idea of *AddSE* procedure is simple. The secondary example is either a positive or negative literal and needs some extra clauses that do not exist in T to explain.

Recall that the indices for q, r are $1, 2$, we can form the following theories by adding secondary

Algorithm 7.7 The NCTSEARCH(h, T^{pos}, i) Procedure

Require: h is a normal program
Require: T is a ground α-Normal Connected Theory
Require: i is positive integer
Ensure: h' is a normal program

```
1: if |T^pos| > i then
2:    h' := h
3: else if |T^pos| = i then
4:    h' := SEARCH(T)
5:    h' := COMPARE(h', h)
6: else
7:    h' := SEARCH(T)
8:    h' := COMPARE(h', h)
9:    for all non-empty T' such that T' := ADDSE(T) do
10:      h' := NCTSEARCH(h', T', i)
11:   end for
12: end if
13: return h'
```

examples $\neg q(2)$ and $\neg q(3)$

$$T_1 = \left\{ \begin{array}{l} q(1) \leftarrow w(1) \\ r(2) \leftarrow w(2), s(2), t(2), \neg q(2) \\ r(3) \leftarrow w(3), s(3) \\ \neg r(1), \neg q(2) \end{array} \right\},$$

$$T_2 = \left\{ \begin{array}{l} q(1) \leftarrow w(1) \\ r(2) \leftarrow w(2), s(2), t(2) \\ r(3) \leftarrow w(3), s(3), \neg q(3) \\ \neg r(1), \neg q(3) \end{array} \right\},$$

$$T_3 = \left\{ \begin{array}{l} q(1) \leftarrow w(1) \\ r(2) \leftarrow w(2), s(2), t(2), \neg q(2) \\ r(3) \leftarrow w(3), s(3), \neg q(3) \\ \neg r(1), \neg q(2), \neg q(3) \end{array} \right\}.$$

No secondary example that is instantiated from $\neg r(+any)$ is added into any of clauses in $T_i (0 \leq i \leq 3)$. It is because $r(X)$ (X can be any terms in P) has the predicate index 2 and none of the clause in $T_i (0 \leq i \leq 3)$ has a head atom whose predicate has a index larger than 2. Similarly, $q(1)$ is not added into the first clause in T_0 since $q(1) \leftarrow w(1)$ has the predicate p in head.

Each of these T_i $(0 \leq i \leq 3)$ will be searched by the procedure *Search*. Assume a hypothesis with less number of clauses and less number of body literals in clauses is more preferable, the

Algorithm 7.8 The AddSE(T) Procedure

Require: normal disjoint open program P, ground α-normal connected theory T, mode declaration M

Ensure: T' is a ground α-normal connected theory

```
1: T' := ∅
2: Choose a clause C ∈ T
3: Choose a body mode mb in M
4: instantiate mb with ground terms in P
5: if mb ∉ C^b
       and true := NoLoop(mb,C,T) then
6:    (∅, Δ_mb) := Abduce(mb)
7:    if mb ∉ body(C)
          and Δ_mb is consistent with T^h then
8:       T' := T − {C}
9:       C' := C
10:      T' := T' ∪ Δ_mb
11:      body(C') := body(C') ∪ {mb}
12:      T' := T' ∪ {C'}
13:      T' := SaturateSet(T')
14:   end if
15: end if
16: return T'
```

hypothesis learned from each T_i is shown below:

$$T_0 = \left\{ \begin{array}{l} q(1) \leftarrow w(1) \\ r(2) \leftarrow w(2), s(2), t(2) \\ r(3) \leftarrow w(3), s(3) \\ \neg r(1) \end{array} \right\} \Rightarrow H_0 = \left\{ \begin{array}{l} q(X) \\ r(X) \leftarrow s(X) \end{array} \right\}$$

$$T_1 = \left\{ \begin{array}{l} q(1) \leftarrow w(1) \\ r(2) \leftarrow w(2), s(2), t(2), \neg q(2) \\ r(3) \leftarrow w(3), s(3) \\ \neg r(1), \neg q(2) \end{array} \right\} \Rightarrow H_1 = \left\{ \begin{array}{l} q(X) \\ r(X) \leftarrow s(X) \end{array} \right\}$$

$$T_2 = \left\{ \begin{array}{l} q(1) \leftarrow w(1) \\ r(2) \leftarrow w(2), s(2), t(2) \\ r(3) \leftarrow w(3), s(3), \neg q(3) \\ \neg r(1), \neg q(3) \end{array} \right\} \Rightarrow H_2 = \left\{ \begin{array}{l} q(X) \\ r(X) \leftarrow s(X) \end{array} \right\}$$

$$T_3 = \left\{ \begin{array}{l} q(1) \leftarrow w(1) \\ r(2) \leftarrow w(2), s(2), t(2), \neg q(2) \\ r(3) \leftarrow w(3), s(3), \neg q(3) \\ \neg r(1), \neg q(2), \neg q(3) \end{array} \right\} \Rightarrow H_3 = \left\{ \begin{array}{l} q(X) \\ r(X) \leftarrow \neg q(X) \end{array} \right\}.$$

Since all of $H_i (0 \leq i \leq 3)$ have same number of clauses and same number of body literals, the best hypothesis can be any of $H_i (0 \leq i \leq 3)$.

So far, we have presented the detail algorithms for ICC. The next section will present an implementation of ICC and critically evaluated the ICC procedure based on that implementation.

CHAPTER 8

Implementation and Evaluation

This chapter will firstly gives a brief description of the implementation of ICC procedure *icc.pl* with respect to its top level predicates and the format of input file that describes the problem. Then, a short evaluation on the system will be provided.

8.1 Implementation

icc.pl is an implementation of the ICC procedure. It was written in Sicstus Prolog. The implementation is designed to prefer hypotheses with smaller structure, i.e. smaller number clauses and smaller number of literals in clauses. The main predicates in *icc.pl* are listed below.

- $main(+FileName, -Best)$
  This is the toppest predicate in *icc.pl*. It integrates the predicates for abduction, saturation, adding secondary examples and searching together. Argument $FileName$ is the name of the input file. And $Best$ is the original data type of the best hypothesis returned. For example, if there is a prolog file $FileName.pl$ that defines an ILP problem, then query $main(FileName, B)$ will give the result for this problem. The $main/2$ is defined in an non-deterministic manner. The number of outputs returned by this predicate depends on the number of abductive explanations for the examples and open program defined in $FileName.pl$.

- $ct\_search/8$
  It is the correspond implementation of $NCTSearch$ procedure which interleaves the $AddSE$ procedure and $Search$ procedure. The function takes a list of different saturated theories as the input and returns the most successful hypothesis derived from these theories by continuously adding secondary examples and update the current best hypothesis.

- *saturate_CT*(+*ToSat*, −*Saturated*)
  It is the correspond implementation of *SaturateSet* procedure. The function contains a sub-function *possibe_body_literal*/2 which is the implementation of *PossibleBodyLiteral* procedure. As noted by Kimber [14], different selection order of clause to saturate could result in different consequences. The function is implemented in a non-deterministic manner and can generate all the possible results (*Saturated*) by saturating *ToSat*. Therefore, this predicate is always used with the Prolog built-in function *findall*/3 to generated a list contains all possible saturated sets and followed by some procedures that can remove duplicates in that list.

- *Abduce*/4
  Abduce/4 is the correspond implementation of *AbduceAll* procedure. And the Kakas-Mancarella's (KM) interpreter is used as the underlying abductive engine. The function takes a list of ground literal example as the input and returns a set of abduciables and a set of transformed form of abduciables for the input examples. The transformed form of an abduciable is simply a list representation of a clause. Such a list contains a ground clause, a corresponding flattened clause, mode declarations for literals in the clause, ground terms in the clause which can be used as inputs, bindings from the flattened clause to the ground clause and the dependencies list (details in Section 7.1) of the clause.

