Abstract

Title of dissertation:	Stochastic Reasoning with Action Probabilistic Logic Programs
	Gerardo Ignacio Simari, Doctor of Philosophy, 2010
Dissertation directed by:	Professor V.S. Subrahmanian Department of Computer Science

In the real world, there is a constant need to reason about the behavior of various entities. A soccer goalie could benefit from information available about past penalty kicks by the same player facing him now. National security experts could benefit from the ability to reason about behaviors of terror groups. By applying behavioral models, an organization may get a better understanding about how best to target their efforts and achieve their goals.

In this thesis, we propose action probabilistic logic (or ap-) programs, a formalism designed for reasoning about the probability of events whose inter-dependencies are unknown. We investigate how to use ap-programs to reason in the kinds of scenarios described above. Our approach is based on probabilistic logic programming, a well known formalism for reasoning under uncertainty, which has been shown to be highly flexible since it allows imprecise probabilities to be specified in the form of intervals that convey the inherent uncertainty in the knowledge. Furthermore, no independence assumptions are made, in contrast to many of the probabilistic reasoning formalisms that have been proposed. Up to now, all work in probabilistic logic programming has focused on the problem of entailment, i.e., verifying if a

given formula follows from the available knowledge. In this thesis, we argue that other problems also need to be solved for this kind of reasoning. The three main problems we address are: Computing most probable worlds: what is the most likely set of actions given the current state of affairs?; answering abductive queries: how can we effect changes in the environment in order to evoke certain desired actions?; and reasoning about promises: given the importance of promises and how they are fulfilled, how can we incorporate quantitative knowledge about promise fulfillment in ap-programs?

We address different variants of these problems, propose exact and heuristic algorithms to scalably solve them, present empirical evaluations of their performance, and discuss their application in real world scenarios.

STOCHASTIC REASONING WITH ACTION PROBABILISTIC LOGIC PROGRAMS

by

Gerardo Ignacio Simari

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2010

Advisory Committee:

Professor V.S. Subrahmanian, Chair/Advisor
Professor Samir Khuller
Professor Sarit Kraus
Professor Dana Nau
Professor Jonathan Wilkenfeld

© Copyright by
Gerardo Ignacio Simari
2010

Dedication

To my family: María Vanina, Amalia, Guillermo, and Patricio

Acknowledgements

First of all, I would like to thank my family for their unconditional support, not only throughout my studies but also in every aspect of my life. I am therefore most grateful to my wife María Vanina, my parents Amalia and Guillermo, and my brother Patricio. I certainly couldn't have done this without them.

I want to thank Professor V.S. Subrahmanian for these years of guidance and advice. His seemingly infinite source of ideas and implacable energy were very important forces in the development of this work. I am sure that his influence will continue to positively affect my academic life in years to come. Thanks are also due to Professors Samir Khuller, Sarit Kraus, Dana Nau, and Jonathan Wilkenfeld for agreeing to serve on my thesis committee, which meant putting valuable time aside to review the manuscript and provide feedback.

My fellow students, visiting scholars, and other members of the Laboratory for Computational Cultural Dynamics have also played an important role in my years at Maryland. Working with Amy Sliva, Paulo Shakarian, Austin Parker, Matthias Bröcheler, Francesco Parisi, Andrea Pugliese, John Dickerson, Avigdor Gal, and Sarit Kraus has taught me a lot about what can be accomplished with good teamwork. A special thanks goes to Amy Sliva, Paulo Shakarian, and Matthias Bröcheler for being excellent office mates, as well as Carlos Castillo for his friendship and all the good times while we were both students at Maryland.

Finally, I want to thank my oldest friends back in Argentina: Patricio Varela, Nieves Agesta, Facundo Osre, Pablo Spagnoli, Carolina Heredia, Natalia Abad, Clara Sellan, and Lucas Tenconi (*"Los de Siempre"*), as well as Emiliano Farías, Angela Cesetti, and Telma Delladio. Their friendship made all the difference.

Contents

1 Introduction **1**
1.1 Action Probabilistic Logic Programs . . . 2
1.2 Applications . . . 3
1.2.1 Promises . . . 4
1.2.2 Sports . . . 4
1.2.3 Groups of Interest . . . 8
1.3 Automated Extraction of Rules . . . 10
1.4 Towards a "Big Picture" . . . 11
1.5 Organization of this Thesis . . . 14

2 Action Probabilistic Logic Programs: Preliminary Notions **18**
2.1 Introduction . . . 18
2.2 Syntax and Semantics . . . 20
2.3 A Fixpoint Operator . . . 26

3 Related Work **31**
3.1 Probabilistic Logic . . . 32
3.1.1 From Leibniz to Boole . . . 32
3.1.2 More Recent Developments . . . 33
3.2 Probabilistic Logic Programming and Related Formalisms . . . 37
3.2.1 The Origins of Probabilistic Logic Programming . . . 37
3.2.2 Further Developments . . . 39
3.3 A Comparison of *ap*-programs to other Approaches to Probabilistic Reasoning . . . 42
3.4 Probabilistic Abduction . . . 45
3.5 Reasoning about Promises, Trust, and Reputation in Autonomous Agents . . . 48
3.5.1 Trust and Reputation in "offer, accept, reject" Negotiations . 48
3.5.2 Other Related Work . . . 52
3.6 Reasoning about Adversaries . . . 52

4 Computing Most Probable Worlds in Action Probabilistic Logic Programs **54**
4.1 Maximally Probable Worlds . . . 56
4.2 Exact Algorithms for finding a Maximally Probable World . . . 61

4.2.1 HOP: Head-Oriented Processing . . . 63
4.2.2 Enhancing HOP: The SemiHOP Algorithm . . . 71
4.3 The Binary Heuristic . . . 74
4.4 Implementation and Experiments . . . 78
4.5 Concluding Remarks . . . 83

5 Focused Most Probable World Computations in Action Probabilistic Logic Programs 87
5.1 Introduction . . . 88
5.2 Preliminaries . . . 90
5.3 A new Linear Program Formulation for Worlds of Interest . . . 91
5.3.1 Refining the Set of Constraints . . . 96
5.3.2 Refinement Algorithms based on Random Sampling . . . 104
5.3.3 Finding Most Probable Worlds of Interest . . . 115
5.4 Experimental Evaluation . . . 116
5.5 Concluding Remarks . . . 124

6 Abductive Inference in Action Probabilistic Logic Programs 127
6.1 Introduction . . . 128
6.2 Basic Abductive Queries to Probabilistic Logic Programs . . . 130
6.3 Algorithms for BAQA . . . 135
6.4 Experimental Results . . . 151
6.5 Concluding Remarks . . . 154

7 Abductive Inference Taking the Adversary into Account 156
7.1 Cost-based Query Answering (CBQA) . . . 158
7.2 An Exact Algorithm for CBQA . . . 160
7.3 A Heuristic Algorithm based on Iterative Sampling of Solutions . . . 164
7.4 Experimental Results . . . 167
7.5 Concluding Remarks . . . 178

8 Advanced Applications of *ap*-programs: Taking Promise Fulfillment into Account 179
8.1 Introduction and Motivating Example . . . 181
8.2 Preliminaries . . . 184
8.3 A Distance Measure between Atoms . . . 188
8.3.1 Distance between Two Action Atoms . . . 188
8.3.2 Distance between a *Promise* and a *Do* Atom . . . 191
8.4 A Function to Measure Degree of Fulfillment . . . 193
8.5 Application and Experiments . . . 203
8.5.1 The US Airline On-Time Performance Dataset . . . 204
8.5.2 Empirical Results . . . 205
8.6 Discussion . . . 209
8.7 Concluding Remarks . . . 210

9 Conclusions **213**

Bibliography **217**

Index **233**

Chapter 1

Introduction

In the real world, there is a constant need to reason about other entities of diverse origin. For example, a goalie in a soccer match wishing to stop a penalty kick could greatly benefit from information available about past kicks taken by the same player or team facing him now. In a more complex setting, a World Bank loan aimed at reducing the cultivation of opium along the Pakistan Afghanistan border would greatly benefit from a socio-economic-political-religious model of the behaviors of the tribes in opium producing regions. By building such models and applying them, the World Bank may get a better understanding about how best to target their loan dollars in order to better achieve their goals. Likewise, a health care organization anxious about the spread of diarrhea (or any other disease) in Kenya might wish to understand socio-economic-cultural-environmental aspects of Kenyan society that cause the diseases to spread extensively in some parts of the country and not in others. In almost all cases, the spread of diseases is not due to biological factors alone, but due to a rash of social behaviors, environmental factors, and economic and educational aspects of the disease-stricken community.

In this thesis, we propose *action probabilistic logic programs* (henceforth many times referred to as *ap*-programs), a formalism designed for reasoning about the probability of certain kinds of events. The main goal is to investigate how we can make use of *ap*-programs to reason in the kind of scenarios described above. We will begin by giving a brief overview of *ap*-programs in the following section.

1.1 Action Probabilistic Logic Programs

Probabilistic logic programming [NS92, NS91, JHdAa90, DS97, Luk98, LKI99] is a well known formalism for reasoning in the presence of uncertainty that is based on probabilistic extensions to classical logic [Nil86, Hai84, FHM90, Hal90]. It has been shown to be highly flexible, since it allows imprecise probabilities to be specified in the form of intervals that convey the uncertainty inherent in the knowledge. Furthermore, it is not necessary to make any assumptions about the knowledge regarding dependencies among events (or lack thereof), so the (not uncommon) situation of *total ignorance* can be modeled adequately; this is in contrast to other formalisms (such as Bayesian Networks) which require conditional independence assumptions to be made when constructing the model.

Up to now, all work in probabilistic logic and probabilistic logic programming has focused on the problem of *entailment* [Nil86], *i.e.*, given a set of formulas deciding whether or not a certain logical formula (not in the original set) can be entailed from the established knowledge base with given probability bounds. The naive approach to solving this problem can be shown to be exponential in the number of ground atoms in the domain, since a linear program specifying the constraints placed by the formulas on every possible world must be built; this has motivated much work

$A : [L, U] \leftarrow C_1 \wedge \ldots \wedge C_n$.

"If the environment in which entity E operates currently satisfies certain conditions $C_1 \ldots C_n$, then the probability that E will take some boolean combination of actions A is between L and U".

Figure 1.1: Formal (above) and informal (below) representation of an action probabilistic rule.

to be focused on making these inferences tractable [JHdAa90, FH94]. Even though entailment plays a very important role in reasoning under uncertainty, we argue that the problem of finding the *most probable worlds* (or most probable models), as well as *abductive query answering*, are also major parts of this process. In this work, we present *action probabilistic logic programs* as a slight variation of the probabilistic logic programs of [NS92, NS91], in which rules are of the form shown in Figure 1.1 As we will discuss in depth in Chapter 2, these rules have two parts: the "heads", which refer to actions taken by the entity being modeled (also referred to as the *adversary*), and rule "bodies", which refer to conditions on the environment [1].

In the following, we will give a brief overview of some of the applications that have motivated this research, and of some of the work that has already been carried out towards reaching those goals.

1.2 Applications

We will now present an overview of some of the real world domains in which reasoning agents (usually people) are interested in associating probabilities to the actions that can be taken by others, which we will refer to as adversaries, or opponents. These domains are *promises made*, *sports*, and *behavior of groups of interest.*

[1]Rule bodies can also contain probabilistic conditions on actions, but we will show in Section 2.3 that we can assume without loss of generality that bodies only refer to the environment.

1.2.1 Promises

Suppose an agent has access to historic records of how others have fulfilled their promises. Using this information, it can derive rules of the form shown in Figure 1.1 and use them during negotiations to decide if it is dealing with a trustworthy party or not, or to perform more complex reasoning such as expected payoff of a given deal. For instance, suppose a reasoning agent wishes to buy an airline ticket and looks up information on past performance of different airlines (publicly available information of this kind can be found at [BTS08]). Upon analyzing this information, it might conclude the following:

> *If airline A made a promise to customer C to fly from a certain airport* X *in the Northeast to airport* Y *in the Midwest, departing at time* DepTime *that is during the holiday season (assuming the time includes the date as well) and arriving at time* ArrTime*, then it is likely that the promise will be fulfilled with probability between 0.85 and 0.92.*

To this end, we will show how the kinds of *ap*-rules shown in Figure 1.1 can be extended with a framework designed especially for reasoning about the *degree of fulfillment* of promises. This will be the topic of Chapter 8.

1.2.2 Sports

The increasing availability of data and near real-time data analysis capabilities have made many sports very attractive for this type of reasoning. In particular, in recent years there has been much work in analyzing certain "well controlled" aspects of some sports. One of the most studied has been the *penalty kick* [2] in soccer [CLG02,

[2] A penalty kick is a type of free kick taken 12 yards from the goal, in which only the kicker and the goalkeeper from the defending team participate.

SK09, Rep10, DFE09, BEAR+05, Mos04, FMH99, BFW08, BEA09, Col07]. A well executed penalty kick by a professional player takes only 0.4 seconds to reach the goal, giving the goalkeeper very little time to execute his defensive action. It is widely assumed that by the time the kicker connects with the ball, the goalkeeper must already have a decision made regarding what to do. Thus, there are two basic strategies that the goalkeeper can adopt: trying to predict the action based on past data alone, thus making up his mind beforehand, or also taking into account the kicker's actions up to the point in which he connects with the ball. The latter strategy, though potentially more fruitful, has the added difficulty of having to read the player's actions and making a decision in an instant. The importance of a single goal in a soccer match, as well as the fact that matches can be decided by penalty kick shootouts in certain tournaments, have made the effort of trying to predict both the actions of the kicker and the goalkeeper a worthwhile one in the eyes of many teams.

The work carried out in this area is diverse. A study carried out by Lucozade and Prozone (see [Rep10] for a report on this study) yielded a fairly complex formula taking into account characteristics of the player himself as well as his technique. The formula is said to be useful in deriving the probability of a successful execution; of course, it can also be used to figure out how to raise such probability by choosing the right players and/or modifying their technique. There has also been a great deal of work from the game-theoretic point of view of penalty kicks [CLG02, Col07, BFW08, SK09] and even from the point of view of norm theory [BEAR+05]. These works find that success probabilities associated with "mixed strategies" (a game-theoretic term meaning that actions are taken according to a probability distribution) used by players are statistically equal. This means that, even though it is unlikely that

players are explicitly applying game theory when deciding how to act, it is clear that they have some sort of "pre-compiled" knowledge equivalent to the game-theoretic equilibria. Another finding in this sort of analysis is that players who are most successful truly play randomly, and don't use their prior executions as part of the decision of how to act next. Finally, there are also studies on trying to anticipate the kicker's actions by observing his behavior up to the point in which the foot connects with the ball [DFE09]. This work yielded interesting results, such as the fact that one of the most reliable indicators is the direction in which the planted foot is pointing. Furthermore, it discovered that certain "distributed movements" (for instance, combinations of leg and arm movements) are also good sources of information, giving players and coaches something completely new to focus on.

Clearly, much of the insight yielded by this kind of analyses can be represented in the form of *ap*-rules. For instance, the following facts found by the work mentioned above could be written using the notation shown in Figure 1.1:

- Kickers have a "natural side" of the goal to aim at: left for right-footed, right for left-footed. The success rate of shots taken to the natural side when the goalkeeper dives the other way is 95%, while 92% for the unnatural side (*i.e.*, missing the goal is somewhat harder when kicking to the natural side).

- When the goalkeeper and the kicker both go to the natural side, the success rate for the kicker is about 70%, and 58% when the opposite side is chosen.

- Given the information above, the goalkeeper will dive to the natural side of the kicker (his own right for right-footed, left for left-footed) with a probability between 60% and 65%. Likewise, the kicker will kick to his natural side with a similar probability.

- Players' characteristics play a major role. For instance, right-footed players have a success rate of about 71%, while left-footed ones have around 52%. Similarly, strikers and defenders have a probability of 75% and 72%, respectively, while midfielders have only 61%.

- Much more focused rules are also possible, such as: right-footed strikers or defenders of age 21 who have played less than 45 minutes in the current match have a probability of scoring of at least 90%. Another example is that players who run up from outside of the box who hit the ball with the inside of the foot and aim at the top-left corner have a probability of scoring of at least 80%.

- Rules designed around specific players are also quite common: "Riquelme kicks left and high with probability around 60%", "Crespo, when running long kicks left (probability 70%), and right when running short (with probability at least 80%)", "Ayala, when waiting long and running long, kicks right with probability at most 75%", etc. [3]

Clearly, this same kind of analysis can be applied to other aspects of soccer, as well as other sports. For instance, [Mos04] focuses on non-penalty kick plays in soccer, analyzing the Nash equilibrium of a model that simplifies how players choose to shoot at the opponent's goal. In [WW01], the authors analyze actions taken during tennis serves from a game-theoretic standpoint, in a manner similar to the ones discussed above for penalty kicks. Also from a game-theoretic standpoint, [ML10] analyzes the "pass vs. run" play calls in the U.S. National Football League, comparing it to the classical matching pennies game [Osb03] (since the offense chooses pass or run, and the defense chooses actions geared towards stopping either passes or runs). Finally,

[3] It is said that precisely this sort of rules were used in a much discussed penalty shootout in the 2006 FIFA World Cup Quarterfinals.

another work [KL09] takes on both the play selection problem in NFL football and the pitch selection problem by pitchers in U.S. Major League Baseball. The authors take into account large datasets from these leagues, and conclude that teams many times do not play according to minimax optimality, and that they could therefore be studied in order to derive ways in which to exploit their strategies.

1.2.3 Groups of Interest

Finally, we will discuss how action probabilistic logic programs can be applied in modeling groups of interest (such as terror groups, or groups at risk of engaging in this sort of activities). In [SMSS07b], we describe how the action probabilistic logic programming framework can be applied to model the behaviors of various stakeholders in the Afghan drug economy, deriving probabilistic rules much in the same way as an expert in any domain could do. In that work, we developed a simple conceptual map involving several actors in this domain and the relationships among them that encourage the cultivation of poppies, production and trafficking of drugs, and corruption in the region, which then allowed us to build preliminary models of what these actors could do. Though these are interesting applications, a more in-depth discussion is outside the scope of this thesis.

We have developed approximately 40 *ap*-programs that are carefully constructed models of 40 different groups from around the world. The groups modeled include tribes (*e.g.*, Shinwari, Waziri, Mohmand tribes in along the Pakistan-Afghanistan border), several terrorist groups (*e.g.*, Hezbollah, Fatah Revolutionary Council – Abu Nidal Organization, the Kurdish group PKK, and others), as well as political parties (*e.g.*, Jamaat-i-Ulema Islami, Pakistan People's Party). For each of these groups, we identified a small set of actions that the group has taken in the

past. For each such action, we tried to find conditions that are good predictors of when those groups would take those actions, and when they would not. These led to rules in the *ap*-program syntax.

The rules themselves have been developed in three ways: by manually having students (and in the case of about 20 groups, terrorism experts) code them, and by automatically extracting them from certain data sets. We started with the manual coding strategy and later transitioned to the use of an automatic extractor that works on a specialized data set called the "Minorities at Risk Organizational Behavior" data set [WAJ+07]. This data set has identified around 150 parameters to monitor for about 300 groups around the world that are either involved in terrorism or are at risk of becoming full-fledged terrorist organizations. The 150 attributes describe aspects of these groups that can change over time, such as whether or not the group engaged in violent attacks, if financial or military support was received from foreign governments, and the type of leadership the group has. The data was easy to divide into outcome conditions, or actions that could be taken by the group (*i.e.*, bombings, kidnappings, armed attacks, etc.), and environmental conditions (*i.e.*, the type of leadership, the kind and amount of foreign support, whether the group has a military wing, etc.). Values for these 150 parameters are available for up to 20 years per group, though it is less for some groups (*e.g.*, groups that have been around for a shorter duration). For each group, MAROB provides a table whose columns correspond to the 150 parameters and the rows correspond to the years.

