PLOG: ITS ALGORITHMS AND APPLICATIONS

by

WEIJUN ZHU, B.S., M.S.

A DISSERTATION

In

COMPUTER SCIENCE

Submitted to the Graduate Faculty

of Texas Tech University in

Partial Fulfillment of

the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Approved

Dr. Michael Gelfond

Committee Chairman

Dr. Rushton Nelson

Dr. Yuanlin Zhang

Peggy Gordon Miller

Dean of the Graduate School

May, 2012

ACKNOWLEDGMENTS

I am thankful to my adviser Michael Gelfond for his help and support during all these years when I studied at Texas Tech University. His strong mathematical background and deep knowledge of knowledge representation helped me to avoid many obstacles and guide me to a path to success. His dedication on his research work inspired me to become what I am today.

I would like to thank Dr. Rushton Nelson for his help on my writing. Being a student with English as second language, he helped me on how to express my ideas precisely and how to write proofs correctly and make them easy to read. During the meetings and seminars, his questions always broad my views and thoughts. I also would like to thank Dr. Yuanlin Zhang for being on my committee. I enjoyed my discussions with him and benefited from it a lot.

I am thankful to all members in the KR lab. I have enjoyed all the seminars presented by them which helped me to understand a wide variety of topics. Staff and faculties in my computer science department did wonderful job to help graduate students to pursue their degrees. I surely learned a lot from the classes I took and got a lot of help from staffs for all kinds of issues.

I shall give my sincerely thanks to my wife for her constant support during all these years. Without her, I probably will not be able to attend and finish my PH.D. program. I am very glad that I met this wonderful woman and she became my wife.

I only can say that I am so lucky to have parents who are willing to come to a foreign county where they don't understand the language at all just to help my wife and me to take care of our child so we can devote all of our time on work and study.

I would like to thank all my friends who have been there for me. They made my stay in Lubbock very memorable. I surely will miss you, the Texas Tech University and the city Lubbock.

Table of Contents

ABSTRACT

This dissertation is a contribution towards combining logical and probabilistic reasoning. The work is based on the language P-log which combines a recently developed non-monotonic logic language, Answer Set Prolog, with the philosophy of causal Bayesian networks. The goal of this dissertation was to design and implement an inference engine for P-log and to develop a methodology of its use. As the result of this research work, we had built two P-log inference engines. Through the experiments on various examples, we have shown the advantages of using plog2.0, a system based on partial grounding algorithm and a concept of partial possible worlds. We introduced a new action description language $\mathcal{NB}$ which allows specifying non-deterministic actions as well as probabilities associated with these actions. We developed an encoding which maps systems written in $\mathcal{NB}$ to P-log programs. We presented systematic methods of representing probabilistic diagnosis and planning problems and algorithms of finding the solutions with P-log systems. Finally, we investigated the performance on these two applications and compared with other similar systems.

List of Tables

List of Figures

Chapter 1

INTRODUCTION

This dissertation is a contribution towards combining logical and probabilistic reasoning. In logical reasoning, problems are represented by some formal logical languages and queries to the problems are solved by using inference engines. In probabilistic reasoning, problems are represented by random variables and probability distributions over those variables. Many methods use networks for the purpose of understanding the dependency relationships among variables. Inference engines are developed based on their graphical representation of these networks. For the last several decades, there have been significant progress on both types of reasoning. While logical reasoning and probabilistic reasoning are very different at their representation methods and algorithms behind their inference engines, recent research shows that in order to build an efficient intelligent agent, it would be beneficial to combine both types of reasoning to achieve a better system.

The language, P-log [1], is one of such attempts. P-log combines a recently developed non-monotonic logic language, Answer Set Prolog, with the philosophy of causal Bayesian networks to achieve strong expressive power as well as neat representation for problems where logical rules and probabilistic distributions are both involved.

1.1 Answer Set Programming

It is well known that human beings can make adequate decisions through reasoning when facing incomplete information of the situation. Furthermore, when new pieces of information come in, one may adjust his/her previous conclusions based on these new information, even though in some cases these new conclusions are contradictory

to what he/she has derived before. Such non-monotonic reasoning mechanism is very common in our daily life. However, many classical logic languages are not adequate for modeling such situations as they are known as *monotonic*. Since late 70's, researchers are looking for theories of non-monotonic logic languages. On the path of finding these languages, Answer Set Programming (ASP) was introduced by Michael Gelfond and Vladimir Lifschitz in 1988 [2]. With *negation as failure* and answer set semantics, ASP is able to represent knowledge that involves defaults, exceptions, intention and various forms of incomplete information.

There are several inference engines developed for ASP. The list includes, but not limited to, *Smodels* [3], *DLV* [4], *Cmodels* [5], *ASSAT* [6] and *Clasp* [7]. With these inference engines available, ASP has been used in many applications such as building supporting systems for space shuttle control [8] including finding plans and reasoning about faulty components; understanding natural languages and answering questions [9]; and checking medical invoices for health insurance companies [10].

1.2 Bayesian Network

Bayesian networks, also commonly known as belief networks, probabilistic networks and knowledge maps, are natural ways to represent conditional independence relations among events. A Bayesian network consists of a directed acyclic graph (DAG) and a set of conditional probability tables for each node in the graph. Judea Pearl [11] emphasizes the importance of causality when modeling probabilistic reasoning problems with Bayesian networks. In [11], the author has shown the advantages of building a Bayesian network around causality rather than associational information.

Many inference algorithms for Bayesian networks have been proposed since 80's. All algorithms can be divided into two groups: exact inference algorithms and approximate algorithms. Exact inference algorithms includes: message propagation algorithm for singly connected network [12], joint-tree algorithm [13], recursive condi-

tioning algorithm[14], variable elimination algorithm[15] and algorithms that compile Bayesian networks to arithmetic circuit [16], to weighted model counting problems [17, 18]. Approximate algorithms such as logic sampling method [19] are also studied for dealing with large Bayesian networks.

1.3 Goals of this Research

The goals of this research are to develop efficient inference engines for language P-log and to investigate how to use P-log to solve different reasoning tasks. Our goals include:

- **Developing algorithms for P-log inference engine:** We develop two algorithms for P-log inference engine. The first algorithm encodes a P-log program to an ASP program Π then uses *Smodels* to find all the answer sets of Π. By decoding probability information and looking up atoms in the answer sets, the algorithm is able to compute the probability of formulas with respect to the P-log program. The second algorithm only builds a ground P-log program that is related to the query. Rules which are not relevant to the current query are dropped from further computation. Furthermore, instead of using other answer set solver, there is an algorithm designed for evaluating possible worlds and the probability of formulas at same time. By taking advantages of properties of input P-log program, it enhances the performance of computing probabilities of formulas.

- **Developing methodology for representing probabilistic diagnosis and planning problems:** We extend an action description language to allow representing actions with uncertain effects and incomplete initial situations. We develop systematic methods to model a probabilistic diagnosis problems and probabilistic planning problems.

- **Investigate the impact of different programs on the performance of P-log engines:** We conduct several tests with P-log systems to understand how different types of P-log program affect the performance of the system. We study the advantage and disadvantage of each systems and list out other options which may help improving the performance of the system.

This dissertation is organized as follows: Chapter 2 will review the syntax and semantics of P-log language. It will also discuss a special type of P-log program, called strongly causally ordered unitary (scou) P-log program. We will show how common probabilistic reasoning problems can be modeled as *scou* P-log programs. In Chapter 3, We will give two algorithms for implementing P-log inference engines. The first one, based on translation from P-log program to ASP, follows the idea presented in [1]. The second one, designed for improving performance of P-log system, will be presented after that. In Chapter 4, we focus on some details of implementation of the system. We give experimental results of running both systems that we have developed. We will discuss the advantage and disadvantage of both systems with respect to different types of P-log program. Then, Chapter 5 talks about extending an action description language to describing non-deterministic results of actions as well as incomplete initial situations. We specifically pay attention to two applications: probabilistic diagnosis and probabilistic planning. We briefly discuss the performance of P-log on these two applications. Chapter 7 presents all the proofs of the theorems in previous chapters. Chapter 8 briefly describes related work on combining logical and probabilistic reasoning as well as probabilistic diagnosis and planning. Finally, Chapter 9 gives our conclusions and future work.

Chapter 2

THE LANGUAGE P-LOG

In this chapter, we review the syntax and semantics of language P-log which was first introduced in [1]. We extend the work of [1] by defining the syntax and semantics of queries of P-log program. The conditions for consistent P-log program are discussed in details in [1] and we narrow down our attention to a special type of P-log program, *scou* P-log program, and to illustrate, by showing some examples, how problems can be modeled as *scou* P-log programs.

2.1 Syntax of P-log

A probabilistic logic program (P-log program) Π consists of (i) a sorted signature, (ii) a declaration, (iii) a regular part, (iv) a set of random selection rules, (v) probabilistic information part and (vi) a set of observations and actions.

1. **Sorted Signature**: The sorted signature Σ of Π contains a set O of objects and a set F of function symbols. The set F is a disjoint union of two sets: F_r, a set of *term building functions* and F_a, a set of attributes used in defining atomic statements. Terms are formed using F_r and O. Expression of the form $a(\bar{t})$ will be referred to as attribute terms where a is an attribute, $\bar{t}$ is a vector of terms of the sorts required by a.

   Technically, every P-log atom is of the form $a(\bar{t}) = y$, where $a(\bar{t})$ is an attribute term and y is a term. A literal is an atomic statement $a(\bar{t}) = y$ or its negation, $a(\bar{t}) \neq y$. The set of terms that an attribute a can take as values is denoted by $range(a)$. If $range(a) = \{true, false\}$, then we say that a is a Boolean attribute. For Boolean attribute, we may simply use $a(\bar{t})$ as a short hand of

$a(\bar{t}) = true$ and $\neg a(\bar{t})$ as short hand of $a(\bar{t}) = false$. An extended literal l is a literal or a **not** l, where **not** is the default negation of Answer Set Prolog.

A P-log statement containing variables is considered as a shorthand for the set of its ground instances, where a ground instance is obtained by replacing unbound occurrences of variables with properly sorted ground terms.

2. **Declaration**: The declaration of a P-log program is a collection of definitions of sorts, attributes and variables.

   - A sort c can be defined by explicitly listing its elements,

   $$c = \{x_1, \ldots, x_n\} \tag{2.1}$$

   If c is a set of continuous integer numbers from m to n. It can be defined as,

   $$c = \{m..n\} \tag{2.2}$$

   - Attributes can be declared as by a statement of the form:

   $$a : c_1 \times \cdots \times c_n \rightarrow c_0 \tag{2.3}$$

   - Variables $V_1, \ldots, V_n$ that can be replaced by elements of a sort c is declared as:

   $$\#domain\ c(V_1; \ldots; V_n) \tag{2.4}$$

3. **Regular Part**: The regular part of a P-log program consists of a collection of rules r of Answer Set Prolog (without disjunction) formed using literals of Σ,

   $$l_0 \leftarrow l_1, \ldots, l_m, \textbf{not}\ l_{m+1}, \ldots, \textbf{not}\ l_n \tag{2.5}$$

4. **Random Selection Rules**: A random selection rule is a rule of the form

   $$[r]\ random(a(\bar{t}) : \{X : p(X)\}) \leftarrow B \tag{2.6}$$

where r is a term used to name the rule and B is a collection of extended literals of Σ. Statement 2.6 says that if B holds, the value of $a(\bar{t})$ is selected at random from $\{X : p(X)\} \cap range(a)$, unless this value is fixed by a deliberate action.

If B is empty, we have

$$[r]\ random(a(\bar{t}) : \{X : p(X)\}) \tag{2.7}$$

If the value of $a(\bar{t})$ can be any value of range of $a(\bar{t})$, we simply write

$$[r]\ random(a(\bar{t})) \leftarrow B \tag{2.8}$$

5. **Probabilistic Information**: Information about probabilities of random attributes taking a particular value is specified by the *pr-atoms*:

$$pr_r(a(\bar{t}) = y|_c B) = v \tag{2.9}$$

where r is the term used to name the *pr-atoms*, B is a collections of extended literals, pr is a special symbol not belonging to Σ and v is a term built from arithmetic functions whose resulting value is a rational number between 0 and 1.

Sometime, we consider statement 2.9 as a rule where B is the body of the rule and $a(\bar{t}) = y$ is the head of the rule. We may refer to such rules as pr rules.

If B is empty, we simply write

$$pr_r(a(\bar{t}) = y) = v \tag{2.10}$$

6. **Observations and Actions**: Observations and actions are statements of the respective forms: $obs(l)$ and $do(a(\bar{t}) = y)$, where l is a literal of Σ and $a(\bar{t}) = y$ is an atom of Σ.

2.2 Semantics of P-log

The semantics of a ground P-log program Π is defined in two aspects: a collection of possible sets of beliefs of a rational agent associated with Π and their probabilities. We refer to these sets as possible worlds. The set of possible worlds is defined through a counterpart of Π, an Answer Set Prolog program $\tau(\Pi)$. The probability of possible world is defined through random selection rules and probabilistic information in Π.

Instead of referring to original Answer Set Prolog syntax, we give our translation based on the syntax that is accepted by a well known answer set solver, *Smodels*. *Smodels* has introduced several new statements which also have been widely used in other answer set solvers. According to [1], the Answer Set Prolog program $\tau(\Pi)$ is defined as follows:

1. Sort declarations: For every sort declaration 2.1 of Π, $\tau(\Pi)$ contains $c(x_1), \ldots, c(x_n)$; for statement 2.2, $\tau(\Pi)$ contains $c = \{m..n\}$.

2. Regular part: for each rule r in the regular part of Π, $\tau(\Pi)$ contains the rule obtained by replacing each occurrence of an atom $a(\bar{t}) = y$ in r by $a(\bar{t}, y)$ and replacing each occurrence of an atom $a(\bar{t}) \neq y$ in r by $\neg a(\bar{t}, y)$. Also we add following rules to make sure that an attribute cannot have more than on value. For each attribute term $a(\bar{t})$, $\tau(\Pi)$ contains the rule:

$$\neg a(\bar{t}, Y_1) \leftarrow a(\bar{t}, Y_2), Y_1 \neq Y_2 \tag{2.11}$$

   where Y_1 and Y_2 are variables and rule 2.11 will be considered as shorthand for a collection of its ground instances with respect to the appropriate typing.

3. Random selection rules:

   (a) For an attribute a, we have the rule:

$$intervene(a(\bar{t})) \leftarrow do(a(\bar{t}, Y)) \tag{2.12}$$

(b) Each random selection rule of the form 2.6 is translated to the following rules:

$$1\{a(\bar{t}, X) : c_0(X) : p(X)\}1 \leftarrow B, not\ intervene(a(\bar{t})) \tag{2.13}$$

where c_0 is the range of a.

4. For each observation $obs(l)$, $\tau(\Pi)$ contains the rule:

$$\leftarrow not\ l \tag{2.14}$$

5. For each action $do(a(\bar{t}, y))$, $\tau(\Pi)$ contains the rule:

$$a(\bar{t}, y). \tag{2.15}$$

Definition 1 *[Possible world]*

An answer set of $\tau(\Pi)$ *is called a* **possible world** *of* Π.

To define the probability measure over all the possible worlds, we start with some definitions.

Definition 2 *[Possible atoms]*

Let W *be a consistent set of literals of* Σ, Π *be a P-log program,* a *be an attribute, and* y *belong to the range of* a. *We say that the atom* $a(\bar{t}) = y$ *is possible in* W *with respect to* Π *if* Π *contains a random selection rule* r *for* $a(\bar{t})$, *where if* r *is of the form 2.6 then* $p(y) \in W$ *and* W *satisfies* B, *and if* r *is of the form 2.8 the* W *satisfies* B. *We say that* y *is a possible outcome of* $a(\bar{t})$ *in* W *with respect to* Π *via rule* r, *and that* r *is a generating rule for the atom* $a(\bar{t}) = y$.

Let $\Omega(\Pi)$ be the set of all possible worlds of a P-log program Π. We define the corresponding probability $P(W, a(\bar{t}) = y)$ as follows:

Definition 3 $[P(W, a(\bar{t}) = y)]$

1. *Assigned probability:*

   If Π *contains* $pr_r(a(\bar{t}) = y|_c B) = v$ *where* r *is the generating rule of* $a(\bar{t}) = y$, $B \subseteq W$, *and* W *does not contain* $intervene(a(\bar{t}))$, *then*

$$PA(W, a(\bar{t}) = y) = v \tag{2.16}$$

2. *Default probability:*

   Let $|S|$ *denote the cardinality of set* S. *Let* $A_{a(\bar{t})}(W) = \{y | PA(W, a(\bar{t}) = y)\ is\ defined\}$. *Then let*

$$\alpha_{a(\bar{t})}(W) = \sum_{y \in A_{a(\bar{t})}(W)} PA(W, a(\bar{t}) = y)$$

$$\beta_{a(\bar{t})}(W) = |\{y : a(\bar{t}) = y\ is\ possible\ in\ W\ and\ y \notin A_{a(\bar{t})}(W)\}|$$

   The default probability of an atom $a(\bar{t}) = y$ *with respect to* W *is defined as:*

$$PD(W, a(\bar{t}) = y) = \frac{1 - \alpha_{a(\bar{t})}(W)}{\beta_{a(\bar{t})}(W)}$$

3. *Finally,*

$$P(W, a(\bar{t}) = y) = \begin{cases} PA(W, a(\bar{t}) = y) & : \quad y \in A_{a(\bar{t})}(W) \\ PD(W, a(\bar{t}) = y) & : \quad otherwise \end{cases}$$

Now we define the measure of each possible world.

Definition 4 *[Measure]*

1. *The unnormalized probability,* $\hat{\mu}_\Pi(W)$, *of a possible world* W *induced by* Π *is*

$$\hat{\mu}_\Pi(W) = \prod_{a(\bar{t},y) \in W} P(W, a(\bar{t}) = y)$$

   where the product is taken over atoms for which $P(W, a(\bar{t}) = y)$ *is defined.*

2. *the measure,* $\mu_\Pi(W)$ *, of a possible world* W *induced by* Π *is the unnormalized probability of* W *divided by the sum of unnormalized probabilities of all possible worlds of* Π*, i.e.,*

$$\mu_\Pi(W) = \frac{\hat{\mu}_\Pi(W)}{\sum_{W_i \in \Omega} \hat{\mu}_\Pi(W_i)}$$

This concludes our review of syntax and semantics of language P-log.

2.3 Query

The query of P-log is not formally defined in [1]. In this research, we extend the idea of query discussed in paper [1]. We start with some definitions of formula and probability of formula with respect to program Π.

Definition 5 *[Formula]*

We define **formula** *as follows:*

1. *An atom of* Σ *of program* Π *is a formula.*
2. *If* f *is a formula, then* **not** f *is a formula.*
3. *If* f_1 *and* f_2 *are formulas, then* $f_1 \ \wedge \ f_2$ *and* $f_1 \ \vee \ f_2$ *are formulas.*

Definition 6 *[Possible world satisfy formula]*

We say that a possible world W *satisfies a formula* f*, denoted by* $W \vdash f$*, if following conditions are satisfied:*

1. *If* f *is an atom* l*, then* $W \vdash f$ *if and only if* $f \in W$*.*
2. *If* f *has form of* **not** l*, then* $W \vdash f$ *if and only if* $f \notin W$*.*
3. *If* $f = f_1 \wedge f_2$ *where* f_1 *and* f_2 *are formulas, then* $W \vdash f$ *if and only if* $W \vdash f_1$ *and* $W \vdash f_2$*.*

4. *If* $f = f_1 \vee f_2$ *where* f_1 *and* f_2 *are formulas, then* $W \vdash f$ *if and only if* $W \vdash f_1$ *or* $W \vdash f_2$.

Definition 7 *[Probability of formula]*

The **probability of a formula** f *with respect to a P-log program* Π *is the sum of the measures of the possible worlds of* Π *in which* f *is satisfied:*

$$P_{\Pi}(f) = \sum_{W \vdash f} \mu_{\Pi}(W)$$

Definition 8 *[Query]*

A query Q *to a P-log program* Π *of signature* Σ *has the form:*

$$\{f_1, \ldots, f_k\} | obs(l_1), \ldots, obs(l_m), do(a_1(\bar{t}_1) = y_1), \ldots, do(a_n(\bar{t}_n) = y_n) \tag{2.17}$$

where $f_1, \ldots, f_k$ *are formulas of* Σ, $l_1, \ldots, l_m$ *are literals and* $a_1(\bar{t}_1) = y_1, \ldots, a_n(\bar{t}_n) = y_n$ *are atoms. By* $formula(Q)$, *we mean the set* $\{f_1, \ldots, f_k\}$ *and by* Π_Q, *we mean a P-log program consisting of do and obs statements in the query* Q.

The answer to a query Q *w.r.t. a P-log program* Π *of signature* Σ *is a set of formulas* F *such that*

$$F = \arg \max_{f \in formula(Q)} P_{\Pi \cup \Pi_Q}(f) \tag{2.18}$$

i.e., F *is a subset of* $formula(Q)$ *such that for every element* f *of* F, *the probability* $P_{\Pi \cup \Pi_Q}(f)$ *has the largest value, or say* f *is one of the formulas in* $formula(Q)$ *that is most likely being true w.r.t. the program* $\Pi \cup \Pi_Q$.

2.4 SCOU P-log Program

The notion of causally ordered P-log program is introduced in [1] in order to understand theorems of consistency of P-log programs. We extend the above notion and define a new class of P-log programs, called *strongly causally ordered unitary P-log programs* (scou P-log program). We are specially interested in this class as programs within this class have following properties:

1. The *scou* P-log programs are coherent. It naturally matches our understanding of causal relationship among events as well as probability axioms.
2. Every problems modeled by Bayesian network can be translated into *scou* P-log programs and the translation is straight forward.
3. Many ideas used for designing efficient inference engine of Bayesian network can be reused for designing of algorithms for P-log systems which take *scou* programs as input.

In this section, we will first review the definitions of causally ordered unitary P-log program, then we will give the definition of *scou* P-log program.

Definition 9 *[Dependency relations]*

Let l_1 and l_2 be literals of Σ. We say that

- *A literal l_1 is immediately dependent on l_2, written as $l_1 >_i l_2$, if there is a rule r of Π such that l_1 occurs in the head of r and l_2 occurs in the r's body;*
- *A literal l_1 depends on l_2, written as $l_1 > l_2$, if $\langle l_1, l_2 \rangle$ belongs to the reflexive transitive closure of relation $l_1 >_i l_2$;*
- *An attribute term $a_1(\bar{t}_1)$ depends on an attribute term $a_2(\bar{t}_2)$ if there are literals l_1 and l_2 formed by $a_1(\bar{t}_1)$ and $a_2(\bar{t}_2)$ respectively such that l_1 depends on l_2.*

Definition 10 *[leveling function]*

A leveling function, $||$, of Π maps attribute terms of Σ onto a set $[0, n]$ of natural numbers. It is extended to other expressions over Σ as follows:

$$|a(\bar{t}) = y| = |a(\bar{t}) \neq y| = |not\ a(\bar{t}) = y| = |not\ a(\bar{t}) \neq y| = |a(\bar{t})|$$

If B is set of expressions then $|B| = max(\{|e| : e \in B\})$.

Definition 11 *[Reasonable leveling]*

A leveling function $||$ of Π is called **reasonable** *if*

1. *no two random attribute terms of* Σ *have the same level under* $||$*;*
2. *for every random selection rule* $[r]$ $random(a(\bar{t}) : \{y : p(y)\}) \leftarrow B$ *of* Π *we have* $|a(\bar{t}) = y| > |\{p(y) : y \in range(a)\} \cup B|$*;*
3. *for every probability atom* $pr_r(a(\bar{t}) = y|_c B)$ *of* Π *we have* $|a(\bar{t})| > |B|$*;*
4. $a_1(\bar{t}_1)$ *is a random attribute term,* $a_2(\bar{t}_2)$ *is a non-random attribute term, and* $a_2(\bar{t}_2)$ *depends on* $a_1(\bar{t}_1)$ *then* $|a_2(\bar{t}_2)| \geq |a_1(\bar{t}_1)|$*.*

Definition 12 *[*$\|$*-induced structure]*

Let Π *be a causally ordered program with signature* Σ *and leveling* $\|$*, and let* $a_1(\bar{t}), \ldots, a_n(\bar{t}_n)$ *be an ordering of its random attribute terms induced by* $\|$*. By* L_i *we denote the set of literals of* Σ *which do not depend on literals formed by* $a_j(\bar{t}_j)$ *where* $i \leq j$*.* Π_i *for* $1 \leq i \leq n+1$ *consists of all declarations of* Π*, along with the regular rules, random selection rules, actions and observations of* Π *such that every literal occurring in the heads or bodies of these rules belong to* L_i*. We'll refer to* $\Pi_1, \ldots, \Pi_{n+1}$ *a* $\|$*-induced structure of* Π*.*

We use the following example to further explain definition 12

Example 1 *[*$\|$*-induced structure of* Π*]*

%All attributes are Boolean.

1. $random(a)$
2. $random(b)$
3. $c \leftarrow$ **not** d, b
4. $d \leftarrow$ **not** c, b
5. $e \leftarrow a, c,$ **not** e
6. $e \leftarrow \neg a, c,$ **not** e

Let F *be a leveling function defined as following:*

$|a| = 1$ *and* $|b| = |c| = |d| = |e| = 2$.

In Example 1, we can see that F is a reasonable leveling according to the definition. Based on this leveling function, we have the following $\|$-induced structure of Π:

$$\Pi_1 = \emptyset$$

Π_2 consists of one rule:

$$\Pi_2 = \{random(a)\}$$

Π_3 consists of all the rules in Π:

$$\Pi_3 = \Pi$$

Definition 13 *[causally ordered P-log program]*

We say that a P-log program is **causally ordered** *if it has a reasonable leveling function* $||$ *such that*

1. Π_1 *has exactly on possible world;*
2. *if* W *is a possible world of* Π_i *and atom* $a_i(\bar{t}_i) = y_0$ *is possible in* W *with respect to* Π_{i+1} *then the program* $W \cup \Pi_{i+1} \cup obs(a_i(\bar{t}_i) = y_0)$ *has exactly one possible world; and*
3. *if* W *is a possible world of* Π_i *and* $a_i(\bar{t}_i)$ *is not active in* W *with respect to* Π_{i+1} *then the program* $W \cup \Pi_{i+1}$ *has exactly one possible world.*

Continuing with Example 1, we now show that the program in Example 1 is a causally ordered P-log program w.r.t. to the leveling we mentioned in Example 1. For Π_1 and Π_2, it is easy to show that they satisfy all the conditions in definition 13. For Π_3, Let $\{a\}$ be a possible world of Π_2, then there has exactly one possible world of program $\{a\} \cup \Pi_3 \cup \{obs(b)\}$, i.e., the set {a, b, d}. For program $\{a\} \cup \Pi_3 \cup \{obs(\neg b)\}$, the set $\{a, \neg b\}$ is the only possible world of it. Similarly for the other possible world $\{\neg a\}$ of Π_2, we can see that all the conditions of definition 13 are satisfied by programs $\{\neg a\} \cup \Pi_3 \cup \{obs(b)\}$ and $\{\neg a\} \cup \Pi_3 \cup \{obs(\neg b)\}$. Therefore the program Π in Example

1 w.r.t. the leveling function where $|a| = 1$ and $|b| = |c| = |d| = |e| = 2$ is a causally ordered P-log program.

Now let us take a look at another reasonable leveling function where $|b| = |c| = |d| = 1$ and $|a| = |e| = 2$. The induced structure of program Π with respect to this leveling function is shown below:

$$\Pi_1 = \emptyset$$

Π_2 consists of three rules (rule (2), (3) and (4)) of Π:

$$\Pi_2 = \{random(b).\ c \leftarrow \textbf{not}\ d, b.\ d \leftarrow \textbf{not}\ c, b.\}$$

$$\Pi_3 = \Pi$$

We can see that the program Π does not satisfy all the conditions in definition 13 if the $\|$-induced structure of Π is created based on the new leveling function. The set $\emptyset$ is a possible world of Π_1 and $b = true$ is possible in $\emptyset$ w.r.t. Π_1. However, the program $\emptyset \cup \Pi_2 \cup obs(b)$ consists of the following rules

1. $random(b)$
2. $c \leftarrow \textbf{not}\ d, b$
3. $d \leftarrow \textbf{not}\ c, b$
4. $obs(b)$

which has two possible worlds $\{b, c\}$ and $\{b, d\}$.

The above example shows that there are causally ordered P-log programs such that when given another reasonable leveling function, it fails to satisfy all the conditions of definition 13. The algorithm we designed for improving the performance of P-log inference engine relies on the assumption that the input P-log program satisfies all the conditions in definition 13 given an arbitrary reasonable leveling function. Therefore, we introduce a new definition called *strongly causally ordered P-log program.*

Definition 14 *[strongly causally ordered P-log program]*

A P-log program Π *is called* **strongly causally ordered** *P-log program if for every reasonable leveling function,* Π *satisfies all the conditions in definition 13.*

Given a leveling function, to check whether it is a reasonable leveling function is not difficult. As all the conditions listed in definition 11 can be checked by only looking at rules in the program. But it will need some extra computation to determine whether the program is a strongly causally ordered P-log program.

There are types of rules which are commonly used for ASP may not be suitable in P-log program as they will result in a P-log program which is not strongly causally ordered P-log program. We list some of these rules as follows:

- Constraint rules should not be used for representing P-log program. In ASP, constraint rules are logical rules with an empty head:

  $$\leftarrow l_1, \ldots, l_k, not\ l_{k+1}, \ldots, not\ l_n$$

  Since constraint rules are used for eliminating unwanted answer sets in ASP, it violates the second condition of causally ordered P-log program if its body is satisfied.

  Notice that a rule

  $$p \leftarrow a, \neg p$$

  is also served as a constraint for atom a, i.e., a cannot be true in any answer sets of an Answer Set Prolog program which containing this rule. Therefore, in general, rules with this form are not useful for building a *scou* P-log program.

- Even number of negative loops are not suitable for P-log program. An example of such rules are:

  $$a \leftarrow not\ b$$

$$b \leftarrow not\ a$$

These types of rules will create two answer sets: $\{a\}$ and $\{b\}$. Obviously, this rules alone will violate the conditions of definition of causally ordered P-log program.

Now we define a unitary P-log program. Let Π be a ground P-log program containing the random selection rule

$$[r]\ random(a(\bar{t}) : \{X : p(X)\}) \leftarrow B$$

We refer to a ground pr-atom

$$pr_r(a(\bar{t}) = y|_c B) = v$$

as a pr-atom indexing r. We will refer to B as the body of the pr-atom.

Let W_1 and W_2 be possible worlds of Π satisfying B. We say that W_1 and W_2 are probabilistically equivalent with respect to r if

1. for all y, $p(y) \in W_1$ if and only if $p(y) \in W_2$, and
2. for every pr-atom q indexing r, W_1 satisfies the body of q if and only if W_2 satisfies the body of q.

A *scenario* for r is an equivalence class of possible worlds of Π satisfying B, under probabilistic equivalence with respect to r.

By $range(a(\bar{t}), r, s)$, we denote the set of possible values of $a(\bar{t})$ in the possible worlds belonging to scenario s of rule r. Let s be a scenario of rule r. A pr-atom q indexing r is said to be *active in* s if every possible world of s satisfies the body of q.

For a random selection rule r and scenario s of r, let $at_r(s)$ denote the set of probability atoms which are active in s.