The input file which describes the ILP problem is compsed by four parts. The first part is a set of parameters constrains the hypothesis space. An example of the constraints is shown in

```
set_max_clause_length(4).
set_max_ground_clauses(10).
set_complete_saturation(0).
set_connected(1).
set_clause_weight(5).
set_literal_weight(1).
```

Figure 8.1: Example of constraints on hypotheses

The *set_max_clause_length*(4) defines the maximum length of each clause in the returned hypothesis, it defines the maximum depth of the search tree (see Section 7.2). Here, this maximum depth is 4. *set_max_ground_clauses*(10) restrict the maximum ground clause with positive head in the bridge formula to be 10. It also means the maximum number clauses in the returned hypothesis is 10. *set_complete_saturation*/1 is the switch of the completed saturation. The completed saturation means try every selection orders of clause when saturate a theory. In ICC procedure, each ground theory is formed by *all* the examples in ICC, the size of the bridge formula would be considerably larger than bridge formulas in those systems which perform an incremental learning. Moreover, in most case of simple ILP problem, the completed saturation of a theory would result in a same saturated theory. Thus, it would be useful to give an restriction on saturation. The parameter 0 in *set_complete_saturation*/1 means

the system do not perform a completed saturation. And 1 means do perform completed saturation when *Saturate_CT* is called. The *set_connected*/1 with parameter 1 means secondary examples are allowed and 0 otherwise. In *icc.pl*, each hypothesis is assigned a score which is calculated from *set_clause_weight*/1 and *set_literal_weight*/1. Here, *set_clause_weight*(5) and *set_literal_weight*(1) means a hypothesis with n clauses and m literals will have a score $n*5+m*1$. Since *icc.pl* is designed to prefer a hypothesis with smaller structure, a hypothesis with a lower score is more preferable.

The other three parts are similar to the way that most ILP systems define the problem. They are a set of Prolog rule which represents background knowledge, a list of *ground* literal examples and two lists of mode declaration (one for head modes and another for body modes). Detailed examples of this input file can be found in Appendix. The indices for the head mode predicates is set automatically. The indeices of predicates appearing at the left hand said of the list are assigned a lower index than the predicates appearing at the right hand side. For example, if we have the following head modes,

Example 8.1. `head_modes( [p(+any), q(+any), r(+any)] ).`

then, the indices assigned to each predicates would be

$$indices = \left\{ \begin{array}{l} index(p/1, 0) \\ index(q/1, 1) \\ index(r/1, 2) \end{array} \right\}.$$

8.2 Evaluation

Generally speaking, the ILP system *icc.pl* successfully retain the idea of using secondary examples to widen the hypotheses space under definite program settings [11] and extend it to solve ILP problems with normal program settings. A typical case study to show this nature of the system is

Example 8.2.

$$B = \left\{ \begin{array}{l} flies(X) \leftarrow bird(X), \neg ab(X) \\ bird(X) \leftarrow penguin(X) \\ bird(tweety) \\ penguin(polly) \\ penguin(peter) \\ has\_big\_wings(peter) \\ has\_big\_wings(X) \leftarrow has\_big\_wings2(X) \\ ab2(X) \leftarrow penguin(X), has\_big\_wings(X) \end{array} \right\},$$

$$E = \left\{ \begin{array}{l} flies(tweety), flies(peter), \\ \neg flies(polly) \end{array} \right\},$$

$$Modeh\ (U) = \{\ has\_big\_wings2(+any), ab(+any)\ \},$$

$$Modeb = \left\{ \begin{array}{ll} bird(+any), & \neg bird(+any), \\ penguin(+any), & \neg penguin(+any), \\ ab2(+any), & \neg ab2(+any) \end{array} \right\}.$$

The above example is modified from classic bird-penguin problem which is widely use in demonstrating the non-monotonicity of normal programs. Aside from *tweety* and *polly*, a new member *peter* who is a penguin can fly is added to the background knowledge and example. And the bird like *peter* are viewed as a new kind of abnormal, defined by $ab2$. Given the disjoint normal open program $P = \langle B, U, I \rangle$ and examples E, We are going to learn rules for ab and has_big_wings2. Note here, has_big_wings2 has a lower index than ab. The task's objective is to learn the rule $ab(X) \leftarrow penguin(X), \neg ab2(X)$.

If *icc.pl* runs with this example with proper constraints, it will firstly compute the following abduciables

$$\Delta = \{ab(polly), \neg flies(polly), \neg ab(peter), \neg ab(tweety)\}.$$

Then by saturation, a theory T_0 without using any secondary examples is computed

$$T_0 = \left\{ \begin{array}{l} ab(polly) \leftarrow bird(polly), penguin(polly) \\ \neg ab(peter) \\ \neg ab(tweety) \end{array} \right\}.$$

and no hypothesis is searched from T_0. If secondary examples are allowed, then there are two possible secondary examples - $ab2(polly)$ and $\neg ab2(polly)$ and obviously they cannot exist in the same clause. By using positive secondary example $ab2(polly)$, a new theory T_1 is formed

$$T_1 = \left\{ \begin{array}{l} ab(polly) \leftarrow bird(polly), penguin(polly), ab2(polly) \\ has\_big\_wings2(polly) \leftarrow bird(polly), penguin(polly) \\ \neg ab(peter), \neg ab(tweety) \end{array} \right\}.$$

and again no hypothesis is drived. However, if the negative secondary example $\neg ab2(polly)$ is used, then a new theory T_3 is computed

$$T_3 = \left\{ \begin{array}{l} ab(polly) \leftarrow bird(polly), penguin(polly), \neg ab2(polly) \\ \neg ab(peter), \neg ab(tweety) \\ \neg has\_big\_wings2(polly) \end{array} \right\}.$$

and the objective rule $ab(X) \leftarrow penguin(X), \neg ab2(X)$ can be derived from T_3. The output generated by *icc.pl* is shown below:

```
---- Abductive Result: ----

[ab(polly),not(flies(polly)),not(ab(peter)),not(ab(tweety))]

---- The GROUND hypothesis is: ----

ab(polly) :- bird(polly), penguin(polly), not(ab2(polly))

not(ab(peter))
not(ab(tweety))
not(has_big_wings2(polly))


----**** The Best hypothesis is: ****----

ab(A) :- penguin(A), not(ab2(A))

Time Cost: 0.010000000000000009 seconds
```

The system is also tested on the soldier example (see Example 1.1) in [15], Example 7.4 and

several ILP tasks with definite program settings including Yamamoto's example (see Example 3.2) [30] and examples collected in [2]. It performs well in most of the case. The details are provided in Appendix. It is also showed that by allowing negative literals to be included in the body of a clause, some better hypotheses may generated (see Example oddeven in Appendix).

However, at the present time, the procedure still have some problems. The most significant problem is that if a negative body mode has output argument, a floundered hypothesis may be searched and even be returned as the best hypothesis. Consider the following example

Example 8.3.

$$B = \left\{ \; q(3,3) \; \right\},$$

$$E = \left\{ \begin{array}{l} p(1), p(2) \\ \neg p(3) \end{array} \right\},$$

$$Modeh \; (U) = \{ \; p(+any) \; \},$$

$$Modeb = \left\{ \; \neg q(+any, -any) \; \right\}.$$

and the rule $h = \{p(X) \leftarrow \neg q(X,Y)\}$ can be derived from the ground formula T by ICC procedure. Note the theory T shown below is not a fully saturated theory produced by ICC procedure.

$$T = \left\{ \begin{array}{l} p(1) \leftarrow \neg q(1,2) \\ p(2) \leftarrow \neg q(2,1) \\ \neg q(3,3) \end{array} \right\}.$$

In Sicstus Prolog, the clause $p(X) \leftarrow \neg q(X,Y)$ will be treated as

$$\left\{ \begin{array}{l} p(X) \leftarrow \neg p'(X) \\ p'(X) \leftarrow q(X,Y) \end{array} \right\} \tag{8.1}$$

and it will be a correct hypothesis if it is run under Sicstus Prolog. However, h is not a valid hypothesis in many other systems since there is a negative non-ground sub-goal in the SLDNF-tree (floundering) when it is used to explain examples in E. Thus, such body modes with outputs like $p(+any, -any)$ is currently not allowed in *icc.pl*. A possible way to solve this is to add a transformation procedure in the *Search* procedure (Section 7.2) such that the procedure will automatically transform clauses like $p(X) \leftarrow \neg q(X,Y)$ into the form of 8.1 by invent new predicates like p'. An informal description of this procedure is shown below

- Let $H \leftarrow Body$ be a clause in a node of the *Search* procedure.
- For any negative body literal $\neg Pred(Terms)$ in $Body$ where $Pred$ is the predicate, and $Terms$ are the terms in this literal, if it is generalised from an instantiation of a body

mode M which has some placemarkers for outputs, then Replace $\neg Pred(Terms)$ by $\neg Pred'(Terms')$ where $Terms'$ are terms in $Terms$ and has corresponding placemarker with '+' or '#' in M.