1.3 Automated Extraction of Rules

Action probabilistic logic programs can be either designed by experts or automatically extracted from data. For the applications discussed above, automatic extraction is certainly possible as real world data has become more and more available. For the "groups of interest" application, automated extraction has been applied thus far to more than 30 groups (such as Hezbollah, FRC-ANO, PKK, Baath Party, Kurdistan Democratic Party of Iran). The automatic *ap*-program extraction (APEX) algorithm requires that we assume that the MAROB columns can be divided into action parameters (those attributes that will form the heads of the *ap*-rules) and environmental parameters (those attributes that will appear in the body of the rules). The APEX algorithm for extracting *ap*-rules consists of three main steps:

1. select an action condition (an action parameter with an instantiated value) to be the head of the rule,
2. fix one environmental condition as part of the body of the rule,
3. add varying combinations of the remaining environmental conditions to the body to determine if significant correlations exist between the body conditions and the outcome condition.

We then use the standard measurements of support and confidence from the literature in order to evaluate potential rule candidates.

The APEX algorithm calculates the difference between the confidence value produced by an environmental condition and by its negation. If this difference is above a given threshold, then an *ap*-rule is extracted. To obtain the probability range for the extracted rule, we use the confidence value initially obtained, plus/minus the standard deviation σ of the values involved in its calculation.

```
Algorithm 1:  APEX(DB, AC, EC, k, t)
1.  set Rules = ∅;
2.  for each action a_i ∈ AC {
3.      set head = a_i;
4.      for each condition e_j ∈ EC {
5.          set fixed_condition = e_j;
6.          for each combination, varied_condition, of
                1, 2, ..., and k of remaining conds. v_1, v_2, ..., v_k ∈ EC
7.              set body = fixed_condition ∧ varied_condition;
8.              compute PosConf = C_{head,body};
9.              set body = fixed_condition ∧ ¬varied_condition;
10.             compute NegConf = C_{head,body};
11.             set prob = |PosConf − NegConf|;
12.             if prob >= t
13.                 add (head : [prob − σ, prob + σ] ← body) to Rules
14.         }
15.     }
16. }
17. Return Rules;
```

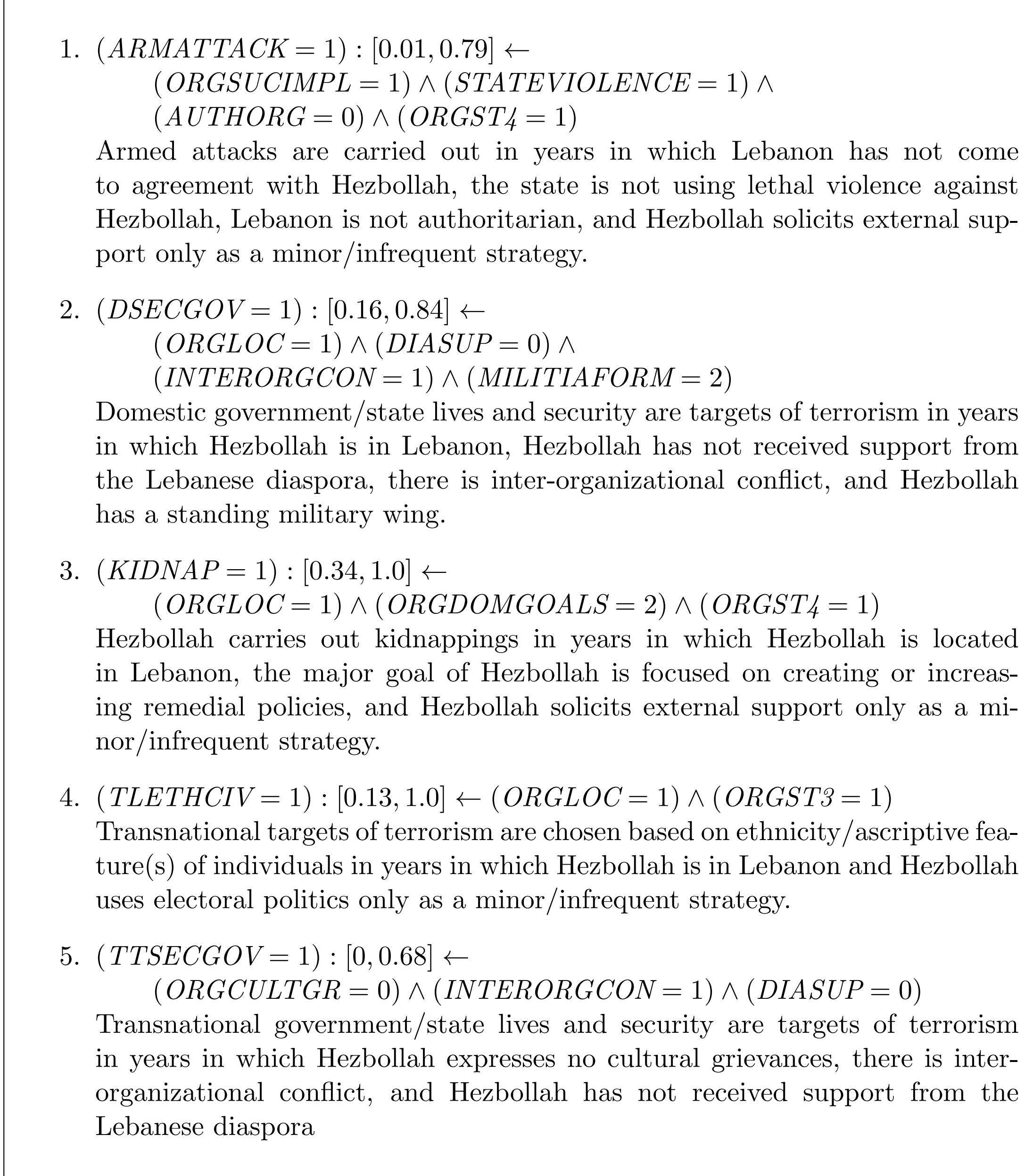
Figure 1.2: The APEX Algorithm.

The complete APEX Algorithm for a database DB with a set of action conditions AC, environmental conditions EC, and confidence difference threshold t is summarized in Figure 1.2. Note that this algorithm is not a novel one, and simply performs calculations to capture interesting variations in the data in order to build rules.

Using this algorithm, we have extracted thousands of *ap*-rules for Hezbollah. Some examples of the *ap*-rules extracted from the data for Hezbollah are given in Figure 1.3.

1.4 Towards a "Big Picture"

The CARA Architecture. In [SAM+07], we describe how our work fits in the "big picture" of modeling agent behavior. This general view is captured in the



1. $(ARMATTACK = 1) : [0.01, 0.79] \leftarrow$
   $(ORGSUCIMPL = 1) \wedge (STATEVIOLENCE = 1) \wedge$
   $(AUTHORG = 0) \wedge (ORGST4 = 1)$
   Armed attacks are carried out in years in which Lebanon has not come to agreement with Hezbollah, the state is not using lethal violence against Hezbollah, Lebanon is not authoritarian, and Hezbollah solicits external support only as a minor/infrequent strategy.
2. $(DSECGOV = 1) : [0.16, 0.84] \leftarrow$
   $(ORGLOC = 1) \wedge (DIASUP = 0) \wedge$
   $(INTERORGCON = 1) \wedge (MILITIAFORM = 2)$
   Domestic government/state lives and security are targets of terrorism in years in which Hezbollah is in Lebanon, Hezbollah has not received support from the Lebanese diaspora, there is inter-organizational conflict, and Hezbollah has a standing military wing.
3. $(KIDNAP = 1) : [0.34, 1.0] \leftarrow$
   $(ORGLOC = 1) \wedge (ORGDOMGOALS = 2) \wedge (ORGST4 = 1)$
   Hezbollah carries out kidnappings in years in which Hezbollah is located in Lebanon, the major goal of Hezbollah is focused on creating or increasing remedial policies, and Hezbollah solicits external support only as a minor/infrequent strategy.
4. $(TLETHCIV = 1) : [0.13, 1.0] \leftarrow (ORGLOC = 1) \wedge (ORGST3 = 1)$
   Transnational targets of terrorism are chosen based on ethnicity/ascriptive feature(s) of individuals in years in which Hezbollah is in Lebanon and Hezbollah uses electoral politics only as a minor/infrequent strategy.
5. $(TTSECGOV = 1) : [0, 0.68] \leftarrow$
   $(ORGCULTGR = 0) \wedge (INTERORGCON = 1) \wedge (DIASUP = 0)$
   Transnational government/state lives and security are targets of terrorism in years in which Hezbollah expresses no cultural grievances, there is inter-organizational conflict, and Hezbollah has not received support from the Lebanese diaspora

Figure 1.3: A sample of the rules extracted by APEX from the Hezbollah dataset. The atoms in the rules are represented as a variable and its value. The English translation of each rule is also provided.

CARA (*Cultural Adversarial Modeling Architecture*) for gathering data about different cultural groups, learning the intensity of opinions that those groups have on various topics, and developing a process that supports building/extracting models of behavior of those groups and continuously refining those models through

shared, multi-person, learning experiences. The CARA architecture is supported via ongoing applications that have been developed, such as tracking tribes along the Pakistan-Afghanistan border with a view to understanding and eventually reducing the burgeoning drug trade there, or focusing on politically active minorities at risk in fragile regions of the world.

The STOP System. Finally, in [SSMS08] we describe the *SOMA Terror Organization Portal* (STOP), a facility that national security analysts can use in order to understand terror threats worldwide. STOP provides a single, Internet or intranet accessible site within which national security analysts can study certain groups. Not only does STOP provide tools they might use, it also provides a valuable social networking capability that allows analysts to often create and expand a network of experts on a given topic. This tool allows them to leverage this network so that different points of view can be incorporated into their analytic task before a final recommendation is made.

To date, STOP allows analysts the ability to examine approximately 36 terror groups from about ten countries ranging from Morocco all the way to Afghanistan. Users can see probabilistic rules extracted automatically about these groups (over 14,000 for Hezbollah; over 20,000 for the PKK), browse them, experiment with them, and mark them as useful or not. They can use these markings to build consensus (or at least identify different camps) about a given topic, and explore the pros and cons of alternative views, all without having to move from their desk.

1.5 Organization of this Thesis

We start off in Chapter 2 by formally presenting the preliminary notions required in the treatment of action probabilistic logic programs. Since these notions are the basis of most of the work described in this thesis, this can be considered to be a basic reference for the rest of the chapters; parts of this chapter first appeared published in [SSNS06], [KMN+07b], [KMN+07a], [SMSS08], [SMSS10], [SS10], [SDS10a], and [SDS10b]. Chapter 3 contains further preliminary material covering the literature that is most relevant to this thesis, and also develops a comparison of action probabilistic logic programs with other formalisms for probabilistic reasoning, as well as an analysis of pros and cons with respect to our own work.

In Chapter 4, we introduce the problem of computing the *most probable worlds* (MPW) given an *ap*-program and the state of the environment. We present and analyze a naive approach to solving this problem, and show that it is highly intractable even for a very small input program. This result motivates searching for various ways in which both exact and approximate answers can be obtained within a reasonable amount of time. We first investigate two techniques for *collapsing* the set of variables into a much smaller set by leveraging *equivalence* classes over the set of possible worlds that can be obtained from the input program. We prove that these techniques are correct (sound and complete), and also analyze the computational cost of the corresponding algorithms, showing that they are in general much more efficient than the naive approach. However, since these improvements are not guaranteed in general, we investigate sampling techniques and a heuristic that can be used to reduce the *number of variables* of the underlying problems that must be solved, which also has the added benefit of reducing the *number* of such problems. Some basic desirable properties of this technique are shown to hold. Finally,

we report on experimental results carried out using prototype implementations of the algorithms described. Parts of the work presented in this chapter were first published in [SSNS06], [KMN$^+$07b], and [KMN$^+$07a].

In Chapter 5, we introduce a problem related to the MPW problem of Chapter 4, in which we assume that a set of atoms has been provided as part of the input indicating that these are *of interest* to the reasoning agent. This information allows us to define *worlds of interest*, and the *most probable worlds of interest* as its corresponding extension. Algorithms are provided for solving this related problem, both exactly and approximately, which in comparison to the original problem afford an exponential speedup due to the extra information provided. To conclude, we report on an extensive experimental evaluation of these algorithms. Parts of the work presented in this chapter were first published in [SMSS08] and [SMSS10].

In Chapter 6, we describe our work on a problem that can be considered in some sense the *dual* of the most probable world problem, since it focuses on effecting *changes in the environment* to evoke actions that are desired by the reasoning agent. We assume that the reasoning agent has the capacity to attempt to make certain changes in the current state of the environment; given an *ap*-program, we are interested in trying to change the environment, subject to some constraints, so that the probability that the entity being modeled takes some action (or combination of actions) is maximized. This is called the Basic Abductive Query Answering Problem (BAQA). We first formally define and study the complexity of BAQA and several variants of it. We then provide an exact (exponential) algorithm to solve BAQA, followed by more efficient algorithms for specific subclasses of BAQA. We also develop appropriate heuristics to solve BAQA efficiently, and report on empir-

ical evaluations performed over synthetic data. Parts of the work presented in this chapter were first published in [SS10] and [SDS10b].

In Chapter 7, we tackle an extension of BAQA in which we wish to take into account how the entity being modeled might react to the changes in the environment brought about by the reasoning agent's actions. This is done by extending the work in Chapter 6 so that state change attempts have associated cost and probability of success. This is a flexible way to incorporate possible reactions by the adversary; some state changes may prove to be more costly than others, and there is an inherent uncertainty in how things might work out when trying to change the current state of the environment. We first show that an exact solution to this problem is highly intractable, and then introduce an algorithm based on iterated density estimation of probability distributions that is shown to work quite well both in terms of scalability and accuracy. Parts of the work presented in this chapter were first published in [SDS10a] and [SDS10b].

Finally, Chapter 8 provides an in-depth presentation of a novel framework which can be used for reasoning about *promises* made in a multi-agent setting. The main goal of this chapter is to show one way in which *ap*-programs can be applied to specific settings by including information about promises made; it also shows how reasoning about the actions of others can be done without involving *ap*-programs directly. In this chapter, we assume that agents have interacted with one another and have agreed on a certain set of promises to perform certain actions. The focus of this framework is then to provide a method by which agents' actions taken toward the fulfillment of these promises can be *evaluated* by the interested parties, so as to obtain a quantitative measure of how well the promises were fulfilled. We then show simple methods for how this information can be used to *predict* future fulfillment

of promises by using historic records of agents' behavior regarding fulfillment. This information can be of great value to agents during the negotiation process (*not* modeled here), since measures of trust and reliability are often based on this kind of data. We show how such predictions can be done first using a simple model based on aggregation of past information and then by reasoning about linear trends over time. Finally, we argue that more complex methods are needed if the agent is required to make more involved decisions. Towards this end, we propose the application of probabilistic logic programming as an approach to building a model of the agent of interest with respect to its behavior in fulfilling promises. Parts of the work presented in this chapter were first published in [SBSK08].

Chapter 2

Action Probabilistic Logic Programs: Preliminary Notions

In this chapter, we will introduce the basic notions pertinent to action probabilistic logic programs that will be used in the rest of the thesis. The goal is for this chapter to be a reference, which is also illustrated with examples to enhance the presentation.

2.1 Introduction

Action probabilistic logic programs (*ap*-programs for short) [KMN$^+$07a] are a class of the extensively studied family of probabilistic logic programs (PLPs) [NS92, NS93, KIL04]. As we discussed in Chapter 1, *ap*-programs have been used extensively to model and reason about the behavior of groups and an application for reasoning about terror groups based on *ap*-programs has users from over 12 US government entities [Gil08]. *ap*-programs use a two sorted logic where there are "state" predicate symbols and "action" predicate symbols and can be used to represent behaviors of arbitrary entities (ranging from users of web sites to institutional

investors in the finance sector to corporate behavior) because they consist of rules of the form:

> *"if a conjunction C of atoms is true in a given state S, then entity E (the entity whose behavior is being modeled) will take action A with a probability in the interval $[L, U]$."*

We emphasize that action atoms only represent the fact that an action is taken, and not the action itself; they are therefore quite different from actions in domains such as AI planning or reasoning about actions, in which effects, preconditions, and post-conditions are part of the specification. We assume that effects and preconditions are generally not known, though later on we show how to represent the information we may have about them.

In this kind of applications, it is essential to avoid making probabilistic independence assumptions, since the approach involves *finding out* what probabilistic relationships exist and then exploit these findings in the forecasting effort. For instance, Figure 2.1 shows a small set of rules automatically extracted from data [ACW08] about Hezbollah's past. Rule 1 says that Hezbollah uses kidnappings as an organizational strategy with probability between 0.5 and 0.56 in years in which no political support was provided to it by a foreign state (`forstpolsup`), and the severity of inter-organizational conflict involving (`intersev1`) it is at level "*c*". Rules 2 and 3, also about kidnappings, state that this action will be performed with probability between 0.8 and 0.86 in years in which no external support is solicited by the organization (`extsup`) and either the organization does not advocate democratic practices (`demorg`) or electoral politics is not used as a strategy (`elecpol`). Similarly, Rules 4 and 5 refer to the action "civilian targets chosen based on ethnicity" (`tlethciv`). The first one states that this action will be taken

r_1. kidnap$(1) : [0.50, 0.56] \leftarrow$ forstpolsup$(0) \wedge$ intersev1(c).
r_2. kidnap$(1) : [0.80, 0.86] \leftarrow$ extsup$(1) \wedge$ demorg(0).
r_3. kidnap$(1) : [0.80, 0.86] \leftarrow$ extsup$(1) \wedge$ elecpol(0).
r_4. tlethciv$(1) : [0.49, 0.55] \leftarrow$ demorg(1).
r_5. tlethciv$(1) : [0.71, 0.77] \leftarrow$ elecpol$(1) \wedge$ intersev2(c).

Figure 2.1: A small set of rules modeling Hezbollah.

with probability 0.49 to 0.55 in years in which the organization advocates democratic practices, while the second states that the probability rises to between 0.71 and 0.77 in years in which electoral politics are used as a strategy and the severity of inter-organizational conflict (with the organization with which the second highest level of conflict occurred) was not negligible" (intersev2). *ap*-programs have been used extensively by terrorism analysts to make predictions about terror group actions [Gil08, MMP+08a].

2.2 Syntax and Semantics

Action probabilistic logic programs (*ap*-programs) are a variant of the probabilistic logic programs introduced in [NS91, NS92]. We will now present the preliminary concepts that will be used throughout this thesis, using the *ap*-program in Figure 2.1 as a running example.