Definition 15 *[Unitary rule]*

A random selection rule r is unitary in Π, or simply unitary, if for every scenario s of r, one of the following conditions holds:

1. *For every y in $range(a(\bar{t}), r, s)$, $at_r(s)$ contains a pr-atom of the form $pr_r(a(\bar{t}) = y|_c B) = v$, and moreover the sum of the values of the probabilities assigned by members of $at_r(s)$ is 1; or*
2. *There is a t in $range(a(\bar{t}), r, s)$ such that $at_r(s)$ contains no pr-atom of the form $pr_r(a(\bar{t}) = y|_c B) = v$, and the sum of the probabilities assigned by the members of $at_r(s)$ is less than or equal to 1.*

Definition 16 *[Unitary Program]*

A P-log program is **unitary** *if each of its random selection rules is unitary.*

A P-log program is called strongly causally ordered unitary P-log program if it is both strongly causally ordered and unitary.

2.5 Representation with P-log

We use this section to show how probabilistic reasoning problems are represented by the language P-log. In [1], it has shown several examples of using P-log to represent various interesting problems, we here focus on representing the reasoning problems as a *scou* P-log program.

We discuss how to represent probability problems that are involved with combination and permutation. For permutation problem, the order of which object is selected matters while for combination problem the order does not matter.

Suppose we have a set O of M objects and we want to select N objects from these M objects. We model this type of problems as every time we select one object from M and we perform such selection for N times (steps). We can use random attribute term $select : steps \rightarrow object$ to represent such random selections. Because, in P-log, random attribute term *select* can only pick one object as its value. Therefore, we

need parameter *steps* to distinguish N steps of selections. The following code is an example of such permutations where repetition is allowed.

```
object={1..m}.
steps={1..n}.
select: steps -> object.
#domain steps(S).
[r(S)] random(select(S)).
```

When repetition is not allowed, we can use auxiliary attribute term $can\_select(N, O)$ to define the dynamic ranges. $can\_select(N, O)$ is true if object O can be selected in Nth steps. The resulting program is shown as follows:

```
object={1..m}.
steps={1..n}.
select: steps -> object.
#domain object(O).
#domain steps(S;S1;S2).
-can_select(S1,O) :- select(S2)=O,  S1<S2.
can_select(S,O):- not -can_select(S,O).
[r(S)] random(select(S):{Y:can_select(S,Y)}).
```

However, this approach increases the number of ground rules and introduces too many auxiliary attribute terms in the program. Consider a problem with 10 objects and 5 steps of selections. we will have 50 attribute terms for *can_select* and have to include hundreds of grounded rules in the program. This will significantly increase the size of ground program. Instead of using dynamic range, another approach is to use constraints:

```
object={1..m}.
$steps={1..n}.
select: steps -> object.
```

```
#domain steps(S).
[r(S)] random(select(S)).
:- select(S1)=O,
   select(S2)=O,
   S1<>S2.
```

The last rule says that we can not select same object O in two different selection steps. While using *lparse* and *smodels* for computing possible worlds, this approach will give correct number of possible worlds and the unnormalized probability of each possible world will be correct too. However, by our definition, this program is not a scou P-log program. Because the condition that every outcome of random selection rules will result in a possible world is not satisfied here. To solve this problem, we may use the following trick:

```
object={1..m}.
steps={1..n}.
select: steps -> object.
#domain steps(S;S1;S2).
#domain object(O).
[r(S)] random(select(S)).
invalid :- select(S1)=O
           select(S2)=O
           S1<>S2.
-invalid :- not invalid.
```

Instead of using constraints, we introduce boolean attribute atom *invalid* as the head of previous constraint rules. Then for each query, we add $obs(invalid = false)$ into the condition part of the query. Since *obs* are acted as constraints according to the semantics of P-log, this will give the same effects as we directly use constraints in the P-log program. The advantage of using this trick is that the program without query is a strongly causally unitary P-log program, therefore, we can apply advanced

inference algorithms to solve the problem.

When ordering does not matter, the selection problem is considered as combination problem. Many combination problems can be solved by using above techniques without any changes. The disadvantage of using the above program is that this representation generates too much more possible worlds than necessary. For example, this approach treats selections (3,2,1) and (1,2,3) as different possible worlds, while in combination problem, there is no difference between these two. We can achieve this by change one of the statement of previous program as:

```
invalid :- select(S1)=O1,
           select(S2)=O2,
           S1<S2, O1<=O2.
```

The above rule says that for every two selections, if the result of an earlier selection is less or equal than the result of later selection, then such selections are invalid. To use such rule, we need a total liner ordering among elements of the sort *object* where $\leq$ relations are well defined.

Now we will show a complete example, called *poker* problem, which uses above techniques to represent combination problems.

Example 2 *[Poker]*

In this problem, we have a desk of 12 cards. With four different suites and 3 different kinds. We randomly pick 5 cards and want to know what is the probability of having a pair in these 5 cards, i.e., having exactly one pair of cards in the same kinds.

The complete program is shown as below. It contains three sorts.

```
#domain card(C;C1;C2).
#domain kind(K;K1;K2).
#domain step(S;S1;S2).
step={1..5}.
```

```
kind={1..3}.
card={1..12}.
```

The only random attribute term *draw* takes *steps* as its parameter and maps to *card*

```
draw: steps -> card.
```

For each card C, relation $kc(C) = K$ means the card C is kind K. We list all these information as facts in the our program for every cards.

```
kc(1)=1. kc(2)=2. kc(3)=3.
kc(4)=1. kc(5)=2. kc(6)=3.
kc(7)=1. kc(8)=2. kc(9)=3.
kc(10)=1. kc(11)=2. kc(12)=3.
```

The following rules are using the same methods we described before to generate all possible combinations of drawing cards.

```
[dr(S)] random(draw(S)).
invalid :- draw(S1)=C1,
           draw(S2)=C2,
           S1<S2,C1<=C2.
-invalid :- not invalid.
```

We say that the picked cards have at least two cards for kind K, if in two steps S_1 and S_2, the cards we have picked, $C1$ and $C2$ are same kind. we use the earlier step S_1 in relation $pair(K, S_1)$ for further determine whether we have exactly two cards in kind K.

```
pair(K,S1) :- kc(C1)=K, kc(C2)=K,
              draw(S2)=C2, draw(S1)=C1,
              S1<S2.
```

Notice that if we have N $(N > 1)$ cards in a kind K, then the above rule will make

$N-1$ atoms $pair(K, S)$ to be true. Therefore, we can use the following rule to decide if the cards have more than two cards in the kind K

```
more_than_two(K) :- pair(K,S1),
                    pair(K,S2),
                    S1<S2.
```

Therefore, we say that we have exactly two cards in a kind K if for some step S, $pair(K, S)$ is true but we do not have more than two cards in K.

```
pair(K) :- pair(K,S),
           not more_than_two(K).
```

We also need to consider that we may have more than two pairs in our hand. This relation *double_pair* can be defined as follow.

```
double_pair :- pair(K1), pair(K2),
               K1<>K2.
```

Now we can say if the hand contains exactly one pair of same kinds if for some kind K, $pair(K)$ is true and there is no other pairs, i.e., *double_pair* is not true.

```
pair :- pair(K),
        not double_pair.
```

Now we can propose our query as follows:

```
{pair}|obs(-invalid).
```

This completes our representation of *poker* example and it is easy to see that the resulting program is a *scou* program.

Chapter 3

ALGORITHM

In this chapter, we presents two algorithms for implementing P-log systems. The first algorithm is called *translation* algorithm. This algorithm translates a P-log program to ASP program and uses existed Answer Set solver to find all possible worlds of P-log program. Since the translation method has been discussed in semantics of P-log, here we only focus on rules which are translated differently from previous chapter and discuss how to encode and decode probabilistic information such that probability of formulas can be computed directly from output of Answer Set solver. The second algorithm consists of two major parts: (1) a *ground* algorithm that produces a ground P-log program such that the resulting ground program only consists of rules that are related to the query. (2) a *solver* algorithm that it utilizes the idea of evaluating *patial* possible worlds to compute probability of formulas.

In section 3.1, we will illustrate the details of *translation* algorithm; In section 3.2, we will discuss the *ground* algorithm and in section 3.3, we will discuss the *solver* algorithm.

3.1 Translation Algorithm

In section 2.2, we have shown how to translate a P-log program into an Answer Set Prolog program such that each possible world of the P-log program is one to one correspondence to the answer set of its counterpart. In this section, we will focus on how to encode probabilistic information of a P-log program into some rules of Answer Set Prolog program and how to extract this information out of answer sets of the Answer Set Prolog program to compute the measure of each possible world.

The idea of encoding probability information of a P-log program is to define atoms $pd/2$ and $pa/3$. such that the resulting Answer Set Prolog program Π has the following properties:

1. For each answer set W of Π, if $a(\bar{t}) = y$ is possible with respect to W via some random selection rule r, then there exists an atom $pd(r, a(\bar{t}, y)) \in W$.
2. For each answer set W of Π and each pr-rule, if $B \subseteq W$ and W does not contain $intervene(a(\bar{t}))$, then $pa(r, a(\bar{t}, y), v) \in W$.

To define $pd/2$ and $pa/3$, we have following rules:

1. For each random selection rule of the form 2.6, we add the following rule to the Answer Set Prolog program:

$$pd(r, a(\bar{t}, y_i)) \leftarrow p(y_i), B, \text{not } intervene(a(\bar{t})). \tag{3.1}$$

   where $y_i \in range(a)$.

2. For each random selection rule of the form 2.8, we add the following rule to the Answer Set Prolog program:

$$pd(r, a(\bar{t}, y_i)) \leftarrow B, \text{not } intervene(a(\bar{t})). \tag{3.2}$$

   where $y_i \in range(a)$.

3. For each pr rules of the form 2.9, we add the following rule to the Answer Set Prolog program:

$$pa(r, a(\bar{t}, y), d(v_1, v_2)) \leftarrow B. \tag{3.3}$$

where $v = v_1/v_2$, and v_1, v_2 are integers. The reason we use v_1 and v_2 instead of v is because most existed Answer Set solvers can only take integers but not float point numbers such as 0.5.

After all answer sets are computed by Answer Set Prolog solver, we can extract all the necessary probabilistic information from answer sets to computer unnormalized measures and normalized measures of each possible world.

Algorithm 1: UnormalizedProbability

Input: W: a possible world of a translated P-log program

Output: p: the unnormalized measure of W

```
1 A_1, A_2: assignments that assign each pd atoms with a value v ∈ [0, 1];
2 begin
3     A_1 := AssignedProbability(W);
4     A_2 := DefaultProbability(W, A_1);
5     p := GeneralProbability(W, A_2);
6     return p;
7 end
```

Algorithm 1 shows how unnormalized measure is computed for a possible world W. The algorithm contains three steps:

1. $AssignedProbability(W)$: In this function, we assign $pd(r, a(\bar{t}) = y)$ to v_1/v_2 if there exists an atom $pa(r, a(\bar{t}) = y, d(v_1, v_2)) \in W$, otherwise, assign $pd(r, a(\bar{t}) = y)$ to -1.

2. $DefaultProbability(W)$: This function computes the default probability of every possible outcomes if the probability of such possible outcome is assigned with -1 in $AssignedProbability$.

3. $GeneralProbability$: In this function, we return the unnormalized probability $\hat{\mu}(W)$ of W. $\hat{\mu}(W)$ is computed as follows:

$$\prod_{pd(r,a(\bar{t})=y)\in W \wedge a(\bar{t})=y \in W} P(pd(r, a(\bar{t}) = y))$$

   where $P(pd(r, a(\bar{t}) = y))$ is the value assigned to atom $pd(r, a(\bar{t}) = y)$ in step 1 or step 2.

With unnormalized measure available for each possible world, the computation of normalized measure of each possible world comes directly from the definition and so is the probability of formulas being true.

3.2 The *Ground* Algorithm

The description of our algorithm will use the following definitions and notations.

Definition 17 *[$Term(S)$]*

Let S be a set of ground literals of a signature Σ. By $Term(S)$, we denote the set of ground attribute terms occurring in the literals from S.

For example, if $S = \{open(1) = true, open(1) = false, select(1) \neq true\}$, then $Term(S) = \{open(1), select(1)\}$.

Definition 18 *[Dependent Set]*

Let Π be a ground P-log program with signature Σ and l be a ground literal of Σ. We define the **dependent set** *of l, written as $Dep_\Pi(l)$, as the minimal set of ground attribute terms from Σ satisfying the following conditions:*

- *$Term(\{l\}) \subseteq Dep_\Pi(l)$;*
- *$Dep_\Pi(l)$ is closed under regular rules of Π, i.e., for every regular rule $l_0 :- B$ of Π if $Term(\{l_0\}) \subseteq Dep_\Pi(l)$, then $Term(B) \subseteq Dep_\Pi(l)$);*
- *For a random selection rule of the form: $[r]$ random $(a(\bar{t})) :- B$,*

  if $a(\bar{t}) \in Dep_\Pi(l)$, then $Term(B) \subseteq Dep_\Pi(l)$;
- *For a random selection rule of the form: $[r]$ random $(a(\bar{t})) : \{y : p(y)\}) :- B$,*

  if $a(\bar{t}) \in Dep_\Pi(l)$, then $Term(B \cup p(y)) \subseteq Dep_\Pi(l)$;
- *For a pr rule $[r]$ $pr(a(\bar{t}) = y|B) = v$,*

  if $a(\bar{t}) \in Dep_\Pi(l)$, then $Term(B) \subseteq Dep_\Pi(l)$.

Theorem 1 *Let Π be scou P-log program with no do and obs statements and Σ be its signature. Let l be a literal of Σ and $\Pi^l = \{r|Term(Head(r)) \in Dep_\Pi(l), r \in \Pi\}$, then $P_{\Pi^l}(l) = P_\Pi(l)$.*

The above theorem shows that to evaluate the probability of a literal l, we may only need to look at rules whose head belongs to the dependent set of literal l. Hence, Π^l is a subset of Π. Notice that, this result can be easily expanded to formulas because it is easy to add a literal l and regular logical rules into the program Π to make a new program Π' such that each possible world W' of program Π' is a super set of some possible world W of Π with same measure and $W \vdash f$ if and only if $l \in W'$.

The $Ground(\Pi, l)$ algorithm presented in this section will generate a ground P-log program Π_G such that W is a possible world of Π_G if and only if W is a possible world of Π^l.

Algorithm 2: *Ground*

Input: a strongly causally ordered P-log program Π with signature Σ and a ground literal l of Σ.

Output: a ground P-log program Π' with signature Σ, such that $P_{\Pi}(l) = P_{\Pi'}(l)$

1 $S_{term} := Term(\{l\})$;
2 $S_{finished} := \emptyset$;
3 $PreProcess(\Pi)$;
4 $\Pi' :=$ declarations in Π;
5 $\langle S^+, S^- \rangle := \langle \emptyset, \emptyset \rangle$;
6 **while** S_{term} *is not empty* **do** // Π' `is constructed inside the loop`
7 Let $a(\bar{t})$ be an element of S_{term};
8 $\Pi' := \Pi' \cup GroundRule(\Pi, a(\bar{t}), \langle S^+, S^- \rangle)$;
9 $S_{finished} := S_{finished} \cup \{a(\bar{t})\}$;
10 $Evaluate(\Pi', S_{finished}, \langle S^+, S^- \rangle)$;
11 $S_{term} := Dep_{\Pi'}(l) \backslash S_{finished}$;
12 **end**
13 $RemoveDanglingRules(l, \Pi')$;
14 **return** Π'

The algorithm *Ground* is presented in Algorithm 2. We explain each steps of the algorithm *Ground* as follows:

1. A variable S_{term} will be used to contain a collection of unprocessed ground attribute terms relevant to l. Initially it is set to $Term(\{l\})$.

2. A variable $S_{finished}$ will store all the ground attribute terms which are already processed by the program. This set is initialized to the empty set.

3. Procedure $PreProcess$ eliminates all arithmetic expressions occurring in the heads of the rules of program Π. Each such expression, e, is replaced by a new variable X_e and an atom $X_e == e$ is added to the body of the rule. For example, a rule

   $$next(N+1, N) \leftarrow number(N).$$

   will be transformed into the rule

   $$next(N', N) \leftarrow number(N), N' == N+1.$$

4. Π' will contain the ground program relevant to l. It is initialized by the declarations of Π.

5. The algorithm maintains two sets of ground literals: S^+ and S^-. The set S^+ stores literals which are known to always be true in all the possible worlds of the program Π, while S^- stores ground literals that are known never to be true in any possible worlds of the program Π. We say that the body of a ground rule r is falsified by sets $\langle S^+, S^- \rangle$ if one of the following conditions are satisfied:

   - $Pos(r) \cap S^- \neq \emptyset$;
   - $Neg(r) \cap S^+ \neq \emptyset$;
   - The body of r contains false arithmetic atom (e.g. $3 > 4 + 1$).

   We use these two sets to decide whether a ground rule of Π' is falsified or not. They are both initialized to empty sets.

6. The **while** loop starts. Inside the loop, the output program Π' is constructed.

7. The algorithm first selects a ground attribute term $a(\bar{t})$ from S_{term}.

8. Function *GroundRule* takes $a(\bar{t})$, together with Π and $\langle S^+, S^- \rangle$ as input. For every rule r of Π it checks if the attribute term occurring in the head of r unifies with $a(\bar{t})$. If it does, i.e. the m.g.u. u is found, the function computes all ground instances of $u(r)$ whose bodies are not falsified by $\langle S^+, S^- \rangle$. Union of such rules is returned and added to Π'.

9. $S_{finished}$ is updated by adding the, now processed, ground attribute term $a(\bar{t})$.

10. The details of procedure *Evaluate* can be found in later section. In short, the procedure computes consequences of $\Pi' \cup S^+ \cup \neg S^-$ (where $\neg S^- =_{def} \{\textbf{not}\ p : p \in S^-\}$); expands the sets S^+ and S^- accordingly; removes all the rules such that their bodies are falsified by the newly updated sets S^+ and S^-.

11. In the end of each iteration, the set S_{term} is updated. S_{term} will contain all the ground attribute terms occurring in the program that are relevant to l but not yet been processed. Since the size of Π' may increase or decrease because of functions *GroundRule* and *Evaluate*, the size of S_{term} may fluctuate during each iteration of the **while** loop. However, the set $S_{finished}$ increases every time and, since the number of ground terms is finite, S_{term} will decrease eventually to the empty set and the loop will terminate.

12. The **while** loop terminates when the set S_{term} is empty.

13. We first gives the definition of *dangling rule*:

    Definition 19 *[Dangling Rule]*

    Let Π *be a ground P-log program with signature* Σ *and* l *be a ground literal of* Σ. *A rule* r *of* Π *is called a* **dangling rule** *with respect to the literal* l *and the program* Π *if the attribute term* $a(\bar{t})$ *defined by the rule* r *does not belong to* $Dep_{\Pi}(l)$.

The procedure $RemoveDanglingRules(l, \Pi)$ removes from Π all the rules dangling with respect to literal l and program Π.

14. The ground program Π' will be returned.

Next, we will give the details of functions *GroundRule* and *Evaluate*.

3.2.1 Function *GroundRule*

Algorithm 3: *GroundRule*

Input: a strongly causally ordered P-log program Π with signature Σ and a ground attribute term t and two sets of ground literals, $\langle S^+, S^- \rangle$, from signature Σ.

Output: Set R such that $r' \in R$ iff r' is a ground instance of some rule of Π, $\{t\} = Term(\{Head(r')\})$ and the body of r' is not falsified by $\langle S^+, S^- \rangle$.

```
1 R := ∅;
2 foreach rule r in Π do
3 |   if IsMatch(r, t) then
4 |   |   R := R ∪ GetGroundRule(r, t, ⟨S+, S−⟩);
5 |   end
6 end
7 return R
```

Function *GroundRule* is presented in Algorithm 3. Steps of function *GroundRule* is described as follows:

1. The set R of ground rules will be initialized to the empty set.
2. The loop will check each rule r (including regular rules, random selection rules and pr rules) of Π to see whether any ground instance of r should be generated and added to R. This is done as follows:
3. Function $IsMatch(r, t)$ checks if there is an m.g.u u which unifies the attribute term d defined by a rule r with ground attribute term t.

If such unification exists and terms assigned by it to variables of d belong to the domains of these variables then function $IsMatch(r, t)$ return true, otherwise it returns false.

4. Function $GetGroundRule(r, t, \langle S^+, S^- \rangle)$ returns the set of all ground instances r' of rule r such that the head of r' is a literal that formed by the ground attribute term t and the body of r' is not falsified by $\langle S^+, S^- \rangle$.

7. The set R is returned at the end.

3.2.2 Procedure *Evaluation*

We start with some definitions.

Definition 20 *[supported literals]*

We say that a ground literal l is **supported** *by a ground program Π if there exists a rule r in Π such that,*

- *if r is a regular rule, then l is the head of r, and*
- *if r is a random selection rule, then $Term(l)$ is defined by r.*

The procedure *Evaluate* uses procedures *One_step* and $Modify$ which we will describe first.

The procedure $One\_step(\Pi, S_f, \langle S^+, S^- \rangle)$ takes a ground p-log program Π, a set, S_f, of ground terms and two sets of ground literals $\langle S^+, S^- \rangle$ as input and it extends $\langle S^+, S^- \rangle$ as follows:

- If there exists a ground regular rule r of Π such that l is the head of r and $Pos(r) \subseteq S^+$ and $Neg(r) \subseteq S^-$, then the literal l is added to S^+.
- If an atome $a(\bar{t}) = y \in S^+$, then every atom of the form $a(\bar{t}) \neq y'$ where $y' \neq y$ is added to S^+ and $a(\bar{t}) = y'$ is added to S^-.
- If $l \in S^+$ then $\neg l$ is added to S^-.

- Every atom $a(\bar{t}) = y$ such that $a(\bar{t}) \in S_f$ and $a(\bar{t}) = y$ is not supported by the program Π is added to S^-.

The procedure $Modify(\Pi, \langle S^+, S^- \rangle)$ simplifies the program Π as follows:

1. All rules whose bodies are falsified by $\langle S^+, S^- \rangle$ are removed from Π.
2. If a literal $l \in S^+$, then all the regular rules in Π whose heads are l and all the random selection rules in Π which define $Term(l)$ are removed from Π.
3. From the remaining rules, extended literals in the body that are valuated to *true* with respect to $\langle S^+, S^- \rangle$ are removed.

Now, we describe the procedure *Evaluate*.

Algorithm 4: *Evaluate*

Input: a ground P-log program Π with signature Σ, a set, S_f, of ground terms and two sets of ground literals $\langle S^+, S^- \rangle$.

```
repeat
    ⟨N+, N−⟩ = ⟨S+, S−⟩;
    ⟨S+, S−⟩ = One_step(Π, S_f, ⟨S+, S−⟩);
    Π = Modify(Π, ⟨S+, S−⟩);
until ⟨N+, N−⟩ = ⟨S+, S−⟩ ;
```

Algorithm 4 shows the procedure $Evaluate(\Pi, S_f, \langle S^+, S^- \rangle)$. Variables N^+ and N^-, storing sets of ground literals, are used for checking whether a fixed point is reached. The procedure extends sets $\langle S^+, S^- \rangle$ and simplifies program Π by calling procedure *One_step* and $Modify$ iteratively until the least fixed point is reached.

Let S_i^- be the values of variables S^+ and S^- at the end of ith iteration, we have $S_j^+ \subseteq S_{j+1}^+$ and $S_j^- \subseteq S_{j+1}^-$. Therefore, the operation on $\langle S^+, S^- \rangle$ is monotonic. Since the number of attribute atoms in the ground program is finite, the loop will reach the least fixed point and terminate.

The following example is used to illustrate the grounding process. We will concentrate on the most interesting procedures: *Evaluate* and *RemoveDanglingRule*.

Example 3 *Consider the following program* Π*:*

```
a, b, c, d, l : boolean.


1. l :- a.
2. l :- b.
3. [a] random(a).
4. b :- c, d.
5. [c] random(c).
6. -d.
```

Let $\langle S^+, S^- \rangle = \langle \emptyset, \emptyset \rangle$ and $S_{finished} = \{l, a, b, c, d\}$, we compute $Evaluate(\Pi, S_{finished}, \langle S^+, S^- \rangle)$ as follows:

Firstly, the sets $\langle S_0^+, S_0^- \rangle = \langle \emptyset, \emptyset \rangle$ and $\Pi_0 = \Pi$. In the first iteration, the procedure $One\_step(\Pi_0, S_{finished}, \langle S_0^+, S_0^- \rangle)$ will add the literal $\neg d$ to S_0^+ since rule (6) has an empty body. Therefore, $\langle S_1^+, S_1^- \rangle = \langle \{\neg d\}, \emptyset \rangle$. The procedure $Modify(\Pi_0, \langle S_1^+, S_1^- \rangle)$ removes rule (6) from the program since the value of its head is known. In the second iteration, the procedure $One\_step(\Pi_1, S_{finished}, \langle S_1^+, S_1^- \rangle)$ adds the literal d to S_1^-. The procedure $Modify(\Pi_0, \langle S_2^+, S_2^- \rangle)$ removes rule (4) since its body is falsified. In the third iteration, the procedure $One\_step(\Pi_1, S_{finished}, \langle S_2^+, S_2^- \rangle)$ adds the literal b to S_2^- since b is not supported by program Π_1 after rule (4) is removed. Therefore, $\langle S_3^+, S_3^- \rangle = \langle \{\neg d\}, \{d, b\} \rangle$. The procedure $Modify(\Pi_1, \langle S_3^+, S_3^- \rangle)$ removes rule (2) from Π_1 and results in program Π_2 as follows:

```
l, a, b, c, d : boolean.


1. l :- a.
2. [a] random(a).
```

```
3. [c] random(c).
```

Since no further attribute atoms will be added to $\langle S_3^+, S_3^- \rangle$ in the forth iteration, a least fixed point is reached and the procedure *Evaluate* terminates.

Now we compute $RemoveDanglingRules(l, \Pi_2)$.

Because the attribute term $c \notin Dep_{\Pi}(l)$, rule (3) is a *dangling* rule with respect to the ground literal l and Π_2. We remove rule (3) from Π_2 and the resulting program, containing only two rules, is shown below:

```
l, a, b, c, d : boolean.


1. l :- a.
2. [a] random(a).
```

3.3 The *Solver* Algorithm

The naive P-log inference engine translates the ground P-log program to an Answer Set Prolog program and uses Answer Set Solver, such as *Smodels*, to compute all the possible worlds and then extracts probability information from each possible worlds. However, we can improve the efficiency of such computation by avoiding computing all the possible worlds. We use the following example to show that sometimes computing all possible worlds may not be necessary.

Example 4 *[A scou P-log program* Π*]*

```
%Beginning of the Program
a, b, f : boolean.
[a] random (a).
[b] random (b).
f :- a.
f :- b.
```

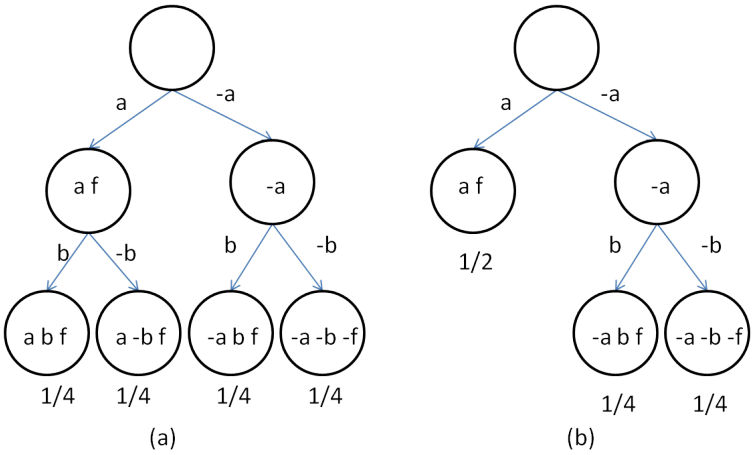

Figure 3.1: Possible worlds of Example 4

```
-f :- not f.
%End of the Program
```

To evaluate the probability of f being true, the previous P-log inference engine will call *Smodels*[3] for computing all possible worlds, which will generate 4 possible worlds, shown in Figure 3.1(a).

Notice that to compute probability of f we don't need to fire random selection rule b for the node $\{a, f\}$. Since the program is strongly causally ordered and unitary, the tree shown in Figure 3.1(a) is a unitary tree that represents Π. By lemma 5 in [1], the literal f is always true in both possible worlds ($\{a, f, b\}$ and $\{a, f, \neg b\}$) extended from $\{a, f\}$, and furthermore, by lemma 1 in [1], $P_{\Pi}(\{a, f\}) = P_{\Pi}(\{a, f, b\}) + P_{\Pi}(\{a, f, \neg b\})$. Therefore, we can compute the probability of f being true by summing up $P_{\Pi}(\{a, f\})$ and $P_{\Pi}(\{\neg a, b, f\})$, as shown in Figure 3.1(b).

The algorithm we will describe next takes advantage of the *scou* property of the input program. This is done by extending standard three valued interpretation to a special 5 valued interpretation and by building a special tree that helps to reduce the unnecessary computation.

3.3.1 Algorithm Closure

Definition 21 *[f-interpretation]*

An **f-interpretation** *is a 5 valued interpretation that maps literals of some signature* Σ *into the set* $\mathcal{V} = \{T, MT, U, MF, F\}$ *of* truth values.

Symbols T, MT, U, MF and F stand for *true*, *must be true*, *unknown*, *must be false*, and *false* respectively. The set $\mathcal{V}$ has a natural truth-based ordering relation $<_t$:

$$F <_t MF <_t U <_t MT <_t T$$

Two truth-values v_1 and v_2 are called *incompatible* if $U <_t v_1$ and $v_2 <_t U$.

The f-interpretation A can be expanded to sets of literals as follows:

$A(\{l_1, \ldots, l_n\}) = min_{1 \leq i \leq n}\{A(l_i)\}$.

To expand A to sets of extended literals we define default negation of truth values: **not** $T = F$, **not** $MT = MF$, **not** $U = U$, **not** $MF = MT$ and **not** $F = T$, and define

$A(\{l_1, \ldots l_n, \textbf{not}\, l_{n+1}, \ldots, \textbf{not}\, l_m\}) = min\{A(l_1), \ldots, A(l_n), \textbf{not}\, A(l_{n+1}), \ldots, \textbf{not}\, A(l_m)\}$

Definition 22 *[Extension]*
An f-interpretation A' *is called an* **extension** *of an f-interpretation* A *(written as* $A \leq_i A'$*) if for every literal* $l \in \Sigma$*, the following two conditions are satisfied:*

- $A(l) \geq_t MT \Longrightarrow A'(l) \geq_t A(l)$,
- $A(l) \leq_t MF \Longrightarrow A'(l) \leq_t A(l)$,

Intuitively, A' is an extension of A if A' gives at least as much knowledge about literals of Σ as A does.

Let A be an f-interpretation. By $true(A)$ we denote the set of literals true in A, i.e.,

$$true(A) = \{a(\bar{t}, y) \ : \ A(a(\bar{t}) = y) = T\} \cup \{\neg a(\bar{t}, y) \ : \ A(a(\bar{t}) \neq y) = T\}$$

Definition 23 *[Consistent Interpretation]*
An f-interpretation A is called **consistent** *with respect to a ground P-log program Π if there exists an extension A' of A, such that $true(A')$ is a possible world of Π.*

Since no possible world contains l and $\bar{l}$ at same time. An f-interpretation A such that $A(l) > U$ and $A(\bar{l}) > U$ is inconsistent. We use symbol A^c to denote inconsistent f-interpretations.

Next, we will introduce the function $Closure(\Pi, A)$. It returns A^c if the given f-interpretation A with respect to a ground P-log program Π is not consistent, or it returns an f-interpretation, A', of A such that

- A' is an extension of A.
- If A_W is an extension of A such that $true(A_W)$ is a possible world of program Π then A_W is also an extension of A';

Before we describe function $Closure(\Pi, A)$, we first describe function $AtLeast(\Pi, A)$ and function $AtMost(\Pi, A)$.