- Lastly, add an extra clause $Pred'(Terms') \leftarrow Pred(Terms)$ to the node.

However, because the time is limited for this project, we cannot decide whether it is a feasible way to solve this problem.

The second problem is concerned with the scalability of current system. The system is very sensitive to the problem size. It performs well on small problems. However, the computation time and memory requirement dramatically increased when the problem size is going large. The system was tested on east-west train problem [18]. In this ILP problem, there are five trains $\{e1, e2, e3, e4, e5\}$ moving towards east and treated as positive examples (details see Appendix). If all five trains are used the Prolog will point out there is no sufficient memory. The same situation happened when four of these eastbound trains are used. However, the correct hypothesis will be returned if three trains are use. But the whole process cost a huge amount of time (See appendix). If only two of the trains are used, then the system will also give the correct hypothesis but just cost a small amount of time. This issue is not easy to solve currently since the procedure is designed to process all examples together, and this makes the system inevitably requires a large memory when running on big tasks. However, *icc.pl* may not be the best way to implement ICC procedure. Improvements like using a better data structure or reduce re-computation are feasiable solutions for this problem.

The last main problem of ICC procedure is the loop checking. As mentioned before, the way ICC procedure deals with loops caused by negative literals is similar to perform a global stratification. This makes some correct hypotheses to be treated as invalided hypothesis as showed by Example 3.5.

$$B = \left\{ \ father(peter, chris) \ \right\},$$

$$E = \{parent(peter, chris)\},$$

$$H = \{parent(peter, chris) \leftarrow father(peter, chris), \neg father(chris, peter), \neg parent(chris, peter)\}.$$

The H is a correct hypothesis for the problem if $parent/2$ is allowed in the head and predicates $father/2$, negations of $father/2$ and negations of $parents/2$ are allowed to be in the body. However, the current ICC procedure do not allow the hypothesis clause with a negative body literal which has the same predicate of the head. A new heuristic that can perform a "locoal stratification" on negative loop checking would be necessary. Since the current procedure lost some hypotheses in the hypotheses space of *α-Coonected Theory Generalisation.*

CHAPTER 9

Conclusion and Future Work

This report has presented a new ILP procedure *Induction with Completion and Connected theories* (ICC) which can solve ILP problems with non-monotonic settings. The procedure is based on the new operational ILP approach called α-*Normal Connected Theory Generalisation* (α-nCTG).

The concept of α-nCTG is generated based on Kimber's *Normal Connected Theory Generalisation* (nCTG). It has been defined and has been proved that this new operational ILP approach will not lose any hypotheses that are learnable from nCTG.

The ICC procedure has been developed to compute α-*Normal Connected Theories* (α-nCT) from normal background programs. The ICC procedure is designed based on Kimber's IoF proof procedure [11] which is designed for learning inductive hypotheses from definite background programs. ICC procedure processes all the examples in one go, and therefore overcome the difficulties associated with iterative leaning approach. Some α-nCTs are build based on examples and are used as a bridge formulas in the learning process. The generalisation process is designed based on the α-nCTG. And new loop checking and subsumption search procedures are introduced. An preliminary implementation *icc.pl* is provided and is evaluated based on several small to medium ILP tasks with both definite and normal program settings. It shows the current procedure is able to solve small ILP tasks but not scalable to ILP problems with medium or large problem size. Other issues including generating floundered hypotheses and incompleteness caused by "global stratification" are remained to solve.

Following from the evaluation, the first and the most urgent future investigation is to redesign the search procedures and make sure that the hypotheses generated by ICC procedure are at least "correct". The way of defining a sub-procedure for inventing new predicates and transforming the clauses into an acceptable form in the search phase would be a possible solution. However, it still needs a much deeper investigation and testing. Reimplementing the

current *icc.pl* to improve the efficiency would also be an good area for the future investigation. The reimplementation includes using new data structures, reducing re-computation in the current system and use more efficient abductive engine. Moreover, the procedure for loop checking can be modified so that relief the problem of incompleteness caused by stratification to some extent. Finally, On the theory side, the completeness of α-nCTG would be an good area to explore, although it seems that it makes no extension on Kimber's nCTG.

Bibliography

[1] Ade, H. and Denecker, M.: *AILP abductive inductive logic programming.* In IJCAI95: Proceedings of the 14th International Joint Conference on Artificial Intelligence, pages 12011207, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[2] Athakravi, D., Corapi, D., Broda, K., Russo, A.: *Example Learning Tasks.* Available at: http://ilp13.cos.ufrj.br/

[3] Clark, K. L.: *Negation as failure.* In H. Gallaire and J. Minker, editors, Logic and Databases, pages 293322. Plenum Press, New York, 1978.

[4] Corapi, D., Russo, A., Lupu, A. E., *Inductive Logic Programming as Abductive Search.* Technical Communications of the Internatinal Conference on Logic Programming, pp.54-64, Edinburgh, 2012.

[5] Denecker, M. and Schreye, D., D.: *Representing incomplete knowledge in abductive logic programming.* Journal of Logic and Computation, 5(5):553577, 1995.

[6] Gelfond, M., and Lifschitz, V.: *The stable model semantics for logic programming.* In R. A. Kowalski and K. Bowen, editors, Proceedings of the Fifth International Conference on Logic Programming, pages 10701080, Cambridge, Massachusetts, 1988. The MIT Press.

[7] Gelfond, M., and Lifschitz, V.: *Logic programs with classical negation.* In Proceedings of the International Conference on Logic Programming, pages 579-597, 1990.

[8] Kakas,A. and Mancarella,P.: *Database updates through abduction.* In Proceedings of the 16th International Conference on Very Large Databases, pages 650661. Morgan Kaufmann, 1990.

[9] Kakas, A., Kowalski, R., and Toni F.: *Abductive logic programming.* Journal of Logic and Computation, 2(6):719770, 1992.

[10] Kowalski, R.,: *Predicate logic as programming language.* Information Processing, 74:569574, 1974.

[11] Kimber, T., Broda, K., Russo, A.: *Induction on Failure: Learning Connected Horn Theories.* In Esra Erdem, Fangzhen Lin, Torsten Schaub, editors, Logic Programming and

Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings. Volume 5753 of Lecture Notes in Computer Science, pages 169-181, Springer, 2003.

[12] Kimber, T.: *Background*. PhD Thesis, Imperial College London, Chapter 2, pp. 31-75 2012.

[13] Kimber, T.: *Inductive Logic Programming: ILP Terminologies*. PhD Thesis, Imperial College London, Chapter 3, pp. 86-88 2012.

[14] Kimber, T.: *Inductive on Failure*. PhD Thesis, Imperial College London, Chapter 5, pp. 117-135 2012.

[15] Kimber, T.: *Normal Conneted Theory*. PhD Thesis, Imperial College London, Chapter 7, pp. 151-187 2012.

[16] Lloyd, J. W.: *Foundations of Logic Programming*. Springer-Verlag, 1987

[17] Lavrac, N., Dzeroski, S.: *Review of "Inductiove Logic Programming: Techniques and Applications"*. Kluwer Academic Publisher, Boston, 1996.

[18] Michalski, R.S., Larson, J.B.: *Inductive inference of VL decision rules*. Paper presented at Workshop in Pattern-Directed Inference Systems, Hawaii, 1977. SIGART Newsletter, ACM, 63, 38-44. 1977

[19] Muggleton, S.: *Bayesian inductive logic programming*. In M. Warmuth, editor. Proceedings of the Seventh Annual ACM Conference on Computational Learning Theory,ACM Press, New York, pages 3-11, 1994.

[20] Muggleton, S.: *Inductive Logic Programming: Theory and methods*. Journal of Logic Programming, 20:629-679, 1994.