We assume the existence of a logical alphabet that consists of a finite set $\mathcal{L}_{cons}$ of constant symbols, a finite set $\mathcal{L}_{pred}$ of predicate symbols (each with an associated arity) and an infinite set $\mathcal{L}_{var}$ of variable symbols: function symbols are not allowed. Terms, atoms, and literals are defined in the usual way [Llo87]. We assume that $\mathcal{L}_{pred}$ is partitioned into disjoint sets: $\mathcal{L}_{act}$ of *action symbols* and $\mathcal{L}_{sta}$ of *state* symbols. Thus, if $t_1, \ldots, t_n$ are terms, and p is an n-ary action (resp., state) symbol, then $p(t_1, \ldots, t_n)$, is called an *action (resp., state) atom.*

Definition 1 (Action formula)**.** *A (ground) action formula is defined as:*

- *a (ground) action atom is a (ground) action formula;*
- *if F and G are (ground) action formulas, then $\neg F$, $F \wedge G$, and $F \vee G$ are also (ground) action formulas.*

The set of all possible action formulas is denoted by $formulas(B_{\mathcal{L}_{act}})$, where $B_{\mathcal{L}_{act}}$ is the Herbrand base associated with $\mathcal{L}_{act}$, $\mathcal{L}_{cons}$, and $\mathcal{L}_{var}$.

Definition 2 (*ap*-formula)**.** *If F is an action formula and $\mu = [\alpha, \beta] \subseteq [0, 1]$, then $F : \mu$ is called an* annotated action formula *(or* ap-*formula), and μ is called the* ap-*annotation of F.*

Without loss of generality, throughout this thesis we will assume that F is in conjunctive normal form (*i.e.*, it is written as a conjunction of disjunctions). Notice that wffs are annotated with probability intervals rather than point probabilities. There are three reasons for this:

1. In many cases, we are told that an action formula F is true in state s with some probability p plus or minus some margin of error e; this naturally translates into the interval $[p - e, p + e]$.
2. As shown by [FHM90, NS92], if we do not know the relationship between events e_1, e_2, even if we know point probabilities for e_1, e_2, we can only infer an interval for the conjunction and disjunction of e_1, e_2.
3. Interval probabilities generalize point probabilities, so our work is also relevant to point probabilities.

Definition 3 (World/State)**.** *A* world *is any finite set of ground action atoms. A* state *is any finite set of ground state atoms.*

It is assumed that all actions in the world are carried out more or less in parallel and at once, given the temporal granularity adopted along with the model. Contrary to (related but essentially different) approaches such as stochastic planning, we are not concerned here with reasoning about the effects of actions. We now define *ap*-rules.

Definition 4 (*ap*-rule)**.** *If F is an action formula, $B_1, \ldots, B_n$ are state atoms, and μ is an* ap-*annotation, then $F : \mu \leftarrow B_1 \ \wedge \ \ldots \wedge \ B_m$ is called an ap*-rule*. If this rule is named r, then Head(r) denotes $F : \mu$ and Body(r) denotes $B_1 \ \wedge \ \ldots \wedge \ B_n$.*

Intuitively, the above *ap*-rule says that an unnamed entity (*e.g.*, a group g, a person p etc.) *will take action F with probability in the range μ if $B_1, \ldots, B_n$ are true in the current state (we will define this term shortly) and if the unnamed entity will take each action A_i with a probability in the interval μ_i for $1 \leq i \leq n$.*

Definition 5 (*ap*-program)**.** *An* action probabilistic logic program *(*ap-*program for short) is a finite set of* ap-*rules. An* ap-*program Π' such that $\Pi' \subseteq \Pi$ is called a* subprogram *of* Π.

Figure 2.1 shows a small portion of an *ap*-program we derived automatically to model Hezbollah's actions. Henceforth, we use *Heads*(Π) to denote the set of all annotated formulas appearing in the head of some rule in Π. Given a ground *ap*-program Π, we will use *sta*(Π) (resp., *act*(Π)) to denote the set of all state (resp., action) atoms that appear in Π.

Example 1 (Worlds and states)**.** *Coming back to the* ap-*program in Figure 2.1, the following are examples of worlds:*

$$\{\texttt{kidnap}(1)\}, \{\texttt{kidnap}(1), \texttt{tlethciv}(1)\}, \{\}$$

The following are examples of states:

$$\{\texttt{forstpolsup}(0), \texttt{elecpol}(0)\}, \{\texttt{extsup}(1), \texttt{elecpol}(1)\}, \{\texttt{demorg}(1)\}.$$

Finally, we will use the concept of *reduction* of an *ap*-program w.r.t. a state:

Definition 6 (Reduction of an *ap*-program w.r.t. a state)**.** *Let* Π *be an* ap-*program and* s *a state. The* reduction of Π w.r.t. s*, denoted* Π_s*, is the set* $\{F : \mu \mid s$ *satisfies* $Body$ *and* $F : \mu \leftarrow Body$ *is a ground instance of a rule in* $\Pi\}$*. Rules in this set are said to be* relevant *in state* s*.*

Note that Π_s *never has any non-action atoms in it.* The following is an example of a reduction with respect to a state.

Example 2. *Let* Π *be the* ap-*program from Figure 2.1, and suppose we have the following state:*

$$s = \{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{extsup}(1), \texttt{elecpol}(0), \texttt{demorg}(0)\}$$

The reduction of Π *with respect to state* s *is:*

$$\Pi_s = \{\texttt{kidnap}(1) : [0.50, 0.56], \texttt{kidnap}(1) : [0.80, 0.86]\}.$$

Key differences. The key differences between *ap*-programs and the PLPs of [NS91, NS92] are that (i) *ap*-programs have a bipartite structure (action atoms and state atoms) and (ii) they allow arbitrary formulas (including ones with negation) in rule heads ([NS91, NS92] do not). They can easily be extended to include variable annotations and annotation terms as in [NS91]. Likewise, as in [NS91], they can be easily extended to allow complex formulas rather than just atoms in rule bodies. We expand on this topic in Section 3.3. *However, the most important difference between*

our approach and [NS91, NS92] is that those papers focus on entailment, while this thesis focuses on fundamentally different problems.

We use $\mathcal{W}$ to denote the set of all possible worlds, and $\mathcal{S}$ to denote the set of all possible states. It is clear what it means for a state to satisfy the body of a rule [Llo87].

Definition 7 (Satisfaction of a rule body by a state)**.** *Let* Π *be an* ap-*program and* s *a state. We say that* s satisfies *the body of a rule* $F : \mu \leftarrow B_1 \ \wedge \ \ldots \wedge \ B_m$ *if and only if* $\{B_1, \ldots, B_M\} \subseteq s$.

Similarly, we define what it means for a world to satisfy a ground action formula:

Definition 8 (Satisfaction of an action formula by a world)**.** *Let* F *be a ground action formula and* w *a world. We say that* w satisfies F *if and only if:*

– *if* $F \equiv a$*, for some atom* $a \in B_{\mathcal{L}_{act}}$*, then* $a \in w$*;*

– *if* $F \equiv F_1 \wedge F_2$*, for action formulas* $F_1, F_2 \in formulas(B_{\mathcal{L}_{act}})$*, then* w *satisfies* F_1 *and* w *satisfies* F_2*;*

– *if* $F \equiv F_1 \vee F_2$*, for action formulas* $F_1, F_2 \in formulas(B_{\mathcal{L}_{act}})$*, then* w *satisfies* F_1 *or* w *satisfies* F_2*;*

– *if* $F \equiv \neg F'$*, for some action formula* $F' \in formulas(B_{\mathcal{L}_{act}})$*, then* w *does not satisfy* F'*.*

The semantics of *ap*-programs uses possible worlds in the spirit of [Hai84, Nil86, FHM90]. Given an *ap*-program Π and a state s, we can define a set $LC(\Pi, s)$ of linear constraints associated with s. Each world w_i expressible in the language $\mathcal{L}_{act}$ has an associated variable v_i denoting the probability that it will actually occur. $LC(\Pi, s)$ consists of the following constraints.

1. For each $Head(r) \in \Pi_s$ of the form $F : [\ell, u]$, $LC(\Pi, s)$ contains the constraint $\ell \leq \sum_{w_i \in \mathcal{W} \wedge w_i \models F} v_i \leq u$.

2. $LC(\Pi, s)$ contains the constraint $\sum_{w_i \in \mathcal{W}} v_i = 1$.

3. All variables are non-negative.

4. $LC(\Pi, s)$ contains only the constraints described in $1 - 3$.

We will provide a more formal model theory for *ap*-programs in the following chapters; for now we merely provide the definition below. Π_s is *consistent* iff $LC(\Pi, s)$ is solvable over $\mathbb{R}$.

Definition 9 (Entailment of an *ap*-formula by an *ap*-program)**.** *Let* Π *be an* ap-*program,* s *a state, and* $F : [\ell, u]$ *a ground action formula.* Π_s entails $F : [\ell, u]$, *denoted* $\Pi_s \models F : [\ell, u]$ *iff* $[\ell', u'] \subseteq [\ell, u]$ *where:*

$\ell' =$ ***minimize*** $\sum_{w_i \in \mathcal{W} \wedge w_i \models F} v_i$ ***subject to*** $LC(\Pi, s)$.

$u' =$ ***maximize*** $\sum_{w_i \in \mathcal{W} \wedge w_i \models F} v_i$ ***subject to*** $LC(\Pi, s)$.

We will show in Example 6 below an example of both $LC(\Pi, s)$ and entailment of an annotated action formula.

Throughout this thesis, we will assume that there is a fixed state s. Hence, once we are given Π and s, Π_s is fixed. We can associate a fixpoint operator T_{Π_s} with Π, s which maps sets of ground *ap*-annotated wffs to sets of ground *ap*-annotated wffs as follows. We first define an intermediate operator $U_{\Pi_s}(X)$.

1.	d: $[0.52, 0.82]$	$\leftarrow$	.
2.	$b \wedge a$: $[0.55, 0.69]$	$\leftarrow$	d: $[0.48, 0.89]$.

Figure 2.2: A simple example of an *ap*-program with action atoms in the body of the rules, which is already reduced with respect to a certain state.

2.3 A Fixpoint Operator

Definition 10. *Suppose X is a set of ground* ap-*wffs. We define* $U_{\Pi_s}(X) = \{F : \mu \mid F : \mu \leftarrow A_1 : \mu_1 \wedge \cdots \wedge A_m : \mu_m$ *is a ground instance of a rule in* Π_s *and for all* $1 \leq j \leq m$*, there is an* $A_j : \eta_j \in X$ *such that* $\eta_j \subseteq \mu_j\}$.

Intuitively, $U_{\Pi_s}(X)$ contains the heads of all rules in Π_s whose bodies are deemed to be "true" if the *ap*-wffs in X are true. However, $U_{\Pi_s}(X)$ may not contain all ground action atoms. This could be because such atoms don't occur in the head of a rule —$U_{\Pi_s}(X)$ never contains any action wff that is not in a rule head. The following is an example of the calculation of $U_{\Pi_s}(X)$.

Example 3. *Consider the simple program depicted in Figure 2.2, and let* $X = \{d : [0.5, 0.55]\}$*. In this case,* $U_{\Pi_s}(X) = \{d : [0.52, 0.82], b \wedge a : [0.55, 0.69]\}$.

In order to assign a probability interval to each ground action atom, we use the same procedure followed in [NS91]. We use $U_{\Pi_s}(X)$ to set up a linear program $\mathsf{CONS}_U(\Pi, s, X)$ as follows.

Definition 11. *Let* Π *be an* ap-*program and* s *be a state. For each world* w_i*, let* p_i *be a variable denoting the probability of* w_i *being the "real world". As each* w_i *is just an Herbrand interpretation (where action symbols are treated like predicate symbols), the notion of satisfaction of an action formula* F *by a world* w*, denoted by* $w \mapsto F$*, is defined in the usual way.*

1. *If* $F : [\ell, u] \in U_{\Pi_s}(X)$, *then* $\ell \leq \Sigma_{w_i \mapsto F}\, p_i \leq u$ *is in* $\mathsf{CONS}_U(\Pi, s, X)$.

2. $\Sigma_{w_i} p_i = 1$ *is in* $\mathsf{CONS}_U(\Pi, s, X)$.

We refer to these as constraints of type (1) and (2), respectively.

The following is an example of how these constraints look.

Example 4. *Let* Π *be the* ap*-program from Figure 2.2, and* $X = \{d : [0.5, 0.55]\}$. *The possible worlds are:* $w_0 = \emptyset$, $w_1 = \{d\}$, $w_2 = \{b\}$, $w_3 = \{a\}$, $w_4 = \{d, b\}$, $w_5 = \{d, a\}$, $w_6 = \{b, a\}$, *and* $w_7 = \{d, b, a\}$. *In this case, the linear program* $\mathsf{CONS}_U(\Pi, s, X)$ *contains the following constraints:*

$$0.52 \leq p_1 + p_4 + p_5 + p_7 \leq 0.82$$

$$0.55 \leq p_6 + p_7 \leq 0.69$$

$$p_0 + p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + p_7 = 1$$

To find the lower (resp., upper) probability of a ground action atom A, we merely minimize (resp. maximize) $\Sigma_{w_i \mapsto A} p_i$ subject to the above constraints. We also do the same w.r.t. each formula F that occurs in $U_{\Pi_s}(X)$ —this is because this minimization and maximization may sharpen the bounds of F. Let $\ell(F)$ and $u(F)$ denote the results of these minimizations and maximizations, respectively. Our operator $T_{\Pi_s}(X)$ is then defined as follows.

Definition 12. *Suppose* Π *is an ap-program, s is a state, and* X *is a set of ground* ap*-wffs. Our operator* $T_{\Pi_s}(X)$ *is then defined to be*

$$\{F : [\ell(F), u(F)] \mid (\exists \mu)\ F : \mu \in U_{\Pi_s}(X)\} \cup$$
$$\{A : [\ell(A), u(A)] \mid A \textit{ is a ground action atom}\}.$$

Thus, $T_{\Pi_s}(X)$ works in two phases. It first takes each formula $F : \mu$ that occurs in $U_{\Pi_s}(X)$ and finds $F : [\ell(F), u(F)]$ and puts this in the result. Once all such $F : [\ell(F), u(F)]$'s have been put in the result, it tries to infer the probability bounds of all ground action atoms A from these $F : [\ell(F), u(F)]$'s. The following is an example of this process.

Example 5. *Consider the* ap*-program presented in Figure 2.2, with the same state* s. *For* $T_{\Pi_s} \uparrow 0$, *we have* $X = \emptyset$. *We first obtain* $U_{\Pi_s}(\emptyset) = \{d : [0.52, 0.82]\}$. *Then,* $T_{\Pi_s}(\emptyset) = \{d : [0.52, 0.82], a : [0, 1.0], b : [0, 1.0]\}$.
To obtain $T_{\Pi_s} \uparrow 1 = T_{\Pi_s}(T_{\Pi_s} \uparrow 0)$, *let* $X = T_{\Pi_s}(\emptyset)$. *Then we have:*

$$U_{\Pi_s}(X) = \{d : [0.52, 0.82], b \wedge a : [0.55, 0.69]\}, \text{ and}$$

$$T_{\Pi_s}(X) = \{d : [0.52, 0.82], b \wedge a : [0.55, 0.69]\} \cup \{A : [\ell(A), u(A)] \mid A \text{ is a ground action atom }\}.$$

In order to infer the probability bounds for all ground action atoms, we need to build a linear program using the formulas from $U_{\Pi_s}(X)$ *and solve it for each ground atom by minimizing and maximizing the objective function of the probabilities of the worlds that satisfy each atom. The possible worlds are:* $w_0 = \emptyset$, $w_1 = \{d\}$, $w_2 = \{b\}$, $w_3 = \{a\}$, $w_4 = \{d, b\}$, $w_5 = \{d, a\}$, $w_6 = \{b, a\}$, *and* $w_7 = \{d, b, a\}$. *The linear program then consists of the following constraints:*

$$0.52 \leq p_1 + p_4 + p_5 + p_7 \leq 0.82$$

$$0.55 \leq p_6 + p_7 \leq 0.69$$

$$p_0 + p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + p_7 = 1$$

In order to obtain $\ell(d)$ *and* $u(d)$ *(that is, bound the probability value for action atom* d*), we minimize and then maximize the objective function* $p_1 + p_4 + p_5 + p_6$

subject to the linear program above, obtaining: $d : [0.52, 0.82]$*. Similarly, we use the objective function* $p_3 + p_5 + p_6 + p_7$ *for atom* a*, obtaining* $a : [0.55, 1.0]$*, and* $p_2+p_4+p_6+p_7$ *for* b*, obtaining* $b : [0.55, 1.0]$*. Therefore, we have finished calculating* $T_{\Pi_s} \uparrow 1$*, and we have obtained*

$$T_{\Pi_s}(X) = \{d : [0.52, 0.82], b \wedge a : [0.55, 0.69], a : [0.55, 1.0], b : [0.55, 1.0]\}.$$

Similar computations with $X = T_{\Pi_s}(T_{\Pi_s}(\emptyset))$ *allows us to conclude that* $T_{\Pi_s} \uparrow 2 = T_{\Pi_s} \uparrow 1$*, which means we reached the fixed point.*

Given two sets X_1, X_2 of *ap*-wffs, we say that $X_1 \leq X_2$ iff for each $F_1 : \mu_1 \in X_1$, there is an $F_1 : \mu_2 \in X_2$ such that $\mu_2 \subseteq \mu_1$. Intuitively, $X_1 \leq X_2$ may be read as "X_1 is less precise than X_2." The following straightforward variation of similar results in [NS91] shows the following results:

Proposition 1. *Given an* ap*-program* Π*, a state* s*, and the* T_{Π_s} *operator defined in Definition 12, the following statements hold:*

1. T_{Π_s} *is monotonic w.r.t. the* $\leq$ *ordering.*

2. T_{Π_s} *has a least fixpoint, denoted* $T^{\omega}_{\Pi_s}$*.*

For the sake of simplicity, in the rest of this thesis we will assume that *ap*-programs do not have annotated action atoms in the bodies of their rules. Since we have shown here how to obtain a set of rules with this property by means of a fixpoint operator, this assumption is without loss of generality.

This concludes the presentation of *ap*-programs and preliminary notions related to their syntax and semantics. In the next chapter, we conclude the introduc-

tory material in this thesis with a discussion of the work in the literature that is most closely related to the material presented in Chapters 4 to 8.

Chapter 3

Related Work

In this chapter, we will review the literature that is related to the work developed in this thesis. In Sections 3.1 and 3.2, we will cover a series of formalisms that are closely related to action probabilistic logic programs, and will discuss why the approaches developed in the literature are fundamentally different from the work presented in later chapters of this thesis. In particular, we will distinguish the *most probable world* problem (the subject of Chapters 4 and 5) as a novel approach to probabilistic reasoning. Section 3.3 will focus on the relationship between *ap*-programs and other well-known formalisms for probabilistic reasoning, presenting examples to support our arguments. Section 3.4 will review the literature on probabilistic abduction, which is closely related to the contents of Chapters 6 and 7. Finally, in support of the work presented in Chapter 8, Section 3.5 reviews how past work on reasoning about trust and reliability in autonomous agents relates to our approach.

3.1 Probabilistic Logic

The subject of *probabilistic logic*, also known as *probability logic*, or *logic of probability* has a long history that dates back to the time of Leibniz (late 17th and early 18th centuries). In his 1984 article [Hai84] (on which we base the discussion in the first part of this section), Hailperin recounts this history in some detail, and also discusses the work of many logicians and mathematicians who were involved in the development of logics of probability that led to the formalisms that have become conventional today.