We start with some definitions and terminology.

Definition 24 *[Weakly Falsified Rule]*

A rule r of P-log program is said to be a **weakly falsified rule** *with respect to to an f-interpretation A, if $A(body(r)) = MF$.*

Definition 25 *[Falsified Rule]*

A rule r of P-log program is said to be a **falsified rule** *with respect to to an f-interpretation A, if $A(body(r)) = F$.*

Definition 26 *[Weakly Supporting Rule]*

A rule r is said to be a **weakly supporting rule** *of a literal l with respect to an f-interpretation A, if it satisfies the following condition:*

- $A(body(r)) \geq U$;
- *if r is a regular rule, then l is the head of r,*
- *if r is a random selection rule, then r defines $Term(l)$.*

Definition 27 *[$LC(\Pi, A)$]*

Let Π be a P-log program and A be an f-interpretation. Let $LC(\Pi, A)$ denote the f-interpretation, A', which is defined as follow:

For each literal $l \in \Sigma(\Pi)$,

1. *We assign the truth value T to $A'(l)$, if one of the following conditions holds:*
   (a) *There exists a regular rule $r \in \Pi$ such that $l = head(r)$ and $A(body(r)) = T$;*
   (b) *The literal l has the form $a(\bar{t}) \neq y$ and there exists a literal $a(\bar{t}) = y'$ such that $y' \neq y$ and $A(a(\bar{t}) = y') = T$;*
   (c) *$A(l) = T$.*
2. *We assign the truth value MT to $A'(l)$, if none of the conditions listed in (1) is satisfied, and one of the following conditions holds:*
   (a) *There exists a regular rule $r \in \Pi$ such that $l = head(r)$ and $A(body(r)) = MT$;*
   (b) *The literal l has the form $a(\bar{t}) \neq y$ and there exists a literal $a(\bar{t}) = y'$ such that $y' \neq y$ and $A(a(\bar{t}) = y') = MT$;*
   (c) *There exists a regular rule r such that $A(head(r)) \leq MF$, such that* **not** $l \in body(r)$, $A($**not** $l) = U$ *and* $A(body(r) \backslash \{$**not** $l\}) > U$;

(d) *There exists a regular rule r such that it is the only rule which weakly supports the literal $head(r)$, and $A(head(r)) = MT$ and $l \in pos(r)$;*

(e) *$A(l) = MT$.*

3. *We assign the truth value F to $A'(l)$ if one of following conditions holds:*

   (a) *All rules which support l are falsified by A.*

   (b) *$A(l) = F$*

4. *We assign the truth value MF to $A'(l)$, if condition (3) is not satisfied, and one of the following conditions holds:*

   (a) *All rules which support l are weakly falsified rules with respect to A;*

   (b) *There exists a regular rule r such that $A(head(r)) \leq MF$, such that $l \in body(r)$, $A(l) = U$ and $A(body(r) \backslash \{l\}) > U$;*

   (c) *There exists a regular rule r such that it is the only rule which weakly supports the literal $head(r)$, and $A(head(r)) = MT$ and $l \in neg(r)$;*

   (d) *$A(\bar{l}) \geq MT$;*

   (e) *$A(l) = MF$.*

5. *We assign the truth value U to $A'(l)$ if none of the conditions listed in (1)-(4) is satisfied.*

If for any literal l, the truth value of $A'(l)$ defined as above is not unique, then A' is not well defined. It indicates there is an inconsistency in A since we attempted to assign incompatible values to a literal l. In this case, we let $LC(\Pi, A) = A^c$.

Proposition 1 *Let Π be a ground P-log program and A be an f-interpretation. Let $A' = LC(\Pi, A)$ where $A' \neq A^c$.*

- *A' is an extension of A.*

- *If A_W is an extension of A such that $true(A_W)$ is a possible world of program Π then A_W is also an extension of A';*

Since $LC(\Pi, A)$ is monotonic with respect to its second argument under the extension ordering $\leq_i$, by Knaster-Tarski theorem it has the least fixpoint which can be computed by an iteration method. This is shown in Algorithm 5. Function $AtLeast(\Pi, A)$ computes the least fixpoint of $LC(\Pi, A)$, in each iteration, if $LC(\Pi, A) = A^c$, the function returns the inconsistent f-interpretation A^c, otherwise it continues until a fixpoint is reached.

Algorithm 5: $AtLeast(\Pi, A)$

Input: A ground P-log program Π and an f-interpretation A.

Output: the least fixpoint of $LC(\Pi, A)$ if it exists, or A^c otherwise

```
1 repeat
2 |   A' := A;
3 |   if A' = A^c then return A^c;
4 |   A := LC(Π, A');
5 until A = A' ;
6 return A';
```

We use following example to illustrate the computation of $LC(\Pi, A)$.

Example 5 *Consider the following program in which all attribute atoms are Boolean.*

```
a :- not b.
b :- not c.
[c] random(c).
```

Let A be an f-interpretation that maps literal a to MT and maps all other literals to U. Then $LC(\Pi, A)$ defines an f-interpretation A_1 in which,

- $A_1(b) = MF$: Because of 4(c) of the definition of $LC(\Pi, A)$;
- For other literals l, $A_1(l) = A(l)$.

Similarly, $LC(\Pi, A_1)$ defines an f-interpretation A_2 in which,

- $A_2(c) = MT$: Because of 2(c) of the definition of $LC(\Pi, A)$;
- For other literals l, $A_2(l) = A_1(l)$.

Since $A_3 = LC(\Pi, A_2) = A_2$, the function $AtLeast(\Pi, A)$ returns A_2.

To describe the function $AtMost(\Pi, A)$, we first define the operator $G^{\Pi}_{L^+,L^-}(X)$, where L^+ and L^- are disjoint sets of extended literals and X is a set of literals:

$G^{\Pi}_{L^+,L^-}(X) = \{a \in Lit(\Pi) | \text{there is a regular rule } r \text{ such that } a = head(r), pos(r) \subseteq X \text{ and } neg(r) \cap L^+ = \emptyset\} \backslash L^-$

Since the operator $G^{\Pi}_{L^+,L^-}(X)$ is monotonic with respect to set inclusion relation, we can compute the least fixed point of $G^{\Pi}_{L^+,L^-}(\emptyset)$ by the standard iteration method.

The function $AtMost(\Pi, A)$ takes a ground P-log program Π and an f-interpretation A as input and returns an f-interpretation. The steps of function $AtMost(\Pi, A)$ are listed as following:

Function $AtMost(\Pi, A)$:

1. If $A = A^c$ then return A^c.
2. Construct
   $L = \{l \mid A(l) = T\} \cup \{\textbf{not } l \mid A(l) = F\}$
   $L^+ = \{l \mid A(l) = T, l \in L\}$
   $L^- = \{l \mid A(l) = F, l \in L\}$.
3. Compute the least fixed point, X, of $G^{\Pi}_{L^+,L^-}(\emptyset)$
4. Define a new mapping A' as follow: If there is no literal $l \in Lit(\Pi)\backslash X$ such that $A(l) > U$ then

$$A'(l) = \begin{cases} F & : \quad l \in Lit(\Pi)\backslash X \\ A(l) & : \quad otherwise \end{cases}$$

otherwise, $A' = A^c$.

5. Return A'.

We use an example below to illustrate the computation of $AtMost(\Pi, A)$:

Example 6 *Consider the following program* Π*:*

```
(1) p :- p.
(2) q :- not p.
(3) r :- p.
```

Let A be an f-interpretation that maps all the literals to truth value U. Hence, we have $L^+ = \emptyset$ and $L^- = \emptyset$. We compute the least fixed point of $G^{\Pi}_{\emptyset,\emptyset}(\emptyset)$ as follows: $G^{\Pi}_{\emptyset,\emptyset}(\emptyset) = \{q\}$.

Because q is the head of rule $q \leftarrow$ **not** p and $\{p\} \cap L^+ = \emptyset$, the literal q is added to $G^{\Pi}_{\emptyset,\emptyset}(\emptyset)$. Literals p and r are not included since for rule (1) and (3), the body $\{p\} \not\subseteq \emptyset$. Notice that $G^{\Pi}_{\emptyset,\emptyset}(\{q\}) = \{q\}$, therefore the least fixed point of $G^{\Pi}_{\emptyset,\emptyset}(\emptyset)$ is $\{q\}$.

By step 4 of algorithm $AtMost(\Pi, A)$, the f-interpretation A' maps literals p and r to truth value F, and leave the truth value of the literal q as U. The function returns A' which is an extension of A.

Proposition 2 *Let* Π *be a P-log program and* A *be an f-interpretation. Let* $A' = AtMost(\Pi, A)$ *and* $A' \neq A^c$*. Then,*

- A' *is an extension of* A*.*
- *If* A_W *is an extension of* A *such that* $true(A_W)$ *is a possible world of program* Π *then* A_W *is also an extension of* A'*;*

Let $Cn(\Pi, A)$ be an operator defined as $AtMost(\Pi, AtLeast(\Pi, A))$. Since both functions $AtLeast(\Pi, A)$ and $AtMost(\Pi, A)$ are monotonic with respect to their second argument A under the extension ordering $\leq_i$, the operator $Cn(\Pi, A)$ is monotonic with respect to its second argument A under the extension ordering $\leq_i$ and therefore

there exist a least fixed point of $Cn(\Pi, A)$.

Algorithm 6: $Closure(\Pi, A)$

Input: A ground P-log program Π and an f-interpretation A.

Output: An f-interpretation A' such that A' is the fixed point of $Cn(\Pi, A)$, or it returns A^c if no such A' exists.

```
1 repeat
2     A' := A;
3     if A' = A^c then return A^c;
4     A = AtLeast(Π, A');
5     A := AtMost(Π, A);
6 until A = A' ;
7 return A';
```

The function $Closure(\Pi, A)$ is shown in Algorithm 6. It calls functions $AtLeast$ and $AtMost$ iteratively until a fixed point is reached, then the result, an f-interpretation will be returned.

Proposition 3 *Let Π be a P-log program and A be an f-interpretation. Let $A' = Closure(\Pi, A)$ and $A' \neq A^c$. Then,*

- *A' is an extension of A.*
- *If A_W is an extension of A such that $true(A_W)$ is a possible world of program Π then A_W is also an extension of A';*

3.3.2 Computing the Probability of a Formula

To describe the computation of probability of a formula f with respect to a P-log program Π we will need the following definitions:

Definition 28 *[Satisfiability]*
An f-interpretation A of literals from Σ satisfies a formula f (denoted by $A \models f$), if:

- *for any literal* l, $A \models l$ *if* $A(l) = T$;
- *for two formulas* f_1 *and* f_2, $A \models f_1 \wedge f_2$ *if* $A \models f_1$ *and* $A \models f_2$;
- *for two formulas* f_1 *and* f_2, $A \models f_1 \vee f_2$ *if* $A \models f_1$ *or* $A \models f_2$;

Sometimes $A \models f$ *is read as* f is true in A.

Definition 29 *[Weighted f-interpretation]*

A **weighted f-interpretation**, I, *is a pair* (A, w), *where* A *is an f-interpretation of literals of* Σ *and* w *is referred to as the weight of* I *which is a real number between [0,1].*

Let $A(I)$ be the first element, the f-interpretation, of I and $w(I)$ be the weight of I. Let Ω be the collection of weighted f-interpretations I such that $true(A(I))$ is a possible world of a program Π. The normalized weight, $\mu(I)$ is defined as follows

$$\mu(I) = \frac{w(I)}{\sum_{I' \in \Omega} w(I')}$$

Definition 30 *[Extension of f-interpretation]*

Let I *and* I' *be two f-interpretations. We say that* I' *extends* I *if* $A(I')$ *is an extension of* $A(I)$.

The weighted f-interpretations will be used to define a *program tree* — the basic data structure used by our query answering algorithm.

Definition 31 *[Program Tree]*

A **program tree** *of* Π *is a tree which satisfies the following properties:*

- *Each node,* N *is labeled by a unique weighted f-interpretation,* I_N.
- *If* N' *is a child of* N *then* $I_{N'}$ *is an extension of* I_N.

- *The set* $true(A(I))$ *of literals is a possible world of program* Π *if and only if weighted f-interpretation* I *is consistent and labels some leaf node* N *of the tree.*

We use function $Initialize$ to create the weighted f-interpretation labeling the root of the tree and function $Branch$ to build weighted f-interpretations of the children of a given node. The function $Initialize(\Pi, Q)$ returns a weighted f-interpretation I, such that for every weighted f-interpretation I' if $true(A(I'))$ is a possible world of $\Pi \cup \Pi_Q$, then I' is an extension of I. Function $Initialize(\Pi, Q)$ returns false if $\Pi \cup \Pi_Q$ is inconsistent.

Function $Initialize(\Pi, Q)$

1. Create a w-interpretation $I = (A, 1)$ where A is initialized by following steps:
   (a) For every sort declaration $c = \{x_1, \ldots, x_n\}$ of Π, the value of $A(c(x_i))$, where $1 \leq i \leq n$, is T;
   (b) For every statement $obs(l)$ occurring in the query Q, the value of $A(l)$ is MT;
   (c) For every statement $do(a(\bar{t}) = y)$ occurring in the query Q, the value of $A(a(\bar{t}) = y)$ is T, and
   (d) The value of remaining literals is U.
2. Let A' be the output of function $Closure(\Pi, A)$. If $A' = A^c$, then return false. Otherwise, return $I' = (A', 1)$.

Notice that in step 1(b) of function $Initialize(\Pi, Q)$, we assigned the truth value MT to all observed literals l. This is because that in the semantics of P-log described by [1], observations are translated as:

$$\leftarrow obs(l), \textbf{not } l. \tag{3.4}$$

Therefore all possible worlds of program $\Pi \cup \Pi_Q$ must falsifies the body of the above rule, i.e., the truth value of l *must be* $true(MT)$. Similar to *do* statement in the query,

since we have rules

$$a(\bar{t}) = y \leftarrow do(a(\bar{t}) = y). \tag{3.5}$$

for each *do* statement, atom $a(\bar{t}) = y$ is supported by rules in Π_Q and hence their truth values are *true* in the initialization.

To describe function *Branch*, we need the following definitions. Notice that these definitions are similar to those in [1]. The difference is that here they are defined with respect to weighted f-interpretations instead of possible worlds.

Definition 32 *[Extended Literal Known by f-interpretation]*

We say that an extended literal l is **known** *with respect to weighted f-interpretation $I = (A, w)$, if A maps l to $\{T, F\}$.*

A set, B, of extended literals is **known** *by I if I knows every element of B.*

Definition 33 *[Ready to Fire]*

We say a random selection rule r is **ready to fire** *with respect to program Π and a weighted f-interpretation $I = (A, w)$, if it satisfies the following conditions:*

- *If r has the form of*

$$[r]\ random(a(\bar{t})) \leftarrow B. \tag{3.6}$$

  then

  1. *there is no $y \in range(a(\bar{t}))$, such that $A(a(\bar{t}) = y) = T$.*
  2. *$A(B) = T$.*
  3. *For every pr rule of the form*

$$pr_r(a(\bar{t}) = y|_c B') = v$$

  B' is known with respect to to I.

- *If r has the form of*

$$[r]\ random(a(\bar{t}) : \{Y : c(Y)\}) \leftarrow B. \tag{3.7}$$

then, in addition to conditions (1), (2) and (3), $c(y)$ should be known with respect to I for every $y \in range(a(\bar{t}))$.

Definition 34 *[Possible outcome]*

We say that y is a **possible outcome** *of $a(\bar{t})$ in a weighted f-interpretation I with respect to program Π via a random selection rule r if the following conditions hold:*

- *if r has the form (3.6), then $y \in range(a(\bar{t}))$ and r is ready to fire with respect to Π and I;*
- *if r has the form (3.7), then $y \in range(a(\bar{t}))$, r is ready to fire with respect to Π and I, and $A(c(y)) = T$.*

Definition 35 *[$P(I, a(\bar{t}) = y)$]*

Let y be a possible outcome of $a(\bar{t})$ in a weighted f-interpretation I with respect to program Π via a random selection rule r. we define $P(I, a(\bar{t}) = y)$ as follows,

1. *Assigned probability: If Π contains $pr_r(a(\bar{t}) = y|_c B) = v$ and I satisfies B, then $PA(I, a(\bar{t}) = y) = v$, otherwise,*
2. *Default probability: Let $|S|$ denote the cardinality of set S. Let $A_{a(\bar{t})}(I) = \{y|PA(I, a(\bar{t}) = y)$ is defined $\}$, and $B_{a(\bar{t})}(I) = \{y|y$ is possible in I and $y \notin A_{a(\bar{t})}(I)\}$. The default probability*

$$PD(I, a(\bar{t}) = y) = \frac{1 - \sum_{y \in A_{a(\bar{t})}(I)} PA(I, a(\bar{t}) = y)}{|B_{a(\bar{t})}(I)|}$$

3. *Finally,*

$$P(I, a(\bar{t}) = y) = \begin{cases} PA(I, a(\bar{t}) = y) & : \quad y \in A_{a(\bar{t})}(I) \\ PD(I, a(\bar{t}) = y) & : \quad otherwise \end{cases}$$

Algorithm 7: $Branch(\Pi, I)$

Input: A ground *scou* P-log program Π and a weighted f-interpretation $I = (A, w)$.

Output: A non-empty set, I_b, of weighted f-interpretations extended from consistent I by firing a random selection rule, or an empty set if no random selection rule is ready to fire, or FALSE if I is inconsistent.

```
1  I_b := ∅;
2  Find a random selection rule r such that r is ready to fire with respect to Π and I;
3  if exists such rule r then
4      foreach possible outcome y of a(t̄) with respect to Π and I via r do
5          Create a new weighted f-interpretation
           I' = (A', w') = (A, w × P(I, a(t̄) = y));
6          if A'(a(t̄) = y) ≥ U then
7              A'(a(t̄) = y) := T;
8              A' := Closure(Π, A');
9              if A' ≠ A^c then add I' = (A', w') to I_b;
10         end
11     end
12     if I_b = ∅ then return false;
13 end
14 return I_b;
```

The function *Branch* is shown in Algorithm 7. It takes a ground *scou* P-log program Π and a consistent weighted f-interpretation I as input and returns a set of consistent weighted f-interpretations, generated by firing a random selection rule r. Steps that are not self-evident are explained as follows:

1. Initialize the set I_b to the empty set.
2. In case there are multiple random selection rules which are ready to fire with respect to I and Π, we pick one randomly.

5. A new weighted f-interpretation $I' = (A', w')$ is generated for each possible outcome y, where $A' = A$ and $w' = w \times P(I, a(\bar{t}) = y)$.

6. If for a possible outcome y, the value of $a(\bar{t}) = y$ is already mapped to MF or F, then attempting to map this literal to value T will cause a contradiction. Therefore, the function discards these weighted f-interpretations.

7. The value of $a(\bar{t}) = y$ is changed from U or MT to T.

8. The changed mapping A' is extended by calling function $Closure(\Pi, A')$,

9. If A' is consistent, then A' will be added to the set I_b.

12. If I_b is an empty set then it means all the possible outcomes of firing the random selection rule r will lead to an inconsistent weighted f-interpretation. In this case, we return false to indicate that the input I itself is inconsistent.

Definition 36 *[Unitary Program Tree]*

Let Π *be a ground scou P-log program,* Q *be a query to* Π *and* $\Pi \cup \Pi_Q$ *be consistent (i.e., there exists at least one possible world of* $\Pi \cup \Pi_Q$*). A* program tree T *with respect to* Π *and* Q *is called* **unitary program tree** *if*

- *The root of* T *is labeled by the output of function* $Initialize(\Pi, Q)$*;*
- *For every two nodes* N *and* N' *of* T*,* N' *is a child of* N *if and only if* $I_{N'} \in Branch(\Pi, I_N)$*.*

We use the following example to show a unitary program tree of a given program Π and its query Q.

Example 7 *Consider the following program* Π *and a query* Q

```
%All atoms are Boolean.
[a] random(a).
[b] random(b).
c :- a.
```

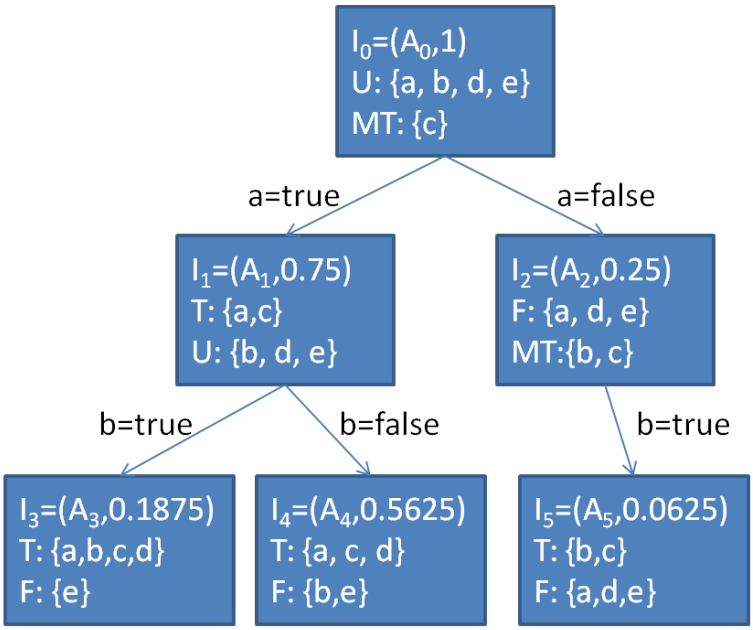

Figure 3.2: Program Tree

```
c :- b.
d :- a, b.
e :- not b.
[a] pr(a)=1/4.
[b] pr(b)=1/4.
%The query Q:
{d, e}|obs(c).
```

Figure 3.2 is a unitary program tree with respect to the program Π and the query Q shown in Example 7. Rectangles are the nodes of the tree and inside a node, $F : \{a, d, e\}$ means that literals a, d and e are mapped to the truth value F by its corresponding weighted interpretation. The label for the arcs indicate the random selection rule we fired in function $Branch(\Pi, Q)$ and as well as the possible outcome we picked to create the children of a given nodes. The calling of function $Initialize(\Pi, Q)$ will return a weighted f-interpretation $I_0 = (A_0, 1)$ where A_0 maps the literal c to truth value MT (since c is observed to be true in the query) and

all other literals to truth value U. This weighted f-interpretation is the root of the programming tree. To compute the children of the root, function $Branch(\Pi, I_0)$ is called. Since both random selection rules are ready to fire, we assume the function $Branch$ picks the random selection rule a to fire. There are two possible outcomes of firing random selection rule a. Let us consider one of the possible outcome where $a = true$. Since $P(I, a = true) = 1/4$, the new created weighted f-interpretation I_1 will be $(A, 0.25)$. Next, we assign the truth value T to the literal $a = true$ as shown in step 7 in Algorithm 7, this results in a new f-interpretation A' where $A'(a = true) = T$ and $A'(l) = A(l)$ for all other literals. In the function $Closure(\Pi, A')$, the truth value of c is changed from MT to T since the present of rule $c \leftarrow a$. The result of function $Closure(\Pi, A')$ is a consistent f-interpretation, therefore I_1 is added to the set I_b which will be returned by function $Branch(\Pi, I_0)$.

Definition 37 *[complete leaf node]*

Let T be unitary program tree with respect to a ground scou program Π and a query Q. A leaf node of N of T is said to be a **complete leaf node** *of T if I_N knows all literals in Σ.*

Definition 38 *[Consistent Leaf Node]*

Let T be unitary program tree with respect to a ground scou program Π and a query Q. A leaf node of N of T is said to be a **consistent leaf node** *of T if $Branch(\Pi, I_N)$ returns an empty set.*

Theorem 2 *Let Π be a ground scou P-log program and Q be a query to Π. Let T be a unitary program tree with respect to Π and Q . Then I is a weighted f-interpretation labeling a complete and consistent leaf node of T if and only if $true(A(I))$ is a possible world of program $\Pi \cup \Pi_Q$.*

Theorem 2 shows that the probability of a formula f with respect to $\Pi \cup \Pi_Q$ can be computed by building the program tree T with respect to Π and Q, then summing up the normalized weight of weighted f-interpretations labeling consistent leaf nodes

of T accordingly. However, it is not always necessary to compute all the leaf nodes of T in order to compute the probability of a formula f. We may utilize the unitary property of our input P-log program to reduce the computation. We describe such method after some definitions.

Definition 39 *[Partial Possible World]*

A weighted f-interpretation $I = (A, w)$ that labels some node of the program tree with respect to a ground scou P-log program Π and a query $Q = F|C$ is called a ***partial possible world*** *of program $\Pi \cup \Pi_Q$, if for all the observations, $obs(l)$, occurring in C, $A(l) = T$.*

Proposition 4 *Let N be a node of a unitary program tree T with respect to Π and Q such that I_N is a partial possible world of program $\Pi \cup \Pi_Q$, then for any node N' descendant from N, $I_{N'}$ is also a partial possible world of program $\Pi \cup \Pi_Q$.*

Definition 40 *[The Smallest Partial Possible World]*

Let N be a node of a unitary program tree T with respect to Π and a query Q such that I_N is a partial possible world of program $\Pi \cup \Pi_Q$ and let N' be the parent of N. I_N is called a **smallest partial possible world** *of $\Pi \cup \Pi_Q$ if $I_{N'}$ is not a partial possible world of $\Pi \cup \Pi_Q$.*

In Example 7, we can see that the weighted f-interpretation I_1 is a smallest partial possible world of the tree since $A_1(c) = T$ and its parent, the root, is not a partial possible world with respect to the program Π and the query Q.

Next we introduce the function $FindPartialWorld(\Pi, Q)$ in which it finds the set, Ω, of smallest partial possible worlds of a unitary program tree T with respect to Π and Q.

Function $FindPartialWorld(\Pi, Q)$ is shown in Algorithm 8. It builds a unitary program tree till all the leaf nodes are labeled by partial possible worlds of program $\Pi \cup \Pi_Q$ or by inconsistent weighted f-interpretation. The weighted f-interpretations

Algorithm 8: $FindPartialWorld(\Pi, Q)$

Input: A ground scou P-log program Π and a query to the program $Q = F|C$

Output: A collection, Φ, of all the smallest partial possible worlds with respect to Π and Q

```
1  I := Initialize(Π, Q);
2  if I is not false then S_I := {I};
3  Φ := ∅;
4  while S_I is not empty do
5  |  Let I be an element of S_I;
6  |  if I knows C_obs then
7  |  |  Φ := Φ ∪ {I};
8  |  else
9  |  |  S_I := S_I ∪ Branch(Π, I);
10 |  end
11 |  Remove I from S_I;
12 end
13 return Φ
```

that are partial possible worlds are collected and returned by the function. Details of steps are explained as follow:

1. The weighted f-interpretation I is the root of the program tree T.

2. If I is consistent, then I is added to the set S_I of weighted f-interpretations. If not, then the while loop will not be executed and the function will return the empty set.

3. The set Φ stores all the partial possible world found in the function and it is initialized with the empty set.

6. By C_{obs} we denote the set of literals occurring in the observations of C, i.e., $C_{obs} = \{l | obs(l) \in C\}$

7. Since all the literals in C_{obs} are initialized with MT in the function *Initialize*, and all weighted f-interpretations in S_I are consistent, see explanation in step 9. Therefore, when I knows C_{obs}, I must satisfy C_{obs}. We add I to Φ as I is a partial possible world of Π and Q.

9. Notice that the function *Branch* returns a set of weighted f-interpretations that are consistent. This guarantees the set S_I only contains all the consistent weighted f-interpretations.

Proposition 5 *Let $\Phi = FindPartialWorld(\Pi, Q)$, then there exists a unitary program tree T such that for every node N of T, the weighted f-interpretation I_N is a smallest partial possible world of T if and only if $I_N \in \Phi$.*

Definition 41 *[Formula Known by weighted f-interpretation]*

*A formula f is **known** by a weighted f-interpretation I if all the literals occurring in f have values other than U.*

A set, F, of formulas are known by a weighted f-interpretation I if all the elements of F are known by I.

Definition 42 *[The Smallest Partial Possible World that Knows Formulas]*

Let F be a set of formulas. Let N and N' be two nodes of a unitary program tree T with respect to Π and Q where N is the parent of N'. We say that $I_{N'}$ is a smallest partial possible world of T with respect to Π and Q that knows F, if $I_{N'}$ is a smallest partial possible world and F is known by $I_{N'}$, and either F is not known by I_N, or I_N is not a partial possible world of T with respect to Π and Q.

In Example 7, we can see that I_1 does not know either formula d nor e, therefore it is not the smallest partial possible world that knows formulas d and e. Instead, weighted f-interpretation I_3, I_4 and I_5 are the smallest partial possible world that knows formulas d and e.

The following theorem shows that the probability of a formula f with respect to a ground *scou* P-log program Π and a query Q can be computed by summing up the normalized weight of all weighted f-interpretations which are the smallest partial possible worlds that know and satisfies f.

Theorem 3 *Let Π be a ground scou P-log program and a query $Q = F|C$ be a query to Π. Let Φ_F be the collection of all the smallest partial possible worlds of T that know the set F of formulas. Then the probability of each formula $f \in F$ with respect to a ground scou P-log program Π and a query Q can be computed by following equation:*

$$P_{\Pi \cup \Pi_Q}(f) = \sum_{I=(A,w) \in \Phi_F \ and \ A \vdash f} \mu(I) \tag{3.8}$$

Theorem 3 shows that we can construct the set Φ_F first, then compute the probability of a formulas by using equation 3.8. To find Φ_F, we use following three steps: (1) we find the set Φ of smallest partial possible worlds of T with respect to Π and Q by calling function $FindPartialWorld(\Pi, Q)$, then (2) we compute the normalized weight of each weighted f-interpretation in the set Φ; finally (3) we further build the set Φ_F from the set Φ of weighted f-interpretations.

Now, we are ready to describe the main function which answers the query to a ground scou P-log program.

Algorithm 9: $Probability(\Pi, Q)$

Input: A ground scou P-log program Π and a query $Q = F|C$ to Π

Output: The set F' of formulas such that for each $f \in F'$ and $f' \in F$,

$$P_{\Pi \cup \Pi_Q}(f) \leq P_{\Pi \cup \Pi_Q}(f')$$

```
1  Φ := FindPartialWorld(Π, Q);
2  Normalize weights of weighted f-interpretations of Φ;
3  F' := F;
4  foreach weighted f-interpretation I ∈ Φ do
5      I_s := {I};
6      while I_s ≠ ∅ do
7          Select a weighted f-interpretation I' from I_s;
8          if I' knows F' then
9              CheckFormula(I', F');
10         else
11             I_s := I_s ∪ Branch(Π, I');
12         end
13         Remove I' from I_s;
14     end
15 end
16 return F'
```

The function $Probability(\Pi, Q)$ is shown in Algorithm 9. Steps of function $Probability(\Pi, Q)$ are explained as follows:

1. If Φ is an empty set, then the program is inconsistent, in such case, we simply return all the formulas in F with undefined probabilities.

2. We use following equation to normalize all the weights of weighted f-interpretations

$I = (A, w)$ in Φ:

$$\mu(I) = \frac{w}{\sum_{I=(A',w')\in\Phi} w'}$$

Then, we replace the weight of each weighted f-interpretation I by its normalized weight $\mu(I)$.

3. The set F' of formulas is initialized with the set F, we obtain the answer of the query by removing formulas f from F' if we know there is another formula $f' \in F$ such that the probability of f' being true is larger than the probability of f being true.

4. The **foreach** loop checks each weighted f-interpretation I of Φ to see whether it knows the set F. If not, then the inner loop, the **while** loop will extend I till every extension of I knows F.