[21] Muggleton, S.: *Inverse Entailment and Progol*. Oxford University Computing Laboratory, Oxford, 1995.

[22] Muggleton, S.: *Learning from positive data*. Oxford University Computing Laboratory, 1997.

[23] Muggleton, S.: *Inductive Logic Programming: issues, results and the challenge of challenge of Learning Language in Logic*. Department of Computer Science, University of York, 1999.

[24] Nienhuys-Cheng, S. H., Wolf, R. D.: *Foundations of inductive Logic Programming*. Springer Verlag, Berlin, 1997.

[25] Ray, O., Broda, K., Russo, A.: *Hybrid Abductive Inductive Learning: A Generalisation of progol*. Proceedings of the 13th International Conference on Inductive Logic Programming, volume 2835 of Lecture Notes in AI, 2003.

[26] Ray, O.: *Nonmonotonic abductive inductive learning.* Journal of Applied Logic, 7:329340, 2009.

[27] Reiter, R.: *On closed world data bases.* In H. Gallaire and J. Minker, editors, Logic and Data Bases, pages 5576. Plenum Press, New York, 1978.

[28] Sakama, C.: *Induction From Answer Sets in Nonmonotonic Logic Programs.* ACM Transactions on Computational Logic, Vol. 6, No. 2, Pages 203231, April 2005.

[29] Shoham, Y.: *Nonmonotonic logics: meaning and utility.* In: Proc. 10th International Joint Conference on Articial Intelligence, Morgan Kaufmann, pp. 388393, 1987.

[30] Yamamoto, A.: *Which hypotheses can be found with inverse entailment?* In S. Dzeroski and N. Lavrac, editors, Proceedings of the 7th International Workshop on Inductive Logic Programming, volume 1297, pages 296308. Springer-Verlag, 1997.

Appendix

Example soldier

The **input** file *soldier.pl*:

```
set_max_clause_length(4).
set_max_ground_clauses(10).
set_complete_saturation(0).
set_connected(0).

set_clause_weight(5).
set_literal_weight(1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Soldier Example
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% background %%
begin_of_background.

obeys(X,Y) :- not(officer(X)),officer(Y).
wears_hat(price).
wears_hat(osbourn).
has_stripe(osbourn).

end_of_background.

%% examples %%
examples( [obeys(price,osbourn), not(obeys(osbourn,price))] ).

%% mode declaration %%
%
%  NOTE: Only '+''#' can be included in not(_) currently
```

```
head_modes( [officer(+any)] ).
body_modes( [
wears_hat(+any), has_stripe(+any),
not(wears_hat(+any)), not(has_stripe(+any))
] ).
```

The **output** produced by *icc.pl*:

```
---- Abductive Result: ----

[not(obeys(osbourn,price)),officer(osbourn),not(officer(price))]

---- The GROUND hypothesis is: ----

officer(osbourn) :- wears_hat(osbourn), has_stripe(osbourn)

not(officer(price))



----**** The Best hypothesis is: ****----

officer(A) :- has_stripe(A)

Time Cost: 0.009999999999999787 seconds
```

Example 7.4 in Section 7

The **input** file *problem2.pl*

```
set_max_clause_length(4).
set_max_ground_clauses(10).
set_complete_saturation(0).
set_connected(1).

set_clause_weight(5).
set_literal_weight(1).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Simple Example
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  background
begin_of_background.
w(1).
w(2).
w(3).
s(2).
s(3).
t(2).
p(X) :- q(X), not(r(X)).
end_of_background.

% examples
examples( [p(1),r(2),r(3)] ).

%  mode declaration
%
%  NOTE: Only '+''#' can be included in not(_) currently
head_modes( [q(+any), r(+any)] ).
body_modes( [w(+any), s(+any), t(+any),
             not(p(+any)), not(q(+any)), not(r(+any))] ).
```

The **output** produced by *icc.pl*

```
---- Abductive Result: ----
```

```
[r(3),r(2),not(r(1)),q(1)]

---- The GROUND hypothesis is: ----

q(1) :- w(1)
r(2) :- w(2), s(2), t(2), not(q(2))
r(3) :- w(3), s(3), not(q(3))

not(r(1))
not(q(2))
not(q(3))



----**** The Best hypothesis is: ****----

q(A)
r(A) :- s(A)

Time Cost: 0.019999999999999574 seconds
```

Example bird

It's the Example 8.2. The **input** file *bird.pl*

```
set_max_clause_length(4).
set_max_ground_clauses(10).
set_complete_saturation(0).
set_connected(1).

set_clause_weight(5).
set_literal_weight(1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Classic bird penguin problem
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%:- dynamic bird/1,penguin/1,flies/1,ab/1,ab2/1.

%  background
begin_of_background.

flies(X) :- bird(X), not(ab(X)).
bird(X) :- penguin(X).
bird(tweety).
penguin(polly).
penguin(peter).
has_big_wings(peter).
has_big_wings(X) :- has_big_wings2(X).
ab2(X) :- penguin(X), has_big_wings(X).

end_of_background.

% examples
examples( [
    flies(tweety),
    flies(peter),
    not(flies(polly))
] ).

%  mode declaration
%
%  NOTE: Only '+' '#'' can be included in not(_) currently
head_modes( [has_big_wings2(+any), ab(+any)] ).
```

```
body_modes( [
    bird(+any),    not(bird(+any)),
    penguin(+any), not(penguin(+any)),
    ab2(+any),     not(ab2(+any))
] ).
```

The **output** produced by *icc.pl*

```
---- Abductive Result: ----

[ab(polly),not(flies(polly)),not(ab(peter)),not(ab(tweety))]

---- The GROUND hypothesis is: ----

ab(polly) :- bird(polly), penguin(polly), not(ab2(polly))

not(ab(peter))
not(ab(tweety))
not(has_big_wings2(polly))



----**** The Best hypothesis is: ****----

ab(A) :- penguin(A), not(ab2(A))

Time Cost: 0.010000000000000009 seconds
```

Example Yamamoto

The **input** file *yamamoto.pl*

```
set_max_clause_length(4).
set_max_ground_clauses(10).
set_complete_saturation(1).
set_connected(1).

set_clause_weight(5).
set_literal_weight(1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Yamamoto's Example
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  background
begin_of_background.
even(s(X)) :- odd(X).
even(0).
end_of_background.

% examples
examples( [odd(s(s(s(0)))),
           odd(s(s(s(s(s(0)))))),
   not(even(s(0))),not(even(s(s(s(0)))))
] ).

% mode declaration
%
% NOTE: Only '+''-' can be included in not(_) currently
head_modes( [odd(+any)] ).
body_modes( [even(+any), =(+any,s(-any))] ).
```

The **output** produced by *icc.pl*

```
---- Abductive Result: ----

[not(odd(s(s(0)))),not(even(s(s(s(0))))),not(odd(0)),
not(even(s(0))),odd(s(s(s(s(s(0)))))),odd(s(s(s(0)))))]
```

```
---- The GROUND hypothesis is: ----

odd(s(0)) :- s(0)=s(0), even(0)

odd(s(s(s(0)))) :- s(s(s(0)))=s(s(s(0))), s(s(0))=s(s(0)),
s(0)=s(0), even(0), even(s(s(0)))

odd(s(s(s(s(0))))) :- even(s(s(s(s(0))))), s(s(s(s(0))))=s(s(s(s(0)))),
s(s(s(0)))=s(s(s(0))), even(s(s(0))), s(s(0))=s(s(0)), s(0)=s(0), even(0)

odd(s(s(s(s(s(0)))))) :- s(s(s(s(s(0)))))=s(s(s(s(s(0))))), even(s(s(s(s(0))))),
s(s(s(s(0))))=s(s(s(s(0)))), s(s(s(0)))=s(s(s(0))), s(s(0))=s(s(0)),
s(0)=s(0), even(0), even(s(s(0))), even(s(s(s(s(s(0))))))

not(odd(s(s(0))))
not(odd(0))



----**** The Best hypothesis is: ****----

odd(A) :- A=s(B), even(B)

Time Cost: 0.060000000000002274 seconds
```