3.1.1 From Leibniz to Boole

As stated in [Hai84], Leibniz already envisioned a “new kind of logic” that would involve degrees of probability. Since the birth of probability theory occurred during his lifetime, this wasn’t a clear proposal, though it was clear that his vision was of a means for estimating likelihoods of truth values, and a proof theory that only led to probability instead of certainty. It wasn’t until the 19th century that modern logic was developed and it was Boole [Boo54] who first combined the logic of “not”, “and”, and “or” with probability theory. There were some (perhaps unnecessary) restrictions to his proposed formalism, since the “or” connective was interpreted as exclusive (and thus probabilities of disjunctions were the result of adding the probabilities of its disjuncts), and all events represented by propositions were assumed to be independent (and so probabilities of conjunctions were the product of the probabilities of its conjuncts). This was indeed the first development in probabilistic logic. Some time later in the 19th century, C.S. Peirce also showed great interest in this endeavor, but his 1883 treatment of *probable inference* [Pei83] provides no formal

discussion of probable inferences. Many-valued logics were developed in the 1920s, and probability was naturally associated with this new formalism. Reichenbach was one of the most distinguished proponents of this association [Rei49]; Tarksi, on the other hand, opposed the extension of many-valued logics to incorporate probability, and also declared that Reichenbach's probability logic was not really a multi-valued logic but a special case of the conventional two-valued logic [Tar36].

3.1.2 More Recent Developments

Hailperin [Hai84] develops a probability logic at the propositional level which slightly generalizes the idea of truth table. The notion of probability is part of the semantics, and thus the meaning of probability plays the same role as truth does in classical logic. This logic is not given syntactically, but rather semantically through the definition of *logical consequence* using the notion of *probability model*. Along the same vein (and published only two years apart), Nilsson [Nil86] proposes a semantical generalization of logic, in which the truth values of sentences are probability values. Nilsson's generalization applies to any logical system for which the consistency of a finite set of sentences can be established. In the same way in which we do for *ap*-programs (see Chapter 2), the semantics of this probabilistic logic assigns probability distributions over the set of possible worlds (*i.e.*, subsets of the Herbrand base). This article was a pioneer in the development of probabilistic logic in various ways. First, it proposed a geometric interpretation of the set of linear constraints induced by the set of sentences and their probabilistic annotations. This interpretation clearly illustrates methods proposed by Nilsson for computing solutions to the intractable problem of obtaining probability distributions that satisfy the constraints. One such method for probabilistic entailment involves approximating the

vector of variables associated with the query formula by a linear combination of the rows of the matrix containing the coefficients of possible worlds, depending on the logical relationship between the formulas in the input set and the query. The other method proposed by Nilsson consists of computing the distribution that has the *maximum entropy*; this approach was later adopted by others for its applications in probabilistic logic programming, as we discuss in the next section. Other approximation methods are discussed in the paper, though none of them are evaluated experimentally for accuracy or computational tractability.

The last two decades of the 20th century continued to prove proliferous with the work of Halpern [Hal90]. This work considers to possibilities for giving semantics to first-order logics of probability. The first is suitable for "statistical knowledge" or "chance setups", and assumes the existence of one world where statements with respect to individuals are either true or false; therefore, *degrees of belief* are not representable. The example used by Halpern to illustrate this semantics is the sentence "the probability that a randomly chosen bird flies is 0.9". The other kind of semantics places probabilities on possible worlds, and therefore chance setups are not well defined and this is more appropriate for degrees of belief. Related to the previous example, an example of this semantics is the sentence "the probability that Tweety (a particular bird) flies is greater than 0.9". Halpern shows that both approaches can actually be combined and tackles the problem of providing sound and complete axioms to characterize probabilistic reasoning in both types of setup. Even though previous complexity results [AH94] show that a *complete* axiomatization is not possible, sound axioms are provided for both semantics that are rich enough to perform interesting probabilistic reasoning. An interesting parallel is drawn between

this situation and Gödels incompleteness result for the axiomatization of arithmetic and Peano's sound (but incomplete) axioms.

The work of Jaumard et al. [JHdAa90] is very interesting in that it is quite possibly the first ever to report on actual implementations of probabilistic logic. The authors consider extensions to Nilsson's probabilistic logic that involve intervals of probability values, conditional probabilities, and a novel problem involving the computation of minimal modifications of probabilistic annotations to restore satisfiability to a set of sentences. The main focus of this work is to propose a *column generation* approach which allows to solve exactly all these extensions. Column generation [GG61, Chv83] is a technique designed to efficiently solve linear programs (such as the ones that arise in the semantics of *ap*-programs, as discussed in Chapter 2) with very large numbers of variables by keeping them implicit. It exploits Duality Theory by mapping variables into constraints and vice versa, and extends the Revised Simplex Method by determining the entering column by solving an auxiliary subproblem (also usually called the *oracle*, or the *column generator*). Finding such a column by enumerating all possibilities is intractable, and this subproblem depends on the type of problem considered but is usually one of combinatorial programming. Solving the subproblem is usually NP-hard; however, it is not necessary to solve it exactly in each round, as long as negative reduced cost is found. When heuristics no longer yield such values, an exact algorithm must be used, which is a weakness of the method. One of the main results of [JHdAa90] is that the *pricing problem* (finding the inputs to the oracle) can be reduced to an instance of weighted MAX-SAT, which implies that PSAT can be solved in polynomial time (via the ellipsoid method) for those classes of formulas where weighted MAX-SAT can be solved in polynomial time.

Another interesting work in probabilistic logic is that of Frisch and Haddawy [FH94], which extends Nilsson's work (discussed above) with conditional probabilities and interval probabilistic annotations, and adopts an inference rule-based approach to deduction (a completely novel approach at the time). Inference rules allow proofs to be produced along with explanations of how conclusions were obtained; they also allow the authors to devise an anytime algorithm that computes increasingly narrow probability intervals for the entailed probabilities, which is the main focus of this work. One of the main advantages is that this process does not require finding *all truth assignments* that are consistent with the given sentences (an NP-complete problem); also, the approach is quite flexible since rules can be easily added. The procedure starts with $[0, 1]$ as a current estimate and proceeds by enumerating all possible proofs given the rules, and intervals are updated by intersection (multiple derivation rule); current estimates decrease monotonically and are thus always correct, allowing a tradeoff between computation time and precision in the derived probabilities.

Finally, the work of Andersen and Pretolani [AP01] is relevant to our own in that they aim towards tractable cases of probabilistic satisfiability (PSAT) in CNF formulas whose clauses are annotated with probabilities. The authors point out that PSAT is computationally harder than SAT, since it remains difficult for cases in which SAT is not. Their goal is then to identify easy cases of PSAT, in which it is possible to give a compact representation of the set of consistent probability assignments. Two different approaches are taken, based on different representations of CNF formulas. The first is based on directed hypergraphs; extending an integer programming formulation of MAX-SAT, it is possible to solve the case in which the hypergraph has no cycles. If a formula is represented by a hypertree, its associated

matrix is balanced, which allows to project constraints on an n-dimensional space, leading to a small linear program. The other approach is based on co-occurrence graphs, and the authors provide a solution for the case in which the graph is a partial 2-tree. In this case, the main result states that PSAT can be reduced to solving a system of $O(n)$ equations in $O(n)$ non-negative variables for formulas representable by partial 2-trees.

In this section, we have presented some of the most important developments in probabilistic logic, which is the basis of the work developed in this thesis. In the next section, we will discuss work on probabilistic logic programs and other related formalisms, and therefore more closely related to our work on action probabilistic logic programs.

3.2 Probabilistic Logic Programming and Related Formalisms

In this section, we will discuss the first approaches to probabilistic logic programs, and then analyze the developments that followed them and how they relate to the work developed in this thesis.

3.2.1 The Origins of Probabilistic Logic Programming

Probabilistic logic programming, which is the basis of our work, was first introduced by Ng and Subrahmanian in [NS91, NS92]; these two papers addressed the problem of combining logic programming [Llo87] with probability theory, adopting semantics in the style of Nilsson and Halpern as discussed above. The authors point out that all semantics proposed for quantitative logic programming prior to their

work had been non-probabilistic, of which [vE86] and [Sha83] are examples; this probabilistic approach aims at developing a probabilistic model theory and fixpoint theory. Logical treatment of probabilities in logic programming is complicated by two facts: first, connectives cannot be interpreted truth-functionally, and second, negation-free definite clause-like sentences can still be (probabilistically) inconsistent. The general form of rules in their formalism is:

$$F_0 : \mu_0 \leftarrow F_1 : \mu_1 \wedge \ldots \wedge F_n : \mu_n$$

where the F_i are *basic formulas* (conjunctions or disjunctions of atoms) and the μ_i are probabilistic annotations in the form of intervals that may contain expressions with variables. In [NS92], heads of rules are restricted to annotated atoms, where negation is still supported by means of $[0, 0]$ annotations, but conditional probabilities are not expressible. This formalism is a general logical framework for expressing probabilistic information, and the authors study its semantics and its relationship with probability theory, model theory, fixpoint theory, and proof theory. They also develop a query processing procedure for answering queries about probabilities of events, which is different from query processing in classical logic programming since most general unifiers are not always unique and therefore *maximally general* unifiers must be computed. In this formalism, atomic formulas assign probability ranges (intervals) to atoms, and using these functions, a formula function determines probability ranges for non-atomic formulas by applying $\oplus$ and $\otimes$ (ignorance). The fixpoint operator is then defined on these formula functions.

3.2.2 Further Developments

Probabilistic logic programming was later studied by several authors: Ngo and Haddawy [NH95], Lukasiewicz and Kern-Isberner [LKI99], Lakshmanan and Shiri [LS01], Dekhtyar and Subrahmanian [DS97], Damasio et al. [DPS99], among others. Ngo and Haddawy [NH95] present a model theory, fixpoint theory, and proof procedure for conditional probabilistic logic programming. Lukasiewicz and Kern-Isberner [LKI99] combine probabilistic logic programming (also adopting an explicit treatment of conditional probabilities) with maximum entropy, as discussed above in relation to Nilsson's original proposal. In a closely related work, Lukasiewicz [Luk98] presents a conditional semantics for probabilistic logic programs where each rule is interpreted as specifying the conditional probability of the rule head, given the body. Lakshmanan and Shiri [LS01] developed a semantics for logic programs in which different general axiomatic methods are given to compute probabilities of conjunctions and disjunctions, and these are used to define a semantics for probabilistic logic programs. In [DS97], Dekhtyar and Subrahmanian consider different conjunction and disjunction strategies, originally introduced by Lakshmanan et al. in [LLRS97], and allow explicit syntax in probabilistic logic programs so that users are able to express their knowledge of a dependency. Damasio et al. [DPS99] present a well-founded semantics for annotated logic programs and show how to compute this well-founded semantics.

Even though there is a rich body of work on probabilistic logic programming, most works to date on this topic have addressed the problem of checking whether a given formula of the form $F : [\ell, u]$ is entailed by a probabilistic logic program [NS91, NS92] or is true in a specific model (*e.g.*, the well-founded model [DPS99]). This usually boils down to finding out if all interpretations that sat-

isfy the probabilistic logic program assign a probability between ℓ and u to F. An interesting extension of the concept of probabilistic entailment is proposed in [YLH08], where the authors propose to go one step further and check to what *degree of satisfaction* the query is entailed by the program, similar to the novel approach of Bröcheler et al. [BSS09], who propose to answer entailment queries with histograms indicating how the density of solutions are distributed in the probabilistic interval. In contrast, Chapters 4 and 5 in this thesis focus on finding *most probable worlds*. This work was motivated by an application that we have developed which tries to identify the most probable set of actions (such sets of actions correspond to worlds) that agents might take (strategically, not tactically; see Chapter 1) during a given real or hypothetic situation. Also quite different from the problems solved in the literature to date is our work on *abductive queries*, presented in Chapters 6 and 7. These queries consist of a condition over action atoms, annotated with a probabilistic interval, along with a set of rules and constraints over how the reasoning agent can change the environment to have the condition hold.

Our work is closely related to the *gp*-program paradigm [NS91]. The framework used here (*ap*-programs) was first presented in [SSNS06, KMN$^+$07a] and differs from *gp*-programs in three ways: (i) *ap*-programs do not allow extensional predicates to occur in rule heads, while *gp*-programs do allow them, (ii) *ap*-programs allow arbitrary formulas to occur in rule heads, whereas *gp*-programs only allow the so-called "basic formulas" to appear in rule heads. (iii) [NS91] solves the problem of probabilistic entailment, while we focus on the most probable world problem, and that of answering abductive queries.

Also related to the MPW problem is work done in obtaining *optimal models* of disjunctive logic programs [LSS04]. An optimal model of a disjunctive logic program

tries to find either a model, a minimal model, or a stable model of the DLP that maximizes an objective function. The techniques there assume no knowledge of the objective function (except for monotonicity). In contrast, our techniques use a form of probabilistic logic program not considered in [LSS04]. Moreover, our techniques set up a linear program that is associated with an *ap*-program, while their work has no such linear program. The complexities associated with finding the most probable world arise from the linear programming formulation because the linear programs are exponential in size, containing one variable for each world. This does not occur in the non-probabilistic framework of [LSS04]. Their work gives sound and complete ways to find optimal models by doing a generate-and-test of models, along with some intelligent pruning. Our work focuses directly on how to solve the linear program to find appropriate worlds. Moreover, our work provides the first heuristic methods that scales to large scenarios.

Another related effort in the agents world is that of optimal feasible status sets [SSD05]. Optimal status sets build upon the status set semantics for agent programs [ESP99]. Status sets are sets of actions that an agent can take in a given situation. They bear the same relationship to agent programs that models bear to logic programs. The work on optimal status sets improves upon work such as that in [LSS04] because it provides a *non-ground* framework which avoids grounding out agent programs until it is necessary. This is a big contribution that we would like to extend to *ap*-programs as well. However, the work of [SSD05] has the same differences from the work in this thesis as [LSS04].

Apart from our own, the closest work to solving large linear programs is that of [JHdAa90] (discussed above) who use column generation methods to solve PSAT, CONDSAT, and minimal modifications to ensure satisfiability. Even though the re-

sults shown in that work suggest that the Column Generation method could applied to our work, it should be noted that the relevant problem (PSAT) that the authors are solving in this paper only requires solving one linear program, instead of once for *each world* as in the case of most probable worlds.

3.3 A Comparison of *ap*-programs to other Approaches to Probabilistic Reasoning

As an important note in this chapter on related work, we would like to compare the power of *ap*-programs with other formalisms designed for probabilistic reasoning. First of all, in the following example we show that *ap*-programs are well-suited for representing uncertainty *at the level of probability distributions.*

Example 6 (Multiple probability distributions given $LC(\Pi, s)$ and entailment)**.** *Consider* ap-*program* Π *from Figure 2.1 and state* s_2 *from Figure 6.2. The set of possible worlds is as follows:* $w_0 = \{\}$, $w_1 = \{\texttt{kidnap}(1)\}$, $w_2 = \{\texttt{tlethciv}(1)\}$, *and* $w_3 = \{\texttt{kidnap}(1), \texttt{tlethciv}(1)\}$. *Suppose we use* p_i *to denote the variable associated with the probability of world* w_i*;* $LC(\Pi, s_2)$ *then consists of the following constraints:*

$$0.5 \leq p_1 + p_3 \leq 0.56$$

$$0.49 \leq p_2 + p_3 \leq 0.55$$

$$p_0 + p_1 + p_2 + p_3 = 1$$

One possible solution to this set of constraints is $p_0 = 0$, $p_1 = 0.51$, $p_2 = 0.05$, *and* $p_3 = 0.44$*; another possible distribution is* $p_0 = 0.5$, $p_1 = 0$, $p_2 = 0$, *and* $p_3 = 0.5$*; yet another one is* $p_0 = 0$, $p_1 = 0.45$, $p_2 = 0.11$, *and* $p_3 = 0.44$. *Finally, formula* $\texttt{kidnap}(1) \wedge \texttt{tlethciv}(1)$ *(satisfied only by world* w_3*) is entailed with probability in*

the interval $[0, 0.55]$*, meaning that one cannot assign a probability greater than* 0.55 *to this formula*[1].

Note that representing a set of distributions is not possible in many other approaches to probabilistic reasoning, such as Bayesian networks [Pea88], Poole's Independent Choice Logic [Poo97] and related formalisms such as [Poo93b]. However, this is a key capability for our approach, as we specifically require a formalism that is not forced to make assumptions about the probabilistic dependence (or independence) of the events we are reasoning about.

On the other hand, it is certainly possible to extend our approach in such a way that the key aspects of Bayesian networks and related formalisms are directly expressible, as was shown in [NS93] when probabilistic logic programs were first introduced. There are two key extensions that we need to make:

1. *Allow annotated action atoms in the bodies of rules:* Even though this is an extension w.r.t. the language introduced here, the original formulation [KMN+07a] already included this capability, and was not introduced here for the sake of brevity. Essentially, by means of a simple fixpoint operator, it was shown that an equivalent program without annotated action atoms in the body of rules can be obtained.

2. *Allow probabilistic annotations to contain variables:* The main goal is to allow probabilistic annotations in the head of rules to depend on those in the body.

Extension 2 is by no means a novel idea. The work of [NS93], on which *ap*-programs are directly based, included variables as part of their language. This extension was

[1] This example shows that, contrary to what one might think, the interval $[0, 1]$ is not necessarily a solution.

not included in the present work (also for reasons of space), but can clearly be incorporated without great effort.

Once our language incorporates Extensions 1 and 2, it is possible to represent the following capabilities (the following is based on [NS93]):

Independence of events: Suppose we wish to represent the fact that any probability distribution that is a solution to $LC(\Pi, s)$ is such that the probability of action atom a is independent of that of action atom b. The following rule will add the necessary constraint to the set of solutions:

$$a \wedge b : [V_1 * V_2, V_1 * V_2] \leftarrow a : [V_1, V_1], b : [V_2, V_2]$$

where V_1, V_2 are variables that can take values in $[0, 1]$.

Conditional probabilities: We will now see how we can represent the knowledge that the probability that action atom a is true given that we know that action atom b is true lies in the interval $[p_1, p_2]$. As before, we can constrain all solutions to obey this relationship by adding the rule:

$$a \wedge b : [p_1 * V, p_2 * V] \leftarrow b : [V, V]$$

where V is a variable in $[0, 1]$. Similarly, suppose we represent the conditional probability of action atom a given b with ab. Then, the following rule constrains the space of solutions to give ab the correct value:

$$ab : [V_2/V_1, V_2, V_1] \leftarrow a \wedge b : [V_2, V_2], b : [V_1, V_1]$$

where V_1, V_2 are variables in $(0, 1)]$ and $[0, 1]$, respectively.

Therefore, even though it is a well known fact that Bayesian networks are capable of representing any *single* probability distribution, we have shown here that: (1) *ap*-programs are especially useful in cases in which we wish to express uncertainty about the probability distribution in question, and (2) *ap*-programs are capable of representing the basic constructs used in this family of formalisms, and therefore no expressivity is lost.

3.4 Probabilistic Abduction

In this section, we will discuss the literature that is most relevant to the work presented in Chapters 6 and 7.

Abduction, both in the classical and probabilistic formulations, is about finding explanations for observations or events. The concept was introduced by the philosopher Charles S. Peirce (1839-1914) [Pei40] as a third form of reasoning (distinct from deduction and induction). In its most basic form, abduction states that "when B is observed, and it is known that A leads to B, then A can be inferred". Peirce referred to the observation (B in this case) "surprising", in the sense that the observer does not have an explanation for it. Now, the abductive inference should only be interpreted as a hypothesis, since clearly there could be other explanations for the observation of B (*i.e.*, that C leads to A as well); this is the well known fallacy of affirming the consequent. Clearly, it is not necessary for the "observation" to have actually happened; as we will see, our work fits better within the frame of *hypothetical* observations.