5. Variable I_s stores a set of weighted f-interpretations to be further extended.

8. If I' knows F then I' is a smallest partial possible world that knows F.

9. Procedure $CheckFormula(I', F')$ updates the set, F', of formulas as well as the probabilities of formulas in F'. To optimize the computation of answering the query, for each formula f we use two variables f_{max} and f_{min} to indicate the range of its probability, $P_{\Pi\cup\Pi_Q}(f)$, of f with respect to the program Π and the query Q. The probability of f satisfies:

$$f_{min} \leq P_{\Pi\cup\Pi_Q}(f) \leq f_{max}$$

At the beginning, f_{min} is initialized to 0 and f_{max} is initialized to 1 for each formula f. Procedure $CheckFormula(I', F')$ is described as follows:

For each $f \in F'$, we do

- If $I' = (A, w)$ satisfies f, then $f_{min} = f_{min} + w$;
- Otherwise $f_{max} = f_{max} - w$;

Then, the procedure removes all the formulas f of F' if there exists a formula f' such that $f'_{min} > f_{max}$.

11. If I' does not know F, then we call function $Branch$ to extend I' and add the result of function $Branch(\Pi, I')$ to I_s.

16. After all the partial possible world are examined, for each formula f in F', its probability is known since we will have that $f_{min} = P_{\Pi \cup \Pi_Q}(f) = f_{max}$. Formulas with smaller probabilities will be all removed from F' and the remaining are the answer to the query Q.

Chapter 4

EXPERIMENTAL RESULT

We conducted our experiments on a machine built with a dual-core 1.60GHz processor and 2GB memory. The operating system is Ubuntu system with version 10.10. We name the P-log inference engine based on the algorithm of partial grounding and computing partial possible worlds as *plog*2.0 and call the one based on translating a P-log program to an A-Prolog problem, as *plog*1.0. We compared performance of *plog*2.0 against *plog*1.0 to see how problems from different domains affect the efficiency of those two P-log inference engines. We also compared the performance of system *plog*2.0 with system ACE [20], an exact inference engine for Bayesian networks to see advantages of using P-log as a language to solve probabilistic reasoning problems.

To test these systems, we have run many well known domains selected from other related literatures. Among all the domains we have tested, we select some of them to present here as they are the most representative examples that illustrate the advantage and disadvantage of both systems. We give brief introduction on each problems first, then the running results as well as our analysis of performance of the systems will be presented after.

4.1 Domain

In this section, we describe all the domains we will use for the purpose of analyzing the performance of both systems.

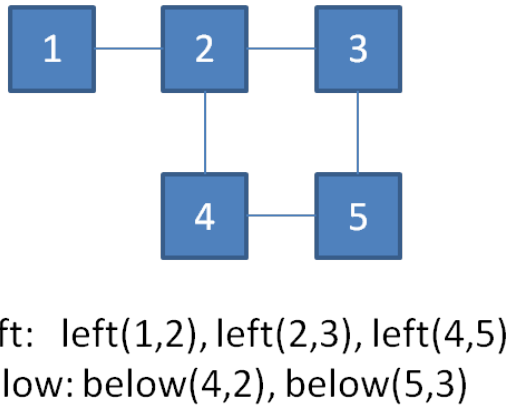

Figure 4.1: A grid map

4.1.1 Block Map Problem

The block map problem was introduced in [16]. The domain describes the random placement of blocks (obstacles) on the location of a map. The problem contains a particular grid map and a set of blocks. The grid map is described by objects *location* and relations $left(loc_1, loc_2)$ and $below(loc_1, loc_2)$, where loc_1 and loc_2 are locations. The relation $left(loc_1, loc_2)$ says that loc_1 and loc_2 are connected and loc_1 is at left of loc_2. Similarly, the relation $below(loc_1, loc_2)$ says that loc_1 and loc_2 are connected and loc_1 is at below of loc_2.

Figure 4.1 shows a simple grid map with 5 locations. Each block can be randomly placed on an unoccupied location and it will block all the paths that go through that location. The typical query of this domain is what is the probability of two locations are still connected after blocks are randomly placed on the map. In our experiment, problem instances *blockmap_m_n* indicates there are m locations and n blocks in the domain. The query used in our test is asking the probability of formula $connected(1, m)$, that is the probability of the first location (labeled by 1) and the last location (labeled by m) being connected.

GenoType	AA	AB	AO	BB	BO	OO
Blood Type	A	AB	A	B	B	O

Figure 4.2: ABO blood group system

4.1.2 Professor and Student Problem

This problem is first introduce in [21]. There are m professors and n students. Professors have two probabilistic attributes: fame(yes/no) and *funding_level*(high/low). The *funding_level* of a professor probabilistically depends on whether he/she is famous or not. Students have one probabilistic attribute: *success*(yes/no). The probability of success of a student is defined conditionally on the funding level of his advisor, a professor. According to the problem, students choose advisor i with higher probability if the professor i is well funded and with lower probability if the professor i is not. Let hi and lo ($hi > lo$) be two positive real numbers. A student chooses a professor with high funding level with probability $hi/(N_h \times hi + N_l \times lo)$ and chose a professor with low funding level with probability $lo/(N_h \times hi + N_l \times lo)$, where $N_h(N_l)$ are the total number of professors with high (low) funding levels. The query of this problem is the probability of a professor's funding level, given the success of one of his students.

4.1.3 Blood Type Problem

The blood type domain is a genetic model of the inheritance of a single gene that determines a person's blood type. Each person has two copies of the chromosome containing this gene, one inherited from his/her mother and one inherited from his/her father. In our model, we are using ABO blood group system in which we have 4 different blood types: *A, B, AB* and *O*. The blood types are controlled by its genotype and it is shown in Figure 4.2.

Our domain contains information of a family tree, see Figure 4.3. If Mary has geno-

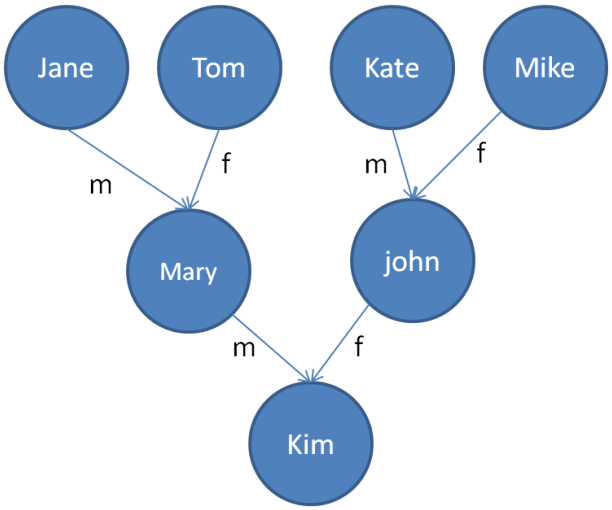

Figure 4.3: Family tree

type AO and John has genotype BO, then Kim can get an A or O with equal probability from his mother and get a B or O with equal probability from his father to make up his own genotype. Therefore, Kim can have one of 4 possible genotypes (AB, AO, OB, OO) with 25% for each possible outcome. If one's parent is not known in our knowledge base, we assume that the genes inherited from his parents agreed with a prior probability distribution. The queries to this domain are what is the probability that the blood type of Kim is O given Mary's blood type is A and John's blood type is B. Problem instances bt_m_n indicates that there are totally m individuals in the domain and n grandparents are known in the family shown in Figure 4.3. For example, an instance bt_7_1 means besides knowing that Kim's parents are Mary and John, we also know that Jane is the mother of Mary and we have no family information about the other three individuals at all.

4.1.4 The Grid Example

There is a grid of $m \times n$ nodes. Each node (i, j), where $1 \leq i \leq m$ and $1 \leq j \leq n$, passes information to $(i+1, j)$ and node $(i, j+1)$ if node (i, j) is not faulty. We assume each nodes in the grid can be faulty with probability of 0.1 and the query is what is

the probability of that node (m, n) can receive information from node $(1, 1)$.

4.1.5 Poker Problem

This example is used to demonstrate the performance of another P-log inference engine which is reported in [22]. The original problem is that from a deck of standard card, we randomly pick up 5 cards and ask what is the probability of having a single pair in this five cards. We use smaller domains in our instances. The problem *poker_m_n* indicate that from $4 \times m$ cards we pick up n cards.

4.2 Running Results

We recorded the running time of Smodels in *plog*1.0 and reported them under the column *Smodels* in our tables. The time used for retrieving probability information from output of Smodels and computing the probability of formulas are recorded under the column *Retrieval*. For the new system *plog*2.0, grounding time are presented under the column *Ground* and the time used for computing partial possible worlds and probability of formulas are shown under the column *Solv*. The total running time, shown under the column *Total* is measured by the *time command* in Linux system. In our tables, all running times are in seconds. We set our time out limit as 5 minutes. For instances that runs more than 5 minutes, we simply recorded them as *time out (T.O.)*. Since the number of possible worlds are one of the most dominant aspects of performance of both systems. We also show the number of possible worlds, under the column *NoA*, in tables where it makes big difference.

In Table 4.1, we present our experimental results of running *plog*2.0 and *plog*1.0 on *professor and students* domain and *blood type* domain. We can see from the table that *plog*2.0 performs much better than *plog*1.0 does on all the instances we have tested. For instances where the number of possible worlds computed by both systems are different, *plog*2.0 is more than 10 times faster than the old one. For instances

Table 4.1: Professor and student domain and blood type domain

	plog1.0				plog3.0			
Instances	NoA	Smodel	Retrieval	Total	NoA	Ground	Solv	Total
p3s2	384	0.020	0.13	0.195	64	0.02	0.01	0.038
p3s4	13824	0.412	7.72	8.549	64	0.02	0.01	0.038
p3s6	n/a	-	-	T.O.	64	0.02	0.01	0.038
p4s1	256	0.024	0.09	0.142	256	0.03	0.02	0.065
p4s2	2084	0.096	0.91	1.086	256	0.03	0.02	0.064
p5s1	16384	0.516	9.37	10.313	256	0.02	0.03	0.064
p6s1	1024	0.076	0.40	0.543	1024	0.04	0.12	0.174
p6s1	4096	0.376	1.92	2.461	4096	0.06	0.55	0.617
p6s1	16384	1.688	8.89	11.002	16384	0.08	2.47	2.567
bt7_1	220	3.732	53.58	59.712	75	0.02	0.01	0.100
bt7_2	560	2.008	29.03	33.900	375	0.02	0.05	0.092
bt7_3	1140	1.132	15.73	17.593	1875	0.03	0.37	0.407
bt7_4	9375	0.570	8.58	9.638	9375	0.03	1.97	2.027
bt8_4	84375	6.530	94.76	105.194	84375	0.04	14.91	15.093

with same number of possible worlds, such as $p4s1$ and $p5s1$, etc., the *solving* time in *plog*2.0 normally is larger than the computation time reported by the *Smodels*, but overall the performance of *plog*2.0 is still better on these instances. The table also shows that the procedure of retrieval probabilistic information from answer sets takes about 90% of the total running time of old P-log inference engine. In *plog*2.0, the grounding time normally is small and insignificant comparing to the *solving* time when the number of possible worlds is large.

The result is not surprising to us. In *professor and students* example, the query only relates to one student and adding new students into this domain will not change the

size of the ground program produced by the new grounding algorithm. Hence, adding new students into this domain has no effect on how many possible worlds will be computed by *plog*2.0. On the other hand, adding new students will increase the number of possible worlds in an exponential rate for old system *plog*1.0. Notice that on examples with the patten $pXs1$, the system *plog*2.0 produces the same ground program as the *plog*1.0 does and so is the number of possible worlds. For these instances, the performance of *plog*2.0 is still better than *plog*1.0. This is because that with large number of possible worlds, the procedure of extracting probability information from answer sets involves heavy string match operations which is much slower than the new approaches where the probability is computed along with the generation of possible worlds. The similar results are seen in *blood type* instances.

Table 4.2: The grid domain

	plog1.0			plog2.0		
Instances	Smodel	Retrieval	Total	Ground	Solv.	Total
grid_3_4	0.088	1.09	1.339	0.01	0.04	0.065
grid_2_7	0.380	5.35	6.126	0.01	0.06	0.082
grid_3_5	0.704	11.03	12.484	0.02	0.26	0.300
grid_4_4	1.540	23.4	26.499	0.02	0.56	0.590
grid_3_6	6.476	106.06	120.064	0.03	1.69	1.793
grid_4_5	-	-	T.O.	0.03	8.08	8.118

In Table 4.2, we show the experimental results on instances of the *grid* domain. The table shows that the *plog*2.0 performs significantly better than *plog*1.0. The total running time grows exponentially with respect to the number of total nodes in the domain for both engines but *plog*2.0 is about 50 times faster in average and appears that the bigger the domain is, the faster the *plog*2.0 is, comparing to *plog*1.0.

Since each node has one corresponding random selection rule in the ground program and each random selection rule has two possible outcome: the node is faulty or not,

the number of possible worlds of an instance $grid\_m\_n$ is $2^{m \times n}$. However, the set of all the least partial possible worlds that knows the formula (here, the formula is the literal $flow(m, n)$) is much smaller. For example, if the node $(1, 1)$ is faulty then we can derive that the literal $flow(m, n)$ is not true in any possible worlds which contains the literal $faulty(1, 1)$ regardless the status of other nodes. We found that the idea of finding the least partial possible worlds that knows formula is specially useful when dealing with diagnosis problems where often a single faulty component may explain the unexpected observations.

Table 4.3: Running results of block map domain

	plog1.0			plog2.0		
Instances	Smodels	Retrieval	Total	Ground	Solv.	Total
Bm_10_2	0.052	0.02	0.167	0.24	0.03	0.273
Bm_10_3	0.108	0.07	0.274	0.24	0.09	0.336
Bm_15_1	0.092	0.01	0.359	1.10	0.04	1.145
Bm_15_2	0.344	0.09	0.712	1.09	0.25	1.36
Bm_15_3	1.110	0.45	1.876	1.13	1.041	2.19
Bm_20_1	0.42	0.03	1.010	3.94	0.14	4.109
Bm_20_2	1.988	0.28	2.895	4.01	1.25	5.292
Bm_20_3	6.624	0.53	7.856	4.04	7.32	11.395

We present the results of running instances of *the block map domain* with *plog*2.0 and *plog*1.0 in Table 4.3. We can see that the new inference engine takes more total running time than those reported by *plog*1.0. Unlike previous examples, where grounding time does not play important role on the overall performance, the grounding time for *the block map* domain significantly affects the overall performance. As shown in Table 4.3, in the worst case, the grounding time could take over 90% of overall running time. The solving time is comparable to the running time of Smodels. Except the last instance, the solving time is slightly smaller than the running time reported by Smodels.

We believe that the large size of ground program is the main reason that causes grounding procedure slow. Both engine need to compute same number of possible worlds and no smaller partial possible worlds are available. Therefore, there is no advantage can be taken by our new grounding algorithm. This explains the reason that the solving time of our system is similar to those reported by Smodels.

Table 4.4: Running results of poker domain

	plog1.0			plog2.0		
Instances	Smodels	Retrieval	Total	Ground	Solv.	Total
Poker_3_3	0.020	0.07	0.254	0.05	0.02	0.084
Poker_3_4	0.052	0.21	0.367	0.09	0.07	0.173
Poker_3_5	0.112	0.54	0.791	0.14	0.15	0.304
Poker_5_3	0.068	0.39	0.583	0.12	0.15	0.288
Poker_5_4	0.408	2.78	3.540	0.22	1.00	1.239
Poker_5_5	1.672	12.60	14.904	0.33	4.14	4.514

The results of running instances of *poker* domain are shown in Table 4.4. Again, the new system *plog*2.0 performs better than the old system *plog*1.0. In average, the new system take about 1/3 of the running time as the old system does.

Similar to the *block map* domain, both engines essentially compute the same size of possible worlds for instances in this domain. Comparing to the block map example, the differences are:

1. the number of possible worlds is bigger, for example, the instance *poker*_5_5 has 15504 possible worlds.

2. each possible worlds is easier to compute than those in *block map* domain.

The large number of possible worlds makes the retrieval procedure become a bottle neck of the performance. It takes more than 80% of the total computing time in several instances. The easiness of computing each possible world make the difference between

the *solving* time of *plog*2.0 and *Smodels* running time of *plog*1.0 ignorable.

Table 4.5: Results for system ACE and *plog*2.0 on *Block Map* problem

-	ACE		plog2.0		
Instances	Total Inf	Comp.	Ground	Solv.	Total
Bm_5_1	1.260	2.417	0.04	0.00	0.062
Bm_5_2	1.484	2.875	0.04	0.00	0.064
Bm_5_3	1.672	3.801	0.04	0.00	0.071
Bm_10_1	3.633	15.964	0.23	0.02	0.248
Bm_10_2	4.451	22.192	0.24	0.03	0.273
Bm_10_3	4.851	32.482	0.24	0.09	0.336
Bm_15_1	8.010	39.954	1.10	0.04	1.145
Bm_15_2	9.785	64.036	1.09	0.25	1.360
Bm_15_3	14.919	132.448	1.13	1.04	2.190
Bm_20_1	14.463	105.894	3.94	0.14	4.109
Bm_20_2	19.194	188.761	4.01	1.25	5.292
Bm_20_3	48.361	740.686	4.04	7.32	11.395

In Table 4.5, we compared our inference engine with ACE on *block map* domain. Notice that ACE improves on-line inference efficiency by pre-compiling the Bayesian network. The compiling procedure normally is much slower than the on-line inference stage. This property allows their system perform better in the situation that many queries will be proposed after a Bayesian network is fixed. However, comparing with the inference time, *plog*2.0 inference engine still does better than ACE on problems of this domain. From the experiment, we believe that one of the advantages of using P-log is that we can define recursive relations easily, while in Bayesian network, it requires a lot of extra nodes to deal with the recursive definition. This results in a large Bayesian network, for instances, problem "blockmap_15_1" leads to a Bayesian network with more than 15,000 random variables while in P-log representation, the

ground problem contains a several hundred atoms and logic rules.

4.3 Summary

Overall, we believe that *plog*2.0 performs much better than system *plog*1.0 does. Different types of problems may have significant impacts on the performance of both system. When the query can be answered by looking at a portion of input P-log program, *plog*2.0 can perform much faster than *plog*1.0 does. Sometimes, it can reduce the computation time from minutes to less than a second. Also, many diagnosis domain benefits from the idea of computing partial possible world in *plog*2.0 instead of computing all complete possible worlds as *plog*1.0 does.

There are domains that the system *plog*1.0 could perform better than *plog*2.0. Programs of these domains may have the following properties:

- All rules have to be considered in order to answer the query. The most improvement in *plog*2.0 comes from the idea of reducing the size of ground P-log program. When this method is not helpful, the performance of *plog*2.0 will be affected.
- The improvement from computing partial possible world is ignorable. This forces the system *plog*2.0 to compute complete possible worlds. Because of using 5 value interpretation, comparing to the algorithm behind *Smodels* which is based on 3 value interpretation, there is some overhead when deriving truth value of literals in *plog*2.0. Therefore, when computing the same number of possible worlds, *plog*2.0 could take slightly longer than *Smodles* does.
- The size of grounded program is large. Because we are using head-to-body approach to ground the input program. Large program increases the overhead of this procedure.

Chapter 5

APPLICATIONS

The goal of this chapter is to describe how to represent probabilistic dynamic domains and how to use P-log to solve probabilistic reasoning problems in these domains. In this chapter, we introduce the probabilistic transition diagram that serves as a model of the probabilistic dynamic domain. We extend the classical system description language $\mathcal{B}$ [23] to system description language $\mathcal{NB}$ which can be viewed as a formal model of parts of natural language that are used for describing the behavior of probabilistic dynamic domains. We present an encoding that translates from system description language $\mathcal{NB}$, history of the domain and probability distribution over possible initial states to a P-log program where many reasoning tasks with respect to the probabilistic dynamic domains can be reduced to answering queries to P-log programs. In particular, we will give details of how to solve probabilistic diagnosis problems and probabilistic planning problems.

5.1 Probabilistic Dynamic Domain

The environment of an intelligent agent normally keeps changing. In order to react properly to its changing environment, we need to arm the agent with tools that can reason about changes. Reasoning about dynamic domains has been well studied for last several decades. In [31], a dynamic domain with no probabilistic information is modeled by the transition diagram whose nodes correspond to physically possible states of the domain and whose arcs are labeled by actions. A transition $\langle \sigma, \alpha, \sigma' \rangle$ of the diagram says that if action α is performed in state σ then the system may move to state σ'.

A probabilistic dynamic domain is a domain containing non-deterministic actions as well as probabilistic information associated with these actions. We model the probabilistic dynamic domain by an extended transition diagram, called probabilistic transition diagram, in which each arc is labeled by a tuple $\langle e, w\rangle$, where e is an action and w is a real number ($0 \leq w \leq 1$). A transition $\langle \sigma, \langle e, w\rangle, \sigma'\rangle$ says that if action e is performed in state σ then the system may move to state σ' with probability w. The w is called the weight of the arc. We may still use notation $\langle \sigma, \alpha, \sigma'\rangle$ to indicate a transition in probabilistic transition diagram when w is not important in our discussion.

5.2 Specifying Transition Diagram

Because the size of the domain could be very large, we need a concise and mathematically accurate description for the purpose of representing such domains and performing reasoning tasks upon it. System description languages are introduced as a tool to describe the behavior of the dynamic domains in a concise way. As our research is largely inherited from previous work of system description $\mathcal{B}$, in this section, we first have a review of system description language $\mathcal{B}$ and then extend to a new system description language $\mathcal{NB}$.

5.2.1 System Description Language $\mathcal{B}$

The signature Σ of system description language consists of two disjoint, non-empty sets of symbols: the set F of fluents and the set A of elementary actions.

A fluent literal is a fluent $f \in F$ or its negation $\neg f$. A set S of fluent literals is called complete if for any fluent $f \in \Sigma$ either f or $\neg f$ is in S.

System description language $\mathcal{B}$, an extension of system description language $\mathcal{A}$ [23], consists of following two propositions laws:

- Dynamic causal laws:

$$e \textbf{ cause } l \textbf{ if } p \tag{5.1}$$

- Static causal laws:

$$l \textbf{ if } p \tag{5.2}$$

where l is a fluent literal, p is a set of fluent literals and e is an elementary action.

The dynamic causal laws say that if the elementary action e is performed at the situation where p holds, then in the resulting situation, l holds. The static causal laws say that for any situations if p holds then l must be true too.

Let A be a system description of a physical system written in $\mathcal{B}$. A set S of fluent literals is closed under a set Z ($Z \subseteq A$) of static causal laws if for every static casual law in Z, the state S includes the head l of every static causal law, l **if** p, such that $p \subseteq S$. The set $Cn_Z(S)$ of consequences of S under Z is the smallest set of fluent literals that contains S and it is closed under Z. Let $E_A(e, \sigma)$ stand for the set of all fluent literals l for which there is a dynamic causal law: e **cause** l **if** p, in A such that $p \subseteq \sigma$.

According to [26], a transition system $T = (S, R)$ described by a system description A is defined as follows:

1. The set S of states of T is the collection of all complete and consistent sets of fluent literals of Σ closed under static causal laws of A.
2. The set R of transition is the collection of all triples $\langle \sigma, a, \sigma' \rangle$ such that $\sigma' = Cn_Z(E(a, \sigma) \cup (\sigma \cap \sigma'))$, where Z is the set of all static causal laws of A.

5.2.2 System Description Language $\mathcal{NB}$

Actions in language $\mathcal{B}$ are considered as deterministic actions. Non-deterministic actions, such as tossing a coin, may have several random results. To model such

actions and the probability distribution associated with these actions, we extend language $\mathcal{B}$ to $\mathcal{NB}$.

Before we introduce system description language $\mathcal{NB}$, we first introduce the definition of *mutually exclusive fluents*

Definition 43 *[Mutually Exclusive Fluents]*

Let T be a transition diagram, two fluents literals l_1 and l_2 are said to be **mutually exclusive** *if for any state σ of T, $\{l_1, l_2\} \not\subseteq \sigma$.*

We extend language $\mathcal{B}$ to language $\mathcal{NB}$ by adding **non-deterministic dynamic causal law** and **probability of possible outcome**:

1. The dynamic causal law in $\mathcal{NB}$ is a natural extension of 5.1:

$$(r)\ e \textbf{ cause } l_1|\ldots|l_n \textbf{ if } p \tag{5.3}$$

   where r is a term which serves as the name of this law, $l_1, \ldots, l_n$ are fluent literals which are pairwise mutually exclusive, p is a set of fluent literals and e is an elementary action. If $n = 1$, the law 5.3 is called deterministic dynamic causal law. Otherwise, it is called non-deterministic dynamic causal law.

   The above law says that if an action e is performed in a situation where p holds, then there are n possible outcomes. For each possible outcome, it will lead to a state σ' where a fluent literal $l \in \{l_1, \ldots, l_n\}$ must be true in σ'.

2. Statements expressing the probability of possible outcome of causal law 5.3 has the following form:

$$(r)\ l : v \textbf{ if } q \tag{5.4}$$

   where r is the name of law 5.3, $l \in \{l_1, \ldots, l_n\}$, v is a real number between 0 and 1, and q is a set of fluent literals which may differ from p in law 5.3.

   Since it is allowed to have multiple dynamic causal laws for an action in $\mathcal{NB}$ as long as their preconditions (literals p in law 5.1) are mutually exclusive, we need

to name the dynamic causal laws and statements that express the probability of possible outcome so that each statements of 5.4 can be associated with a unique dynamic causal law.

The statement 5.4 says that if e is performed in a state σ where q is true, then the probability of the system moving to a state containing l is v.

The semantics of language $\mathcal{NB}$ consists of two parts: the transitions defined by $\mathcal{NB}$ and the probability distribution over transitions.

We assume that for any given state σ there exists at most one non-deterministic dynamic causal law 5.3 for action e such that $p \subseteq \sigma$. This is a natural and not very limiting assumption.

To define the first part, we need the following definitions.

Definition 44 *[Possible Direct Effect]*

Let e be an action. A collection $E(e, \sigma)$ of fluent literals is called a **possible direct effect** *of e in σ, if for each fluent literal l', $l' \in E(e, \sigma)$ if and only if there exists a dynamic causal law 5.3 where $l' \in \{l_1, \ldots, l_n\}$, $p \subseteq \sigma$ and no other fluent literal in $\{l_1, \ldots, l_n\}$ belongs to $E(e, \sigma)$.*

We have the following assumptions on the transition system through the rest of this chapter:

1. In each step, only one elementary action is performed. Concurrent actions are not allowed. We believe our approach can be extended to systems with concurrent actions. However, in our dissertation, we use this assumption to simplify our definitions and proofs.

2. For each $E(e, \sigma)$, there exists exactly one state σ' such that $\sigma' = Cn_Z(E(e, \sigma) \cup (\sigma \cap \sigma'))$. We require such assumptions as to make sure that non-determinism only comes from non-deterministic dynamic causal laws. It is well known that the transition diagram may be non-deterministic even if there is no non-

deterministic dynamic causal laws. In [34], the authors have shown some necessary conditions for this assumption being true in a system.

3. For each pair of action e and state σ, the direct effects of deterministic laws and the direct effects of non-deterministic laws are mutually exclusive.

Now we are going to define the probability distribution associated with non-deterministic actions. Notice that if any of the previous assumptions does not hold, the following definition may define a probability distribution which is not intended from the intuitive meaning of laws 5.3 and 5.4.

Definition 45 *Let σ be a state of a transition diagram T and e be a non-deterministic action described by 5.3 where $p \subseteq \sigma$ and l be a fluent literal in $\{l_1, \ldots, l_n\}$ in 5.3. We define the probability $P_e(l, \sigma)$ (we omit the parameter T as we consider that T is fixed in the context) of fluent literal l being true after e is performed in σ as follows:*

1. *If there is a statement of form 5.4 for action e and l such that $q \subseteq \sigma$, then*

$$P_e(l, \sigma) = v$$

2. *Otherwise, let k be the sum of $P_e(l, \sigma)$ such that $P_e(l, \sigma)$ is defined in (1), and let m be the number of fluent literals in $\{l_1, \ldots, l_n\}$ whose probabilities are not defined by 5.4. We define*

$$P_e(l, \sigma) = \frac{1 - k}{m}$$

Definition 46 *[Probability of a Transition]*

Let $\langle \sigma, e, \sigma' \rangle$ be a transition in T. We define the probability of an agent being in state σ' after performing action e in state σ, $P(\sigma'|e, \sigma)$, as follows:

1. *If e is deterministic with respect to σ, i.e., there is no non-deterministic dynamic law for e such that its precondition $p \subseteq \sigma$, then*

$$P(\sigma'|e, \sigma)_{def} = 1$$

2. *otherwise, let* $l \in \{l_1, \ldots, l_n\} \cap \sigma'$, *we have*

$$P(\sigma'|e,\sigma)_{def} = P_e(l,\sigma)$$

Definition 47 *[Semantics of* $\mathcal{NB}$*]*

We say that a system description SD written in $\mathcal{NB}$ *represents a probabilistic transition diagram* $T = (S, R)$*, if,*

- *The set* S *of states of* T *is the collection of all complete and consistent sets of fluent literals of* Σ *closed under static causal laws of SD.*
- *A transition* $\langle \sigma, \langle e, w\rangle, \sigma'\rangle$ *belongs to* R *of* T*, if and only if there is a set* $E(e,\sigma)$*, such that* $\sigma' = Cn_{SD}(E(e,\sigma) \cup (\sigma \cap \sigma'))$ *and* $P(\sigma'|e,\sigma) = w$

A path $\langle \sigma_0, e_0, \sigma_1, \ldots, e_{n-1}, \sigma_n\rangle$ describes how the system transforms from σ_0 to σ_n as a result of executing a sequences of action: $e_0, \ldots, e_{n-1}$. We assume that our transition diagram has Markovian properties. This means that the likelihood of being in a next state only depends on the current state and the action executed in current state. It does not depend on how the system has reached to its current state. Based on this assumption, we define the probability of a path as follows:

Definition 48 *[Probability of Path]*

Let $p = \langle \sigma_0, e_0, \sigma_1, \ldots, e_{n-1}, \sigma_n\rangle$ *be a path of length* n *of* T*. We define the probability of the path as*

$$P(p) = \prod_{i=0}^{n-1} P(\sigma_{i+1}|e_i, \sigma_i)$$

Definition 49 *[Probabilistically Consistent Transition Diagram]*

Let $\Omega_{\sigma,n}$ *be the collection of all paths of length* n *starting from an initial state* σ*. We say that the transition diagram is probabilistically consistent if for every* n *and* σ*:*

$$\sum_{p \in \Omega_{\sigma,n}} P(p) = 1$$

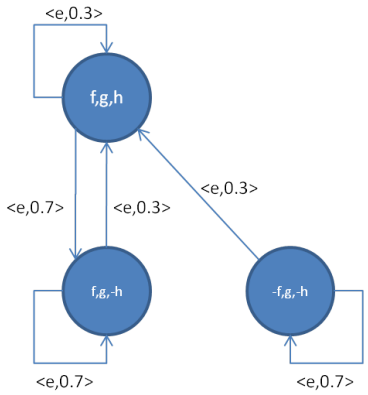

Figure 5.1: An extended transition diagram with a non-deterministic action

Given a transition diagram, to check whether it is probabilistically consistent or not is not difficult. For every state σ and action e, if the sum of weights of arcs which are labeled by e and leaving from σ is 1, then the transition diagram is consistent.

Example 8 *[Simple System Description]*

Consider a system description SD_1 consisting of the following statements:

$$(r)\ e\ \textbf{cause}\ \neg h|h$$

$$f\ \textbf{if} h$$

$$g\ \textbf{if} \neg h$$

$$g\ \textbf{if} f$$

$$(r)\ h : 0.3$$

The Figure 5.1 shows the extended transition diagram of Example 8. In Example 8, if action e is performed in state $\{g, \neg f, \neg h\}$, then the agent may move to the state $\{f, g, h\}$ as the result of h being true after action e being executed, or it will stay at $\{g, \neg f, \neg h\}$ as the result of $\neg h$ being true after action e being executed. Because of the mutually exclusive requirements on the direct effects of non-deterministic actions, we can see that different possible outcomes always lead to different states.