Example mother

The **input** file *mother.pl*

```
set_max_clause_length(4).
set_max_ground_clauses(10).
set_complete_saturation(0).
set_connected(0).

set_clause_weight(5).
set_literal_weight(1).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Mother Example
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  background
begin_of_background.
person(s1).
person(s2).
person(s3).
person(s4).
person(s5).
person(m1).
person(m2).
person(m3).
person(m4).
person(m5).
male(s1).
male(s2).
male(s3).
male(s4).
male(s5).
female(m1).
female(m2).
female(m3).
female(m4).
female(m5).
child(s1,m1).
child(m5,m4).
child(s2,m1).
```

```
child(m3,s1).
child(m2,m1).
child(s3,s1).
end_of_background.

% examples
examples( [
mother(m1,s1),     mother(m1,m2),
not(mother(s1,m3)), not(mother(s1, s3)),
not(mother(m2,m4)), not(mother(s2, s3)),
not(mother(m1,m3))
] ).

%  mode declaration
%
%  NOTE, Only '+''#' can be included in not(_) currently
head_modes( [mother(+person,+person)] ).
body_modes( [male(+person),female(+person), child(+person,+person)] ).
```

The **output** produced by *icc.pl*

```
---- Abductive Result: ----

[not(mother(m1,m3)),not(mother(s2,s3)),not(mother(m2,m4)),
not(mother(s1,s3)),not(mother(s1,m3)),mother(m1,m2),mother(m1,s1)]

---- The GROUND hypothesis is: ----

mother(m1,m2) :- female(m1), female(m2), child(m2,m1)
mother(m1,s1) :- male(s1), female(m1), child(s1,m1)

not(mother(m1,m3))
not(mother(s2,s3))
not(mother(m2,m4))
not(mother(s1,s3))
not(mother(s1,m3))



----**** The Best hypothesis is: ****----

mother(A,B) :- female(A), child(B,A)
```

```
Time Cost: 0.0 seconds
```

Example odd & even

The example is slightly different from the original one. The positive examples are modified from $\{even(0), even(4), odd(3), odd(5)\}$ to $\{even(0), even(2), even(4), odd(3), odd(5)\}$ where a new positive example $even(2)$ is added. Without doing this, the system will give a 'better' hypothesis $\{even(0), even(4), odd(X) \leftarrow succ(Y, X), even(Y)\}$ over the target one. Moreover, in order to get the objective hypothesis, connected theories are used.

The **input** file *oddeven.pl*

```
set_max_clause_length(4).
set_max_ground_clauses(10).
set_complete_saturation(0).
set_connected(1).

set_clause_weight(5).
set_literal_weight(1).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% oddeven Example
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  background
begin_of_background.
num(0).
num(s(0)).
num(s(s(0))).
num(s(s(s(0)))).
num(s(s(s(s(0))))).
num(s(s(s(s(s(0)))))).
num(s(s(s(s(s(s(0))))))).
num(s(s(s(s(s(s(s(0)))))))).
succ(X,s(X)) :- num(X),num(s(X)).
end_of_background.

% examples
examples( [
even(0),
even(s(s(0))),
even(s(s(s(s(0))))),
odd(s(s(s(0)))),
odd(s(s(s(s(s(0)))))),
```

```
not(even(s(0))), not(even(s(s(s(0))))),
not(odd(0)),     not(odd(s(s(s(s(0))))))
] ).

%  mode declaration
%
%  NOTE, Only '+''#' can be included in not(_) currently
head_modes( [even('#'(num)), even(+num), odd(+num)] ).
body_modes( [succ(-num,+num), odd(+num), even(+num)] ).
```

The **output** produced by *icc.pl*

```
---- Abductive Result: ----

[not(odd(s(s(s(s(0))))))),not(odd(0)),not(even(s(s(s(0))))),not(even(s(0))),odd(s(s(s(s(s(0)))

---- The GROUND hypothesis is: ----

even(0).

even(s(s(0))) :- succ(s(0),s(s(0))), succ(0,s(0)), odd(s(0)), even(0)

even(s(s(s(s(0))))) :- succ(s(s(s(0))),s(s(s(s(0))))), succ(s(s(0)),s(s(s(0)))),
odd(s(s(s(0)))), succ(s(0),s(s(0))), even(s(s(0))), succ(0,s(0)), odd(s(0)), even(0)

odd(s(0)) :- succ(0,s(0)),even(0)

odd(s(s(s(0)))) :- succ(s(s(0)),s(s(s(0)))), succ(s(0),s(s(0))), even(s(s(0))),
succ(0,s(0)), odd(s(0)), even(0)

odd(s(s(s(s(s(0)))))) :- succ(s(s(s(s(0)))),s(s(s(s(s(0)))))),
succ(s(s(s(0))),s(s(s(s(0))))), even(s(s(s(s(0))))), succ(s(s(0)),s(s(s(0)))),
odd(s(s(s(0)))), succ(s(0),s(s(0))), even(s(s(0))), succ(0,s(0)), odd(s(0)), even(0)

not(odd(s(s(s(s(0))))))
not(odd(0))
not(even(s(s(s(0)))))
not(even(s(0)))


----**** The Best hypothesis is: ****----
```

```
even(A) :- succ(B,A), odd(B)
even(0).
odd(A) :- succ(B,A), even(B)

Time Cost: 4.89000000000000001 seconds
```

If a new mode bode $not(odd(+num))$ is allowed, a better hypothesis is then be learned

```
...

----**** The Best hypothesis is: ****----

even(A) :- not(odd(A))
odd(A) :- succ(B,A), even(B)

Time Cost: 17.58 seconds
```

Example nonealike

The **input** file *nonealike.pl*

```
set_max_clause_length(4).
set_max_ground_clauses(10).
set_complete_saturation(0).
set_connected(0).

set_clause_weight(5).
set_literal_weight(1).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Simple Example
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  background
begin_of_background.
face(1).
face(2).
face(3).
face(4).
face(5).
face(6).
eq(X,X) :-  face(X).
diff(X,Y) :-  face(X),face(Y), X \== Y.
end_of_background.

% examples
examples( [
nonealike(1, 2, 3, 4, 5),
nonealike(1, 3, 4, 5, 6),

not(nonealike(2, 2, 3, 4, 5)), not(nonealike(1, 3, 3, 4, 6)),
not(nonealike(1, 2, 3, 3, 5)), not(nonealike(2, 3, 4, 5, 5))
] ).

%  mode declaration
%
%  NOTE, Only '+''#' can be included in not(_) currently
head_modes( [nonealike(+face,+face,+face,+face,+face)] ).
```

```
body_modes( [eq(+face,+face), diff(+face,+face)] ).
```

The **output** produced by *icc.pl*

```
---- Abductive Result: ----

[not(nonealike(2,3,4,5,5)),not(nonealike(1,2,3,3,5)),not(nonealike(1,3,3,4,6)),
not(nonealike(2,2,3,4,5)),nonealike(1,3,4,5,6),nonealike(1,2,3,4,5)]

---- The GROUND hypothesis is: ----

nonealike(1,2,3,4,5) :- eq(1,1), eq(2,2), eq(3,3), eq(4,4), eq(5,5),
diff(1,2), diff(1,3), diff(1,4), diff(1,5), diff(2,1), diff(2,3),
diff(2,4), diff(2,5), diff(3,1), diff(3,2), diff(3,4), diff(3,5),
diff(4,1), diff(4,2), diff(4,3), diff(4,5), diff(5,1), diff(5,2),
diff(5,3), diff(5,4)
nonealike(1,3,4,5,6) :- eq(1,1), eq(3,3), eq(4,4), eq(5,5), eq(6,6),
diff(1,3), diff(1,4), diff(1,5), diff(1,6), diff(3,1), diff(3,4),
diff(3,5), diff(3,6), diff(4,1), diff(4,3), diff(4,5), diff(4,6),
diff(5,1), diff(5,3), diff(5,4), diff(5,6), diff(6,1), diff(6,3),
diff(6,4), diff(6,5)

not(nonealike(2,3,4,5,5))
not(nonealike(1,2,3,3,5))
not(nonealike(1,3,3,4,6))
not(nonealike(2,2,3,4,5))



----**** The Best hypothesis is: ****----

nonealike(A,B,C,D,E) :- diff(A,B), diff(B,C), diff(C,D), diff(D,E)

Time Cost: 3.84 seconds
```