In the field of Artificial Intelligence, abduction has been studied as a tool for non-monotonic reasoning and a complement to deductive approaches (theorem prov-

ing, etc.). Among others, it has been applied to diagnosis [PR90, CT91, Poo92], reasoning with non-monotonic logics [EGL97b, EG95], default reasoning [Poo88, Poo89], probabilistic reasoning [CS90, Pea91, Poo92, Poo93a, Poo93b, BK93, AH98, Jos08, Jos09], argumentation [KBH02], planning [Esh88, Sha00, dLPdB04], temporal reasoning [Esh88, Sha89] and belief revision [Pag96]; furthermore, it has been combined quite naturally with different variants of logic programs [BGMP97, EGL97a, DK02, Poo92, Poo93a, Sha00, KMM00, CMS08, Chr08].

Abductive logic programming refers to any of the many instantiations/extensions of the general model usually specified in the following manner [DK02]: an *abductive logic programming theory* is a triple (P, A, IC), where P is a logic program, A is a set of ground *abducible* atoms (that do not occur in the head of a rule in P), and IC is a set of classical logic formulas called *integrity constraints*. Then, an abductive explanation for a query Q is a set $\Delta \subseteq A$ such that $P \cup \Delta \models Q$, $P \cup \Delta \models IC$, and $P \cup \Delta$ is consistent. This is an abstract definition, independent of syntax and semantics; the variations in how such important aspects are defined has lead to many different models.

The work that is most relevant to ours lies at the intersection of probabilistic reasoning and logic programming. David Poole *et al.* worked in combining probabilistic and non-monotonic reasoning, leading to the development of the Theorist system [PGA87], Probabilistic Horn Abduction [Poo93b, Poo93a], and eventually the Independent Choice Logic [Poo97]. Similar recent work by Christiansen [Chr08] addresses the problem of probabilistic abduction with logic programs based on constraint handling rules. Though these models are related to our work, they either make general assumptions of pairwise independence of probabilities of events (such as in [Poo97] or [Chr08]) or are based on the class of so-called graphical models,

which includes the well-known Bayesian Networks (BNs) [Pea88]. In BNs, domain knowledge is represented in a directed acyclic graph in which nodes represent attributes and edges represent *direct probabilistic dependence*, whereas the lack of an edge is interpreted as independence. Joint probability distributions can therefore be obtained from the graph, and abductive reasoning in this domain is carried out by applying Bayes's theorem given these joint distributions and a set observations (or hypothetical events). Another important problem in BNs that is directly related to abductive inference is that of obtaining the *maximum a posteriori probability* (usually abbreviated MAP, and also called *most probable explanation*, or MPE) [AH98]. The main difference between graphical model-based work and our own is that we *make no assumptions on the dependence or independence of probabilities of events.*

While AI planning may seem relevant, there are several differences. First, we are not assuming knowledge of the effects of actions; second, we assume the existence of a probabilistic model underlying the behavior of the entity being modeled. In this framework, we want to find a state such that *when the atoms in the state are added to the* ap-*program, the resulting combination* entails *the desired goal with a given probability.* While the italicized component of the previous sentence can be achieved within planning, it would require a state space that is exponentially larger than the one we use (in this space, the search space would be the set of all sets of atoms closed under consequence that are jointly entailed by any subprogram of the *ap*-program and any state (under the definition in our approach). This would cause states to be potentially exponentially bigger than those in this chapter and would also exponentially increase their number.

To the best of our knowledge, this is the first work that tackles the problem of abductive reasoning in probabilistic logic programming under no independence

assumptions, in the tradition of the works of [NS91] and [NS92] for probabilistic logic programming, and [Hai84], [Nil86], and [FHM90] for probabilistic logic in general. As we are adopting the class of *action* probabilistic logic programs, it is natural to consider abductive reasoning with respect *goals* instead of observations (as is done sometimes when the logic programming perspective of abductive inference is adopted). Hence, we are not looking for the most probable explanation for an observation, but rather the state of affairs that *would explain* the truth of a given goal; furthermore, in order for this state of affairs to be a solution, it is necessary for it to be *attainable* from the current one. As we will see, this last constraint adds a layer of complexity to the problem.

3.5 Reasoning about Promises, Trust, and Reputation in Autonomous Agents

This section presents the literature most relevant to the formalism we present in Chapter 8 for incorporating into *ap*-programs the capability of representing knowledge about fulfillment of promises.

3.5.1 Trust and Reputation in "offer, accept, reject" Negotiations

There is extensive past work on developing models of trust in agent systems. The most relevant perhaps is that of Sierra et al. [SD07a], which presents a model of decision making based on trust in simple *Offer, Accept, Reject* negotiations between autonomous agents. Decision-making in this model integrates the utilitarian, infor-

mation, and semantic views of the exchange of information, and the authors present summary measures that generalize trust, reliability, and reputation as an illustration of the model's capabilities. However, promises of the kind we discuss in Chapter 8 are not considered. Another important difference with our approach is that these measures assume the availability of probability distributions that describe the *ideal* enactments with respect to a given commitment, expected enactments, a more general *semantic similarity* measure that allows to gauge the similarity between the commitment and its actual enactment, and a measure of how much uncertainty we expect to have given a certain commitment. In a related paper by the same authors [SD07b], the focus is on measuring trust as the foundation for confidence between agents that sign contracts. Their modeling of trust is based on a conditional probability that describes the probability of observing a certain enactment given a previously established contract and a context. Every contract execution will represent a point in this distribution. Information held by agents is updated based on the passage of time (decay) for information regarding previous enactments, as well as from perceived preferences, for information regarding how likely other agents are to accept certain offers. As before, the authors show how different probability distributions can be used to obtain measures of trust according to different views (as expected behavior, as expected preferability, and as certainty in contract execution).

The following example illustrates the way trust is modeled in [SD07a, SD07b]. Suppose two agents, s and vs engage in a negotiation which ends in the following promise made by s to vs: *"If it doesn't rain tomorrow, I will come to the office"*. We model this promise in the *content language* of [SD07b] as agents s and vs agreeing on a deal $\delta = (a, b)$, where $b = \emptyset$ and

$$a = \text{if } noRain(w, t_{now} + 1) \text{ then } goToOffice(s, t_{now} + 1)$$

Where $noRain(w, t_{now}+1)$ means that agent w (for *weather*) will execute the action *no rain* tomorrow, and $goToOffice(s, t_{now}+1)$ means that agent s will go to the office tomorrow.

For the first measure of trust discussed in [SD07b] ("trust as expected behavior"), we require a distribution of *ideal enactments* of the contract from the point of view of agent vs, that is, what vs hopes s will do in this situation. This distribution involves *all possible* enactments a' of the contract; even though the framework allows for more richness in this consideration, we will adopt a simplified case here. One such distribution, denoted by $P_I^{t_{now}}(a' \mid a)$ could be:

$$P_I^{t_{now}}(noRain(w, t_{now}+1) \wedge goToOffice(s, t_{now}+1) \mid a) = 0.40$$
$$P_I^{t_{now}}(rain(w, t_{now}+1) \wedge goToOffice(s, t_{now}+1) \mid a) = 0.30$$
$$P_I^{t_{now}}(noRain(w, t_{now}+1) \wedge notGoToOffice(s, t_{now}+1) \mid a) = 0$$
$$P_I^{t_{now}}(rain(w, t_{now}+1) \wedge notGoToOffice(s, t_{now}+1) \mid a) = 0.30$$

We also require a distribution that reflects what vs actually *expects* s to do in this scenario, that is, what is the probability assigned by vs to each possible enactment a' of a. We denote this distribution by $P_{vs}^{t_{now}}(a' \mid a)$, and one possibility is the following:

$$P_{vs}^{t_{now}}(noRain(w, t_{now}+1) \wedge goToOffice(s, t_{now}+1) \mid a) = 0.40$$
$$P_{vs}^{t_{now}}(rain(w, t_{now}+1) \wedge goToOffice(s, t_{now}+1) \mid a) = 0.15$$
$$P_{vs}^{t_{now}}(noRain(w, t_{now}+1) \wedge notGoToOffice(s, t_{now}+1) \mid a) = 0.05$$
$$P_{vs}^{t_{now}}(rain(w, t_{now}+1) \wedge notGoToOffice(s, t_{now}+1) \mid a) = 0.40$$

Finally, the measure of trust that vs has in s with respect to this contract is calculated as the relative entropy between these two probability distributions, which

is calculated as:

$$T(vs, s, a) = 1 - \sum_{a' \in A(a)} P_I^{t_{now}}(a' \mid a) \log \left(\frac{P_I^{t_{now}}(a' \mid a)}{P_{vs}^{t_{now}}(a' \mid a)} \right)$$

In this case, this calculation yields:

$$T(vs, s, a) = 1 - \left(0.40 \log \left(\frac{0.40}{0.40} \right) + 0.30 \log \left(\frac{0.30}{0.15} \right) + 0 + 0.30 \log \left(\frac{0.40}{0.40} \right) \right)$$

which reduces to $1 - (0 + 0.09 + 0 + -0.03) = 0.94$.

The second measure defined in [SD07b] is called *trust as expected preferability*, and is based on the definition of a predicate $\mathit{Prefer}(a', a)$, meaning that an agent prefers the enactment a' to the actual agreement a. Then, this predicate is used in a distribution $P^{t_{now}}(\mathit{Prefer}(a', a))$, and the measure is defined as:

$$T(vs, s, a) = \sum_{b'} P^{t_{now}}(\mathit{Prefer}(a', a)) P_{vs}^{t_{now}}(a' \mid a)$$

The third and last measure proposed is called *trust as certainty in contract execution*, and is based on the fact that trust can be thought of as "the lack of expected uncertainty in those possible executions that are better than the contract specified". They define $A_+(a, \kappa) = \{a' \mid P^{t_{now}}(\mathit{Prefer}(a', a)) > \kappa\}$. The measure is then defined as follows:

$$T(vs, s, a) = 1 + \frac{1}{A^*} \sum_{a' \in A_+(a,\kappa)} P_+^{t_{now}}(a' \mid a) \log P_+^{t_{now}}(a' \mid a)$$

where $P_+^{t_{now}}(a' \mid a)$ is the normalization of $P_{vs}^{t_{now}}(a' \mid a)$ for $a' \in A_+(a, \kappa)$, and $A^* = 1$ if $|A_+(a, \kappa)| = 1$ and $\log |A_+(a, \kappa)|$ otherwise.

Finally, we note that in each case the authors define extensions of these measures to apply to classes of promises (for instance, promises made by s to be at some place at some time), and also to measure the overall confidence that an agent has in another agent.

3.5.2 Other Related Work

The area of trust and reputation in agent systems (which can include both artificial and human agents), has seen a lot of activity in the last few years. Though not as closely related to our work as that of Sierra et al. (discussed above), the following deserves mention. Dellarocas [Del06] provides a recent survey of the area, focusing on Internet-based mechanisms such as for online auctions, one of the most important areas of application of work in this area. In [DB07], Dondio and Barrett propose a generic method of selecting evidence that is recognized as support for trust, attacking one of the basic questions regarding trust. Game-theoretic treatments of this topic have also been developed, such as in the work of Ely et al. [EFL02]. However, game-theoretic approaches has been criticized for placing too much importance on probability while underestimating its cognitive aspects; an example of such criticism is the work of Falcone and Castelfranchi [FC01]. In this respect, our work takes a step in this direction by allowing agents to influence the measure of fulfillment according to their own preferences (see Chapter 8 for details and examples).

3.6 Reasoning about Adversaries

In this last section, we will briefly discuss how work in adversarial reasoning is related to this thesis.

In terms of applications of *ap*-programs, there is growing need for reasoning about how diverse cultural, political, industrial and other organizations make decisions. Past work on adversarial reasoning in artificial intelligence has focused primarily on games such as Chess [Hsu02, GAKB02, GMSB10, NLP+10], Bridge [SNT96], Go [BC01, SHG06], Poker [SL02, BBD+03, GS06], and Checkers [SBB+07], where the rules of the game are well articulated and where the state is well defined. Reasoning about real world adversaries can be viewed as a complex game-tree search problem [Nau82, Nau83, NLP+10], but it is difficult to know the "rules" of the game, the state space is huge, and the state is often largely unknown. Often, even the variables constituting the state are unclear, let alone their values. Past adversarial reasoning work in AI can be a great asset even in real-world cultural reasoning, but a major problem is to identify the adversary's objectives and payoffs as well as the rules that the adversary adheres to, and determine how best to "play" the adversary given our knowledge of his behavioral rules. In this thesis, we show how some of these problems can be addressed by applying efficient algorithms for solving different kinds of problems using action probabilistic logic programs under various different assumptions.

Chapter 4

Computing Most Probable Worlds in Action Probabilistic Logic Programs

Probabilistic logic programs (PLPs) [NS92] have been proposed as a paradigm for probabilistic logical reasoning with no independence assumptions. PLPs used a possible worlds model based on prior work by [Hai84], [FHM90], and [Nil86] to induce a set of probability distributions on a space of possible worlds. Past work on PLPs [NS91, NS92] focuses on the entailment problem of checking if a PLP entails that the probability of a given formula lies in a given probability interval.

However, we have recently been developing several applications for cultural adversarial reasoning [SAM+07, Bha07] where PLPs and their variants are used to build a model of the behavior of certain socio-cultural-economic groups in different parts of the world. Our research group has thus far built models of approximately 30 groups around the world including tribes such as the Shinwaris and Waziris, terror groups like Hezbollah and the PKK, political parties such as the Pakistan People's Party and the Harakat-e-Islami as well as nation states. Of course, all these models only capture a few actions that these entities might take. Such PLPs contain rules that state things like

"There is a 50 to 70% probability that group g will take action(s) a when condition C holds in the current state."

In such applications, the problem of interest is that of finding the most probable action (or sets of actions) that the group being modeled might do in a given situation. This corresponds precisely to the problem of finding a "most probable world" that is the focus of this chapter.

In Section 4.1, we describe the *most probable world* (MPW) problem by immediately using the linear programming methods of [NS91, NS92] —these methods are exponential because the linear programs are exponential in the number of ground atoms in the language. The novel content of this chapter starts in Section 4.2 where we present the *Head Oriented Processing (HOP)* approach; HOP reduces the linear program for *ap*-programs, and we show that using HOP, we can often find a much faster solution to the MPW problem. We define a variant of HOP called SemiHOP that has slightly different computational properties, but are still guaranteed to find the most probable world. Thus, we have three exact algorithms to find the most probable world.

Subsequently, in Section 4.3, we develop a heuristic called the *Binary* heuristic that can be applied in conjunction with the *Naive*, HOP, and SemiHOP algorithms. The basic idea is that rather than examining all worlds corresponding to the linear programming variables used by these algorithms, only some fixed number k of worlds is examined. This leads to a linear program whose number of variables is k. Finally, Section 4.4 describes a prototype implementation of our *ap*-program framework and includes a set of experiments to assess combinations of exact algorithm and the heuristic. We assess both the efficiency of our algorithms, as well as the accuracy of the solutions they produce. We show that the SemiHOP algorithm with the binary

heuristic is quite accurate (at least when only a small number of worlds is involved) and then show that it scales very well, managing to handle situations with over 10^{27} worlds in a few minutes.

4.1 Maximally Probable Worlds

We are now ready to introduce the problem of, given an *ap*-program and a current state, finding the most probable world. As explained through our Hezbollah example, we may be interested in knowing what actions a group might take in a given situation.

Definition 13 (lower/upper probability of a world)**.** *Suppose* Π *is an* ap-*program and* s *is a state. The* lower probability*,* $\mathsf{low}(w_i)$ *of a world* w_i *is defined as:* $\mathsf{low}(w_i) =$ **minimize** p_i ***subject to*** $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$*. The* upper probability*,* $up(w_i)$ *of world* w_i *is defined as* $up(w_i) =$ **maximize** p_i ***subject to*** $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$*.*

Thus, the lower probability of a world w_i is the lowest probability that that world can have in any solution to the linear program. Similarly, the upper probability for the same world represents the highest probability that that world can have. It is important to note that for any world w, we cannot *exactly* determine a point probability for w. This observation is true even if all rules in Π have a point probability in the head because our framework *does not make any simplifying assumptions* (e.g. independence) about the probability that certain things will happen.

We now present two results that state that checking if the low (resp. up) probability of a world exceeds a given bound (called the BOUNDED-LOW and BOUNDED-UP problems respectively) is intractable. The NP-hardness results, in

both cases, are by reduction from the problem of checking consistency of a generalized probabilistic logic program (PLP-CONS), whose proof is given below.

First, we reproduce a result that states that we can be guaranteed a solution to a linear program where only a number of the variables linear in the number of constraints are set to a non-zero value. This result was first introduced in [Chv83], and later used in [FHM90] to show that deciding the validity of a formula in their logic is NP-Complete.

Lemma 1 ([Chv83, FHM90]). *If a system of m linear equalities and/or inequalities has a nonnegative solution, then it has a nonnegative solution with at most m positive variables.*

We now show that checking consistency of an *ap*-program is NP-complete.

Proposition 2. *The problem of deciding if a probabilistic logic program* Π *is consistent in a state* s *is* NP*-complete.*

Proof. Membership in NP: By Lemma 1, we know that if Π is consistent, then there will be a polynomial number of variables in $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ with non-zero values. Therefore, this set of values constitutes our witness, whereas the rest of the variables are implicitly assigned a value of zero.

NP-Hardness: We will perform a reduction of the *boolean formula satisfiability* problem (SAT) to checking consistency of Π with respect to a given state s (PLP-CONS). In order to perform the reduction, we must define a polynomial time computable function R that maps an arbitrary boolean formula f into an instance of PLP-CONS such that f is satisfiable if and only if $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ is solvable (note that $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ is solvable if and only if Π is consistent in state s; this well known property was proved in [NS92]). The PLP-CONS instance will correspond to

a simplified version of the problem, in which only one rule is present, and the upper and lower probabilities are equal to 1. Define:

$$R(f) = \{f : [1, 1] \leftarrow\}$$

to be the PLP built from an arbitrary ground formula f. It is clear that this transformation can be performed in polynomial time. We will now prove that f is satisfiable if and only if Π is consistent with respect to the empty state, (*i.e.*, $\mathsf{CONS}_U(R(f), \emptyset, T^{\omega}_{\Pi_s})$ is solvable):

- f is satisfiable $\Rightarrow \mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ is solvable: By hypothesis, there exists an assignment of variables in f such that f is true. We use such values to build a world w such that formula f is satisfied by w. Because the upper and lower probabilities in the rule are both 1, we can assign 1 to p_w, the probability of world w, whereas every other world receives probability 0. We have therefore constructed a solution to the constraints that proves that Π is consistent in s.

- $\mathsf{CONS}\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ is solvable $\Rightarrow f$ is satisfiable: By hypothesis, there exists a solution to $\mathsf{CONS}\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ that assigns a probability p_{w_i} to each world that satisfies f. Because the set of worlds satisfying f is nonempty, this means it is possible to find at least one assignment for the variables in f such that f is satisfied.

We have therefore shown that SAT is polynomial time reducible to checking consistency in PLP, and therefore this problem is *NP*-hard. □

The results above are used in proving the following propositions.