5.2.3 History of the System

To perform reasoning tasks such as diagnosis, the agent not only needs the knowledge of the transition system, but also needs the knowledge of which state it was in and what actions it has executed as well as observations after those actions. Those knowledge is called the history of the system.

The history of the system (up to step n) is described by two types of statements [23]. By statements of the form

$$e \ \textbf{happened at} \ i \tag{5.5}$$

The history records actions that happened at step i where $0 \leq i < n$. A fluent literal l we observed at step j where $0 \leq j \leq n$ is recorded as:

$$\textbf{observe} \ l \ \textbf{at} \ j \tag{5.6}$$

Definition 50 *[models of history]*

A path, $\langle \sigma_0, e_0, \sigma_1, \ldots, e_{n-1}, \sigma_n \rangle$, of transition diagram is a **model** *of a history H if*

- *For every statement: e* **happened at** *i in H, $e = e_i$;*
- *For every statement:* **observe** *l* **at** *j in H, $l \in \sigma_j$;*

5.2.4 Beliefs about Initial States

To reason about the transition diagram, the agent may need knowledge of the initial state. When the agent does not observe all the truth values of fluents at step 0, its knowledge about its initial state may be incomplete.

Definition 51 *[Partial State]*

A set s of fluent literals is called a partial state of transition diagram T if there is a state σ of T such that $s \subseteq \sigma$.

Let H be a history and let $s = \{l : \textbf{observe } l \textbf{ at } 0 \in H\}$ be a partial state. We say that the partial state s is then initial partial state defined by H. By $compl(s)$, we denote the set S of completed states which are compatible with s, i.e., $\sigma \in S$ if and only if $s \subseteq \sigma$. By P_s we denote the probability distribution over all the subsets of $compl(s)$. The representation of P_s may have many forms. It can be represented by a table which lists the probability of each state σ of $compl(s)$, or in case fluent literals which do not belong to s are independent, as we will show in section 5.5, we can use random selection rules and *pr* atoms to define P_s.

Definition 52 *Let P_s be a probability distribution over all the subsets of $compl(s)$. The probability $P_s(p)$ of a path p starting at a state σ_0 compatible with s is*

$$P_s(p) = P_s(\sigma_0) \times P(p) \tag{5.7}$$

5.2.5 Domain Description

As we have pointed out, many reasoning tasks require system descriptions, history of the system and probability distribution over initial states. We group these knowledge together as we refer to domain description.

Definition 53 *[Domain Description]*

A **domain description** $\mathcal{K}$ *is a triple $\langle SD, H, P_s \rangle$, where SD is a system description written in language $\mathcal{NB}$, H is the history of SD and P_s is the probability distribution over possible initial states.*

Just as in Bayesian network, the probability of some random variables may change given some new information of the domain. The probability of paths of a transition diagram may change given extra information from history and probability distribution of initial state. We give the definition of conditional probability of paths as follows:

Definition 54 *[Conditional Probability of a Path]*

Let $\mathcal{K}$ be a domain description and s be the initial partial state defined by H of $\mathcal{K}$. Let $PATH$ be the set of models of H. The probability of a path conditioned on history H is defined as follows:

$$P_{\mathcal{K}}(p|H) = \frac{P_s(p)}{\sum_{p \in PATH} (P_s(p))} \tag{5.8}$$

Given a domain description $\mathcal{K}$, an intelligent agent can perform many interesting and important reasoning tasks based on it. Many studies (such as [31] and [32]) has shown that we can encode system description languages to certain logic programs. The reasoning task then can be reduced to computation of logic programs. We adopt the same method as we encode the domain description $\mathcal{K}$ to P-log program $\Pi(\mathcal{K})$. With proper queries, the answer of these queries to $\Pi(\mathcal{K})$ will contain the solution of reasoning tasks.

5.3 Encoding $\mathcal{K}$ with P-log

In this section, we give an algorithm which maps a domain description $\mathcal{K}$ to rules of P-log program $\Pi(\mathcal{K})$ such that there is one-to-one correspondence of a model, p, of H of $\mathcal{K}$ to a possible world, W, of $\Pi(\mathcal{K})$, and $\mu(W) = P_{\mathcal{K}}(p|H)$.

Our translation starts with some declarations. For a system with a history up to step n, we have the following declaration:

$$time = \{0..n\}$$

$$\#domain\ time(T)$$

We have declaration in $\Pi(\mathcal{K})$:

$$fluent = \{f_1, \ldots, f_m\}$$

where $\{f_1, \ldots, f_m\}$ is the set of all fluents in the language.

We introduce a new boolean predicate h:

$$h : fluent, time \rightarrow boolean$$

As each possible world of program $\Pi(\mathcal{K})$ represents a path $p = \langle \sigma_0, e_0, \sigma_1, \ldots, e_{n-1}, \sigma_n \rangle$, a literal $h(f,t)$ belonging to a possible world W means that fluent f is true in state σ_t of p. We use $\neg h(f,t)$ to represent that $\neg f$ is true in state σ_t. We use $h(l,t)$ stands for $h(f,t)$ if l is a fluent f or stands for $\neg h(f,t)$ if l is $\neg f$. If $p = \{l_1, \ldots, l_n\}$ is a set of fluent literals, $h(p,t)$ is a short hand of $h(l_0,t), \ldots, h(l_n,t)$.

Inertia Axiom: For each fluent, their truth values stay unchanged if no laws make them change. For each inertial fluent, we have the following rules:

$$h(f, T+1) \leftarrow h(f,T), \textbf{not } \neg h(f, T+1)$$

$$\neg h(f, T+1) \leftarrow \neg h(f,T), \textbf{not } h(f, T+1)$$

The translation of dynamic causal laws and static causal laws to rules in ASP program can be found in [31]. The dynamic causal laws are translated to

$$h(l, T+1) \leftarrow occurs(e,T), h(p,T)$$

The static causal laws are translated to

$$h(l,T) \leftarrow h(p,T)$$

The non-deterministic dynamic causal laws, (r) e **cause** $l_1 | \ldots | l_n$ **if** p, are translated to the following statements:

- We declare a sort $outcome_r$ consisting of n $(n > 1)$ symbols where each symbol represents a possible outcome of action e.

$$outcome_r = \{l_1, \ldots, l_n\}$$

- We introduce a new random attribute:

$$result_r : time \rightarrow outcome_r$$

  This random attribute is defined by a random selection rule to enumerate all possible outcomes of a non-deterministic dynamic causal law (r).

- We use a random selection rule to say that if e is performed in a state which satisfies p. Then there are n possible outcomes.

$$[r]\ random(result_r(T+1)) \leftarrow occurs(e,T), h(p,T)$$

- for the i'th possible outcome l_i, we have the following rules:

$$h(l_i,T) \leftarrow result_r(T) = l_i$$

The statements for probabilities of possible outcome, $(r)\ l : v$ **if** q, is translated to pr rules:

$$pr_r(result_e(T+1) = l|_c h(q,T)) = v$$

The history of actions e performed at t is translated to:

$$occurs(e,t).$$

The history of observations l at step t where $t > 0$ is translated to:

$$obs(h(l,t)).$$

For observations of l at step 0, we add

$$h(l,0).$$

into the P-log program.

In case, the agent's knowledge of initial state is incomplete, we add the following program to represent the probability distribution over initial states with P-log. Let

s be the partial state defined by a history H and let $n = |compl(s)|$. We introduce a sort ini_state as follows:

$$ini\_state = \{c_1, \ldots, c_n\}$$

where each symbol c_i is the name of a complete state in $compl(s)$.

We declare a random attribute $at\_initial\_state$ as follows:

$$at\_initial\_state : ini\_state$$

Then we have a random selection rule:

$$[ini]\ random(at\_initial\_state)$$

Let $\sigma_i \in compl(s)$. For each fluent literal $l \in \sigma_i$, we add the following rules to P-log program:

$$h(l, 0) \leftarrow at\_initial\_state = c_i$$

Suppose $P_s(\{\sigma_i\}) = v$ where $0 \leq v \leq 1$. We can have the following pr atoms

to describe the probability distribution:

$$[ini]\ pr(at\_initial\_state = c_i) = v$$

This concludes our translation from domain description to a P-log program. The next theorem shows that the obtained P-log program can be used for computing the probability of paths conditioned on histories.

Theorem 4 *Let $\Pi(\mathcal{K})$ be the P-log program obtained from a domain description $\mathcal{K}$. Let H be a history of $\mathcal{K}$ which records all the actions executed up to step $n-1$. A path $p = \langle \sigma_0, e_0, \sigma_1, \ldots, e_{n-1}, \sigma_n \rangle$ is a model of H if and only if there is a possible world W of $\Pi(\mathcal{K})$ such that for any $0 \leq t \leq n$,*

1. $e_t = \{e : occurs(e, t) \in W\}$

2. $h(l,t) \in W$ *iff* $l \in \sigma_t$

and $\mu(W) = P_{\mathcal{K}}(p|H)$.

Next, we will demonstrate how to use this encoding to evaluate the probability of some fluent being true given the history of the system. Evaluating the probability, $P_{\mathcal{K}}(l,j)$, of some fluents l being true at some step j is common and important reasoning tasks, such as computing the probability of success of a given plan.

5.4 Evaluating $P_{\mathcal{K}}(l,j)$

In this section, we discuss which query can be used for computing the probability, $P_{\mathcal{K}}(l,j)$, of a given fluent literal l being true in a certain step j of H with respect to the encoded program $\Pi(\mathcal{K})$. We first give a formal definition of this probability:

Definition 55 *[$P_{\mathcal{K}}(l,j)$]*

*Let $\mathcal{K}$ be a domain description and l be a fluent literal and j be an integer between 0 and n where n is the last step of H. Let $H' = H \cup \{$ **observe** l **at** $j\}$, the probability $P_{\mathcal{K}}(l,j)$ of fluent literal l being true at a given step j is defined as follows:*

$$P_{\mathcal{K}}(l,j) = \sum_{p \text{ is a model of } H'} P_{\mathcal{K}}(p|H') \tag{5.9}$$

Proposition 6 *Let $\Pi(\mathcal{K})$ be the P-log program obtained from a domain description $\mathcal{K}$. Let l be fluent literal of $\mathcal{K}$ and let j ($0 \leq j \leq n$) be an integer.*

$$P_{\Pi(\mathcal{K})}(h(l,j)) = P_{\mathcal{K}}(l,j)$$

Because the query $\{h(l,j)\}|\emptyset$ to P-log program $\Pi(\mathcal{K})$ returns $P_{\Pi(\mathcal{K})}(h(l,j))$, our system is capable of computing the probability $P_{\mathcal{K}}(l,j)$.

In the next two sections, we will illustrate how to use similar method to solve another two important tasks (the diagnosis problem and the planning problem) with P-log.

5.5 Probabilistic Diagnosis Problem

In this section, we will discuss how to solve probabilistic diagnosis problem with P-log. Our work on probabilistic diagnosis has roots in [24] in which the author defines diagnosis problems in terms of first order logic. In our approach, we introduce probabilistic information into the domain. With probability information considered, the quality of diagnostic results could be enhanced.

Definition 56 *[probabilistic diagnosis problem]*

Let T be a physical system which is modeled by system description language $\mathcal{NB}$. A probabilistic diagnosis problem $\mathcal{D}$ is a tuple $\langle \mathcal{K}, C \rangle$, where $\mathcal{K}$ is a domain description of T and C is a set of components that may be faulty in T.

The states of the transition diagram defined by SD of $\mathcal{K}$ consisting of fluents $faulty(c)$ for each $c \in C$ as well as other fluents used for describing the behavior of the system. We assume actions in SD are deterministic. While the method described here can be easily extended to non-deterministic domains, for the purpose of simplicity, we represent our methods based on this assumption. We further assume that no action can damage a component nor fix a broken one. This is different from [33] in which exogenous actions are the cause of broken components. Last, we assume that the probability of a component being faulty is independent from the status of other components.

We divide the recorded history H of $\mathcal{K}$ to three disjoint parts:

- O_{n-1}: A collection of statement of form: **observe** l **at** j, where $0 \leq j \leq n-1$. In diagnosis problem, we assume that all the fluents are known in the initial state except fluents of form $faulty(c)$. By s_0, we denote the set of fluent literals which are known being true in the initial state.
- O_e: A collection of statement of form: e **happened at** i. We assume that all the actions are recorded and the history is up to step n.

- O_n: A collection of statement of form: **observe** l **at** n.

Because the agent has incomplete knowledge of its initial state, we need to specify the probability distribution over all possible initial sates.

Let $D \subseteq C$ be the set of faulty components. We define a set W_D of fluent literals as follows:

$$W_D = \{faulty(c) : c \in D\} \cup \{\neg faulty(c) : c \in C_T \backslash D\} \tag{5.10}$$

Notice that for each D there is at most one state σ such that $W_D \subseteq \sigma$ and σ satisfies s_0. If such state exists, we denote it by s_D. The state s_D is the initial state that the agent believes it might be in given the assumption that all components in D are faulty and the rest are not faulty. Let $P(c)$ denote the probability of a component c being faulty. Let s_0 be the initial partial state defined by H. Because we assumed that components are independent of each other, for each possible initial state s_D, the probability, $P_s(\{s_D\})$, can be defined as follows:

$$P_s(\{s_D\}) = \prod_{c \in D} P(c) \times \prod_{c \notin D} (1 - P(c))$$

In general, the probability $P(c)$ can be obtained from many resources. A typical one will be some statistical results obtained by the producer. For example, if for over one thousand bulbs produced by a company A, about 3 bulbs are faulty, then the probability of this type of bulb being faulty can be modeled as 0.003.

Our definitions of symptoms and candidate diagnosis are very similar to those in [33]. The main difference between this work with [33] is that, in [33], the candidate diagnosis is a set of unrecorded actions (called exogenous actions) which happened and caused unexpected behaviors, while in our work, it is a set of components that the agent believes to be faulty at the beginning.

Definition 57 *[Configuration][33]*

Let $\Gamma_{n-1} = O_{n-1} \cup O_e$. *A pair* (Γ_{n-1}, O_n) *is called a* **configuration**.

Definition 58 *[Symptom]*

Let $D_\emptyset = \emptyset$. A symptom *is a configuration (Γ_{n-1}, O_n) such that there exists a model p of Γ_{n-1} which starts from the initial state $s_\emptyset$ but has no model of $\Gamma_{n-1} \cup O_n$ which starts from $s_\emptyset$.*

Now we give our definition of candidate diagnosis.

Definition 59 *[Candidate diagnosis]*

We say that a non-empty set D of components is a **candidate diagnosis** *of symptom $S = (\Gamma_{n-1}, O_n)$ w.r.t. a transition diagram T if there is a model of $\Gamma_{n-1} \cup O_n$ in T which starts from the initial state s_D.*

If D is a candidate diagnosis of symptom $S = (\Gamma_{n-1}, O_n)$, the set, W_D, of fluent literals, defined as 5.10, is called a possible world that is compatible with the symptom S.

Let P_S be a probability distribution defined as follows:

$$P_S(\{W_D\}) = P_{\mathcal{K}}(p|H)$$

where path p starts at the initial state s_D.

By $P_S(D)$, where D is a candidate diagnosis of S, we denote

$$P_S(\{W_D : \forall c \in D, faulty(c) \in W_D, W_D \in \Omega\})$$

Definition 60 *[Best Diagnosis]*

A candidate diagnosis D is a **best diagnosis** *w.r.t. a probability distribution P_S, if for every candidate diagnosis D', we have $P_S(D') \leq P_S(D)$.*

Given a diagnosis problem $\mathcal{D}$, the task of solving $\mathcal{D}$ is to find the set of best diagnosis of $\mathcal{D}$. To solve such problem, we use the translation method illustrated in previous section to obtain P-log program $\Pi(\mathcal{K})$. We introduce a new type of query and show that the answer of this query to $\Pi(\mathcal{K})$ is the solution to the probabilistic diagnosis

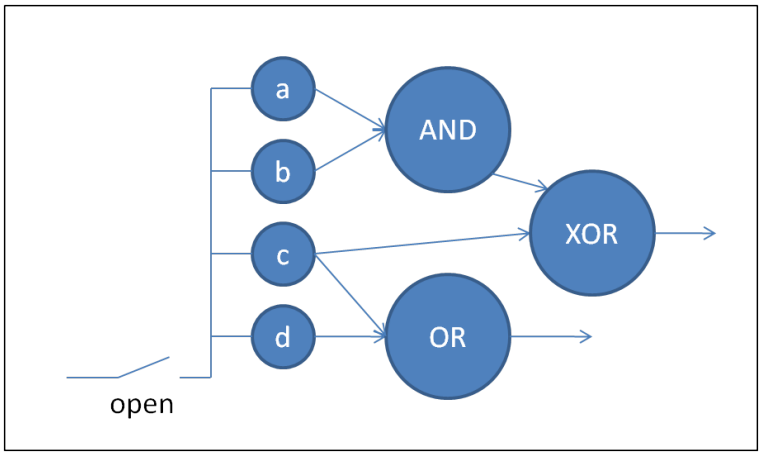

Figure 5.2: A digit circuit.

problem. We use the following example to show how to represent a physical domain of diagnosis problem with language $\mathcal{NB}$ and to explain how the definition of best diagnosis can be used for checking and repairing components efficiently.

Example 9 *[Digital Circuit]*

Consider a system T consisting of a digital circuit DC from Figure 5.2. Components a, b, c and d are digital devices. Those components output signal 1 if they are activated and work normally. Otherwise, they output 0. All components will be activated by an action *open*. The node "AND" of the diagram performs AND operation on outputs of components a and b. The output of this node and the output of component c are the inputs for node "XOR" for XOR operation. Similarly for node "OR" which takes the output of c and d as its input and outputs signal based on logical OR operation. Initially, no component is activated. After performing action *open* which activates all the components, the agent expects to see an output of 0 from node "XOR". Suppose however that the received signal is 1.

The system description, SD, of T given below defines the dynamic system of the above example. Variable C ranges over components, a, b, c and d of the system.

The only action *open* in our example will cause the system to be activated:

open **causes** *activated*

A component C will output signal 1 if the system is activated and C works properly, otherwise, it will output 0:

$output(C, 1)$ **if** $activated, \neg faulty(C)$

$output(C, 0)$ **if** $\neg activated$

$output(C, 0)$ **if** $faulty(C)$

P-log, as an extension of Answer Set Prolog, is good at representing knowledge in a general form. For the purpose of saving space, instead of using general axioms for digital gate operations, we represent the functionality of these gates with domain related input and output.

The following three static laws describe how an AND node works:

$output(and\_node, 1)$ **if** $output(a, 1), output(b, 1)$

$output(and\_node, 0)$ **if** $output(a, 0)$

$output(and\_node, 0)$ **if** $output(b, 0)$

The following two static laws describe how an XOR node works. The inputs of the XOR node are the output of the AND node and the output of the component c.

$output(xor\_node, 1)$ **if** $output(and\_node, S1), output(c, S2), S1 <> S2$

$output(xor\_node, 0)$ **if** $output(xor\_node, S), output(c, S)$

The following three static laws describe how an OR node works. The inputs of the OR node are the output of components c and d:

$output(or\_node, 1)$ **if** $output(c, 1)$

$output(or\_node, 1)$ **if** $output(d, 1)$

$output(or\_node, 0)$ **if** $output(c, 0), output(d, 0)$

The history of initial states is described as follows:

observe $\neg activated$ **at** 0

observe $output(C, 0)$ **at** 0

observe $output(and\_node, 0)$ **at** 0

observe $output(xor\_node, 0)$ **at** 0

observe $output(or\_node, 0)$ **at** 0

The rest of the history of the system is described by the following two statements:

1. the action is performed at step 0:

   $open$ **happened at** 0

2. the observation O_1 consists of the statement:

   observe $output(xor\_node, 1)$ **at** 1

The probability of each components being faulty is written as follows:

$$P(a) = P(b) = 0.4$$

$$P(c) = P(d) = 0.5$$

This completes the description of domain description $\mathcal{K}$. Since there is no path that starts from the initial state $s_{O_0 \cup D_\emptyset}$ and satisfies Γ_0 and O_1 the configuration $S_1 = (\Gamma_0, O_1)$ is a symptom of the system T.

There are 8 candidate diagnosis of symptom S_1 and the probability distribution P_{S_1} is described as follows:

1. $D_1 = \{a\}$, $P_{S_1}(W_{D_1}) = 0.12$.
2. $D_2 = \{b\}$, $P_{S_1}(W_{D_2}) = 0.12$.
3. $D_3 = \{a, b\}$, $P_{S_1}(W_{D_3}) = 0.08$.

4. $D_4 = \{c\}$, $P_{S_1}(W_{D_4}) = 0.18$.
5. $D_5 = \{a, d\}$, $P_{S_1}(W_{D_5}) = 0.12$.
6. $D_6 = \{b, d\}$, $P_{S_1}(W_{D_6}) = 0.12$.
7. $D_7 = \{a, b, d\}$, $P_{S_1}(W_{D_7}) = 0.08$.
8. $D_8 = \{c, d\}$, $P_{S_1}(W_{D_8}) = 0.18$.

For each candidate diagnosis D of S, we compute $P_{S_1}(D)$ as follows:

1. $P_{S_1}(D_1) = P_{S_1}(W_{D_1}) + P_{S_1}(W_{D_3}) + P_{S_1}(W_{D_5}) + P_{S_1}(W_{D_7}) = 0.4$.
2. $P_{S_1}(D_2) = P_{S_1}(W_{D_2}) + P_{S_1}(W_{D_3}) + P_{S_1}(W_{D_6}) + P_{S_1}(W_{D_7}) = 0.4$.
3. $P_{S_1}(D_3) = P_{S_1}(W_{D_3}) + P_{S_1}(W_{D_7}) = 0.16$.
4. $P_{S_1}(D_4) = P_{S_1}(W_{D_4}) + P_{S_1}(W_{D_8}) = 0.36$.
5. $P_{S_1}(D_5) = P_{S_1}(W_{D_5}) + P_{S_1}(W_{D_7}) = 0.20$.
6. $P_{S_1}(D_6) = P_{S_1}(W_{D_6}) + P_{S_1}(W_{D_7}) = 0.20$.
7. $P_{S_1}(D_7) = P_{S_1}(W_{D_7}) = 0.08$.
8. $P_{S_1}(D_8) = P_{S_1}(W_{D_8}) = 0.18$.

From computation, we can see that D_1 and D_2 are the best diagnosis of symptom S_1 according to our definition. This means components a and b are the ones that are most likely to be faulty given the symptom in this example. This information helps agent check and repair components a or b first, instead of checking c or d first. Furthermore, component d is unrelated to the symptom S_1 at all and the best diagnosis in our definition will guarantee containing no unrelated components.

5.5.1 Solving Diagnostic Problem

To obtain the P-log program $\Pi(\mathcal{K})$, we need to translate SD, H and P_s to rules of P-log program. The translation of SD and H is same as those described in section

1.2. The translation of P_s is different because of the assumption of in-dependency among components, which gives us a concise representation of P_s:

We declare a boolean random attribute $faulty$ as follows:

$$faulty : component \rightarrow boolean$$

For each component $c \in C$ we have

$$[c]\ random(faulty(c))$$

$$h(faulty(c), 0) \leftarrow faulty(c)$$

$$\neg h(faulty(c), 0) \leftarrow \neg faulty(c)$$

and

$$[c]\ pr(faulty(c)) = v$$

where $v = P(c)$.

The complete P-log program Π_d of Example 9 can be found in [47].

Proposition 7 *[Soundness]*
Let $\mathcal{D} = \langle \mathcal{K}, C \rangle$ be a probabilistic diagnosis problem. Let W be a possible world of $\Pi(\mathcal{K})$ and let

$$C(W) = \{c : faulty(c) = true \in W\}$$

be a collection of components that are faulty in W. Then $C(W)$ is a candidate diagnosis of symptom S with respect to $\mathcal{D}$.

Proposition 8 *[Completeness]*
For each candidate diagnosis D of symptom S of a probabilistic diagnosis problem $\mathcal{D} = \langle \mathcal{K}, C \rangle$ there exists a possible world W of $\Pi(\mathcal{K})$ such that $D = C(W)$.

By $Conj(L)$ where $L = \{l_1, \ldots, l_n\}$ is a set of literals, we denote the propositional formula: $l_1 \wedge \cdots \wedge l_n$.

Proposition 9 *[Probability Equivalence]*
Let $L = \{l_1, \ldots, l_n\}$ be a set of literals. For each possible world W of $\Pi(\mathcal{K})$, we have

$$P_S(C(W)) = P_{\Pi(\mathcal{K})}(Conj(\{faulty(c_1) = true, \ldots, faulty(c_n) = true\} \cap W))$$

Let $\Omega(\Pi(\mathcal{K}))$ be the collection of all possible worlds of $\Pi(\mathcal{K})$. We can define a set of formulas F as follows:

$$F = \{Conj(\{faulty(c_1) = true, \ldots, faulty(c_n) = true\} \cap W) : W \in \Omega(\Pi(\mathcal{K}))\}$$

Then for each formula f in the answer F' to query $Q = F|\emptyset$ w.r.t. the P-log program $\Pi(\mathcal{K})$, the set of components that forms the formula f is the best diagnosis of symptom S.

Notice that F is not available until all the possible worlds are computed. Therefore, we need first translate $\Pi(\mathcal{K})$ to an ASP program $\tau(\Pi(\mathcal{K}))$ (exclude all *pr* atoms) and use answer set solver (e.g. smodels) to compute all the answer sets of $\tau(\Pi(\mathcal{K}))$. Then we can construct the set F of formulas. Let Π be a P-log program obtained from $\Pi(\mathcal{K})$ by excluding all the *obs* and *do* statements in $\Pi(\mathcal{K})$. Let C be the collection of *obs* and *do* statements in $\Pi(\mathcal{K})$. By proposing the query $Q = F|C$ to program Π, we can solve the finding best diagnosis problem.

5.5.2 Extending P-log system

In this section, we will first extend the syntax of the query we introduced in definition 8 and then give a new algorithm designed to improve the performance of finding best diagnosis with P-log.

The procedure described in the last paragraph of section 5.5.1 has two major problems and both are related to the efficiency of solving diagnosis problems. First, we use two similar inference engines (answer set solver and *plog* inference engine) for completing the tasks. If we are able to merge these two computations to one, then surely the

performance can be improved. Second, the number of possible worlds can be very large and so is the size of the set F which consists of all possible faults. As we will point out later in this section that it is not necessary to compute the whole set F, a small subset of F can be sufficient to use in order to find best diagnosis.

In *plog*2.0, the statement $[l_1, \ldots, l_n]$ defines a set of formulas w.r.t. a P-log program Π and a query Q as follows:

$$[l_1, \ldots, l_n] = \{Conj(\{l_1, \ldots, l_n\} \cap W) : W \in \Omega(\Pi \cup \Pi_Q)\}$$

Let Π_1 be the P-log program obtained from the system description $\mathcal{K}$ of Example 9 without *do* and *obs* statements. Let $\Pi_Q = \Pi(\mathcal{K}) \backslash \Pi_1$, i.e., the collection of *do* and *obs* statements occurring in $\Pi(\mathcal{K})$. The notation $[faulty(a), faulty(b), faulty(c), faulty(d)]$ with respect to the program Π_1 and Π_Q defines the following set of formulas:

$\{faulty(a), faulty(b), faulty(c), faulty(a) \wedge faulty(b), faulty(a) \wedge faulty(d),$
$faulty(b) \wedge faulty(d), faulty(c) \wedge faulty(d), faulty(a) \wedge faulty(b) \wedge faulty(d)\}$.

The answer to the following query

$$[faulty(a), faulty(b), faulty(c), faulty(d)]|do(occurs(open, 0)), obs(h(output(xor\_node, 1), 1))$$

w.r.t. the program Π_1 returns the best diagnoses we are looking for.

Let Π be a ground *scou* P-log program and the query $Q = [l_1, \ldots, l_n]|C$. The *plog2.0* system solves above query in two steps:

1. Find a set of formulas F ($F \subseteq [l_1, \ldots, l_n]$) such that if a formula f belongs to the answer of Q, then $f \in F$. Notice that the size of the set $[l_1, \ldots, l_n]$ can be very large when n is large. However, for two sets of literal L_1 and L_2, if $L_1 \subseteq L_2$, then $P_{\Pi \cup \Pi_Q}(Conj(L_1)) \geq P_{\Pi \cup \Pi_Q}(Conj(L_2))$. This property makes efficient computation of finding F possible.

2. Find the most likely formulas from F w.r.t. the query Q and the program Π.

Let $f = l_1 \wedge \cdots \wedge l_n$ be a formula. By $[f]$, we denote the collection of literals occurring in f. We say that two formulas, $f_1 = Conj(\{l_1, \ldots, l_n\})$ and $f_2 = Conj(\{k_1, \ldots, k_m\})$, are incompatible if $[f_1] \not\subseteq [f_2]$ and $[f_2] \not\subseteq [f_1]$.

The algorithm shown in Algorithm 10 returns the set of formulas F such that for each $f_1 \in F$ and $f_2 \in F$, f_1 and f_2 are incompatible and query $Q' = F|C$ and Q are equivalent. We explain steps of the algorithm as follows:

Algorithm 10: $ReduceFormula(\Pi, Q)$

Input: A ground scou P-log program Π and a query $Q = [l_1, \ldots, l_n]|C$ to Π

Output: The set $F \subseteq [l_1, \ldots, l_n]$ of formulas such that for each $f_1 \in F$ and $f_2 \in F$, f_1 and f_2 are incompatible, and the query $F|C$ and Q are equivalent w.r.t. Π.

```
1  Φ := FindPartialWorld(Π, Q);
2  F := ∅;
3  while Φ ≠ ∅ do
4      Let I ∈ Φ such that rank(I) is the smallest;
5      if I knows {l_1, ..., l_n} then
6          Let L = {l : I(l) = T and l ∈ {l_1, ..., l_n}} and f = Conj(L);
7          if for all formulas f' ∈ F, f and f' are incompatible then
8              Add f to F;
9              Φ = Prune(Φ, f)
10         end
11     else
12         Φ := Φ ∪ Branch(Π, I);
13     end
14     Remove I from Φ;
15 end
16 return F
```

1. The function $FindPartialWorld$ is described in Algorithm 8. It returns a set, Φ, of smallest partial possible world of $\Pi \cup \Pi_Q$. If Φ is an empty set, then the program is inconsistent. In such case, we simply return an empty set.

2. The variable F stores all the formulas we are interested in. Initially, this set is an empty set.

3. The stop condition for the **while** loop is that when Φ becomes an empty set. Since the total number of weighted f-interpretation is finite and each weighted f-interpretation will only be checked at most once and then removed from the set Φ, therefore the set Φ will become an empty set eventually. Hence the loop will stop.

4. In our implementation, each weighted f-interpretation $I = (A, w)$ has a rank $rank(I)$ defined as follows:

$$rank(I) = |\{l : A(l) = T \ where \ l \in \{l_1, \ldots, l_n\}\}|$$

   where $||$ is the cardinality of the set. The algorithm always selects a possible worlds with the lowest rank to further investigate.

5. If I knows all the literals in $\{l_1, \ldots, l_n\}$, i.e., $A(l) \in \{T, F\}$ for each literal $l \in \{l_1, \ldots, l_n\}$, then the algorithm may find a formula we are interested in.