Example highroll

The **input** file *highroll.pl*

```
set_max_clause_length(3).
set_max_ground_clauses(10).
set_complete_saturation(0).
set_connected(0).

set_clause_weight(5).
set_literal_weight(1).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% highroll Example
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  background
begin_of_background.
face(1).
face(2).
face(3).
face(4).
face(5).
face(6).
sum(2).
sum(3).
sum(4).
sum(5).
sum(6).
sum(7).
sum(8).
sum(9).
sum(10).
sum(11).
sum(12).
add(X,Y,Z) :- face(X), face(Y), Z is X + Y, sum(Z).
greaterThan(X,Y) :- sum(X), sum(Y), X > Y.
end_of_background.

% examples
examples( [
```

```
    high(3, 5), high(6, 3), high(6, 6),

    not(high(1, 1)), not(high(2, 3)), not(high(4, 1)),
    not(high(3, 3)), not(high(5, 2))
] ).

%  mode declaration
%
%  NOTE, Only '+' '#' can be included in not(_) currently
head_modes( [high(+face,+face)] ).
body_modes( [
    add(+face,+face,-sum),
    greaterThan(+sum,'#'(sum))
] ).
```

The **output** produced by *icc.pl*

```
---- Abductive Result: ----

[not(high(5,2)),not(high(3,3)),not(high(4,1)),not(high(2,3)),not(high(1,1)),
high(6,6),high(6,3),high(3,5)]

---- The GROUND hypothesis is: ----

high(3,5) :- add(3,3,6), add(3,5,8), add(5,3,8), add(5,5,10),
greaterThan(3,2), greaterThan(5,4), greaterThan(5,3), greaterThan(5,2),
add(3,6,9), add(5,6,11), add(6,3,9), add(6,5,11), add(6,6,12),
greaterThan(6,5), greaterThan(6,4), greaterThan(6,3), greaterThan(6,2),
greaterThan(8,7), greaterThan(8,6), greaterThan(8,5), greaterThan(8,4),
greaterThan(8,3), greaterThan(8,2), greaterThan(10,9),
greaterThan(10,8), greaterThan(10,7), greaterThan(10,6),
greaterThan(10,5), greaterThan(10,4), greaterThan(10,3),
greaterThan(10,2), greaterThan(9,8), greaterThan(9,7), greaterThan(9,6),
greaterThan(9,5), greaterThan(9,4), greaterThan(9,3), greaterThan(9,2),
greaterThan(11,10), greaterThan(11,9), greaterThan(11,8),
greaterThan(11,7), greaterThan(11,6), greaterThan(11,5),
greaterThan(11,4), greaterThan(11,3), greaterThan(11,2),
greaterThan(12,11), greaterThan(12,10), greaterThan(12,9),
greaterThan(12,8), greaterThan(12,7), greaterThan(12,6),
greaterThan(12,5), greaterThan(12,4), greaterThan(12,3),
greaterThan(12,2)
```

```
high(6,3) :- add(3,3,6), add(3,6,9), add(6,3,9), add(6,6,12),
greaterThan(3,2), greaterThan(6,5), greaterThan(6,4), greaterThan(6,3),
greaterThan(6,2), greaterThan(9,8), greaterThan(9,7), greaterThan(9,6),
greaterThan(9,5), greaterThan(9,4), greaterThan(9,3), greaterThan(9,2),
greaterThan(12,11), greaterThan(12,10), greaterThan(12,9),
greaterThan(12,8), greaterThan(12,7), greaterThan(12,6),
greaterThan(12,5), greaterThan(12,4), greaterThan(12,3),
greaterThan(12,2)

high(6,6) :- add(6,6,12), greaterThan(6,5), greaterThan(6,4),
greaterThan(6,3), greaterThan(6,2), greaterThan(12,11),
greaterThan(12,10), greaterThan(12,9), greaterThan(12,8),
greaterThan(12,7), greaterThan(12,6), greaterThan(12,5),
greaterThan(12,4), greaterThan(12,3), greaterThan(12,2)

not(high(5,2))
not(high(3,3))
not(high(4,1))
not(high(2,3))
not(high(1,1))



----**** The Best hypothesis is: ****----

high(A,B) :- add(A,B,C), greaterThan(C,7)

Time Cost: 2.21 seconds
```

Example trains

The **input** file *soldier.pl*

```
set_max_clause_length(4).
set_max_ground_clauses(10).
set_complete_saturation(0).
set_connected(0).

set_clause_weight(5).
set_literal_weight(1).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Trains Example
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%  background
begin_of_background.
% eastbound train 1
short(car_12). % 0
closed(car_12). % 1
long(car_11). % 2
long(car_13).
short(car_14).
open_car(car_11). % 3
open_car(car_13).
open_car(car_14).
shape(car_11,rectangle). % 4,5
shape(car_12,rectangle).
shape(car_13,rectangle).
shape(car_14,rectangle).
load(car_11,rectangle,3). % 6,7,8
load(car_12,triangle,1).
load(car_13,hexagon,1).
load(car_14,circle,1).
wheels(car_11,2).   % 9,10
wheels(car_12,2).
wheels(car_13,3).
wheels(car_14,2).
has_car(east1,car_11). % 11,12
has_car(east1,car_12).
has_car(east1,car_13).
```

```
has_car(east1,car_14).

% eastbound train 2
has_car(east2,car_21).
has_car(east2,car_22).
has_car(east2,car_23).
short(car_21).
short(car_22).
short(car_23).
shape(car_21,u_shaped).
shape(car_22,u_shaped).
shape(car_23,rectangle).
open_car(car_21).
open_car(car_22).
closed(car_23).
load(car_21,triangle,1).
load(car_22,rectangle,1).
load(car_23,circle,2).
wheels(car_21,2).
wheels(car_22,2).
wheels(car_23,2).

% eastbound train 3
has_car(east3,car_31).
has_car(east3,car_32).
has_car(east3,car_33).
short(car_31).
short(car_32).
long(car_33).
shape(car_31,rectangle).
shape(car_32,hexagon).
shape(car_33,rectangle).
open_car(car_31).
closed(car_32).
closed(car_33).
load(car_31,circle,1).
load(car_32,triangle,1).
load(car_33,triangle,1).
wheels(car_31,2).
wheels(car_32,2).
wheels(car_33,3).
```

```
% eastbound train 4
has_car(east4,car_41).
has_car(east4,car_42).
has_car(east4,car_43).
has_car(east4,car_44).
short(car_41).
short(car_42).
short(car_43).
short(car_44).
shape(car_41,u_shaped).
shape(car_42,rectangle).
shape(car_43,elipse).
shape(car_44,rectangle).
double(car_42).
open_car(car_41).
open_car(car_42).
closed(car_43).
open_car(car_44).
load(car_41,triangle,1).
load(car_42,triangle,1).
load(car_43,rectangle,1).
load(car_44,rectangle,1).
wheels(car_41,2).
wheels(car_42,2).
wheels(car_43,2).
wheels(car_44,2).

% eastbound train 5
has_car(east5,car_51).
has_car(east5,car_52).
has_car(east5,car_53).
short(car_51).
short(car_52).
short(car_53).
shape(car_51,rectangle).
shape(car_52,rectangle).
shape(car_53,rectangle).
double(car_51).
open_car(car_51).
closed(car_52).
closed(car_53).
load(car_51,triangle,1).
```

```
load(car_52,rectangle,1).
load(car_53,circle,1).
wheels(car_51,2).
wheels(car_52,3).
wheels(car_53,2).

% westbound train 6
has_car(west6,car_61).
has_car(west6,car_62).
long(car_61).
short(car_62).
shape(car_61,rectangle).
shape(car_62,rectangle).
closed(car_61).
open_car(car_62).
load(car_61,circle,3).
load(car_62,triangle,1).
wheels(car_61,2).
wheels(car_62,2).

% westbound train 7
has_car(west7,car_71).
has_car(west7,car_72).
has_car(west7,car_73).
short(car_71).
short(car_72).
long(car_73).
shape(car_71,rectangle).
shape(car_72,u_shaped).
shape(car_73,rectangle).
double(car_71).
open_car(car_71).
open_car(car_72).
jagged(car_73).
load(car_71,circle,1).
load(car_72,triangle,1).
load(car_73,nil,0).
wheels(car_71,2).
wheels(car_72,2).
wheels(car_73,2).

% westbound train 8
```

```
has_car(west8,car_81).
has_car(west8,car_82).
long(car_81).
short(car_82).
shape(car_81,rectangle).
shape(car_82,u_shaped).
closed(car_81).
open_car(car_82).
load(car_81,rectangle,1).
load(car_82,circle,1).
wheels(car_81,3).
wheels(car_82,2).

% westbound train 9
has_car(west9,car_91).
has_car(west9,car_92).
has_car(west9,car_93).
has_car(west9,car_94).
short(car_91).
long(car_92).
short(car_93).
short(car_94).
shape(car_91,u_shaped).
shape(car_92,rectangle).
shape(car_93,rectangle).
shape(car_94,u_shaped).
open_car(car_91).
jagged(car_92).
open_car(car_93).
open_car(car_94).
load(car_91,circle,1).
load(car_92,rectangle,1).
load(car_93,rectangle,1).
load(car_93,circle,1).
wheels(car_91,2).
wheels(car_92,2).
wheels(car_93,2).
wheels(car_94,2).

% westbound train 10
has_car(west10,car_101).
has_car(west10,car_102).
```

```
short(car_101).
long(car_102).
shape(car_101,u_shaped).
shape(car_102,rectangle).
open_car(car_101).
open_car(car_102).
load(car_101,rectangle,1).
load(car_102,rectangle,2).
wheels(car_101,2).
wheels(car_102,2).
end_of_background.

% type definitions
int(0). int(1). int(2). int(3).

car(car_11).  car(car_12).  car(car_13).  car(car_14).
car(car_21).  car(car_22).  car(car_23).
car(car_31).  car(car_32).  car(car_33).
car(car_41).  car(car_42).  car(car_43).  car(car_44).
car(car_51).  car(car_52).  car(car_53).
car(car_61).  car(car_62).
car(car_71).  car(car_72).  car(car_73).
car(car_81).  car(car_82).
car(car_91).  car(car_92).  car(car_93).  car(car_94).
car(car_101).  car(car_102).

shape(elipse).  shape(hexagon).  shape(rectangle).  shape(u_shaped).
shape(triangle). shape(circle). shape(nil).

train(east1).  train(east2).  train(east3).  train(east4).  train(east5).
train(west6).  train(west7).  train(west8).  train(west9).  train(west10).

% examples
examples( [
eastbound(east1),
eastbound(east2),
eastbound(east3),
eastbound(east4),
eastbound(east5),

not(eastbound(west6)),
not(eastbound(west7)),
```

```
not(eastbound(west8)),
not(eastbound(west9)),
not(eastbound(west10))
] ).

% mode declaration
%
% NOTE: Only '+' '#' can be included in not(_) currently
head_modes( [eastbound(+train)] ).

body_modes( [
short(+car),
closed(+car),
long(+car),
open_car(+car),
double(+car),
jagged(+car),
shape(+car,'#'(shape)),
load(+car,'#'(shape),'#'(int)),
wheels(+car,'#'(int)),
has_car(+train,-car)
] ).
```