Proposition 3 (BOUNDED-LOW complexity)**.** *Given a ground* ap-*program* Π*, a state* s*, a world* w*, and a probability threshold* p_{th}*, deciding if* $low(w) \geq p_{th}$ *is* NP-*complete.*

Proof. Membership in NP: In the same way membership was shown in Proposition 2, Lemma 1 tells us that for any "yes" instance of the problem there will be a polynomial number of variables in $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ with non-zero values. Therefore, this set of values constitutes our witness, whereas the rest of the variables are implicitly assigned a value of zero.

NP-hardness: We will reduce the PLP-CONS problem to the problem of deciding if a certain world w is such that $low(w) \geq p_{th}$ for a certain probability value p_{th}. Because PLP-CONS was proven to be *NP*-hard above, this reduction will prove that BOUNDED-LOW is *NP*-hard as well.

Given an instance of PLP-CONS consisting of a program Π and a state s, we build an instance of BOUNDED-LOW, consisting of an *ap*-program Π', a state s', a world w, and a probability threshold p_{th} in the following manner: program Π' is equal to Π and state s' is equal to s, world w is an arbitrary world, and $p_{th} = 0$. We must now show that this transformation yields a reduction by proving that Π is consistent in state s if and only if $low(w) \geq 0$ with respect to Π' and state s':

- Π is consistent $\Rightarrow$ $low(w) \geq 0$ with respect to Π' in state s': If Π is consistent, this means that $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ is solvable. Therefore, it is clear that any possible world will receive a probability value greater than or equal to zero.

- $low(w) \geq 0$ with respect to Π' in state s' $\Rightarrow$ Π is consistent: If $low(w) \geq 0$ with respect to Π' in state s', this means that w has received a probability

value greater than or equal to zero, subject to $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$. This is only possible if $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ is solvable, which means that Π is consistent.

Note that, whenever Π is inconsistent, the value of $low(w)$ is undefined, for any possible world w. To complete the proof, we note that the transformation from a PLP-CONS instance to a BOUNDED-LOW instance can be done in polynomial time with respect to the size of the *ap*-program given for PLP-CONS. $\square$

Proposition 4 (BOUNDED-UP complexity). *Given a ground* ap-*program* Π*, a state* s*, a world* w*, and a probability threshold* p_{th}*, deciding if* $up(w) \leq p_{th}$ *is* NP-*complete.*

Proof. Membership in NP: Analougous to Proposition 3

NP-hardness: This proof is very similar to the one for the *NP*-hardness of BOUNDED-LOW (Proposition 3). As before, we will reduce the PLP-CONS problem to the problem of deciding if a certain world w is such that $up(w) \leq p_{th}$ for a certain probability value p_{th}. Because PLP-CONS was proven to be *NP*-hard above, this reduction will prove that BOUNDED-UP is *NP*-hard as well.

Given an instance of PLP-CONS consisting of a program Π and a state s, we build an instance of BOUNDED-UP, consisting of an *ap*-program Π', a state s', a world w, and a probability threshold p_{th} in the following manner: program Π' is equal to Π and state s' is equal to s, world w is an arbitrary world, and $p_{th} = 1$. We must now show that this transformation yields a reduction by proving that Π is consistent in state s if and only if $up(w) \leq 1$ with respect to Π' and state s':

- Π is consistent $\Rightarrow up(w) \leq 1$ with respect to Π' in state s': If Π is consistent, this means that $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ is solvable. Therefore, it is clear that any possible world will receive a probability value less than or equal to one.

- $up(w) \leq 1$ with respect to Π' in state $s' \Rightarrow \Pi$ is consistent: If $up(w) \leq 0$ with respect to Π' in state s', this means that w has received a probability value less than or equal to one, subject to $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$. This is only possible if $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ is solvable, which means that Π is consistent.

Note that, whenever Π is inconsistent, the value of $up(w)$ is undefined, for any possible world w. To complete the proof, we note that the transformation from a PLP-CONS instance to a BOUNDED-UP instance can be done in polynomial time with respect to the size of the *ap*-program given for PLP-CONS. □

We will now present the *Most Probable World* problem (MPW for short).

The MPW Problem. The *most probable world* problem (MPW for short) is the problem where, given an *ap*-program Π and a state s as input, we are required to find a world w_i where $low(w_i)$ is maximal.[1]

The MPW problem can therefore easily be shown to be in Σ^p_2; given the *NP*-completeness result of simply checking if the lower probability exceeds a threshold, computing the most probable world is highly intractable. In the next section, we will present exact algorithms to solve the MPW problem.

4.2 Exact Algorithms for finding a Maximally Probable World

In this section, we develop algorithms to find the most probable world for a given *ap*-program and a current state. As the above results show us, there is no unique probability associated with a world w; the probability could range anywhere

[1]A similar **MPW-Up Problem** can also be defined. The *most probable world-up* problem (MPW-Up) is: given an *ap*-program Π and a state s as input, find a world w_i where $up(w_i)$ is maximal. In this thesis, we will mainly address the MPW problem.

```
Algorithm 2:  NaiveMPW
1. Compute T^ω_{Π_s};
2. Best = NIL;
3. Bestval = 0;
4. For each world w_i,
5.     Compute low(w_i) by minimizing p_i subject to the set
         CONS_U (Π, s, T^ω_{Π_s}) of constraints.
6.     If low(w_i) > Bestval then set
         Best = w_i and Bestval = low(w_i);
7. If Best = NIL then
      return any world whatsoever
8. else return Best.
```

Figure 4.1: The Naive algorithm for finding a most probable world.

between $low(w)$ and $up(w)$. Hence, in the rest of this chapter, we will assume the worst case, i.e. the probability of world w is given by $low(w)$. We will try to find a world for which $low(w)$ is maximized.

In this section, we study the following problem: given an *ap*-program Π and a state s, find a world w such that $low(w)$ is maximized. If we replace $low(w)$ by $up(w)$, the techniques to find a world w such that $up(w)$ is maximal are similar (though not all apply directly). There may also be cases in which we are interested in using some other value (e.g. the average of $low(w)$ and $up(w)$ and so on).

A Naive Algorithm. The *naive* algorithm to find the most probable world is the direct implementation of the definition of the problem, and it basically consists of the steps described in Figure 4.1.

The Naive algorithm does a brute force search after computing $T^{\omega}_{\Pi_s}$. It finds the low probability for each world and chooses the best one. Clearly, we can use it to solve the MPW-Up problem by replacing the minimization in Step 2(a) by a maximization.

There are two key problems with the naive algorithm. The first problem is that in Step (1), computing $T^{\omega}_{\Pi_s}$ is very difficult. When some syntactic restrictions

are imposed, this problem can be solved without linear programming at all as in the case when Π is a probabilistic logic program (or p-program as defined in [NS92]) where all heads are atomic.

The second problem is that in Step 2(a), the number of (linear program) variables in $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ is exponential in the number of ground atoms. When this number is, say 20, this means that the linear program contains over a million variables. However, when the number is say 30 or 40 or more, this number is inordinately large. In this chapter, when we say that we are focusing on lowering the computation time of our algorithms, we are referring to improving Step 2(a).

In this section, we will present two algorithms, the HOP and the SemiHOP algorithms, both of which can significantly reduce the number of variables in the linear program by collapsing multiple linear programming variables into one. They both stem from the basic idea that when variables *always* appear in certain groups in the linear program, these groups can be *collapsed* into a single variable. As we will see, the basic idea can lead to great savings, but being too ambitious in trying to collapse all possible sets can be detrimental to our benefits; this last observation is the root of the second algorithm.

4.2.1 HOP: Head-Oriented Processing

We can do better than the naive algorithm without losing any precision in the calculation of a most probable world. In this section we present the HOP algorithm, prove its correctness, and propose an enhancement that also provably yields a most probable world.

Given a world w, state s, and an *ap*-program Π, let $Sat(w) = \{F \mid c$ is a ground instance of a rule in Π_s and $Head(c) = F : \mu$ and $w \models F\}$. Intuitively, $Sat(w)$ is

the set of heads of rules in Π_s (without probability annotations) whose heads are satisfied by w.

Definition 14. *Suppose* Π *is an* ap-*program,* s *is a state, and* w_1, w_2 *are two worlds. We say that* w_1 *and* w_2 *are equivalent, denoted* $w_1 \sim w_2$*, iff* $Sat(w_1) = Sat(w_2)$*.*

In other words, we say that two worlds are considered equivalent if and only if the two worlds satisfy the formulas in the heads of exactly the same rules in Π_s. It is easy to see that $\sim$ is an equivalence relation; we use $[w_i]$ to denote the $\sim$-equivalence class to which a world w_i belongs. The intuition for the HOP algorithm is given in Example 7.

Example 7. *Consider the set* $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ *of constraints. For example, consider a situation where* $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ *contains just the three constraints below:*

$$0.7 \leq p_2 + p_3 + p_5 + p_6 + p_7 + p_8 \leq 1 \tag{4.1}$$

$$0.2 \leq p_5 + p_7 + p_8 \leq 0.6 \tag{4.2}$$

$$p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + p_7 + p_8 = 1 \tag{4.3}$$

In this case, each time one of the variables p_5*,* p_7*, or* p_8 *occurs in a constraint, the other two also occur. Thus, we can replace these by one variable (let's call it* y *for now). In other words, suppose* $y = p_5 + p_7 + p_8$*. Thus, the above constraints can be replaced by the simpler set*

$0.7 \leq p_2 + p_3 + p_6 + y \leq 1$

$0.2 \leq y \leq 0.6$

$p_1 + p_2 + p_3 + p_4 + p_6 + y = 1$

The process in the above example leads to a reduction in the size of the set $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$. Moreover, suppose we minimize y subject to the above constraints. In this case, the minimal value is 0.2. As $y = p_5 + p_7 + p_8$, it is immediately obvious that the low probability of any of the p_i's is 0. Note that we can also group p_2, p_3, and p_6 together in the same manner.

We build on top of this intuition. The key insight here is that for any $\sim$-equivalence class $[w_i]$, the entire summation $\sum_{w_j \in [w_i]} p_j$ either appears *in its entirety* in a constraint of type (1) in $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ or does not appear at all. This is what the next result states.

Proposition 5. *Suppose* Π *is an* ap*-program,* s *is a state, and* $[w_i]$ *is a* $\sim$*-equivalence class. Then for each constraint* c *of the form*

$$\ell \leq \sum_{w_r \models F} p_r \leq u \tag{4.4}$$

in $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$*, either every variable in the summation* $\sum_{w_j \in [w_i]} p_j$ *appears in the summation in (4.4) above or no variable in the summation* $\sum_{w_j \in [w_i]} p_j$ *appears in the summation in (4.4).*

Proof. Let c be a constraint of the form (4.4) and suppose for a contradiction that there exist two variables, p_x and p_y such that $w_x, w_y \in [w_i]$ and p_x appears in the constraint c, while p_y does not. In this case, $w_x \models F$ and $w_y \not\models F$. However, in this case, $w_x \not\sim w_y$, and therefore cannot be in the same equivalence class $[w_i]$, yielding a contradiction. $\square$

Example 8. *Here is a toy example of this situation. Suppose* Π_s *consists of the two very simple rules:*

$$(a \vee b \vee c \vee d) : [0.1, 0.5] \leftarrow .$$
$$(a \wedge e) : [0.2, 0.5] \leftarrow .$$

Assuming our language contains only the predicate symbols a, b, c, d, e*, there are* 32 *possible worlds. However, what the preceding proposition tells us is that we can group the worlds into four categories. Those that satisfy both the above head formulas (ignoring the probabilities), those that satisfy the first but not the second head formula, those that satisfy the second but not the first head formula, and those that satisfy neither. This is shown graphically in Figure 4.2, in which* p_i *is the variable in the linear program corresponding to world* w_i*. For simplicity, we numbered the worlds according to the binary representation of the set of atoms. For instance, world* $\{a, c, d, e\}$ *is represented in binary as* 10111*, and thus corresponds to* w_{23}*. Note that only three variables appear in the new linear constraints; this is because it is not possible to satisfy* $\neg(a \vee b \vee c \vee d \vee e)$ *and* $(a \wedge e)$ *at the same time.*

Effectively, what we have done is to modify the number of variables in the linear program from $2^{card(\mathcal{L}_{act})}$ to $2^{card(\Pi_s)}$ —a saving that can be significant in some cases (though not always, and in some cases it can actually result in an increase in size). The number of constraints in the linear program stays the same. Formally speaking, we define a *reduced set of constraints* as follows.

Definition 15 ($\mathsf{RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$)**.** *For each equivalence class* $[w_i]$*, the linear program* $\mathsf{RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ *uses a variable* p'_i *to denote the summation of the probability of each of the worlds in* $[w_i]$*. For each* ap-*wff* $F : [\ell, u]$ *in* $T^{\omega}_{\Pi_s}$*,*

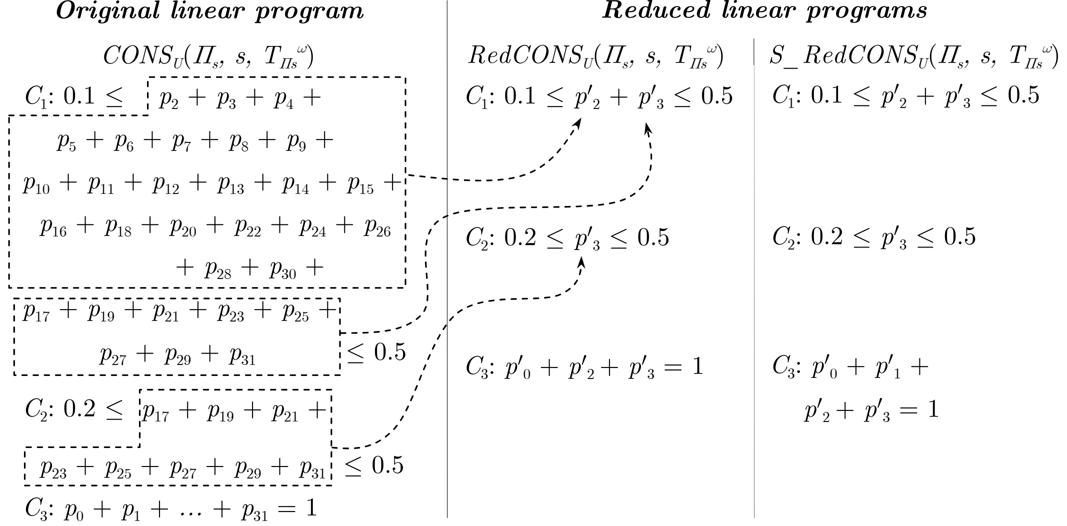


Figure 4.2: Reducing $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ by grouping variables. The new LPs are called $\mathsf{RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ and $\mathsf{S_RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$, as presented in Definitions 15 and 17.

$\mathsf{RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ *contains the constraint:*

$$\ell \leq \sum_{[w_i] \models F} p'_i \leq u.$$

Here, $[w_i] \models F$ *means that some world in* $[w_i]$ *satisfies* F*. In addition, the set of constraints* $\mathsf{RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ *contains*

$$\sum_{[w_i]} p'_i = 1.$$

When reasoning about $\mathsf{RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$, we can do even better than mentioned above. The result below states that to find the most probable world, we only need to look at the equivalence classes that are of cardinality 1.

Theorem 1. *Suppose* Π *is an* ap*-program,* s *is a state, and* w_i *is a world. If* $card([w_i]) > 1$*, then* $low(w_i) = 0$.

Proof. Immediate, by observing that there are no restrictions on the values assigned to the variables that correspond to worlds in the same $\sim$-class. If there is more than one world in a class $[w_x]$, there is always a solution that assigns zero to each variable p_i such that $w_i \in [w_x]$, and therefore $low(w_i) = 0$. □

Going back to Example 7, we can conclude that $low(w_5) = low(w_7) = low(w_8) = 0$. As a consequence of this result, we can suggest the Head Oriented Processing (HOP) algorithm which works as follows. First we present some simple notation. Let $\mathit{FixedWff}(\Pi, s) = \{F \mid F : \mu \in U_{\Pi_s}(T^{\omega}_{\Pi_s})\}$. Given a set $X \subseteq \mathit{FixedWff}(\Pi, s)$, we define $Formula(X, \Pi, s)$ to be

$$\bigwedge_{G \in X} G \wedge \bigwedge_{G' \in \mathit{FixedWff}(\Pi,s)-X} \neg G'.$$

Here, $Formula(X, \Pi, s)$ is the formula which says that X consists of all and only those formulas in $\mathit{FixedWff}(\Pi, s)$ that are true. Given sets $X_1, X_2 \subseteq \mathit{FixedWff}(\Pi, s)$, we say that $X_1 \sim X_2$ if and only if $Formula(X_1, \Pi, s)$ and $Formula(X_2, \Pi, s)$ are logically equivalent.

Theorem 2 (correctness of HOP)**.** *Algorithm HOP is correct, i.e. it is guaranteed to return a world whose low probability is greater than or equal to that of any other world.*

Proof. We will prove this property in two stages:

- *Soundness:* We wish to show that if HOP returns a world w_{sol}, then there is no other world w_i such that $low(w_i) > low(w_{sol})$. Suppose HOP does return w_{sol} but that there is a world w_i such that $low(w_i) > low(w_{sol})$. Clearly, $[w_i]$ and $[w_{sol}]$ must be different $\sim$-equivalence classes. In this case, step 3 of the

Algorithm 3: HOP Algorithm.

1. Compute $T^{\omega}_{\Pi_s}$. $bestval = 0$; $best = NIL$.
2. Let $[X_1], \ldots, [X_n]$ be the $\sim$-equivalence classes defined above for Π, s.
3. For each equivalence class $[X_i]$ do:
   (a) If there is exactly one interpretation that satisfies $Formula(X_i, \Pi, s)$ then:
      i. **Minimize** p'_i **subject to** $\mathsf{RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ where $[w_i]$ is the set of worlds satisfying exactly those heads in X_i. Let Val be the value returned.
      ii. If $Val > best$, then $\{best = w_i; bestval = Val\}$.
4. If $bestval = 0$ then return any world whatsoever, otherwise return $best$.

Figure 4.3: The Head-Oriented Processing (HOP) algorithm.

HOP algorithm will consider both these equivalence classes. As $bestval$ is set to the highest value of $low(w_j)$ for all equivalence classes $[w_j]$, it follows that $low(w_{sol}) \leq low(w_i)$, yielding a contradiction.

- *Completeness:* We wish to show that if there exists a world w_{max} such that $low(w_{max}) \geq low(w_i) \forall w_i \in \mathcal{W}$, then HOP will return a world w_{sol} such that $low(w_{sol}) = low(w_{max})$. Similar to the case made for soundness, if there exists a world w_{max} with the highest possible low value, it is either in the same class as the world that is returned by the algorithm, or in a different class. In the former case, the world returned clearly has the same value as w_{max}; in the latter, this must also be the case, since otherwise the algorithm would have selected the variable corresponding to $[w_{max}]$ instead.

This concludes the proof, and we therefore have that HOP is guaranteed to return a world whose low probability is greatest. □

Step 3(a) of the HOP algorithm is known as the UNIQUE-SAT problem—it can be easily implemented via a SAT solver as follows.

1. If $\bigwedge_{F \in X} F \wedge \bigwedge_{G \in \bar{X}} \neg G$ is satisfiable (using a SAT solver that finds a satisfying world w) then
   (a) If $\bigwedge_{F \in X} F \ \wedge \ \bigwedge_{G \in \bar{X}} \neg G \ \wedge \ (\bigvee_{a \in w} \neg a \ \ \vee \ \ \bigvee_{a' \in \bar{w}} a')$ is satisfiable (using a SAT solver) then return "two or more" (two or more satisfying worlds exist) else return "exactly one"
2. else return "none."