6. Let the variable f be the formula induced from the weighted f-interpretation I.

7. We check whether this formula f is incompatible with formulas in F, if there exists a formula f' which is compatible with f, then $[f'] \subseteq [f]$. This is because we always choose the weighted f-interpretation with smallest rank to check first. If there exists a formula f' such that $[f'] \subseteq [f]$ then $P_{\Pi \cup \Pi_Q}(f') > P_{\Pi \cup \Pi_Q}(f)$, hence we can discard the formula f.

8. If no such f' exists, we then add f to the set F.

9. Each time a new formula f is added to the set F, we may use the new formula to prune the set Φ. Let f' be the formula induced from a weighted f-interpretation

$I' \in \Phi$ such that $[f] \subset [f']$. If there exists a non zero weighted possible world W of Π such that $W \not\models f'$ but $W \models f$, then the partial possible world labeled by I' can be removed from Φ. This because that any formula induced from the children of I' or I' itself is strict superset of f and whose probability must be less than f w.r.t. Π because of the existence of W. The function $Prune(\Phi, f)$ removes all such partial possible words from Φ and returns the remaining set of Φ.

12. If I does not know all the literals occurring in $\{l_1, \ldots, l_n\}$, we need to further expand the weighted f-interpretation until all the literals are known. The function $Branch(\Pi, I)$, described in Algorithm 7, generates children of I.

14. Removing the checked weighted f-interpretation makes sure that every weighted f-interpretation I will only be checked once.

We use the following example to trace this algorithm and to show how function $Prune(\Phi, f)$ may help to improve the efficiency.

Example 10 *Consider the following ground scou P-log program* Π_{10}*:*

```
a, b, c, f : boolean.
[a] random(a).
[b] random(b).
[c] random(c).
f:-a.
%end
```

The query Q *to this program is* $[a, b, c]|obs(f)$.

If we compute all the possible worlds of $\Pi_{10} \cup \Pi_Q$, we will have 4 possible worlds and therefore, we will have 4 formulas: a, $a \wedge b$, $a \wedge c$ and $a \wedge b \wedge c$. It is easy to see that the formula a is the one with the highest probability with respect to $\Pi_{10} \cup \Pi_Q$. Now let us see what is returned from the function $ReduceFormula(\Pi_{10}, Q)$. First, by call-

ing function $FindPartialWorld$, we assume that it returns a set of partial possible worlds with only one element: $\{a, f\}$ (here, we denote the corresponding weighted f-interpretation by only listing the literals which have been assigned to value T). Therefore, initially, $\Phi_0 = \{\{a, f\}\}$. Because $\{a, f\}$ does not know literals b and c, we need to expand this partial possible world. Assume we fired random selection rule $[b]$ and we get $\Phi_1 = \{\{a, b, f\}, \{a, \neg b, f\}\}$. Because $rank(\{a, \neg b, f\}) = 1 < rank(\{a, b, f\}) = 2$, we select $I_1 = \{a, \neg b, f\}$ in step 4 in the second the iteration. Since I_1 does not know c, we further expand this node and result in $\Phi_2 = \{\{a, b, f\}, \{a, \neg b, c, f\}, \{a, \neg b, \neg c, f\}\}$. In the third iteration, $\{a, \neg b, \neg c, f\}$ will be picked and it knows $\{a, b, c\}$. The atomic formula a will be added to F and the function $Prune(\Phi, a)$ will be called. Now we can see that $\{a, b, f\}$ can be removed from Φ_2 as the formula $a \wedge b$ is a superset of a and there exists a non-zero weighted partial possible world (for example, $\{a, \neg b, \neg c, f\}$) which does not entail formula $a \wedge b$. With the same reason, the partial possible world $\{a, \neg b, c, f\}$ will also be removed from Φ_2. Therefore, at the end of third iteration, $\Phi_3 = \emptyset$ and the function returns $F = \{a\}$. The whole process created 4 partial possible worlds while computing all possible worlds will create 6 partial possible worlds in total, and furthermore, the set returned from function $ReduceFormula(\Pi, Q)$ give us a much smaller set than the set of formulas built from computing all possible worlds. When dealing with large domain, this function is critical to the performance of our system.

5.5.3 Diagnosis on Space Shuttle System

Reaction Control System (RCS) is a subsystem of Space Shuttle which has primary responsibility for maneuvering the space shuttle. In [8], the RCS has been represented by A-Prolog for performing plan checking and generating plans. The system consists of three subsystems positioned at left, right and front. Each subsystem is a plumbing system where tanks, switches and valves are used for delivering fuels and oxidizer to the jets to fire. The astronauts on board have pre-scripted plans which can be used

for normal operations. However, if some components are faulty, a new plan must be generated in order to finish the mission and ensure the safety of crews. Whether a new plan works or not largely depends on the knowledge of which components are faulty. Since there are many of those components, astronauts need to determine which components to check first and which one to check later. In this context, the methods of finding best diagnosis may help astronauts make such decisions.

The diagnosis problem can be solved by the translation based *plog*1.0 system given some extras assumptions such as there are at most two components are faulty. Without assumptions like this, the number of possible worlds will be too large to handle for the *plog*1.0 system.

The *plog*2.0 system can find the best diagnosis without such assumptions. While subsystems are not completely separate from each other, many possible faulty symptoms are only related to one subsystems. When performing diagnostic task, it is critical to eliminate unrelated components out of consideration. This is done by the ground algorithm of *plog*2.0 system. Similar to the *grid* domain example, the algorithm of computing partial possible worlds reduces the amount of computations needed for evaluating probabilities of formulas. With *plog*2.0, most instances can be solved within 1 minutes.

The complete P-log program for RCS diagnosis can be found in [47].

5.6 Probabilistic Planning Problem

In this section, we will show how to use P-log to find the best probabilistic plans. Planning problems are one of those most important and widely studied problems in knowledge representation. Planning problems can be characterized as classical planning problems if all actions are deterministic and the agent has complete knowledge of initial state. When there are nondeterministic actions and probability distribution is available, the planning problem can be called as probabilistic planning problems. For

probabilistic planning problems, if the agent is able to perform observations during the execution of its plan, then such problems are called conditional probabilistic planning problems in which plans are conditioned on its observation results. Otherwise, the problems are called probabilistic planning problems. Based on the requirements of the quality of the plan, one may look for a plan that always reaches the goal (conformant planning problem), or plans that achieve the goal with at least θ probability, or plans that achieve the goal with the highest probability. In this chapter, we focus on finding probabilistic plans that achieve the goal with the highest probability.

We start with a probabilistic planning example which is introduced in [27].

Example 11 *The task of the robot is to grasp a box with its grips. However, the action* grasp *is not deterministic. It may succeed with probability 0.7 if the grips are dry, or it may succeed with the probability 0.5 if the grips are wet. The robot also can perform another action* dry *which will make the grips dry. Different from original example, we assume this action is deterministic which means it will always succeed. We also assume if the robot already holds the block then performing action* grasp *will keep the robot holding the block, but performing action dry drops the block. The robot knows that the block is not held initially but has no idea whether the grips are wet or not. Instead, it believes that the grips are wet with probability 0.1 initially.*

Given planning length of 2, the robot can have the following two intuitive simple probabilistic plans:

- the robot can first *dry* the grip then performs the action *grasp*, or
- the robot can perform action *grasp* twice this certainly increase its chance of holding the block after that.

Now the question is which plan is better?

Definition 61 *[Planning Problem]*

A **planning problem** *is a tuple* $\langle \mathcal{K}, s_0, s_f \rangle$ *where* s_0 *and* s_f *are consistent partial*

states and $\mathcal{K}$ is a domain description which defines a transistion diagram T.

Definition 62 *[Probabilistic Plan]*

We say a chain of action $\alpha = \langle a_0, \ldots, a_{n-1}\rangle$ is a **probabilistic plan** *for a planning problem $\langle \mathcal{K}, s_0, s_f\rangle$ if there exists a path $p = \langle \sigma_0, a_0, \sigma_1, a_1, \ldots, a_{n-1}, \sigma_n\rangle$ such that $\sigma_0 \in compl(s_0)$, $\sigma_n \in compl(s_f)$ and $P_{\mathcal{K}}(p|H) > 0$.*

Certainly, we are interested in a chain of actions that leads an agent starting from initial partial state s_0 to the final partial state s_f with higher probabilities. Let $\alpha = \langle a_0, \ldots, a_{n-1}\rangle$ and $PT(\alpha, s_0, s_f)$ be the collection of all the paths $\langle \sigma'_0, a'_0, \sigma'_1, a'_1, \ldots, a'_{n-1}, \sigma'_n\rangle$ of length n in T such that $\sigma'_0 \in compl(s_0)$, $\sigma'_n \in compl(s_f)$ and $a_i = a'_i$ (for $0 \leq i \leq n-1$). Let P_{s_0} be the probability distribution over all possible initial states. We define the probability $P_{s_0}(s_f|\alpha, s_0)$ of a probabilistic plan α for a planning problem $\langle \mathcal{K}, s_0, s_f\rangle$ as follows:

$$P_{s_0}(s_f|\alpha, s_0) = \sum_{p \in PT(\alpha, s_0, s_f)} P_{s_0}(p)$$

Now we introduce the definition of *best probabilistic plan* as follows.

Definition 63 *[Best Probabilistic Plan]*

Let $\mathcal{P} = \langle \mathcal{K}, s_0, s_f\rangle$ be a planning problem and A_n be the collection of all action chains with length n. Let

$$S = \arg\max_{\alpha \in A_n} \{P_{s_0}(s_f|\alpha, s_0)\}$$

An action chain α is said to be the **best probabilistic plan** *if $\alpha \in S$.*

Figure 5.3 shows the transition diagram of our example. There are four states and four possible action chains with length 2. However, only two of them are probabilistic plans. For the other two action chains $\{dry, dry\}$ and $\{grasp, dry\}$, both will lead to the agent staying in the state $\{\neg wet, \neg hold\}$.

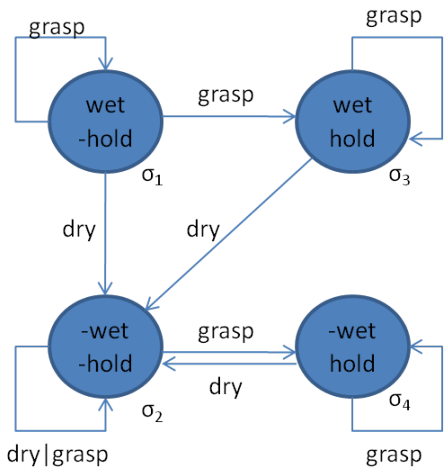

Figure 5.3: Trasition diagram of Exmaple 2

For the other two action chains $\alpha_1 = \{dry, grasp\}$ and $\alpha_2 = \{grasp, grasp\}$, we compute $P_T(\{hold\}|\langle dry, grasp\rangle, \{\neg hold\})$ and $P_T(\{hold\}|\langle grasp, grasp\rangle, \{\neg hold\})$ as follows:

There are two possible initial states: $\sigma_1 = \{\neg hold, wet\}$ and $\sigma_2 = \{\neg hold, \neg wet\}$. According to our example, the probability distribution of these two states are:

- $P_{\{\neg hold\}}(\{\neg hold, wet\}) = 0.1$
- $P_{\{\neg hold\}}(\{\neg hold, \neg wet\}) = 0.9$

For plan α_1, there are two paths in $PT(\alpha_1, \{\neg hold\}, \{hold\})$ and their probabilities are:

$P_{\{\neg hold\}}(\langle \sigma_1, dry, \sigma_2, grasp, \sigma_4\rangle) = 0.1 \times 1 \times 0.7 = 0.07$, and

$P_{\{\neg hold\}}(\langle \sigma_2, dry, \sigma_2, grasp, \sigma_4\rangle) = 0.9 \times 1 \times 0.7 = 0.63$.

Therefore, for plan α_1, we have

$P_{\{\neg hold\}}(\{hold\}|\alpha_1, \{\neg hold\}) = 0.07 + 0.63 = 0.7$.

For plan α_2, there are two paths in $PT(\alpha_2, \{\neg hold\}, \{hold\})$ starting at state σ_1 and their probabilities are:

- $P(\langle \sigma_1, grasp, \sigma_1, grasp, \sigma_3 \rangle) = 0.1 \times 0.5 \times 0.5 = 0.025$;
- $P(\langle \sigma_1, grasp, \sigma_3, grasp, \sigma_3 \rangle) = 0.1 \times 0.5 \times 1 = 0.05$.

There are two paths in $PT(\alpha_2, \{\neg hold\}, \{hold\})$ starting at state σ_2 and their probabilities are:

- $P(\langle \sigma_2, grasp, \sigma_2, grasp, \sigma_4 \rangle) = 0.9 \times 0.3 \times 0.7 = 0.189$;
- $P(\langle \sigma_2, grasp, \sigma_4, grasp, \sigma_4 \rangle) = 0.9 \times 0.7 \times 1 = 0.63$.

Overall, for plan α_2, we have

$P_{\{\neg hold\}}(\{hold\}|\alpha_2, \{\neg hold\}) = 0.025 + 0.05 + 0.189 + 0.63 = 0.894$

Hence, instead of drying the grip first, the robot can simply try to grab the block twice and it is more likely to be successful than the other plans.

5.6.1 Solving Probabilistic Planning Problem

The transition diagram of Example 11 can be represented by language $\mathcal{NB}$. The completed description of Example 11 can be found in [47]. Together with history and probability distribution over initial states, the domain description $\mathcal{K}$ can be mapped to the P-log program $\Pi(\mathcal{K})$. In addition, we need some extra rules to describe the goals and to generate plans. In Example 11, the goal is to hold the block at time step 2, we have a following rule:

$$goal \leftarrow h(grasp, 2) \tag{5.11}$$

To generate plans, we introduce a new random attributes:

$$o : time \rightarrow action \tag{5.12}$$

and have random selection rules for each step T where $0 \leq T < n$:

$$[o(T)]\ random(o(T)) \leftarrow time(T), 0 \leq T < n \tag{5.13}$$

The following rules say that if we select the action A at time T, the action A occurs at time T.

$$occur(A, T) \leftarrow o(T) = A \tag{5.14}$$

We denote the P-log program consisting of rules (5.11) - (5.14) by Π_{goal}.

Let Ω be all the possible action sequences from step 0 to step $n-1$. By $Conj(\omega_{n-1})$ we mean a conjunction formed formula $o(0) = a_0 \wedge \cdots \wedge o(n-1) = a_{n-1}$. We define $< o(0), \ldots, o(n-1) >$, a set of formulas, as:

$$< o(0), \ldots, o(n-1) > =_{def} \{Conj(\omega_{n-1}) : \omega \in \Omega\}$$

The following query to program $\Pi(\mathcal{K}) \cup \Pi_{goal}$ contains the best probabilistic plans.

$$< o(0), \ldots, o(n-1) > |obs(goal)$$

Theorem 5 *Let $\Pi(\mathcal{K})$ be a P-log program obtained from a domain description $\mathcal{K}$ of a probabilistic planning problem $\langle \mathcal{K}, s_0, s_f \rangle$. An action sequence $o(0) = a_0, \ldots, o(n-1) = a_{n-1}$ is a best probabilistic plan if and only if the formula: $o(0) = a_0 \wedge \cdots \wedge o(n-1) = a_{n-1}$ is in the answer of the query $< o(0), \ldots, o(n-1) > |obs(goal)$ with respect to the P-log program $\Pi(\mathcal{K}) \cup \Pi_{goal}$.*

5.6.2 Performance

We use the *Sand and Castle* [28] example to test the performance of plog systems. The tests are conducted under Ubuntu 10.10 environment with Intel dual-core processor at 1.60GHz, 2GB memory.

The Figure 5.4 shows the time spent on finding the best probabilistic plans with different lengths. We can see that the system, *plog*2.0, can only compute plans with shorter length. Due to the number of possible worlds growing exponentially, the time used for finding all the best probabilistic plans grows exponentially too. Comparing

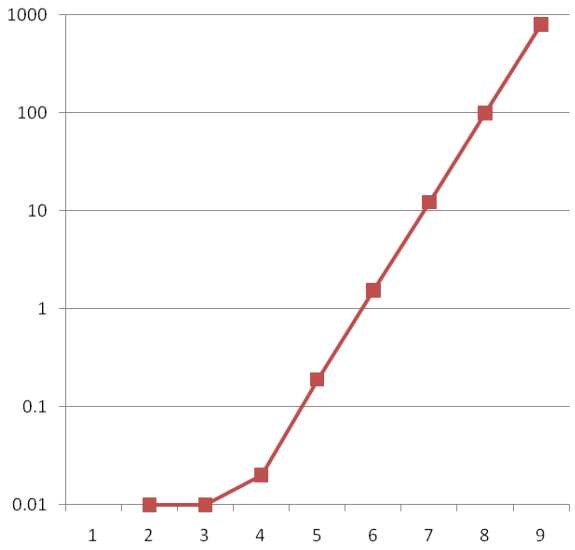

Figure 5.4: Performance of plog2.0 on *Sand and Castle* problem

to other probabilistic planning system, such as MAXPlan [28] and CPPlan [29], the ability of our system of finding probabilistic plans is limited.

When the length of the plan goes longer, so is the number of random selection rules. This hurts the performance of our *plog* system in two ways: first, previous analysis has shown that the performance of *plog* system is largely related to the number of possible worlds, or say the

search space. In probabilistic planning problem, longer plans means more selections on actions and therefore larger search space. Second, the deeper the program tree goes, the smaller is the unnormalized weight of leaf nodes. When the system finds a plan, it updates the global lower bound v, that is if a formula f is an answer of the query, then the probability of f must be larger than or equal to v. Since such v is tend to be very small in a large and deep programming tree, such bounding mechanism becomes useless and can not prune any search space of the problem.

Chapter 6

PROOFS

This chapter presents proofs of theories and lemmas shown in the Chapter 3 and Chapter 5.

6.1 Proof of Theorem 1

Before we move to the proof of theorem 1, we first introduce the following lemma and its proof which will be used for the proof of the theorem.

Lemma 1 *Let* Π *be a ground scou P-log program with no obs and do statement,* l *be a literal of signature of* Σ *and* $\|$ *be a causal leveling function. Let* $L_1, \ldots, L_{n+1}$ *be the sets of literals as defined in definition 12 and* $\Pi_1, \ldots, \Pi_{n+1}$ *be the* $\|$*-induced structure of* Π*. If* $l \in L_k$*, then* $P_\Pi(l) = P_{\Pi_k}(l)$*.*

PROOF: Let $a_1(\bar{t}_1), \ldots, a_m(\bar{t}_m)$ be the ordering of random attribute terms induced by $\|$. We construct a sequence of trees $T_0, \ldots, T_n$ as described in [1]: T_0 is a tree with one node n_0 labeled by *true* and T_j $(0 < j \leq n)$ is obtained from T_{j-1} by expanding every leaf of T_{j-1} which is ready to branch on $a_j(\bar{t}_j)$ via any rule relative to Π_j by this term. By proposition 1 of [1], we have $\Pi_{n+1} = \Pi$ and T_n is a tableau of Π which represents Π. Since Π is a unitary P-log program, T_n is a unitary tree.

Let Ω_k be the collection of all possible worlds of Π_k. Since Π is a causally ordered and unitary program, by theorem 1 in [1], Π is coherent. Therefore, Π contains at least one possible worlds with non-zero unnormalized probability. Since every possible world of Π expands some possible world of Π_k, Π_k contains at least one possible world with non-zero unnormalized probability. By definition of probability, we can compute the

probability, $P_{\Pi_k}(l)$, of literal l being true with respect to program Π_k as follow:

$$P_{\Pi_k}(l) = \sum_{V \in \Omega_k, l \in V} \mu(V) \tag{6.1}$$

where $\mu(V)$ is the normalized probability of possible world V. $\mu(V)$ can be computed from unnormalized probability of possible worlds in Ω_k as follow:

$$\mu(V) = \frac{\hat{\mu}(V)}{\sum_{V \in \Omega_k} \hat{\mu}(V)} \tag{6.2}$$

From 6.1 and 6.2, we derive that

$$P_{\Pi_k}(l) = \frac{\sum_{V \in \Omega_k, l \in V} \hat{\mu}(V)}{\sum_{V \in \Omega_k} \hat{\mu}(V)} \tag{6.3}$$

By definition of $\hat{\mu}(V)$,

$$\hat{\mu}(V) = \prod_{a(\bar{t})=y \in V} P(V, a(\bar{t}) = y) \tag{6.4}$$

where the product range over all the random attribute atom $a(\bar{t}) = y \in V$. Let p_k be a path from the root of T_k to some leaf node and p_n be a path in T_n expanding p_k. Let V be the possible world represented by the leaf node of path p_k and W be the possible world represented by the leaf node of path p_n. Let $a(\bar{t}) = y$ be a random attribute atom such that $a(\bar{t}) = y \in V$ and r be the random selection rule of Π_k that generates $a(\bar{t}) = y$.

1. If Π contains a pr atom

$$pr_r(a(\bar{t}) = y|_c B) = v$$

such that B is satisfied by V, we have that $PA(V, a(\bar{t}) = y) = v$. By proposition 3 in [1], we have $V \models B \Rightarrow W \models B$ and this implies that $PA(W, a(\bar{t}) = y) = v$. Therefore, $PA(V, a(\bar{t}) = y) = PA(W, a(\bar{t}) = y)$.

2. Similarly, if random attribute $a(\bar{t}) = y$ is assigned with default probability, then $PD(V, a(\bar{t}) = y) = PD(W, a(\bar{t}) = y)$.

From (a), (b) and definition 3, we have

$$P(V, a(\bar{t}) = y) = P(W, a(\bar{t}) = y) \tag{6.5}$$

From equation 6.4 and 6.5, we have

$$\hat{\mu}(V) = \prod_{a(\bar{t})=y \in V} P(W, a(\bar{t}) = y) \tag{6.6}$$

for every possible world W of Π_{n+1} expands V.

Let V' be a possible world of program Π_m, where $1 \leq m \leq n+1$. By $s_m(V')$, we denote the leaf node of T_{m-1} which represents V'. From the construction of $T_0, \ldots, T_n$ and definition of path value, for every possible world V of program Π_k we can compute the path value, $pv_{T_n}(s_k(V))$, of a node $s_k(V)$ of T_n as follows:

$$pv_{T_n}(s_k(V)) = \prod_{a(\bar{t})=y \in p_{T_n}(s_k(V))} P(W, a(\bar{t}) = y) \tag{6.7}$$

where $p_{T_n}(s_k(V))$ is the set of labels of nodes lying on the path from the root of T_n to node $s_k(V)$, i.e., the set of random attributes in V. From equation 6.6 and 6.7, we have

$$\hat{\mu}(V) = pv_{T_n}(s_k(V)) \tag{6.8}$$

From equation 6.3 and 6.8, we have

$$P_{\Pi_k}(l) = \frac{\sum_{V \in \Omega_k, l \in V} pv_{T_n}(s_k(V))}{\sum_{V \in \Omega_k} pv_{T_n}(s_k(V))} \tag{6.9}$$

Notice that T_{k-1} is a unitary tree. By definition of unitary tree, the sum of path value of all the leaf nodes equals to 1. Therefore, 6.9 can be rewritten as

$$P_{\Pi_k}(l) = \sum_{V \in \Omega_k, l \in V} pv_{T_n}(s_k(V)) \tag{6.10}$$

Let Ω_{n+1} be the set of all answer sets of program Π_{n+1}. We can similarly have the following equation:

$$P_{\Pi_{n+1}}(l) = \sum_{W \in \Omega_{n+1}, l \in W} pv_{T_n}(s_{n+1}(W)) \tag{6.11}$$

Let s be a node of T_n. By $leaf_{T_n}(s)$, we denote the set of all the leaf nodes of T_n which are descendents of the node s. Since T_n is a unitary tree, from proposition 2 in [1], we have

$$pv_{T_n}(s_k(V)) = \sum_{W \in \Omega_{n+1}, s_{n+1}(W) \in leaf_{T_n}(s_k(V))} pv_{T_n}(s_{n+1}(W)) \tag{6.12}$$

From equation 6.10 and 6.12

$$P_{\Pi_k}(l) = \sum_{l \in V, V \in \Omega_k} (\sum_{W \in \Omega_n, s_{n+1}(W) \in leaf_{T_n}(s_k(V))} pv_{T_n}(s_{n+1}(W))) \tag{6.13}$$

For every possible world, V, of program Π_k and possible world, W of program Π_{n+1} such that W expands V, we have $V \models l \Leftrightarrow W \models l$. Therefore, from equation (12), we have

$$P_{\Pi_k}(l) = \sum_{W \in \Omega_{n+1}, l \in W} pv_{T_n}(s_{n+1}(W)) \tag{6.14}$$

From equation 6.11 and 6.14,we can conclude that,

$$P_{\Pi_{n+1}}(l) = P_{\Pi_k}(l). \tag{6.15}$$

Since $\Pi_{n+1} = \Pi$, we have that

$$P_{\Pi}(l) = P_{\Pi_k}(l). \tag{6.16}$$

This ends the proof of the lemma.

Theorem 1

Let Π be scou P-log program with no *do* and *obs* statements and Σ be its signature. Let l be a literal of Σ and $\Pi^l = \{r | Term(Head(r)) \in Dep_\Pi(l), r \in \Pi\}$, then $P_{\Pi^l}(l) = P_\Pi(l)$.

PROOF:

To prove the theorem, we first construct a strict probabilistic leveling F_2 of Π, such that for any random attribute term $a(\bar{t})$,

$$F_2(a(\bar{t})) \leq F_2(\alpha_l) \text{ iff } a(\bar{t}) \in Dep(\alpha_l). \tag{6.17}$$

where α_l is the attribute term forming the literal l.

Since Π is a causally ordered P-log program, there exists a strict probabilistic leveling function F. To define F_2 we will need an auxiliary function, F_1, defined as follow:

$$F_1(a(\bar{t})) = \begin{cases} F(a(\bar{t})) & a(\bar{t}) \text{ is a random attribute term} \\ m(a(\bar{t})) & otherwise \end{cases} \tag{6.18}$$

where $m(a(\bar{t}))$ is the highest leveling of random attribute terms in the dependent set of $a(\bar{t})$. In case $a(\bar{t})$ does not depend on any random attribute term, $m(a(\bar{t})) = 0$.

Let us consider the following program:

$$a, b, l, r : boolean$$

$$[a]\ random(a)$$

$$[b]\ random(b)$$

$$l \leftarrow a$$

$$r \leftarrow l$$

Let F be a strict probabilistic leveling function defined as follow:

$$F(b) = 1$$

$$F(a) = 2$$

$$F(l) = 3$$

$$F(r) = 4$$

By definition of F_1, we have

$$F_1(b) = 1$$

$$F_1(a) = 2$$

$$F_1(l) = 2$$

$$F_1(r) = 2$$

The following shows that F_1 is a strict probabilistic leveling over Σ:

Clearly, F_1 is a leveling function as it maps every attribute term of Σ onto a set [0,n] of integers. Since F is a strict probabilistic leveling and $F_1(a(\bar{t})) = F(a(\bar{t}))$ for random attribute term $a(\bar{t})$, condition 1 of definition 4 is satisfied by F_1.

For a random selection rule: $[r]\ random(a(\bar{t}) : \{Y : p(Y)\}) : -B$, because F is a strict probabilistic leveling,

$$F(a(\bar{t})) > max\{F(a_1(\bar{t_1})) : a_1(\bar{t_1}) \in Term(B \cup \{p(Y)\})\}. \tag{6.19}$$

From 6.19 we have that

$$F(a(\bar{t})) > F(a_1(\bar{t_1})), \tag{6.20}$$

Now consider two cases:

- Suppose $a_1(\bar{t}_1)$ is a random attribute term. Then, by definition of F_1, we have that

$$F_1(a_1(\bar{t}_1)) = F(a_1(\bar{t}_1)). \tag{6.21}$$

  Therefore, from 6.20 and 6.21 we have that

$$F(a(\bar{t})) > F_1(a_1(\bar{t}_1)). \tag{6.22}$$

- Suppose $a_1(\bar{t}_1)$ is a regular attribute term. From condition 4 of definition 11, we know that for any random attribute term $a_2(\bar{t}_2)$ which $a_1(\bar{t}_1)$ depends on, we have

$$F(a_1(\bar{t}_1)) \geq max\{F(a_2(\bar{t}_2)) : a_1(\bar{t}_1) \text{ depends on } a_2(\bar{t}_2)\}. \tag{6.23}$$

  From definition of F_1, we reassign the level of $a_1(\bar{t}_1)$ to the highest level of those random attribute terms that it depends on. This means

$$F_1(a_1(\bar{t}_1)) = max\{F(a_2(\bar{t}_2)) : a_1(\bar{t}_1) \text{ depends on } a_2(\bar{t}_2)\}. \tag{6.24}$$

  From 6.23 and 6.24, we have that

$$F(a_1(\bar{t}_1)) \geq F_1(a_1(\bar{t})). \tag{6.25}$$

  From 6.19 and 6.25, we have that $F(a(\bar{t})) > F_1(a_1(\bar{t}_1))$.

Hence, we can have the following formula

$$F(a(\bar{t})) > max\{F_1(a_1(\bar{t}_1)) : a_1(\bar{t}_1) \in Term(B \cup \{p(Y)\})\} \tag{6.26}$$

Because $a(\bar{t})$ is a random attribute term and by definition of F_1, we have

$$F_1(a(\bar{t})) = F(a(\bar{t})) \tag{6.27}$$

From 6.26 and 6.27, we have

$$F_1(a(\bar{t})) > max\{F_1(a_1(\bar{t}_1)) : a_1(\bar{t}_1) \in Term(B \cup \{p(Y)\})\} \tag{6.28}$$

Therefore, condition 2 of definition 11 is satisfied. Similarly, condition 3 of definition 11 is also satisfied.

Condition 4 follows directly from the definition of F_1 for non-random attribute terms.

Hence F_1 is a strict probabilistic leveling function.

Now we will use F_1 to construct the strict probabilistic leveling F_2 which satisfying 6.17.

Let $k = F_1(\alpha_l) + 1$ we define F_2 as follows:

$$F_2(a(\bar{t})) = \begin{cases} F_1(a(\bar{t})) & a(\bar{t}) \in Dep(\alpha_l) \\ F_1(a(\bar{t})) + k & otherwise \end{cases}$$

Example:

$$F_2(a) = 2$$

$$F_2(l) = 2$$

$$F_2(r) = 2$$

$$F_2(b) = 3$$

First, we will show that F_2 satisfies 6.17:

- $F_2(a(\bar{t})) \leq F_2(\alpha_l) \Rightarrow a(\bar{t}) \in Dep(\alpha_l)$ (if part).

  To prove the *if part* of 6.17, we assume that $a(\bar{t}) \notin Dep(\alpha_l)$ and show that $F_2(a(\bar{t})) > F_2(\alpha_l)$.

  Since for every $a(\bar{t})$, if $a(\bar{t}) \notin Dep(\alpha_l)$, then by definition of F_2,

  $$F_2(a(\bar{t})) = F_1(a(\bar{t})) + k. \tag{6.29}$$

where $k = F_1(\alpha_l) + 1$. Since $F_1(a(\bar{t})) \geq 0$, from 6.29 we have that

$$F_2(a(\bar{t})) > F_1(\alpha_l). \tag{6.30}$$

Since $\alpha_l \in Dep(\alpha_l)$, by definition of F_2, we have that

$$F_2(\alpha_l) = F_1(\alpha_l) \tag{6.31}$$

From 6.30 and 6.31, we have that

$$F_2(a(\bar{t})) > F_2(\alpha_l) \tag{6.32}$$

Hence $F_2(a(\bar{t})) \leq F_2(\alpha_l) \Rightarrow a(\bar{t}) \in Dep(\alpha_l)$

- $F_2(a(\bar{t})) \leq F_2(\alpha_l) \Leftarrow a(\bar{t}) \in Dep(\alpha_l)$ (only if part)

  Suppose $a(\bar{t}) \in Dep(\alpha_l)$. By definition of F_2, we have

  $$F_2(a(\bar{t})) = F_1(a(\bar{t})) \tag{6.33}$$

  Now consider the following two cases:

  - α_l is a random attribute term. Since F_1 is a strict probabilistic leveling, it satisfies conditions 2,3 and 4 of definition 11. Because $a(\bar{t}) \in Dep(\alpha_l)$, this implies that $F_1(\alpha_l) \geq F_1(a(\bar{t}))$.
  - α_l is a regular attribute term. By definition of F_1 for regular attribute term, we can see that for any random attribute term $a(\bar{t})$ which α_l depends on, $F_1(\alpha_l) \geq F_1(a(\bar{t}))$.