Since *icc.pl* has limited scalability, the trains example is only tested with three or two positive examples. The **outputs** produced by *icc.pl*

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Test with e1, e2, e3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

---- Abductive Result: ----

[not(eastbound(west10)),not(eastbound(west9)),not(eastbound(west8)),
not(eastbound(west7)),not(eastbound(west6)),eastbound(east3),eastbound(east2),
eastbound(east1)]

---- The GROUND hypothesis is: ----

eastbound(east1) :- has_car(east1,car_14), has_car(east1,car_13),
has_car(east1,car_12), has_car(east1,car_11), short(car_12), short(car_14),
closed(car_12), long(car_11), long(car_13), open_car(car_11), open_car(car_13),
open_car(car_14), shape(car_11,rectangle), shape(car_12,rectangle),
```

```
shape(car_13,rectangle), shape(car_14,rectangle), load(car_11,rectangle,3),
load(car_12,triangle,1), load(car_13,hexagon,1), load(car_14,circle,1),
wheels(car_11,2), wheels(car_12,2), wheels(car_13,3), wheels(car_14,2)

eastbound(east2) :- has_car(east2,car_23), has_car(east2,car_22),
has_car(east2,car_21), short(car_21), short(car_22), short(car_23),
closed(car_23), open_car(car_21), open_car(car_22), shape(car_21,u_shaped),
shape(car_22,u_shaped), shape(car_23,rectangle), load(car_21,triangle,1),
load(car_22,rectangle,1), load(car_23,circle,2), wheels(car_21,2),
wheels(car_22,2), wheels(car_23,2)

eastbound(east3) :- has_car(east3,car_33), has_car(east3,car_32),
has_car(east3,car_31), short(car_31), short(car_32), closed(car_32),
closed(car_33), long(car_33), open_car(car_31), shape(car_31,rectangle),
shape(car_32,hexagon), shape(car_33,rectangle), load(car_31,circle,1),
load(car_32,triangle,1), load(car_33,triangle,1), wheels(car_31,2),
wheels(car_32,2), wheels(car_33,3)

not(eastbound(west10))
not(eastbound(west9))
not(eastbound(west8))
not(eastbound(west7))
not(eastbound(west6))



----**** The Best hypothesis is: ****----

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

Time Cost: 83.07 seconds

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Test with e2, e3, e4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

---- Abductive Result: ----

[not(eastbound(west10)),not(eastbound(west9)),not(eastbound(west8)),
not(eastbound(west7)),not(eastbound(west6)),eastbound(east4),eastbound(east3),
eastbound(east2)]
```

```
---- The GROUND hypothesis is: ----

eastbound(east2) :- has_car(east2,car_23), has_car(east2,car_22),
has_car(east2,car_21), short(car_21), short(car_22), short(car_23),
closed(car_23), open_car(car_21), open_car(car_22), shape(car_21,u_shaped),
shape(car_22,u_shaped), shape(car_23,rectangle), load(car_21,triangle,1),
load(car_22,rectangle,1), load(car_23,circle,2), wheels(car_21,2),
wheels(car_22,2), wheels(car_23,2)

eastbound(east3) :- has_car(east3,car_33), has_car(east3,car_32),
has_car(east3,car_31), short(car_31), short(car_32), closed(car_32),
closed(car_33), long(car_33), open_car(car_31), shape(car_31,rectangle),
shape(car_32,hexagon), shape(car_33,rectangle), load(car_31,circle,1),
load(car_32,triangle,1), load(car_33,triangle,1), wheels(car_31,2),
wheels(car_32,2), wheels(car_33,3)

eastbound(east4) :- has_car(east4,car_44), has_car(east4,car_43),
has_car(east4,car_42), has_car(east4,car_41), short(car_41), short(car_42),
short(car_43), short(car_44), closed(car_43), open_car(car_41),
open_car(car_42), open_car(car_44), double(car_42), shape(car_41,u_shaped),
shape(car_42,rectangle), shape(car_43,elipse), shape(car_44,rectangle),
load(car_41,triangle,1), load(car_42,triangle,1), load(car_43,rectangle,1),
load(car_44,rectangle,1), wheels(car_41,2), wheels(car_42,2), wheels(car_43,2),
wheels(car_44,2)

not(eastbound(west10))
not(eastbound(west9))
not(eastbound(west8))
not(eastbound(west7))
not(eastbound(west6))