The following example shows how the HOP algorithm would work on the program from Example 8.

Example 9. *Consider the program from Example 8, and suppose* $X = \{(a \vee b \vee c \vee d \vee e), (a \wedge e)\}$. *In Step* (3a), *the algorithm will find that* $\{a, d, e\}$ *is a model of* $(a \vee b \vee c \vee d \vee e) \wedge (a \wedge e)$; *afterwards, it will find* $\{a, c, e\}$ *to be a model of* $(a \vee b \vee c \vee d \vee e) \wedge (a \wedge e) \wedge ((\neg a \vee \neg d \vee \neg e) \vee (b \vee c))$. *Thus,* X *has more than one model and the algorithm will not consider any of the worlds in the equivalence class induced by* X *as a possible solution, which avoids solving the linear program for those worlds.*

The worst-case complexity of HOP is, as its Naive counterpart, exponential. However, HOP can sometimes (but not always) be preferable to the Naive algorithm. The number of variables in $\mathsf{RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ is $2^{card(\Pi_s)}$, which is much smaller than the number of variables in $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ when the number of ground rules whose bodies are satisfied by state s is smaller than the number of ground atoms. The checks required to find all the equivalence classes $[X_i]$ take time proportional

to $2^{2*card(\Pi_s)}$. Lastly, HOP avoids solving the reduced linear program for all the non-singleton equivalence classes (for instance, in Example 9, the algorithm avoids solving the LP three times). This last saving, however, comes at the price of solving SAT twice for each equivalence class and the time required to find the $[X_i]$'s. We will now explore a way in which we can trade off computation time against how many of these savings we obtain, again without giving up obtaining an exact solution.

4.2.2 Enhancing HOP: The SemiHOP Algorithm

A variant of the HOP algorithm, which we call the SemiHOP algorithm, tries to avoid computing the full equivalence classes. As in the case of HOP, SemiHOP is also guaranteed to return a most probable world. The SemiHOP algorithm avoids finding pairs of sets that represent the same equivalence class, and therefore does not need to compute the checks for logical equivalence of every possible pair, a computation which can prove to be very expensive.

Proposition 6. *Suppose* Π *is an* ap-*program,* s *is a state, and* X *is a subset of* $\mathit{FixedWff}(\Pi, s)$*. Then there exists a world* w_i *such that* $\{w | w \models Formula(X, \Pi, s)\} \subseteq [w_i]$*.*

Proof. Immediate from Definition 14. □

We now define the concept of a sub-partition.

Definition 16. *A* sub-partition *of the set of worlds of* Π *w.r.t.* s *is a partition* $W_1, \ldots, W_k$ *where:*

1. $\bigcup_{i=1}^{k} W_i$ *is the entire set of worlds.*

2. *For each* W_i*, there is an equivalence class* $[w_i]$ *such that* $W_i \subseteq [w_i]$*.*

Algorithm 4: SemiHOP Algorithm.

1. Compute $T^{\omega}_{\Pi_s}$.
2. $bestval = 0$; $best = NIL$.
3. For each set $X \subseteq \mathit{FixedWff}(\Pi, s)$ do:
   (a) If there is exactly one interpretation that satisfies $Formula(X, \Pi, s)$ then:
      i. **Minimize** $p^{\star}_i$ **subject to** $\mathsf{S_RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ where W_i is a subpartition of the set of worlds of Π w.r.t. s. Let Val be the value returned.
      ii. If $Val > best$, then $\{best = w_i; bestval = Val\}$.
4. If $bestval = 0$ then return any world whatsoever, otherwise return $best$.

Figure 4.4: The SemiHOP algorithm.

The following result, which follows immediately from the preceding proposition, says that we can generate a subpartition by looking at all subsets of the set $\mathit{FixedWff}(\Pi, s)$.

Proposition 7. *Suppose* Π *is an* ap-*program, s is a state, and* $\{X_1, \ldots, X_k\}$ *is the power set of* $\mathit{FixedWff}(\Pi, s)$. *Then the partition* $W_1, \ldots, W_k$ *where* $W_i = \{w \mid w \models Formula(X_i, \Pi, s)\}$ *is a sub-partition of the set of worlds of* Π *w.r.t. s.*

Proof. Immediate from Proposition 6. □

The intuition behind the SemiHOP algorithm is best presented by going back to constraints 4.1 and 4.2 given in Example 7. Obviously, we would like to collapse all three variables p_5, p_7, p_8 into one variable y. However, if we were to just collapse p_7, p_8 into a single variable y', we would still reduce the size of the constraints (through the elimination of one variable), though the reduction would not be maximal (because we could have eliminated two variables). The SemiHOP algorithm allows us to

use subsets of equivalence classes instead of full equivalence classes. We define a *semi-reduced set of constraints* as follows.

Definition 17 ($\mathsf{S_RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$)**.** *Let $W_1, \ldots, W_k$ be a subpartition of the set of worlds for Π and s. For each W_i, $\mathsf{S_RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ uses a variable $p^{\star}_i$ to denote the summation of the probability of each of the worlds in W_i. For each* ap-*wff $F : [\ell, u]$ in $T^{\omega}_{\Pi_s}$, $\mathsf{S_RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ contains the constraint:*

$$\ell \leq \sum_{W_i \models F} p^{\star}_i \leq u.$$

Here, $W_i \models F$ implies that some world in W_i satisfies F. In addition, the set of constraints $\mathsf{S_RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ contains

$$\sum_{W_i} p^{\star}_i = 1$$

Example 10. *Returning to Example 7, $\mathsf{S_RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ could contain the following constraints: $0.7 \leq p_2 + p_3 + p_5 + p_6 + y' \leq 1$, $0.2 \leq p_5 + y' \leq 0.6$, and $p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + y' = 1$ where $y' = p_7 + p_8$.*

The pseudo-code for the SemiHOP algorithm is depicted in Figure 4.4. The following theorem ensures the correctness of this algorithm.

Theorem 3 (correctness of SemiHOP)**.** *Algorithm SemiHOP is correct, i.e. it is guaranteed to return a world whose low probability is greater than or equal to that of any other world.*

Proof. The proof is completely analogous to that of Theorem 2, with the only difference in this case being that some of the equivalence classes will be partitioned. □

The key advantage of $\mathsf{SemiHOP}$ over HOP is that we do not need to construct the set $[w_i]$ of worlds, i.e. we do not need to find the equivalence classes $[w_i]$. This is a potentially big saving because there are 2^n possible worlds (where n is the number of ground action atoms) and finding the equivalence classes can be expensive. However, this advantage comes with a drawback, since the size of the set $\mathsf{S_RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ can be bigger than the size of the set $\mathsf{RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$. It is hard to quantify how much larger $\mathsf{S_RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ is compared to $\mathsf{RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$; in general, the more logically equivalent rule heads we have, the more unnecessary variables will be included in $\mathsf{S_RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$.

4.3 The Binary Heuristic

In this section, we introduce a heuristic called the *Binary Heuristic* that can be utilized in conjunction with any of the three exact algorithms described thus far (Naive, HOP, and $\mathsf{SemiHOP}$) in the chapter. The basic idea behind the Binary Heuristic is to limit the number of variables in the linear programs associated with the Naive, HOP, and $\mathsf{SemiHOP}$ algorithms to a fixed number k that is chosen by the user.

Suppose we use $\mathcal{V}_{Naive}, \mathcal{V}_{\mathsf{HOP}}$, and $\mathcal{V}_{\mathsf{SemiHOP}}$ to denote the set of variables occurring in the linear programs $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$, $\mathsf{RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ and $\mathsf{S_RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$, respectively. Note that all these linear programs contain two kinds of constraints:

- Interval constraints which have the form $\ell \leq p_{i_1} + \cdots + p_{i_m} \leq u$ and
- A single equality constraint of the form $p_1 + \cdots + p_n = 1$.

Let $\mathcal{V}^k_{Naive}, \mathcal{V}^k_{\mathsf{HOP}}, \mathcal{V}^k_{\mathsf{SemiHOP}}$ be some subset of k variables from each of these sets, respectively. Let CONS be one of $\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$, $\mathsf{RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$, or $\mathsf{S_RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$. We now construct a linear program CONS' from CONS as follows.

- For all constraints of the form

$$\ell \leq p_{i_1} + \cdots + p_{i_m} \leq u$$

  remove all variables in the summation that do not occur in the selected set of k variables and re-set the lower bound to 0.

- For the one constraint of the form $p_1 + \cdots + p_n = 1$, remove all variables in the summation that do not occur in the selected set of k variables and replace the equality "=" by "≤".

Example 11. *Consider the program from Example 8, and suppose $m = 10$ and* CONS *refers to the constraints associated with the naive algorithm which has 32 worlds altogether. Then, we can select a sample of ten worlds such as*

$$\mathcal{W}_m = \{w_2, w_4, w_8, w_{10}, w_{12}, w_{16}, w_{18}, w_{22}, w_{23}, w_{25}\}$$

Now, $\mathsf{CONS}'\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ *contains the following constraints:*

$$0 \leq p_2 + p_4 + p_8 + p_{10} + p_{12} + p_{16} + p_{18} + p_{22} + p_{23} + p_{25} \leq 0.5$$

$$0 \leq p_{23} + p_{25} \leq 0.5$$

$$p_2 + p_4 + p_8 + p_{10} + p_{12} + p_{16} + p_{18} + p_{22} + p_{23} + p_{25} \leq 1$$

Theorem 4. *Let Π be an* ap-*program, $m > 0$ be an integer, and s be a state. Then every solution of* CONS *is also a solution of* CONS' *where* CONS *is one of*

$\mathsf{CONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$, $\mathsf{RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$, *or* $\mathsf{S_RedCONS}_U\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ *and* CONS' *is constructed according to the above construction.*

Proof. (i) Suppose σ is a solution to CONS. For any interval constraint

$$\ell \leq p_{i_1} + \cdots + p_{i_m} \leq u$$

deleting some terms from the summation preserves the upper bound and clearly the summation still is greater than or equal to 0. Hence, σ is a solution to the modified interval constraint in CONS'. For the equality constraint $p_1 + \ldots + p_n = 1$, removing some variables from the summation causes the resulting sum (under the solution σ) to be less than or equal to 1 and hence the corresponding constraint in CONS' is satisfied by σ. □

A major problem with the above result is that CONS' is always satisfiable because setting all variables to have value 0 is a solution. The binary algorithm tries to tighten the lower bound in the interval constraints involved so that we have a set of solutions that more closely mirror the original set. It does this by looking at each interval constraint in CONS' and trying to set the lower bound of that constraint first to $\ell/2$ where ℓ is the lower bound of the corresponding constraint in CONS. If the resulting set of constraints is satisfiable, it increases it to $3\ell/4$, otherwise it reduces it to $\ell/4$. This is repeated for different interval constraints until reasonable tightness is achieved. It should be noted that the order in which the constraints are processed is important - different orders can lead to different CONS' being generated. The detailed algorithm is shown in Figure 4.5. The algorithm is called with $\Pi' = T^{\omega}_{\Pi_s}$, and CONS equal to one of CONS_U, $\mathsf{RedCONS}$, or $\mathsf{S_RedCONS}$.

```
Algorithm 5:  Binary(Π′, m, ε, CONS){
 1.     CONS′ = new set of linear constraints;
 2.     W_m = select a set of m worlds in W;
 3.     for each rule r_i in Π {
 4.        let r_i = F : [ℓ, u] ← body;
 5.        add 0 ≤ Σ_{w_i ∈ W_m ∧ w_i ⊨ F} p_i ≤ u to CONS′;
 6.     }
 7.     for each constraint c_i ∈ CONS′; {
 8.        let L be the lower bound in c_i;
 9.        let L* be c_i's original lower bound in CONS;
10.        while not done(CONS′, c_i, ε) {
11.           L′ = (L* + L)/2
12.           let c′_i be constraint c_i with lower bound L′;
13.           if solvable((CONS′ − c_i) ∪ c′_i) {
14.              CONS′ = (CONS′ − c_i) ∪ c′_i;  L = L′
15.           }
16.           else {
17.              L* = L′;
18.              L′ = (L′ − L)/2;
19.              if solvable((CONS′ − c_i) ∪ c′_i) {
20.                 CONS′ = (CONS′ − c_i) ∪ c′_i;  L = L′
21.              }
22.              else { L* = L′; }
23.           }
24.        }
25.     }
26.     add Σ_{w_i ∈ W_m} p_i ≤ 1 to CONS′;
27.     return CONS′;
28.  }
```

Figure 4.5: The Binary Heuristic Algorithm.

The Binary algorithm takes a chance. Rather than use a very crude estimate of the lower bound in the constraints (such as 0, the starting point), it tries to "pull" the lower bounds as close to the original lower bounds as possible in the expectation that the revised linear program is closer in spirit to the original linear program. Here is an example of this process.

Example 12. *Consider the following very simple program:*

$a \wedge b : [0.8, 0.9] \leftarrow .$

$a \wedge c : [0.2, 0.3] \leftarrow .$

Let $\mathcal{W} = \{w_0 = \emptyset, w_1 = \{a\}, w_2 = \{b\}, w_3 = \{c\}, w_4 = \{a, b\}, w_5 = \{a, c\}, w_6 = \{b, c\}, w_7 = \{a, b, c\}\}$*, but suppose* $m = 4$ *and we select a sample of four worlds* $\mathcal{W}_m = \{w_0, w_2, w_6, w_7\}$*. Now, assuming* $s = \emptyset$*,* $\mathsf{CONS}'\left(\Pi, s, T^{\omega}_{\Pi_s}\right)$ *contains the following constraints:*

$0 \leq p_7 \leq 0.9$

$0 \leq p_7 \leq 0.3$

$p_0 + p_2 + p_6 + p_7 \leq 1$

which is clearly solvable, but yielding the all-zero solution. The binary heuristic will then modify the first constraint so that its lower bound is 0.4 *and, since this new program is unsolvable, will subsequently adjust it to* 0.2*. At this point, the program is now back to being solvable, and one more iteration leaves the lower bound at* $(0.4 + 0.2)/2 = 0.3$*, which results once again in a solvable program. At this point, we decide to stop, and the final value of the lower bound is thus* 0.3*. The algorithm then moves on to the following constraint, and adjusts its lower bound first to* 0.1 *and then to* 0.15*, and decides to stop. The final set of constraints is then:*

$0.3 \leq p_7 \leq 0.9$

$0.15 \leq p_7 \leq 0.3$

$p_0 + p_2 + p_6 + p_7 \leq 1$

4.4 Implementation and Experiments

We have implemented several of the algorithms described above—the Naive, HOP, SemiHOP, and the binary heuristic algorithms—using approximately 6,000 lines of

Java code. The binary heuristic algorithm was applied to each set of constraints: $\mathsf{CONS}_U(\Pi, s, T^{\omega}_{\Pi_s})$, $\mathsf{RedCONS}_U(\Pi, s, T^{\omega}_{\Pi_s})$, and $\mathsf{S_RedCONS}_U(\Pi, s, T^{\omega}_{\Pi_s})$; we refer to these approximations as the Naive_{binary}, HOP_{binary}, and $\mathsf{SemiHOP}_{binary}$ algorithms respectively. Our experiments were performed on a Linux computing cluster comprised of 64 8-core, 8-processor nodes with between 10GB and 20GB of RAM. The linear constraints were solved using the QSopt linear programming solver library [ACDM09], and the logical formula manipulation code from the COBA belief revision system and SAT4J satisfaction library were used in the implementation of the HOP and SemiHOP algorithms.

For each experiment, we randomly generated *ap*-programs; we held the number of rules constant at 10, where each rule consisted of an empty body (we assume they are the rules that are relevant in the state, and after computing the fixpoint). The rule heads had a number of clauses distributed uniformly between 1 and 5, and a number of variables per clause also distributed uniformly between 1 and 4 [2]. The probability intervals were also generated randomly from the uniform distribution, making sure that the lower bound was less than or equal to the upper bound. All random number selection were implemented using the random number generator provided by JAVA. The experiments then consisted of the following: (i) generate a new *ap*-program and send it to each of the three algorithms, (ii) vary the number of worlds from 32 to 16,384, performing at least 10 runs for each value and recording the average time taken by each algorithm, and (iii) measure the quality of SemiHOP and all algorithms that use the binary heuristic by calculating the average distance from

[2]The *maximum number of clauses* and *variables per clause* parameters were kept relatively low to simulate the kinds of rules that could be generated by either human experts or machine learning algorithms such as APEX (see Page 11). In the former case this make sense because human beings typically can't reason about formulas that are too complex; in the latter, it is unlikely that more complex (and useful) formulas could be learned automatically because the data supporting such rules would become too sparse.

the solution found by the exact algorithm. Due to the immense time complexity of the HOP algorithm, we do not directly compare its performance to the Naive algorithm or SemiHOP. In the discussion below we use the metric $ruledensity = \frac{\mathcal{L}_{act}}{card(T^{\omega}_{\Pi_s})}$ to represent the size of the *ap*-program; this allows for the comparison of the Naive and HOP and SemiHOP algorithms as the number of worlds increases.

Running time. Figure 4.6 shows the running times for each of the Naive, SemiHOP, Naive$_{binary}$, and SemiHOP$_{binary}$ algorithms for increasing number of worlds. As expected, the binary search approximation algorithm is superior to the exact algorithms in terms of computation time, when applied to both the Naive and SemiHOP constraint sets. With a sample size of 25%, Naive$_{binary}$ and SemiHOP$_{binary}$ take only about 132.6 seconds and 58.19 seconds for instances with 1,024 worlds, whereas the Naive algorithm requires almost 4 hours (13,636.23 seconds). This result demonstrates that the Naive algorithm is more or less useless and takes prohibitive amounts of time, even for small instances. Similarly, the checks for logical equivalence required to obtain each $[w_i]$ for HOP cause the algorithm to consistently require an exorbitant amount of time; for instances with only 128 worlds, HOP takes 58,064.74 seconds, which is much greater even than the Naive algorithm for 1,024 worlds. Even when using the binary heuristic to further reduce the number of variables, HOP$_{binary}$ still requires a prohibitively large amount of time.

At low rule densities, SemiHOP runs slower than the Naive algorithm; with 10 rules, SemiHOP uses 18.75 seconds and 122.44 seconds for 128 worlds, while the Naive algorithm only requires 1.79 seconds and 19.99 seconds respectively. However, SemiHOP vastly outperforms Naive for problems with higher densities—358.3 seconds versus 13,636.23 seconds for 1,024 worlds—which more accurately reflect real-world problems in which the number of possible worlds is far greater than the

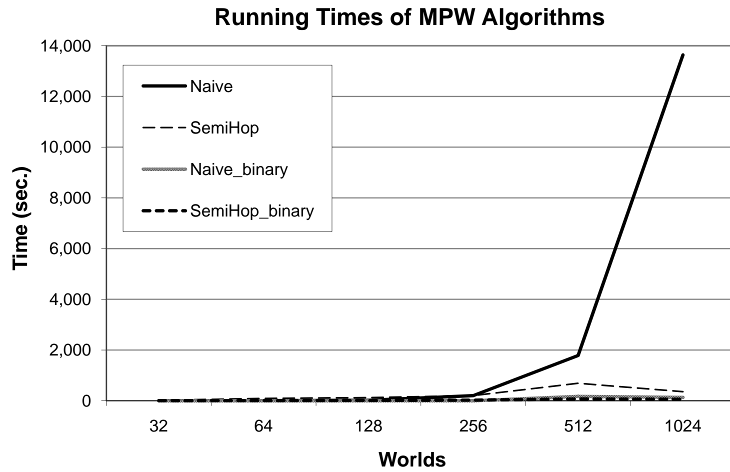


Figure 4.6: Running time of the algorithms for increasing number of worlds.