  Hence for every α_l we have that

  $$F_1(\alpha_l) \geq F_1(a(\bar{t})) \tag{6.34}$$

  From 6.33 and 6.34, we have that

  $$F_1(\alpha_l) \geq F_2(a(\bar{t})) \tag{6.35}$$

Since $\alpha_l \in Dep(\alpha_l)$, from definition of F_2, we have

$$F_2(\alpha_l) = F_1(\alpha_l) \tag{6.36}$$

From 6.35 and 6.36, we have that

$$F_2(\alpha_l) \geq F_2(a(\bar{t})) \tag{6.37}$$

Hence F_2 satisfies 6.17.

To prove that F_2 is a strict probabilistic leveling, we will show that F_2 satisfies all the conditions of definition 11.

1. To show that for every two random attribute terms $a_1(\bar{t}_1)$ and $a_2(\bar{t}_2)$ in Σ, $F_2(a_1(\bar{t}_1)) \neq F_2(a_2(\bar{t}_2))$, we have to consider the following three cases:

   (a) $a_1(\bar{t}_1) \notin Dep(\alpha_l)$ and $a_2(\bar{t}_2) \notin Dep(\alpha_l)$: Because F_1 is a strict probabilistic leveling, we have that

   $$F_1(a_2(\bar{t}_2)) \neq F_1(a_1(\bar{t}_1)). \tag{6.38}$$

   By definition of F_2, we have that

   $$F_2(a_1(\bar{t}-1)) = F_1(a_1(\bar{t}_1)) + kF_2(a_1(\bar{t}_1)) = F_1(a_1(\bar{t}_1)) + k \tag{6.39}$$

   From 6.38 and 6.39 we have that $F_2(a_1(\bar{t}_1)) \neq F_2(a_2(\bar{t}_2))$. Hence condition 1 is satisfied.

   (b) $a_1(\bar{t}_1) \in Dep(\alpha_l)$ and $a_2(\bar{t}_2) \in Dep(\alpha_l)$: Similar to case a.

   (c) $a_1(\bar{t}_1) \in Dep(\alpha_l)$ and $a_2(\bar{t}_2) \notin Dep(\alpha_l)$: Form 6.17, we know that $F_2(a_1(\bar{t}_1)) \leq F_2(\alpha_l)$ and $F_2(a_2(\bar{t}_2)) > F_2(\alpha_l)$. This implies $F_2(a_1(\bar{t}_1)) \neq F_2(a_2(\bar{t}_2))$.

2. To prove that for every random selection rule $[r]$ $random(a(\bar{t}) : \{Y : p(Y)\}) : -B$ of Π we have $|a(\bar{t}) = y| > |\{p(Y) : Y \in range(a) \cup B|$, we need to consider the following two cases:

(a) If $a(\bar{t}) \in Dep(\alpha_l)$, then attribute terms that forms $p(y)$ and literals in B also belong to $Dep(\alpha_l)$, By definition of F_2 and given F_1 is a strict probabilistic leveling, F_2 satisfies condition 2 of definition 11.

(b) If $a(\bar{t}) \notin Dep(\alpha_l)$, then $F_2(a(\bar{t})) = F_1(a(\bar{t})) + k$. For every attribute term $a_1(\bar{t}_1) \in Term(\{p(y)\} \cup B)$, $F_2(a_1(\bar{t}_1))$ either equals to $F_1(a_1(\bar{t}_1)) + k$ or $F_1(a_1(\bar{t}_1))$. In both cases, clearly, $F_2(a(\bar{t})) > F_2(a_1(\bar{t}_1))$.

3. Condition 3 is similar to condition 2.

4. To prove F_2 satisfies condition 4 of definition 11, we need to consider the following three cases. Given F_1 being a strict probabilistic leveling and the definition of F_2 , condition 4 is hold for the following two cases:

   (a) $a_1(\bar{t}_1) \notin Dep(\alpha_l)$ and $a_2(\bar{t}_2) \notin Dep(\alpha_l)$;

   (b) $a_1(\bar{t}_1) \in Dep(\alpha_l)$ and $a_2(\bar{t}_2) \in Dep(\alpha_l)$.

   For the third case: $a_1(\bar{t}_1) \notin Dep(\alpha_l)$, $a_2(\bar{t}_2) \in Dep(\alpha_l)$ and non random attribute $a_1(\bar{t}_1)$ depends on random attribute term $a_2(\bar{t}_2)$, from (17) we have $F_2(a_1(\bar{t}_1)) > k \geq F_2(a_2(\bar{t}_2))$. Therefore, condition 4 is satisfied.

Since F_2 is a strict probabilistic leveling we can have $\Pi_1, \ldots, \Pi_{n+1}$ as a F_2-induced structure of Π. Notice that $l \in L_k$ with respect to F_2. By lemma 1, $P_{\Pi}(l) = P_{\Pi_k}(l)$. Now we will show that Π_k can be split to two programs Π' and Π_R, where Π_R may only contain regular rules and answer sets of program Π' are one to one correspond to the answer sets of Π_k.

It is not difficult to see that the set, U, of literals that formed by the attribute term from $Dep(\alpha_l)$ is a splitting set of program Π_k. As we define $\Pi' = \{r | Term(Head(r)) \in Dep(\alpha_l), r \in \Pi\}$, it is also not difficult to see that $\Pi' = bot_U(\Pi_k)$. Because random attribute term $a_1(\bar{t}_1), \ldots, a_k(\bar{t}_k)$ belong to $Dep(\alpha_l)$, all random selection rules in Π_k belong to $bot_U(\Pi_k)$. This indicates that $top_U(\Pi_k)$ only consists of regular rules. As Π is a strongly causally ordered P-log program, for every answer set X of $bot_U(\Pi_k)$,

there exists exactly one answer set Y for $eval_U(top_U(\Pi_k), X)$ and $X \cup Y$ is consistent. From splitting theorem, we know that for every answer set V of program Π_k, there exists a solution $\langle X, Y \rangle$ to Π_k with respect to U such that $V = X \cup Y$. Therefore, we can rewrite equation 6.1 as

$$P_{\Pi_k}(l) = \sum_{X \cup Y \models l, X \cup Y \in \Omega_k} \mu(X \cup Y) \tag{6.40}$$

Because $l \in U$, we know that $l \notin Y$. Also, Y contains only regular attribute terms. We rewrite above equation as

$$P_{\Pi_k}(l) = \sum_{X \models l, X \text{ is an answer set of } \Pi'} \mu(X) \tag{6.41}$$

From the definition of the probability of l being true with respect to program Π', we have the following equation:

$$P_{\Pi'}(l) = \sum_{X \models l, X \text{ is an answer set of } \Pi'} \mu(X) \tag{6.42}$$

From equation 6.41 and 6.42, we can establish that $P_{\Pi'}(l) = P_{\Pi_k}(l)$. From Lemma 1, $P_{\Pi}(l) = P_{\Pi_k}(l)$, we conclude that $P_{\Pi'}(l) = P_{\Pi}(l)$.

6.2 Proof of Propostion 1

Proposition 1

Let Π be a ground P-log program and A be an f-interpretation. Let $A' = LC(\Pi, A)$ where $A' \neq A^c$.

- A' is an extension of A.
- If A_W is an extension of A such that $true(A_W)$ is a possible world of program Π then A_W is also an extension of A';

PROOF:

First we prove that A' is an extension of A. Let l be a literal in the signature Σ of Π.

1. Suppose that $A(l) = T$, then by **rule 1(c)**, $A'(l) = T$. Hence $A'(l) \geq_i A(l)$.
2. Suppose that $A(l) = MT$. Because $A(l) \neq U \Rightarrow A'(l) \neq U$, and $A' \neq A^c$, therefore, $A'(l) \in \{MT, T\}$. Hence $A'(l) \geq_i A(l)$.
3. Suppose that $A(l) = MF$. Similar to (2), we have that $A'(l) \in \{MF, F\}$. Hence $A'(l) \geq_i A(l)$.
4. Suppose that $A(l) = F$. Then by **rule 3(b)**, $A'(l) = F$. Hence $A'(l) \geq_i A(l)$.
5. Suppose that $A(l) = U$. Since U is minimal value w.r.t. the ordering $>_i$, we have $A'(l) \geq_i A(l)$.

Therefore, A' is an extension of A.

Now we prove that if A_W is an extension of A such that $true(A_W)$ is a possible world of program Π then A_W is also an extension of A'.

Suppose there exists a literal l, such that $A'(l) >_i A(l)$ and there exists a possible world A_W such that $A_W(l)$ is incompatible with $A'(l)$. Because for each literal l', $A_W(l') \in \{T, F\}$ and A_W is also an extension of A, we can conclude that $A(l) = U$. Now we look at each rule of $LC(\Pi, A)$ in definition 27 (page 44) where $A(l) = U$ and $A'(l) \neq U$.

- Suppose that $A'(l) = T$ because of **1(a)**. Then, based on **1(a)**, $A(body(r)) = T$. $A(body(r)) = T \Rightarrow A_W(body(r)) = T$. Because A_W is a possible world, $A_W(body(r)) = T \Rightarrow A_W(l) = T$. $A_W(l)$ and $A(l)$ are compatible. Contradictory.
- Suppose that $A'(l) = T$ because of **1(b)**. Since $a(\bar{t}) \neq y \leftarrow a(\bar{t}) = y' \in \Pi$, $A(a(\bar{t}) = y') = T \Rightarrow A_W(a(\bar{t}) = y') = T \Rightarrow A_W(a(\bar{t}) = y) = T$. Hence, $A_W(l)$

and $A(l)$ are compatible. Contradictory.

- Suppose that $A'(l) = MT$ because of **2(a)**. Because A_W is a possible world and A_W is an extension of A, $A(body(r) \geq_i MT \Rightarrow A_W(body(r)) = T$. Because A_W is a possible world, $A_W(body(r)) = T \Rightarrow A_W(l) = T$. $A_W(l)$ and $A(l)$ are compatible. Contradictory.

- Suppose that $A'(l) = MT$ because of **2(b)**. The arguments are similar to what we did for 1(b) and 2(a).

- Suppose that $A'(l) = MT$ because of **2(c)**. Because $A'(l) = MT$ and $A_W(l)$ is incompatible with $A'(l)$, $A_W(l) \in \{F, MF\}$. Because $true(A_W)$ is a possible world, therefore $A_W(l) = F$.

$$A(body(r)\backslash\{\textbf{not } l\}) > U \Rightarrow A_W(body(r)\backslash\{\textbf{not } l\}) = T$$

  $A_W(l) = F \Rightarrow A_W(\textbf{not } l) = T$. Together, we have that $A_W(body(r)) = T$. $A_W(body(r)) = T \Rightarrow A_W(head(r)) = T$. A_W is not an extension of A. Contradictory.

- Suppose that $A'(l) = MT$ because of **2(d)**. From $A(head(r)) = MT$, $true(A_W)$ is a possible world and A_W is an extension of A, we have that $A_W(head(r)) = T$. As all other rules are falsified, the body of r must be true in $true(A_W)$. For $l \in pos(r)$, $A_W(l) = T$. $A_W(l)$ and $A(l)$ are compatible. Contradictory.

- Suppose that $A'(l) = F$ because of **3**: If all rules that support l are falsified by A, then $l \notin true(A_W)$. Therefore, $A_W(l) = A'(l) = F$. Contradictory.

- Suppose that $A'(l) = MF$ because of **4(a)**. Because $true(A_W)$ is a possible world of Π and A_W is an extension of A. All rules which support l are weakly falsified by A means all rules which support l are falsified by A_W. Therefore, $A_W(l) = F$. $A_W(l)$ and $A'(l)$ are compatible. Contradictory.

- Suppose that $A'(l) = MF$ because of **4(b)**: Similar to 2(c).

- Suppose that $A'(l) = MF$ because of **4(c)**: Similar to 2(d).
- Suppose that $A'(l) = MF$ because of **4(d)**: $A(\bar{l}) = MF \Rightarrow A_W(\bar{l}) = T \Rightarrow A_W(l) = F$. $A_W(l)$ and $A(l)$ are compatible. Contradictory.

In summary, If A_W is an extension of A such that $true(A_W)$ is a possible world of program Π then A_W is also an extension of A'.

6.3 Proof of Proposition 2

Proposition 2

Let Π be a ground P-log program and A be an f-interpretation. Let $A' = AtMost(\Pi, A)$ where $A' \neq A^c$.

- A' is an extension of A.
- If A_W is an extension of A such that $true(A_W)$ is a possible world of program Π then A_W is also an extension of A';

PROOF:

From step 4 of function $AtMost(\Pi, A)$, we can see that for a literal l, if $A(l) \neq A'(l)$ then $A(l) \leq U$ and $A'(l) = F$. Hence, A' is an extension of A.

The operator $G^{\Pi}_{L^+,L^-}(X)$ is the same operator used for $Atmost(\Pi, M)$ function in *Smodels* which has the following property ([45]):

Let P be a ground answer set program and M be a set of extended literals. If S is an answer set of P such that S agrees with M then $S \subseteq Atmost(P, M)$.

Since $true(A_W)$ is an answer set of Π such that $true(A_W)$ agrees with $true(A)$, $true(A_W) \subseteq Atmost(\Pi, true(A))$. Hence for each literal l, if $l \notin Atmost(\Pi, true(A))$, then $l \notin true(A_W)$, i.e., $A_W(l) = F$.

As the function $Atmost(\Pi, true(A))$ of *Smodels* computes the same least fixed point

as $G^{\Pi}_{L^+,L^-}(X)$ does, $l \notin Atmost(\Pi, true(A))$ means $l \notin G^{\Pi}_{L^+,L^-}(\emptyset)$. This means $A'(l) = F$. Therefore A_W is also an extension of A'.

6.4 Proof of Theorem 2

Theorem 2

Let Π be a ground *scou* P-log program and Q be a query to Π. Let T be a unitary program tree with respect to Π and Q . Then I is a weighted f-interpretation labeling a complete and consistent leaf node of T if and only if $true(A(I))$ is a possible world of program $\Pi \cup \Pi_Q$.

PROOF:

We prove this theorem based on the results of [46]. In [46], an answer set solver system can be represented by a graph G where each node is labeled by a list of decorated literals and each arch from a node N to N' in G means N' can be reached from N by applying certain operations (similar to $AtLeast$ and $AtMost$ used in system *smodels*). Each answer set computed by the system can be matched to a terminal node (a complete and consistent node in which every literal l in the signature Σ is assigned) which is not a $FailState$.

Our approach to this proof starts with some definitions and lemmas, then we will give a proof of the theorem based on these lemmas. Finally, we prove each lemmas to finish the proof.

We introduce a new notation $\mathcal{M}(N)$ where N is a node of T:

Definition 64 *[$\mathcal{M}(N)$]*

Let I be the weighted f-interpretation labeling the node N. Let S be a set of atoms such that for each atom $a(\bar{t}) = y \in S$, $a(\bar{t}) = y$ is assigned to T in line 7 in Algorithm 7. Let M be an SM_Π graph (a graph represents the application of Smodels algorithm) to program Π. We said that $\mathcal{M}(N)$ is a node of M relative to Σ such that

1. $l \in \mathcal{M}(N)$ *if and only if* $A(I_N)(l) = T$ *and* $l \notin S$;

2. $\neg l \in \mathcal{M}(N)$ *if and only if* $A(I_N)(l) = F$;

3. $l^d \in \mathcal{M}(N)$ *if and only if* $l \in S$

Lemma 2 *Let N be the root node of T, then $\mathcal{M}(N)$ is reachable from $\emptyset$ in a SM_Π graph.*

Lemma 3 *If N and N' are consistent nodes of unitary program tree where N' is a child of N, then there exists a SM_Π graph in which $\mathcal{M}(N')$ is reachable from $\mathcal{M}(N)$.*

Thus for each leaf node N, $\mathcal{M}(N)$ is reachable from $\emptyset$ in a SM_Π graph.

Because for each consistent and complete leaf node N of T, $\mathcal{M}(N)$ is a terminal state of $\mathcal{M}$. By proposition 2 of [46], we can see that for each consistent and complete leaf node N of T, $\mathcal{M}(N)^+$ is an answer set of program $\Pi \cup \Pi_Q$ where $\mathcal{M}(N)^+$ is a set of literals which is assigned to *true* w.r.t. the node $\mathcal{M}(N)$.

By definition of $\mathcal{M}(N)$ and $true(A(I_N))$, we know that $\mathcal{M}(N)^+ = true(A(I))$. Therefore, the set $true(A(I_N))$ is a possible world of program $\Pi \cup \Pi_Q$.

Because in scou P-log program, each possible world is correspond to a **unique** assignment of random attributes. As we can see that each possible assignment of random attributes in T either leads to a consistent leaf nodes, which is a possible world or an inconsistent partial assignment A. Therefore, for each possible world W of program $\Pi \cup \Pi_Q$ there is a consistent leaf node N of T such that $true(A(N)) = W$.

Now, we focus on the proof of Lemma 2 and Lemma 3.

Lemma 2

The label for the root node of a unitary programming tree T is the output of function $Initialize(\Pi, Q)$. We list all the steps in which some literals are assigned to T or F in this function and show that there is a corresponding transition rule in graph SM_Π:

1. **step 1(a)**: Because sort declaration $c = \{x_1, \ldots, x_n\}$ is translated to a list of facts in A-Prolog which are rules with empty body. By applying the Unit Propagate LP [46], we have a path in SM_Π:

$$\emptyset \Rightarrow c(x_1) \Rightarrow c(x_1), c(x_2) \Rightarrow \cdots \Rightarrow c(x_1), c(x_2), \ldots, c(x_n)$$

2. **step 1(c)**: Because we will have the following rules:

$$a(\bar{t}) = y \leftarrow do(a(\bar{t}) = y)$$

$$do(a(\bar{t}) = y)$$

for each *do* statement in the query Q, it is clear that, based on Unit Propagate LP, $M \Rightarrow M\ a(\bar{t}) = y$ is a transition in SM_Π.

3. **step 2**: Now we need to prove that $\mathcal{M}(A) \Rightarrow^* \mathcal{M}(Closure(\Pi, A))$ is a path in graph SM_Π. As we have proved the function $AtMost$ in our dissertation is equivalent to function *Atmost* in *Smodels*. We only need to show that $\mathcal{M}(A) \Rightarrow \mathcal{M}(AtLeast(\Pi, A))$.

   In function $LC(\Pi, A)$ (Called by function $AtLeast(\Pi, A)$), we have

   (a) If $A(l) = T$, then $LC(\Pi, A)(l) = T$;

   (b) If $A(l) \neq T$ and $LC(\Pi, A)(l) = T$, there exists a Unit Propagate LP transition in SM_Π such that $\mathcal{M}(A) \Rightarrow \mathcal{M}(LC(\Pi, A))$. See rule 1(a) and 1(b) in function $LC(\Pi, A)$;

   (c) If $A(l) = F$, then $LC(\Pi, A)(l) = F$;

   (d) If If $A(l) \neq F$ and $LC(\Pi, A)(l) = F$, there exists an All Rules Cancelled transition in SM_Π such that $\mathcal{M}(A) \Rightarrow \mathcal{M}(LC(\Pi, A))$.

Therefore, $\mathcal{M}(A) \Rightarrow^* \mathcal{M}(AtLeast(\Pi, A))$.

Hence, $\mathcal{M}(A) \Rightarrow^* \mathcal{M}(Closure(\Pi, A))$ is a path in graph SM_Π. In summary, $\mathcal{M}(Initialize(\Pi, Q))$ is reachable from the emptyset in graph SM_Π.

Lemma 3

As we can see in algorithm 7, a new interpretation A' is obtained by first assigning a random atom $a(\bar{t}) = y$ to truth value T and then calling function $Closure$. In line 7 in function $Branch$, A new interpretation A' is obtained from A with a random atom $a(\bar{t}) = y$ (where $A(a(\bar{t}) = y) \neq F$) assigned to T, therefore $\mathcal{M}(A) \Rightarrow \mathcal{M}(A) \cup a(\bar{t}) = y$ based on transition $Decide$ in SM_Π. We have already proved that $\mathcal{M}(A) \Rightarrow^* \mathcal{M}(Closure(\Pi, A))$. Overall, $\mathcal{M}(A) \Rightarrow^* \mathcal{M}(A')$ where $A' \in Branch(\Pi, A)$.

6.5 Proof of Theorem 3

Theorem 3

Let Π be a ground *scou* P-log program and a query $Q = F|C$ be a query to Π. Let Φ_F be the collection of all the smallest partial possible worlds of T that know the set F of formulas. Then the probability of each formula $f \in F$ w.r.t. a ground *scou* P-log program Π and a query Q can be computed by following equation:

$$P_{\Pi \cup \Pi_Q}(f) = \sum_{I=(A,w) \in \Phi_F \text{ and } A \vdash f} \mu(I) \tag{6.43}$$

PROOF:

We start with the proof with showing the following two statements:

1. Each possible world W of program $\Pi \cup \Pi_Q$ where $W \vdash f$ is labeling a consistent leaf node N which is a descendant of a node N' such that $N' \in \Phi_F$.

2. Let $\Omega_{N'}$ be all the consistent leaf nodes descendant from node N' where N' is a smallest partial possible world that knows f, then

$$\sum_{W \in \Omega_{N'}} \mu(W) = \mu(I) \tag{6.44}$$

where I is the f-interpretation that labels N'.

Prove of Statement 1

Because each consistent leaf node N of program tree T w.r.t. program $\Pi \cup \Pi_Q$ where the formula f is true in N is also a partial possible world of $\Pi \cup \Pi_Q$ that knows f. From definition of smallest partial possible world that knows f, there exists a node in T which is the smallest partial possible world of T that knows f and is an ascendant of N.

Prove of Statement 2

Because each possible world which is the super set of N' must label a consistent leaf node descended from the node N and the programming tree w.r.t. program Π is unitary, therefore

$$\mu(I) = \sum_{W \in des(I)} \mu(W) \tag{6.45}$$

where $des(I)$ is the collection of all possible worlds that descendant from a node N' labeled by I. We claim that if $I \vdash f$, then $I' \vdash f$. This is because for each $W \in des(I)$, the f-interpretation I' that labels W is an extension of I. Hence, $W \in des(I)$ if and only if $W \in \Omega_{N'}$. From Equation (6.45), we have that

$$\sum_{W \in \Omega_{N'}} \mu(W) = \mu(I) \tag{6.46}$$

Given the above two statements, the proof of Theorem 3 is listed below.

From the definition $P_{\Pi \cup \Pi_Q}(f)$, we have that

$$P_{\Pi \cup \Pi_Q}(f) = \sum_{W \vdash f} \mu(W) \tag{6.47}$$

Because of statement 1 and 2, $W \vdash f$ if and only if there is exists a node N' such that $W \in \Omega_{N'}$ and $N' \in \Phi_F$.

$$\sum_{W \vdash f} \mu(W) = \sum_{N' \in \Phi_F} (\sum_{W \in \Omega_{N'}} \mu(W)) \tag{6.48}$$

From Equation (6.46)

$$\sum_{N' \in \Phi_F} (\sum_{W \in \Omega_{N'}} \mu(W)) = \sum_{N' \in \Phi_F} \mu(I) \tag{6.49}$$

From definition of Φ_F and Equation (6.49)

$$\sum_{N' \in \Phi_F} (\sum_{W \in \Omega_{N'}} \mu(W)) = \sum_{I=(A,w) \in \Phi_F \text{ and } A \vdash f} \mu(I) \tag{6.50}$$

Therefore, from Equation (6.47), (6.48) and (6.50), we have that

$$P_{\Pi \cup \Pi_Q}(f) = \sum_{I=(A,w) \in \Phi_F \text{ and } A \vdash f} \mu(I) \tag{6.51}$$

6.6 Proof of Theorem 4

Theorem 4

Let $\Pi(\mathcal{K})$ be the P-log program obtained from a system knowledge $\mathcal{K}$. A path $p = \langle \sigma_0, e_0, \sigma_1, \ldots, e_{n-1}, \sigma_n \rangle$ is a model of H if and only if there is a possible world W of $\Pi(\mathcal{K})$ such that for any $0 \leq t \leq n$,

1. $e_t = \{e : occurs(e, t) \in W\}$
2. $h(l, t) \in W$ iff $l \in \sigma_t$

and $\hat{\mu}(W) = \hat{P}_{\mathcal{K}}(p|H)$.

PROOF:

Part 1:

We construct a set W of fluent literals from p as follows:

- for each fluent literal l and j, $0 \leq j \leq n$, if $l \in \sigma_j$, then $h(l, j) \in W$;

- for each action e_i, $0 \leq i < n$, $occurs(e, i) \in W$;
- If the knowledge of initial state is incomplete, then let c be the symbol which is mapped to σ_0 in $\Pi(\mathcal{K})$, we have that $at\_initial\_state = c \in W$;
- If for some e_i, there exists a non-deterministic dynamic causal law whose body is satisfied by σ_i, then let $l \in \{l_1, \ldots, l_n\} \cap \sigma_{i+1}$, we add that $result_r(i+1) = l$ into W;

We prove that W is a possible world of $\Pi(\mathcal{K})$ which satisfies condition (1) and (2).

Let $W(i)$ where $0 \leq i \leq n$ be a subset of W such that $W(i) = \{h(l, j) : h(l, j) \in W \text{ and } j \leq i\} \cup \{occurs(e, j) : occurs(e, j) \in W \text{ and } j < i\} \cup \{result_r(j) = l : result_r(j) = l \in W \text{ and } j \leq i\} \cup \{at\_initial\_state = c : at\_initial\_state = c \in W\}$. By $\Pi(\mathcal{K})_{W(i)}$, $0 \leq i \leq n$, we denote a P-log program such that if a rule $r \in \Pi(\mathcal{K})_{W(i)}$ if and only if $r \in \Pi(\mathcal{K})$ and all literals in r belong to $W(i)$.

Now we use induction on i to prove that for each i, $0 \leq i < n$, $W(i)$ is a possible world of $\Pi(\mathcal{K})_{W(i)}$.

Base case $i = 0$:

If the knowledge of initial state is complete, then

$$\Pi(\mathcal{K})_{W(0)} = \{h(l, 0) \leftarrow : l \in \sigma_0\} \cup \{occurs(e, 0) \leftarrow\}$$

By definition,

$$W(0) = \{h(l, 0) : l \in \sigma_0\} \cup \{occurs(e, 0)\}$$

It is clear that $W(0)$ is a possible world of $\Pi(\mathcal{K})_{W(0)}$.

If the knowledge of initial state is incomplete, we have following rules in $\Pi(\mathcal{K})_{W(0)}$:

- For each **observe** l **at** $0 \in H$, $obs(h(l, 0)) \leftarrow \in \Pi(\mathcal{K})_{W(0)}$;
- $[ini]$ $random(at\_initial\_state) \in \Pi(\mathcal{K})_{W(0)}$;

- $occurs(e, 0) \leftarrow\in \Pi(\mathcal{K})_{W(0)}$;
- For each literal $l \in \sigma_0$, $h(l, 0) \leftarrow at\_initial\_state = c \in \Pi(\mathcal{K})_{W(0)}$, where c is a symbol which is mapped to σ_0.

By definition,

$$W(0) = \{h(l, 0) : l \in \sigma_0\} \cup \{occurs(e, 0)\} \cup \{at\_initial\_state = c\}$$

It is not difficult to see that $W(0)$ is a possible world of $\Pi(\mathcal{K})_{W(0)}$.

Induction hypothesis: Suppose for some j, $0 \leq j < n$, $W(j)$ is a possible world of $\Pi(\mathcal{K})_{W(j)}$. we will manage to show that $W(j+1)$ is a possible world of $\Pi(\mathcal{K})_{W(j+1)}$. To achieve that, firstly, we shall prove that for each rule in $\Pi(\mathcal{K})_{W(j+1)}$ it is satisfied by $W(j+1)$.

- For each rule $h(l, j+1) \leftarrow h(p, j+1) \in \Pi(\mathcal{K})_{W(j+1)}$ whose body is satisfied by $W(j+1)$:

  From the definition of $\Pi(\mathcal{K})_{W(j+1)}$, we have

  $$h(l, j+1) \leftarrow h(p, j+1) \in \Pi(\mathcal{K})_{W(j+1)} \Rightarrow l \textbf{ if } p \in \mathcal{K} \tag{6.52}$$

  By the definition of W,

  $$h(p, j+1) \subseteq W \Rightarrow \ p \subseteq \sigma_{j+1} \tag{6.53}$$

  From (6.52), (6.53) and definition of state, we have that

  $$l \in \sigma_{j+1} \tag{6.54}$$

  From the definition of $W(j+1)$ and (6.54)

  $$h(l, j+1) \in W(j+1) \tag{6.55}$$

  Hence, each rule $h(l, j+1) \leftarrow h(p, j+1) \in \Pi(\mathcal{K})_{W(j+1)}$ is satisfied by $W(j+1)$.

- Similarly, it is easy to see that for each rule

$$h(l, j+1) \leftarrow occurs(e, j), h(p, j) \in \Pi(\mathcal{K})_{W(j+1)}$$

  it is satisfied by $W(j+1)$.

- For inertia axioms r: $h(l, j+1) \leftarrow h(l, j), \textbf{not } \neg h(l, j+1)$: If the body of r is satisfied by $W(j+1)$, then $\neg h(l, j+1) \notin W(j+1)$. By the definition of $W(j+1)$, we have that $\neg l \notin \sigma_{j+1}$. By the definition of state, we have that $l \in \sigma_{j+1}$. Hence, $h(l, j+1) \in W(j+1)$.

- For logical rule: $h(l, j+1) \leftarrow result_r(j+1) = l$. By definition of $W(j+1)$, $result_r(j+1) = l \in W(j+1)$ iff $l \in \sigma_{j+1}$. Hence $h(l, j+1) \in W(j+1)$.

- If $[r]$ $random(result_r(j+1)) \leftarrow occurs(e, t), h(p, t) \in \Pi(\mathcal{K})_{W(j+1)}$, and its body is satisfied by $W(j+1)$. Recall that in this case, rule r is satisfied by $W(j+1)$ if there exists only one $l \in range(result_r)$ such that $result_r(j+1) = l \in W(j+1)$. Because the body of r is satisfied by $W(j+1)$ iff the body of corresponding non-deterministic dynamic causal law is satisfied by σ_j. By the definition of W, $result_r(j+1) = l \in W(j+1)$.

- For a rule $h(l, j+1) \leftarrow result_r(j+1) = l$, if $result_r(j+1) = l \in W(j+1)$, then $l \in \sigma_{i+1}$. therefore, $h(l, j+1) \in W$. Hence the rule $h(l, j+1) \leftarrow result_r(j+1) = l$ is satisfied by $W(j+1)$.

In summary, we know that all the rules in $\Pi(\mathcal{K})_{W(j+1)}$ is satisfied by $W(j+1)$.