----**** The Best hypothesis is: ****----

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

Time Cost: 48.9 seconds


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Test with e3, e4, e5
```

%%

---- Abductive Result: ----

[not(eastbound(west10)),not(eastbound(west9)),not(eastbound(west8)),
not(eastbound(west7)),not(eastbound(west6)),eastbound(east5),eastbound(east4),
eastbound(east3)]

---- The GROUND hypothesis is: ----

eastbound(east3) :- has_car(east3,car_33), has_car(east3,car_32),
has_car(east3,car_31), short(car_31), short(car_32), closed(car_32),
closed(car_33), long(car_33), open_car(car_31), shape(car_31,rectangle),
shape(car_32,hexagon), shape(car_33,rectangle), load(car_31,circle,1),
load(car_32,triangle,1), load(car_33,triangle,1), wheels(car_31,2),
wheels(car_32,2), wheels(car_33,3)

eastbound(east4) :- has_car(east4,car_44), has_car(east4,car_43),
has_car(east4,car_42), has_car(east4,car_41), short(car_41), short(car_42),
short(car_43), short(car_44), closed(car_43), open_car(car_41),
open_car(car_42), open_car(car_44), double(car_42), shape(car_41,u_shaped),
shape(car_42,rectangle), shape(car_43,elipse), shape(car_44,rectangle),
load(car_41,triangle,1), load(car_42,triangle,1), load(car_43,rectangle,1),
load(car_44,rectangle,1), wheels(car_41,2), wheels(car_42,2), wheels(car_43,2),
wheels(car_44,2)

eastbound(east5) :- has_car(east5,car_53), has_car(east5,car_52),
has_car(east5,car_51), short(car_51), short(car_52), short(car_53),
closed(car_52), closed(car_53), open_car(car_51), double(car_51),
shape(car_51,rectangle), shape(car_52,rectangle), shape(car_53,rectangle),
load(car_51,triangle,1), load(car_52,rectangle,1), load(car_53,circle,1),
wheels(car_51,2), wheels(car_52,3), wheels(car_53,2)

not(eastbound(west10))
not(eastbound(west9))
not(eastbound(west8))
not(eastbound(west7))
not(eastbound(west6))

----**** The Best hypothesis is: ****----

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

Time Cost: 42.94 seconds

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Test with e1, e2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

---- Abductive Result: ----

[not(eastbound(west10)),not(eastbound(west9)),not(eastbound(west8)),
not(eastbound(west7)),not(eastbound(west6)),eastbound(east2),eastbound(east1)]

---- The GROUND hypothesis is: ----

eastbound(east1) :- has_car(east1,car_14), has_car(east1,car_13),
has_car(east1,car_12), has_car(east1,car_11), short(car_12), short(car_14),
closed(car_12), long(car_11), long(car_13), open_car(car_11), open_car(car_13),
open_car(car_14), shape(car_11,rectangle), shape(car_12,rectangle),
shape(car_13,rectangle), shape(car_14,rectangle), load(car_11,rectangle,3),
load(car_12,triangle,1), load(car_13,hexagon,1), load(car_14,circle,1),
wheels(car_11,2), wheels(car_12,2), wheels(car_13,3), wheels(car_14,2)

eastbound(east2) :- has_car(east2,car_23), has_car(east2,car_22),
has_car(east2,car_21), short(car_21), short(car_22), short(car_23),
closed(car_23), open_car(car_21), open_car(car_22), shape(car_21,u_shaped),
shape(car_22,u_shaped), shape(car_23,rectangle), load(car_21,triangle,1),
load(car_22,rectangle,1), load(car_23,circle,2), wheels(car_21,2),
wheels(car_22,2), wheels(car_23,2)

not(eastbound(west10))
not(eastbound(west9))
not(eastbound(west8))
not(eastbound(west7))
not(eastbound(west6))



----**** The Best hypothesis is: ****----

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

```
Time Cost: 1.1400000000000006 seconds


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Test with e2, e3
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

---- Abductive Result: ----

[not(eastbound(west10)),not(eastbound(west9)),not(eastbound(west8)),
not(eastbound(west7)),not(eastbound(west6)),eastbound(east3),eastbound(east2)]

---- The GROUND hypothesis is: ----

eastbound(east2) :- has_car(east2,car_23), has_car(east2,car_22),
has_car(east2,car_21), short(car_21), short(car_22), short(car_23),
closed(car_23), open_car(car_21), open_car(car_22), shape(car_21,u_shaped),
shape(car_22,u_shaped), shape(car_23,rectangle), load(car_21,triangle,1),
load(car_22,rectangle,1), load(car_23,circle,2), wheels(car_21,2),
wheels(car_22,2), wheels(car_23,2)

eastbound(east3) :- has_car(east3,car_33), has_car(east3,car_32),
has_car(east3,car_31), short(car_31), short(car_32), closed(car_32),
closed(car_33), long(car_33), open_car(car_31), shape(car_31,rectangle),
shape(car_32,hexagon), shape(car_33,rectangle), load(car_31,circle,1),
load(car_32,triangle,1), load(car_33,triangle,1), wheels(car_31,2),
wheels(car_32,2), wheels(car_33,3)

not(eastbound(west10))
not(eastbound(west9))
not(eastbound(west8))
not(eastbound(west7))
not(eastbound(west6))



----**** The Best hypothesis is: ****----

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

Time Cost: 0.45000000000000284 seconds
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Test with e3, e4
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

---- Abductive Result: ----

[not(eastbound(west10)),not(eastbound(west9)),not(eastbound(west8)),
not(eastbound(west7)),not(eastbound(west6)),eastbound(east4),eastbound(east3)]

---- The GROUND hypothesis is: ----

eastbound(east3) :- has_car(east3,car_33), has_car(east3,car_32),
has_car(east3,car_31), short(car_31), short(car_32), closed(car_32),
closed(car_33), long(car_33), open_car(car_31), shape(car_31,rectangle),
shape(car_32,hexagon), shape(car_33,rectangle), load(car_31,circle,1),
load(car_32,triangle,1), load(car_33,triangle,1), wheels(car_31,2),
wheels(car_32,2), wheels(car_33,3)

eastbound(east4) :- has_car(east4,car_44), has_car(east4,car_43),
has_car(east4,car_42), has_car(east4,car_41), short(car_41), short(car_42),
short(car_43), short(car_44), closed(car_43), open_car(car_41),
open_car(car_42), open_car(car_44), double(car_42), shape(car_41,u_shaped),
shape(car_42,rectangle), shape(car_43,elipse), shape(car_44,rectangle),
load(car_41,triangle,1), load(car_42,triangle,1), load(car_43,rectangle,1),
load(car_44,rectangle,1), wheels(car_41,2), wheels(car_42,2), wheels(car_43,2),
wheels(car_44,2)

not(eastbound(west10))
not(eastbound(west9))
not(eastbound(west8))
not(eastbound(west7))
not(eastbound(west6))



----**** The Best hypothesis is: ****----

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

Time Cost: 1.2400000000000009 seconds
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Test with e4, e5
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

---- Abductive Result: ----

[not(eastbound(west10)),not(eastbound(west9)),not(eastbound(west8)),
not(eastbound(west7)),not(eastbound(west6)),eastbound(east5),eastbound(east4)]

---- The GROUND hypothesis is: ----

eastbound(east4) :- has_car(east4,car_44), has_car(east4,car_43),
has_car(east4,car_42), has_car(east4,car_41), short(car_41), short(car_42),
short(car_43), short(car_44), closed(car_43), open_car(car_41),
open_car(car_42), open_car(car_44), double(car_42), shape(car_41,u_shaped),
shape(car_42,rectangle), shape(car_43,elipse), shape(car_44,rectangle),
load(car_41,triangle,1), load(car_42,triangle,1), load(car_43,rectangle,1),
load(car_44,rectangle,1), wheels(car_41,2), wheels(car_42,2), wheels(car_43,2),
wheels(car_44,2)

eastbound(east5) :- has_car(east5,car_53), has_car(east5,car_52),
has_car(east5,car_51), short(car_51), short(car_52), short(car_53),
closed(car_52), closed(car_53), open_car(car_51), double(car_51),
shape(car_51,rectangle), shape(car_52,rectangle), shape(car_53,rectangle),
load(car_51,triangle,1), load(car_52,rectangle,1), load(car_53,circle,1),
wheels(car_51,2), wheels(car_52,3), wheels(car_53,2)

not(eastbound(west10))
not(eastbound(west9))
not(eastbound(west8))
not(eastbound(west7))
not(eastbound(west6))



----**** The Best hypothesis is: ****----

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

Time Cost: 1.1299999999999955 seconds
```