To prove that $W(j+1)$ is the minimal set that satisfies $\Pi(\mathcal{K})_{W(j+1)}$, we use contradictory. Suppose that there exists a set $W' \subset W(j+1)$ such that W' also satisfies all the rules in $\Pi(\mathcal{K})_{W(j+1)}$. We assume that for some fluent literal l, $h(l, j+1) \in W(j+1)$ and $h(l, j+1) \notin W'$. Because $h(l, j+1) \in W(j+1)$, $\neg h(l, j+1) \notin W(j+1)$. Therefore, neither $h(l, j+1)$ nor $\neg h(l, j+1)$ belongs to W'. Because there must be either $h(l, j)$ or $\neg h(l, j)$ belongs to W'. We can see that the body of one of inertia axioms must be satisfied. Since the head $h(l, j+1) \notin W$ therefore, W' is not a possible

world.

It is trivial to see that if other literals, such as $occurs(e, j+1)$ and $result_r(j+1) = l$ are missing in W' then W' is not a possible world of $\Pi(\mathcal{K})_{W(j+1)}$. Therefore, $W(j+1)$ is a possible world of $\Pi(\mathcal{K})_{W(j+1)}$.

Overall, for each i, $0 \leq i < n$, $W(i)$ is a possible world of $\Pi(\mathcal{K})_{W(i)}$.

It is clear that $W(n) = W$ and $\Pi(\mathcal{K}) = \Pi(\mathcal{K})_{W(n)}$. Therefore W is a possible world of $\Pi(\mathcal{K})$.

From the definition of W, It is clear that both conditions (1) and (2) are satisfied by W and the path p.

Part 2:

Let W be a possible world of $\Pi(\mathcal{K})$. Let $\sigma(W, i)$ be the collection of fluent literals l such that $h(l, i) \in W$. Let $a(W, i)$ be an action e such that $occurs(e, i) \in W$. We prove that $\langle \sigma(W, 0), a(W, 0), \ldots, a(W, n-1), \sigma(W, n) \rangle$ is a path of SD which is a model of H and condition (1) and (2) are satisfied.

The proof consists of four steps:

1. We prove that $\sigma(W, j)$ is a state of the transition diagram.
2. We prove that for each j where $0 \leq j < n$, $\langle \sigma(W, j), a(W, j), \sigma(W, j+1)$ is a transition of the system.
3. We prove that $\langle \sigma(W, 0), a(W, 0), \sigma(W, 1), a(W, 1), \ldots, a(W, n-1), \sigma(W, n) \rangle$ is a path which is a model of H.
4. This path p and the possible world W satisfies condition (1) and (2).

Step 1

By definition of answer set, the possible world W cannot be a set of literals in which both $h(l, j)$ and $\neg h(l, j)$ belong to W. Hence, we should only need to prove that at

least one of $h(l, j)$ and $\neg h(l, j)$ belong to W for each l and j. We use induction on the step of path to prove this claim:

For the base, where $i = 0$, there are two possible cases:

1. The knowledge of initial state is complete: in this case, for each fluent $f \in \sigma_0$ there is an observation for f or $\neg f$. This statement is translated into the fact:

   $$h(l, 0) \leftarrow$$

   where l is f or $\neg f$ respectively. Therefore, at least one of $h(l, 0)$ and $\neg h(l, 0)$ belong to W.

2. The knowledge of initial state is incomplete: According to our translation, W must contains exactly one atom $initial\_at = i$. Therefore, for each fluent $f \in \Sigma$, either

   $$h(f, 0) \leftarrow initial\_at = i \in \Pi(\mathcal{K})$$

   or

   $$\neg h(f, 0) \leftarrow initial\_at = i \in \Pi(\mathcal{K})$$

Because W is possible world of $\Pi(\mathcal{K})$, for each rule r

$$h(l, 0) \leftarrow h(p, 0)$$

r is satisfied by W. This means if $p \subseteq \sigma(W, 0)$ then $l \in \sigma(W, 0)$. Therefore $\sigma(W, 0)$ is consistent. Hence, for $i = 0$, $\sigma(W, 0)$ is a state.

Induction hypothesis: Suppose that for some j, $0 \leq j \leq n - 1$, $\sigma(W, j)$ is a state of transition diagram. We will show that $\sigma(W, j + 1)$ is also a state.

Because $\sigma(W, j)$ is a state of T, at least one of $h(f, j)$ and $\neg h(f, j)$ belong to W for each f. Suppose that for some f that $h(f, j + 1) \notin W$ and $\neg h(f, j + 1) \notin W$. If $h(f, j) \in W$ then the body of the inertia axiom

$$h(f, j + 1) \leftarrow h(f, j), \textbf{not } \neg h(f, j + 1)$$

is satisfied, therefore $h(f, j+1) \in W$. Contridictory. Similar results can be derived from the case where $\neg h(l, j) \in W$. Therefore, at least one of $h(f, j+1)$ and $\neg h(f, j+1)$ belong to W for each f.

Conclusion: for each fluent $f \in \Sigma$ and step $j+1$ where $0 \leq j < n$, one and only one of f and $\neg f$ belongs to $\sigma(W, j+1)$

By our translation, for each static causal laws in SD of $\mathcal{K}$, there exist a rule $h(l, T) \leftarrow h(p, T) \in \Pi(\mathcal{K})$. By the definition of answer sets, the possible world W must satisfies each of these rules. Hence $\sigma(W, j+1)$ $(0 \leq j < n)$ is consistent w.r.t. the set of static causal laws, which means $\sigma(W, j+1)$ is a state of T.

In summary, for each j, $0 \leq j < n$, $\sigma(W, j)$ is a state of the transition diagram.

Step 2

By the definition of transition diagram, this is equivalent to prove that for each fluent literal l, $l \in Cn(E'(e_j, \sigma(W, j)) \cup (\sigma(W, j) \cap \sigma(W, j+1))$ iff $l \in \sigma(W, j+1)$. We present the prove with two steps.

1. Step 2a: If $l \in Cn(E'(e_j, \sigma(W, j) \cup (\sigma(W, j) \cap \sigma(W, j+1))$, then $l \in \sigma(W, j+1)$.
2. Step 2b: If $l \in \sigma(W, j+1)$, then $l \in Cn(E'(e_j, \sigma(W, j)) \cup (\sigma(W, j) \cap \sigma(W, j+1))$

Step 2a:

- **case 1:** $l \in E'(e_j, \sigma(W, j))$. Let r be a dynamic causal law in which l is the head of r and the body p of r is satisfied by $\sigma(W, j)$. Since

  $$h(l, T+1) \leftarrow occurs(e, T), h(p, T) \in \Pi(\mathcal{K})$$

  and $occurs(e, T) \in W$ and according to the definition of $\sigma(W, j)$, we have that $p \subseteq \sigma(W, j) \Rightarrow h(p, t) \subseteq W$, replacing varialbe T by constant j, we have that $h(l, j+1) \in W$. That is $l \in \sigma(W, j+1)$.

  If r is a non-deterministic causal law and l is a fluent literal in the head of r such that $l \in \sigma(W, j+1)$ and the body p of r is satisfied by $\sigma(W, j)$. It is not

difficult to see that the body of the random selection rule is satisfied by W. By the semantics of P-log, there exists one and only one fluent literal l' such that $result_r(j+1) = l' \in W$. Because $l' \in \{l_1, \ldots, l_n\}$, therefore $l' = l$. Hence $h(l, j+1) \in W \Rightarrow l \in \sigma(W, j+1)$.

- **case 2:** $l \in \sigma(W,j) \cap \sigma(W,j+1)$. Clearly, $h(l,j+1) \in W$. Therefore $l \in \sigma(W, j+1)$.
- **case 3:** $l \notin (\sigma(W,j) \cap \sigma(W,j+1)) \cup E'(e_j, \sigma(W,j))$ and $l \in Cn(E'(e,\sigma_j) \cup (\sigma_j \cap \sigma_{j+1}))$. Because for each literal $l' \in (\sigma(W,j) \cap \sigma(W,j+1)) \cup E'(e_j, \sigma(W,j))$, we have proved that $l \in \sigma(W, j+1)$. It is easy to see that for every static causal law r whose body is satisfied by $\sigma(W, j+1)$, the body of its corresponding rules in P-log program are satisfied by $\sigma(W, j+1)$. Therefore, $l \in Cn(E'(e,\sigma_j) \cup (\sigma_j \cap \sigma_{j+1})) \Rightarrow l \in \sigma(W, j+1)$.

In summary, if $l \in Cn(E'(\sigma(W,j), e_j) \cup (\sigma(W,j) \cap \sigma(W,j+1))$, then $l \in \sigma(W, j+1)$.

Step 2b:

A fluent literal l can belong to $\sigma(W, j+1)$ because of one of the following four rules:

1. $h(l, j+1) \leftarrow occurs(e,j), h(p,j) \in \Pi(\mathcal{K})$ where $h(p,j) \subseteq W$ and $occurs(e,j) \in W$: In this case, we can see that there exists a dynamic causal law

   $$e \textbf{ cause } l \textbf{ if } p \in SD$$

   Therefore, $l \in E'(e, \sigma_j) \Rightarrow l \in \sigma_{j+1}$.

2. $h(l, j+1) \leftarrow h(l,j), \textbf{not } \neg h(l, j+1) \in \Pi(\mathcal{K})$ whose body is satisfied by W. Clearly, that $h(l, j+1)$ and $h(l,j)$ belong to W. Therefore, $l \in \sigma(W,j) \cap \sigma(W, j+1)$, which means that $l \in Cn(E'(e_j, \sigma(W,j)) \cup (\sigma(W,j) \cap \sigma(W,j+1))$.

3. $h(l, j+1) \leftarrow h(p, j+1) \in \Pi(\mathcal{K})$ where $h(p, j+1) \subseteq W$. Form lemma 1, we have that $l \in Cn(E'(e_j, \sigma(W,j)) \cup (\sigma(W,j) \cap \sigma(W,j+1))$.

4. $h(l, j+1) \leftarrow result_r(j+1) = l$ where $result_r(j+1) = l \in W$. Because $result_r(j+1) = l \in W$, there exists a non-deterministic dynamic causal law in SD whose body is satisfied by $\sigma(W, j)$. Since l is the head of such rule, $l \in E'(e, \sigma(W, j))$ and therefore $l \in Cn(E'(e_j, \sigma(W, j)) \cup (\sigma(W, j) \cap \sigma(W, j+1))$.

Step 3:

To prove that the path p is a model of H, we know that **observe** l **at** $j \in H$ iff $obs(h(l, j)) \leftarrow\in \Pi(\mathcal{K})$. According to the semantics of P-log $obs(h(l, j)) \leftarrow\in \Pi(\mathcal{K})$ iff $\leftarrow$ **not** $h(l, j) \in \tau(\Pi(\mathcal{K}))$ where τ is a translation from P-log program to Answer Set Prolog defined in [1]. Clearly, for every possible world W of $\Pi(\mathcal{K})$, $h(l, j) \in W$. Hence, $l \in \sigma(W, j)$. By definition of model of history, p is a model of H.

Step 4:

Condition (1) and (2) are obviously satisfied because of the way we define $a(W, j)$ and $\sigma(W, j)$.

Part 3

The probability equivalence between path p and possible world W can be established based on the following claims:

1. For each atom $result_r(i) = l \in W$, $P(result_r(i) = l, W) = P_{a(W,i)}(l, \sigma(W, i))$; From the definition of $P(result_r(i) = l, W)$ and the definition of $P_{a(W,i)}(l, \sigma(W, i))$, this claim is easy to see.

2. For each atom $at_initial_state = c \in W$ where c is mapped to $\sigma(W, 0)$, we have $P(at\_initial\_state = c, W) = P_s(\sigma(W, 0))$ where s is partial state defined by H.

3. $result_r$ and $at\_initial\_state$ are the only random attributes in $\Pi(\mathcal{K})$.

From the definition of $P_T(p)$ and the unnormalized measure of possible world, we can derive that $\mu(W) = \hat{P}_{\mathcal{K}}(p)$. From the definition of normalized measure of possible world and the definition of $\hat{P}_{\mathcal{K}}(p|H)$, we have that $\hat{\mu}(W) = \hat{P}_{\mathcal{K}}(p|H)$.

6.7 Proof of Theorem 5

Theorem 5

Let $\Pi(\mathcal{K})$ be a P-log program obtained from a domain description $\mathcal{K}$ of a probabilistic planning problem $\langle \mathcal{K}, s_0, s_f \rangle$. An action sequence $o(0) = a_0, \ldots, o(n-1) = a_{n-1}$ is a best probabilistic plan if and only if the formula: $o(0) = a_0 \wedge \cdots \wedge o(n-1) = a_{n-1}$ is in the answer of the query $< o(0), \ldots, o(n-1) > |obs(goal)$ with respect to the P-log program $\Pi(\mathcal{K}) \cup \Pi_{goal}$.

PROOF:

The probability of $\alpha = (o(0) = a_0, \ldots, o(n-1) = a_{n-1})$ with respect to the program $\Pi(\mathcal{K}) \cup \Pi_{goal}$ is calculated as follows:

$$P_{\Pi(\mathcal{K}) \cup \Pi_{goal} \cup \{\leftarrow obs(goal)\}}(\alpha) = \frac{\sum\limits_{W \vdash \alpha, W \in \Omega} \mu(W)}{\sum\limits_{W \in \Omega} \mu(W)}$$

Notice that $\sum\limits_{W \in \Omega} \mu(W)$ is a constant. Let p_W be the path in a possible world W. We have

$$\sum_{W \vdash \alpha, W \in \Omega} \mu(W) = \sum_{W \vdash \alpha, W \in \Omega} P_{s_0}(p) \times \prod_{0 \le i < n} P(W, o(i) = a_i)$$

Since $\prod_{0 \le i < n} P(W, o(i) = a_i)$ is a constant given fixed number of actions in the domain and fixed length of the plan, we have

$$\sum_{W \vdash \alpha, W \in \Omega} \mu(W) = c \sum_{W \vdash \alpha, W \in \Omega} P_{s_0}(p)$$

where $c = \prod_{0 \le i < n} P(W, o(i) = a_i)$. Because we know that: $W \vdash \alpha$ if and only if $p \in PT(\alpha, s_0, s_f)$, therefore

$$c \sum_{W \vdash \alpha, W \in \Omega} P_{s_0}(p) = c \times P_{s_0}(s_f | \alpha, s_0)$$

Hence,

$$P_{\Pi(\mathcal{K}) \cup \Pi_{goal} \cup \{\leftarrow obs(goal)\}}(\alpha) = c \times P_{s_0}(s_f | \alpha, s_0)$$

We conclude that, for any plan β, $P_{s_0}(s_f|\alpha, s_0) \geq P_{s_0}(s_f|\beta, s_0)$, if and only if

$$P_{\Pi(\mathcal{K})\cup\Pi_{goal}\cup\{\leftarrow obs(goal)\}}(\alpha) \geq P_{\Pi(\mathcal{K})\cup\Pi_{goal}\cup\{\leftarrow obs(goal)\}}(\beta)$$

If a plan α is an answer of query $< o(0), \ldots, o(n-1) > |obs(goal)$ with respect to the P-log program $\Pi(\mathcal{K}) \cup \Pi_{goal}$, there's no such plan β such that

$$P_{\Pi(\mathcal{K})\cup\Pi_{goal}\cup\{\leftarrow obs(goal)\}}(\alpha) < P_{\Pi(\mathcal{K})\cup\Pi_{goal}\cup\{\leftarrow obs(goal)\}}(\beta)$$

Therefore, there is no such plan β such that

$$P_{s_0}(s_f|\alpha, s_0) < P_{s_0}(s_f|\beta, s_0)$$

This means the plan α is a best probabilistic plan.

Chapter 7

RELATED WORK

In this chapter, we describe the related work in fields of algorithms for combining logic and probabilistic reasonings, action languages with non-deterministic dynamic domains, probabilistic diagnostic reasonings and probabilistic planning.

7.1 Probabilistic Reasoning Algorithms

For the last several decades, many studies have been conducted on how to combine probabilistic reasoning with logical based reasoning. In late 80's, the Bayesian network became one of the most popular methods for representing probabilistic reasoning problems. In the following decades, algorithms designed for fast computation of Bayesian network are proposed in several literatures including [12, 13, 14].

Meanwhile, several research work has shown that some algorithms could improve the performance of Bayesian networks inference engine, if the network contains certain local structures or a lot of deterministic information, i.e., a lot of 0 and 1's in the conditional probability table. Those work includes 1) compiling a Bayesian network to arithmetic circuits and 2) transforming Bayesian network to weighted model checking problem.

The idea of compiling Bayesian network to arithmetic circuits was introduced in [16]. By compiling the Bayesian network in an off-line stage, it can improve the inference time in the on-line stage. The authors proposed a new encoding scheme that facilitates the representation of local structure in the form of parameter equality, and identified some of its properties that improve compile time. Comparing to classical methods used for Bayesian networks, it has shown significant improvement over its

performance.

In [18], they proposed an algorithm that encodes the Bayesian network to weighted proposition formulas. By taking the advantage of state-of-art SAT solvers, they are able to compute the models rather quickly and computes the probability of formulas by model counting. The performance of this approach largely depends on the percentage of determinism in the Bayesian network. According to their experiments, if over 90% of the entries of conditional probability tables of a Bayesian network are 0's and 1's, then such encoding may improve the overall performance of the inference.

The closest work on P-log inference engine is another system developed in [22]. The system is very similar to the translation based system we have discussed in this dissertation. Besides introducing some new syntax for concise representation of certain knowledge, the system uses XSB [35] with XASP[36] to allowing using Prolog rules. Like the system ACE, the inference procedure is divided into two stages: an off-line computation stage where each encoded possible worlds are computed and an on-line stage where probability of formulas is computed. Because of the separation of computing possible worlds and computing probability of formulas, the overall performance of their system is better than *plog*1.0 when a number of queries are proposed w.r.t. a fixed P-log program. However, their system may only be able to deal with P-log programs with less possible worlds, they reported that when the number of possible worlds is large, their system may not return the answer in given time due to large IO operations and limitation of memory. Because *plog*2.0 reduce the size of the ground P-log program w.r.t. the query, the number of possible worlds computed in *plog*2.0 on the same problem may be much less than their system does. We can handle problems with much larger domains.

7.2 Probabilistic Action Language

In [37], the author has introduced a new action description language $\mathcal{PAL}$ as an extension of language $\mathcal{AL}$. By introducing *unknown variables*, it allows dynamic causal laws and static causal laws to represent non-deterministic actions. Comparing to our language $\mathcal{NB}$ introduced in Chapter 5, in their work, the probability of unknown variable cannot be conditioned on the truth values of fluents. Therefore, they need more dynamic laws to represent the probability distributions. Furthermore, since the truth values of unknown variables in the static causal laws are fixed, it is not possible to represent unknown fluents whose truth values might be changed in the transition diagram. In our extension, we are not limited to this restriction.

7.3 Probabilistic Diagnosis

Our method for solving diagnostic problems has roots in the earlier work by Reiter [24]. In [24], the author defined a diagnosis as a minimal set of abnormality assumptions such that the observations are consistent with all others components acting normally. ([39]). The work of [24] has been extended in [38] where probabilistic information are introduced on faulty components. Comparing to our work, the logical languages used for describing the system are different. In [38], the underlying language is first order logic and while in our work, we are using P-log which is based on Answer Set Prolog. Furthermore, in [38], based on the method they use for computing the joint probabilities over faulty components, they assumed that the probability of faulty components are independent of each other. In our work, we don't have such assumptions on faulty components.

Our work also has some similarity with [33]. In fact, much of our key definitions are borrowed from [33]. However, the fundamental difference between these two works lie on the view of how the components become faulty. In [33], the system become faulty because there are exogenous actions that the agent is unaware of and those actions

are causing certain components faulty. We ignored the cause of faulty components and considered them unchanged during the history of the system.

7.4 Probabilistic Planning

The work presented in [28] solved the same planning problem as we do in this research. But their approaches are very different from ours. Firstly, they represent the planning domain with so called sequential-effects-tree (ST) representation [40], which is a syntactic variant of two-time-slice Bayesian nets with conditional probability tables represented as trees [42]. Secondly, they solved the planning problem by converting an ST representation into an E-MAJSAT formula with the property that, given an assignment to the choice variables, the probability of a satisfying assignment with respect to the chance variables is the probability of success for the plan specified by the choice variables. The E-MAJSAT problem is solved by an extension of the Davis-Putnam-Logemann-Loveland (DPLL) procedure [41]. From their primitive experimental results, they have shown that their techniques performed better than other systems, namely BURIDAN [27] and POMDP [43], available at that time.

Our understanding of planning problems and methods of formalizing goals in the planning problems are under the influence of results in [44]. In [44], the author managed to show that how to use ASP to solve planning problems in a deterministic environment. The emphasis on representing both planning problems and diagnosis problems with same language and solving them with same inference engines helps to build a more powerful intelligent multi-task agent.

Chapter 8

CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

This dissertation investigates algorithms for P-log inference engine. In particular, we have discussed two algorithms and shown that their performance on various types of problems. We also extended an action description language to allow representing non-deterministic actions as well as statements for describing incomplete initial states. We have shown that P-log can be used for finding best probabilistic diagnosis and best probabilistic plans.

We list the major contribution of this dissertation as follows:

1. In the translation based algorithm, we have shown how probabilistic information can be encoded into logical rules such that probability of formulas can be computed from information extracted from output of existed Answer Set Prolog solver.

2. In the second algorithm, we have discussed how to selectively grounds the input program with respect to the specific query given by the user. When a smaller ground P-log program is produced, often it will result in a much smaller set of possible worlds and faster computation. We also designed the algorithms which take advantage of unitary property of P-log program and improve the performance of its inference engine.

3. We implemented and tested two P-log systems with various types of domains. We have shown that the running results of some representative domains, and discussed the advantage and disadvantage of both systems.

4. We extended the deterministic action description language $\mathcal{B}$ to a new language $\mathcal{NB}$ to allow representing non-deterministic actions. We also introduced new statement for describing incomplete initial states of transition systems. We developed methods that solves common reasoning tasks in a non-deterministic dynamic environment with P-log.

5. We defined the notion of best probabilistic diagnosis and gave a systematic method for representing such problems and shown how to use P-log to solve such problems.

6. We used P-log to solve probabilistic planning problems. We have run the preliminary tests and discussed the performance of P-log inference engine on this task.

7. Finally, we proved some of propositions and theories shown in this dissertation.

8.2 Future Work

We list our future work as follows:

- It is very common that once a P-log program is fixed, user may propose many different queries to the program. Our implementation so far has not taken advantage of this usage patterns. With proper cache techniques introduced, we should be able to avoid repeated computations for different queries. One of our future work is to add this feature so that a sequence of queries can be answered more quickly.

- If the ground program still contains a lot of random selection rules. Evaluating the probability of a formula may become infeasible. Many sophisticated algorithms, such as recursive conditioning [14], allow to compute probabilities with respect to a large scale of Bayesian networks. We are interested in how to combine these techniques with our algorithms so we may be able to solve

problems with large number of random selection rules in the program.

- The ground time of the new system is comparatively small in many domains we have tested. However in some problems, when the ground program is large, it becomes the bottle neck of the overall performance of P-log system. We expect some overhead of our grounding algorithm comparing to other systems, say $lparse$. However, we believe a better design of grounding algorithm may help us to reduce the grounding time on large programs.
- Our performance on probabilistic planning tasks is not as good as other systems. We need to investigate better heuristic functions and more sophisticated algorithms for faster performance.

REFERENCE

[1] Baral. C, M. Gelfond and N. Rushton. Probabilistic Reasoning with Answer Sets. Theory and Practice of Logic Programming (2009) 57–144.

[2] Gelfond M. and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, Logic Programming: Proceedings of the Fifth International Conf. and Symp., pages 1070-1080, 1988.

[3] Niemela. I and P. Simons. Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. Lecture Notes in Computer Science (1997) Vol. 1265 420–429.

[4] Leone. N, G. Pfeifer, W. Faber, F. Calimeri, T. Dell'Armi, T. Eiter, G. Gottlob, G. Ianni, G. Ielpa, C. Koch, S. Perri, and A. Polleres. The dlv system. In Sergio Flesca and Giovanbattista Ianni, editors, Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002), September 2002.

[5] Lierler. Y, and M. Maratea. Cmodels-2: SAT-Based Answer Set Solver Enhanced to Non-tight Programs. In Vladimir Lifschitz and Ilkka Niemela, editors, Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning Conference (LPNMR'04), volume 2923, pages 346-350. Springer Verlag, LNCS 2923, 2004.

[6] Lin. F, and Y. Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In AAAI, pages 112-117, 2002.

[7] Gebser. M, B. Kaufmann, A. Neumann and T. Schaub. clasp: A Conflict-driven Answer Set Solver. In Proceedings of Logic Programming and Non-Monotonic Reasoning (LPNMR), pages 260-265. 2007.

[8] Nogueira. N, M. Balduccini, M. Gelfond, R. Watson and M. Barry. An A-Prolog Decision Support System for the Space Shuttle. In PADL 2001 (2000) 169–183.

[9] Brooks. D.R, E. Erdem, J.W. Minett and D. Ringe. Character-based Cladistics and Answer Set Programming. In Proceedings of International Symposium on Practical Aspects of Declarative Languages pages 37-51.2005.

[10] Beierle. C, O. Dusso and G. Kern-Isberner. Modelling and Implementing a Knowledge Base for Checking Medical Invoices with DLV. In Gerhard Brewka and Ilkka Niemela and Torsten Schaub and Miroslaw Truszczynski editors, Nonmonotonic Reasoning, Answer Set Programming and Constraints. Dagstuhl Seminar Proceedings, 2005.

[11] Pearl. J. Causality - Models, Reasoning and Inference. Cambridge University Press. 2000.

[12] Pearl. J. Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann Publishers. 1988.

[13] Dechter. R and J. Pearl. Tree Clustering for Constraint Networks. Artificial Intelligence, Volume 38, Issue 3, April 1989. Pages 353-366.

[14] Darwiche. A. Recursive Conditioning. Artificial Intelligence, Volume 126, Issues 1-2, February 2001. Pages 5-41.

[15] Dechter. R. Bucket Elimination: A Unifying Framework for Probabilistic Inference. In: Proc. 12th Conference on Uncertainty in Artificial Intelligence (UAI),1996. pp. 211219.

[16] Chavira. M, A. Darwiche and M. Jaeger. Compiling Relational Bayesian networks for Exact Inference. International Journal of Approximate Reasoning. Volume 42, Issues 1-2, PGM'04, May 2006. Pages 4-20.

[17] Sang. T, P. Beame and H. Kautz. Solving Bayesian Networks by Weighted Model Counting. In: Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05) AAAI Press, 2005. Pages 475-482.

[18] Chavira. M and A. Darwiche. On Probabilistic Inference by Weighted Model Counting. Artificial Intelligence, Volume 172, Issues 6-7, April 2008. Pages 772-799.

[19] Henrion. M. Propagation Uncertainty in Bayesian Networks by Probabilistic Logic Sampling. In John F. Lemmer and Laveen N. Kanal, editors, Uncertainty in Artificial Intelligence. Elsevier/North-Holland, Amsterdam, London, New York, 1988. Pages 149 -163.

[20] http://reasoning.cs.ucla.edu/ace/

[21] Pasula. H and S. Russell. Approximate Inference for First-Order Probabilistic Languages. International Joint Conference on Artificial Intelligence. 2001, VOL 17; PART 1, pages 741-748.

[22] Anh. H, C. Ramli and C. Damasio. An Implementation of Extended P-Log Using XASP. Logic Programming: Lecture Notes in Computer Science (2008), Vol. 5366. 739–743.

[23] Gelfond. M and V. Lifschitz. Action Languages. Electronic Transactions on AI. (1998) Vol 3.

[24] Reiter. R. A Theory of Diagnosis from First Principles. Artificial Intelligence, Volume 32, Issue 1, April 1987.

[25] Pednault. E. Formulating Multi-agent, Dynamic World Problems in the Classical Planning Framework. In Michael Georgeff and Amy Lansky, editors, Reasoning about Actions and Plans. pages 47-82. Morgan Kaufmanm, San Mateo, CA, 1987.

[26] McCain. N and H. Turner. Causal Theories of Action and Change. In Proc. AAAI-07, pages 460-465, 1997.

[27] Kushmerick. N, S. Hanks and D. Weld. An Algorithm for Probabilistic Planning. Artificial Intelligence, Volume 76, Issues 1-2, Pages 239-286. July 1995.

[28] Majercik. S.M and M.L. Littman MAXPLAN: A New Approach to Probabilistic Planning. In Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS), page 86-93. 1998.

[29] Hyafil. N and F. Bacchus. Conformant Probabilistic Planning via CSPs. In Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS), page 205-214. 2003.

[30] Turner. H. Representing Actions in Logic Programs and Default Theories: A Situation Calculus Approach. In Journal of Logic Programming, Vol. 31, No. 1-3, pages 245-298, 1997.

[31] Baral. C and M. Gelfond Reasoning Agents in Dynamic Domains Kluwer International Series in Engineering and Computer Science. 2000, ISSU 597, pages 257-280. Kluwer Academic Publishers Group.

[32] Giunchiglia. E, J. Lee, V. Lifschitz, N. McCain and H. Turner. Nonmonotonic Causal Theories. Artificial Intelligence, Volume 153, Issues 1-2, March 2004, pages 49-104.

[33] Balduccini. M and M. Gelfond. Diagnostic Reasoning with A-Prolog. Theory and Practice of Logic Programming (2003), pages 425-461.

[34] M. Balduccini. Answer Set Based Design of Highly Autonomous, Rational Agents. PhD thesis, Texas Tech University, Dec 2005.

[35] Rao. P, K. Sagonas, T. Swift, D. S. Warren and J. Freire. XSB: A System for Efficiently Computing WFS. Lecture Notes in Computer Science, 1997, Issue 1265, pages 430-440, SPRINGER VERLAG KG.

[36] Castro. L, T. Swift and D.S. Warren. Xasp: Answer Set Programming with Xsb and Smodels. http://xsb.sourceforge.net/packages/xasp.pdf

[37] Baral. C, N. Tran, and L.-C. Tuan. Reasoning about Actions in a Probabilistic Setting. Proceedings of the National Conference on Artificial Intelligence. 2002, No. 18, pages 507-512. MIT Press.

[38] Kohlas. J, P.A. Monney, B. Anrig and R. Haenni. Model-based Diagnostics and Probabilistic Assumption-based Reasoning. Artificial Intelligence. Vol. 104, Issues 1-2, pages 71 -106, 1998.

[39] Poole. D. Representing Knowledge for Logic-based Diagnosis. Proceedings of the International Conference of Fifth Generation Computer Systems. 1998.

[40] Littman. M.L. Probabilistic Propositional Planning: Representations and Complexity. In Proceedings of the Fourteenth National Conference on Artificial Intelligence. Pages 748 - 754, 1997. AAAI Press. The MIT Press.

[41] Davis. M, G. Logeman and D. Loveland. A Machine Program for Theorem-Proving. Communication of the ACM. VOL 5. Issue 7. pages 394–397, 1962.

[42] Boutilier. C, R. Dearden and M. Goldszmidt Exploiting Structure in Policy Construction. In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence. Page 1104-1113. 1995.

[43] Cassandra. A, M.L. Littman and N.L. Zhang. Incremental Pruning: A Simple Fast, Exact Method for Partially Observable Markov Decision Processes. In Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-97). Page 54-61, 1997. Morgan Kaufmann Publishers.

[44] Balduccini. M, M. Gelfond, R. Watson and M. Nogueira. The USA-Advisor: A case study in Answer Set planning. Lecture Notes in Computer Science. Issue 2173, pages 439-442, 2001.

[45] Simons. P. Extending the Stable Model Semantics with More Expressive Rules. Lecture Notes in Computer Science. Logic Programming and Nonmonotonic Reasoning, Issue 1730 pages 305 -316, 1999.

[46] Lierler. Y. Abstract Answer Set Solvers. In Proceedings of the 24th International Conference on Logic Programming (ICLP08). Page 377-391, Dec 2008.

[47] http://www.cs.ttu.edu/ wzhu/dissertation