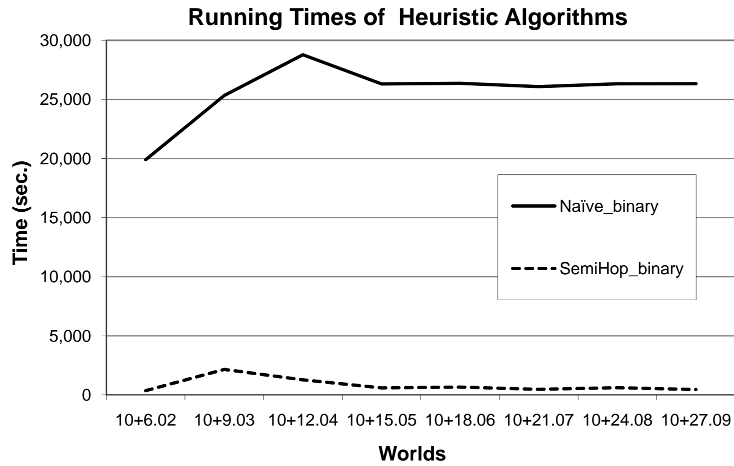


Figure 4.7: Running time of Naive_{binary} and $\mathsf{SemiHOP}_{binary}$ for large number of worlds.

number of *ap*-rules. Because the SemiHOP algorithm uses subpartitions rather than unique equivalence classes in the $\mathsf{RedCONS}_U(\Pi, s, T^{\omega}_{\Pi_s})$ constraints, the algorithm overhead is much lower than that of the HOP algorithm, and thus yields a more efficient running time.

The reduction in the size of the set of constraints afforded by the binary heuristic algorithm allows us to apply the Naive and SemiHOP algorithms to much larger *ap*-programs. In Figure 4.7, we examine the running times of the Naive$_{binary}$ and SemiHOP$_{binary}$ algorithms for large numbers of worlds (up to 2^90 or about 1.23794×10^{27} possible worlds) with a sample size for the binary heuristic of 2%; this is to ensure that the reduced linear program is indeed tractable. SemiHOP$_{binary}$ consistently takes less time than Naive$_{binary}$, though both algorithms still perform rather well. For 1.23794×10^{27} possible worlds, Naive$_{binary}$ takes an average 26,325.1 seconds while SemiHOP$_{binary}$ requires only 458.07 seconds. This difference occurs because $|\mathsf{S_RedCONS}_U(\Pi, s, T^{\omega}_{\Pi_s})| < |\mathsf{CONS}_U(\Pi, s, T^{\omega}_{\Pi_s})|$ that is the heuristic algorithm is further reducing an already smaller constraint set. In addition, because SemiHOP only solves the linear constraint problem when there is exactly one satisfying interpretation for a subpartition, it performs fewer computations overall. Because of this property, experiments running SemiHOP$_{binary}$ on problems with very large *ap*-programs (from 1,000 to 100,000 ground atoms) only take around 300 seconds using a 2% sample rate. However, this aspect of the SemiHOP algorithm can also lead to some anomalous behavior, where the running time will appear to decrease as the number of worlds increases. Figure 4.8 illustrates this anomaly, as the computation time appears to decrease with very large numbers of worlds. This occurs when we have taken a small sample of subpartitions in a problem with very high rule density, and there are no subpartitions with a single satisfying interpretation; as a result, no "most probable world" computations are performed, which obviously leads to a drastic reduction in the running time. Further experimentation is necessary to determine the optimal balance between an efficient running time and a sample large enough to produce meaningful results.

# Action Atoms	*# Worlds*	*Time (s)*
1,000	$\approx 10^{301}$	363.62
10,000	$\approx 10^{3,010}$	254.41
100,000	$\approx 10^{30,102}$	211.65

Figure 4.8: Running time of the $\mathsf{SemiHOP}_{binary}$ algorithm for very large numbers of possible worlds.

Quality of solution. Figure 4.9 compares the accuracy of the probability found for the most probable world by SemiHOP, Naive_{binary}, and $\mathsf{SemiHOP}_{binary}$ to the solution obtained by the Naive algorithm, averaged over at least 10 runs for each number of worlds. The results are given as a percentage of the solution returned by the Naive algorithm, and are only reported in cases where both algorithms found a solution. The SemiHOP and $\mathsf{SemiHOP}_{binary}$ algorithms demonstrate near perfect accuracy; this is significant because in the $\mathsf{SemiHOP}_{binary}$ algorithm, the binary heuristic was only sampling 25% of the possible subpartitions. However, in many of these cases, both the Naive and the SemiHOP algorithms found most probable worlds with a probability of zero. The most probable world found by the Naive_{binary} algorithm can be between 75% and 100% as likely as those given by the regular Naive algorithm; however, the Naive_{binary} algorithm also was often unable to find a solution.

4.5 Concluding Remarks

We have presented the theory and algorithms of *ap*-programs. *ap*-programs are a variant of probabilistic logic programs and their syntax and semantics is not very different from them. What we have done, however, is to present the following contributions:

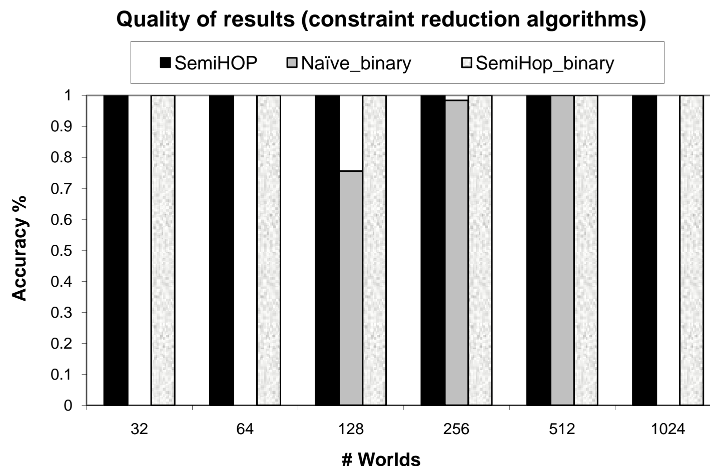


Figure 4.9: Quality of the solutions produced by SemiHOP, Naive$_{binary}$, and SemiHOP$_{binary}$ as compared to Naive.

1. Dealing with the problem of reducing the size of the linear programs that are generated by *ap*-programs - this problem has not been addressed in the literature for this problem, for any kind of PLPs.

2. Studying the problem of finding the most probable world, given an *ap*-program; this problem has not been addressed in the literature either, for any kind of PLPs.

3. Three algorithms to find the most probable world, along with the Binary heuristic that can be used in conjunction with any of them.

4. Our theory has produced tangible results of use to US military officers [Bha07, Sub07].

5. Our implementation is the only one we are aware of that can work for large numbers of ground atoms with reasonable accuracy and levels of efficiency

much superior to past efforts (we could only evaluate accuracy in cases with small numbers of ground atoms).

Furthermore, we have developed methods to reduce the size of the linear programs involved by taking into account the set of action atoms that a specific user might be interested in. By taking this into account, one can come up with a smaller linear program than that obtained by our other algorithms that often, but not always, leads to a fast solution. The SemiHOP algorithm also proposes a reduction in number of variables in the resulting linear program by defining equivalence classes of worlds. However, that algorithm bases the equivalence classes on co-occurrence of worlds in constraints, whereas the equivalence classes that arise here are purely a consequence of the input provided by the user through the set $\mathcal{Q}$ of actions that are interesting to him. It remains to be seen what the accuracy of the Monte Carlo approach is, and whether it can be improved using the heuristics presented here.

There are many problems that remain open. First, we need an accurate estimation of the computational complexity of the MPW problem. We have proven *NP*-hardness results, but were unable to establish membership in *NP*. A more accurate classification would be desirable. Moreover, it would be desirable to come up with efficient parallel algorithms. Third, it would be nice to get some concrete theoretical results about the accuracy of solutions produced by the binary heuristic. It is possible also that a judicious selection of variables in the binary heuristic may yield better results.

In the next chapter, we show that the approach described in this chapter can be significantly improved through the incorporation of yet another piece of knowledge. Under certain conditions, when we know what actions the user is interested in predicting (which is often the case and can be easily communicated by the user to a

system implementation), we can reduce the size of the linear program significantly, while guaranteeing that an exact solution will be found, not an approximate solution. Furthermore, we develop a Monte Carlo sampling approach that, when used in conjunction with the reduced linear program, is enormously helpful in scaling the performance of the system.

Chapter 5

Focused Most Probable World Computations in Action Probabilistic Logic Programs

The "Most Probable World" (MPW) problem in probabilistic logic programming, presented in Chapter 4, is that of finding a possible world with the highest probability. We have shown that this problem is computationally intractable and involves solving exponentially many linear programs, each of which is of exponential size. In this chapter, we study what happens when the user focuses his interest on a set of atoms in such a PLP. We show that we can significantly reduce the number of worlds to be considered by defining a "reduced" linear program whose solution is in one-to-one correspondence with the exact solution to the MPW problem. However, the problem is still intractable. We develop a Monte Carlo sampling approach that enables us to build a quick approximation of the reduced linear program that allows us to estimate (inexactly) the solution to the MPW problem. We show experimentally that our approach works well in practice, scaling well to problems where the exact solution is intractable to compute.

5.1 Introduction

As we have discussed in previous chapters, action probabilistic logic programs [KMN+07b, KMN+07a] (*ap*-programs for short) provide a logic programming paradigm through which we can develop stochastic models of the behavior of real world organizations *without making any assumptions about independence* of events and/or conditions. In view of the fact that no such assumptions are made, this paradigm is fundamentally different from others such as Bayesian Networks which are based on the assumption that certain variables are only dependent on a specific subset of variables. *ap*-programs and their variants have been used extensively over the last couple of years to develop models of the behaviors of the various stakeholders in the Afghan drug trade [SMSS07a] as well as terrorist groups such as Hezbollah [MMP+08a] and Hamas [MMP+08b]. These models, which are now available through a secure site [MSSS08] to registered users from over 12 defense organizations, exist for over 36 groups ranging from Morocco to Afghanistan. The behavioral models themselves are expressed through a set of stochastic rules that are informally of the form "If condition C holds, then group g will take a given action a with a probability in the range $[\ell, u]$." Note that the use of *ap*-programs to model organizational behavior is not limited to organizations with suspicious activities—in theory, they could just as well be used to learn conditions about when an investment bank will buy or sell a certain stock, or when an insurance company will pay or deny a given claim, or when OPEC will raise oil prices. However, these latter applications have not been built to the best of our knowledge.

A group's behavior is thus characterized by a set of such rules. Naturally, there is much interest in what a group will do in a given situation S (real or hypothetical) that may or may not have been encountered in the past. Past work by us has

studied this problem in considerable detail. A "world" informally refers to a set of actions that the group might take in situation S [1]. It is clear that the number of all possible worlds is exponential in the number of actions considered. The key question for decision makers is: what is the most (or k most) probable world(s) in a given situation S for group g?

The naive approach to solving this problem as described in Chapter 4 is to derive a linear program from the *ap*-program and situation S, and to try to compute the probability of world w for each and every world w. This approach has two fundamental problems. First, the size of the linear program is exponential in the number of actions considered. Second, we need to solve an exponential number of such linear programs (one for each world) in order to determine the most probable world. In Chapter 4, we made two major improvements to alleviate these problems. First, we proposed a method to reduce the size of the linear program (sometimes but not always) from being exponential in the number of atoms, to being exponential in the number of rules; and second, we developed a heuristic that explores only a fixed number of worlds.

In this chapter, we develop an approach based on the action probabilistic logic programs of [KMN$^+$07b, KMN$^+$07a] that investigates how reasoning with these programs can take advantage of the incorporation of yet another piece of knowledge. Under certain conditions, when we know what actions the user is interested in predicting (which is often the case and can be easily communicated by the user to a system implementation), we can define a set of "worlds of interest" that abstracts away the differences that only occur with respect to actions that are not of interest

[1] It is assumed that all actions in the world are carried out more or less in parallel and at once, given the temporal granularity adopted along with the model. Contrary to (related but essentially different) approaches such as stochastic planning, we are not concerned here with reasoning about the effects of actions.

to the user. One of the main reasons for doing this is that probabilities assigned to these worlds "focus" the probability mass that would otherwise be dispersed across the worlds that are abstracted away by this approach. Another positive aspect is that doing this yields a linear program that is significantly reduced in size, while guaranteeing that an exact solution will be found. Furthermore, we develop approximation algorithms based on Monte Carlo sampling that, when used in conjunction with the reduced linear program, are enormously helpful in scaling the performance of the system while not losing a great deal of precision. We describe these methods and provide experimental results showing that our system performs well even when a relatively large number of actions is considered.

5.2 Preliminaries

As we have seen in previous chapters, Action probabilistic logic programs (*ap*-programs) are a variant of the probabilistic logic programs introduced in [NS91, NS92]. In this chapter, we assume that we have been provided with a distinguished set $\mathcal{Q}$ of *action atoms of interest* chosen from the set of all possible ground action atoms. For instance, when a user is reasoning about a given group, he might specify what actions he is interested in. The powerset of $\mathcal{Q}$ will be denoted by $\mathcal{W}^{\mathcal{Q}}$, and represents the *worlds of interest*; their importance in the reduction of the size of the resulting constraints when performing most probable world computations will be discussed below.

Definition 18. *Let $F \in bf(\mathcal{L}_{act})$ be a ground basic action formula, and $\mathcal{Q}$ be a set of ground action atoms. The* reduction *of F with respect to $\mathcal{Q}$, denoted by $red(F, \mathcal{Q})$ is defined as a new formula F', which is obtained by removing from F, the atoms*

that do not appear in $\mathcal{Q}$. If F is a conjunction (resp. disjunction) and does not contain any atoms from $\mathcal{Q}$, then $red(F, \mathcal{Q}) = \top$ (resp. $\bot$).

We use $comp(F, \mathcal{Q})$ to denote the part of formula F that does not contain the atoms in $\mathcal{Q}$ (this intuitively corresponds to the complement of $red(F, \mathcal{Q})$).

Definition 19. *Let $F \in bf(\mathcal{L}_{act})$ be a ground basic action formula, and $\mathcal{Q}$ be a set of ground action atoms. The* complement *of F with respect to $\mathcal{Q}$, denoted by $comp(F, \mathcal{Q})$ is defined as a new formula F', which is obtained by removing from F the atoms that do not appear in $red(F, \mathcal{Q})$. If F is a conjunction (resp. disjunction) and , $F = red(F, \mathcal{Q})$ then $comp(F, \mathcal{Q}) = \top$ (resp. $\bot$).*

The example below illustrates these concepts.

Example 13. *Let $F = a \vee b \vee c \vee d \vee e$, and $\mathcal{Q} = \{a, b, c\}$. In this case, we have $red(F, \mathcal{Q}) = a \vee b \vee c$, while $comp(F, \mathcal{Q}) = d \vee e$. If $\mathcal{Q}' = \{f, g\}$, then $red(F, \mathcal{Q}') = \bot$ and $comp(F, \mathcal{Q}') = F$.*

As can be seen from the examples, reductions and complements of a formula are defined in such a way that their combination (depending on the basic formula's type) results in the original formula.

5.3 A new Linear Program Formulation for Worlds of Interest

Let Π be a basic *ap*-program, s be a state, $\mathcal{L}_{act}$ be the set of all possible action predicates, and $gr(\mathcal{L}_{act})$ be the set of all possible ground action atoms. Following the work of [Hai84, Nil86, Hal90, FHM90, NS92, Luk98], [KMN$^+$07a] shows how

we can associate a set of linear constraints with $\Pi, s, \mathcal{L}_{act}$. If $\mathcal{W}$ is the set of all possible worlds, and p_i is a variable denoting the (as yet unknown) probability of world $w_i \in \mathcal{W}$, then [KMN+07a] creates the set $\mathsf{CONS}(\Pi, s)$ of linear constraints defined by:

1. If $F_i : [\ell, u] \in \Pi_s$, then $\ell \leq \left(\sum_{w_j \models F_i} p_j\right) \leq u$ is in $\mathsf{CONS}(\Pi, s)$.
2. $\sum_{w_i} p_i = 1$ is in $\mathsf{CONS}(\Pi, s)$.

Now suppose the user selects a set $\mathcal{Q}$ of ground action atoms that he considers to be of interest for his work. We can take advantage of $\mathcal{Q}$ to significantly reduce the size of the linear constraints $\mathsf{CONS}(\Pi, s)$, and focus the probability masses of the possible worlds over these atoms of interest. We now show an example of how the latter can prove to be useful.

Example 14. *Suppose a given* ap-*program contains three possible atoms, a, b, and c, giving rise to eight possible worlds: $w_0 = \{\}$, $w_1 = \{a\}$, $w_2 = \{b\}$, $w_3 = \{c\}$, $w_4 = \{a, b\}$, $w_5 = \{a, c\}$, $w_6 = \{b, c\}$, and $w_7 = \{a, b, c\}$. Suppose further that the most probable world according to the techniques of [KMN+07a] is w_6 with probability* 0.25*, and that the rest of the probability distribution is: $p(w_0) = 0$, $p(w_1) = 0.2$, $p(w_2) = 0.05$, $p(w_3) = 0.08$, $p(w_4) = 0.12$, $p(w_5) = 0.16$, $p(w_7) = 0.14$. If the user were only interested in whether or not a is true (*i.e.*, $\mathcal{Q} = \{a\}$), then the most probable world w.r.t. $\mathcal{Q}$ is $\{a\}$, with probability $p(w_1) + p(w_4) + p(w_5) + p(w_7) =$* $0.2+0.12+0.16+0.14 = 0.62$*, versus world $\{\}$ (representing all worlds where a is not true) with probability $p(w_0) + p(w_2) + p(w_3) + p(w_6) = 0 + 0.05 + 0.08 + 0.25 = 0.38$. Note that a wasn't true in the most probable world originally computed.*

In connection to our discussion of applications in Chapter 1, the difference between the two kinds of probability distributions in Example 14 can also be visu-

alized by the soccer penalty kick example in Section 1.2.2, as shown in the following example.

Example 15. *Suppose a goalkeeper assumes that the player can kick in six different ways, with the following probabilities:*

$$p(high \wedge right) = 0.25$$
$$p(high \wedge center) = 0.025$$
$$p(high \wedge left) = 0.35$$
$$p(low \wedge right) = 0.25$$
$$p(low \wedge center) = 0.025$$
$$p(low \wedge left) = 0.10$$

Therefore, the most probable kick would be "high to the left", since it has probability 0.35. However, if the goalkeeper decides to abstract away the height of the kick, he would get the following, more "focused" distribution:

$$p(right) = 0.50$$
$$p(center) = 0.05$$
$$p(left) = 0.45$$

Similar to what happened in Example 14, note that the most probable kick is now towards the right instead of to the left.

In this chapter we will tackle the problem of building a set of constraints that will allow us to carry out the kind of computations discussed in Examples 14 and 15. Given $\mathcal{Q}$, we define a set of linear constraints $\mathsf{CONS}_0(\Pi, s, \mathcal{Q})$. The intuition behind this set of constraints, as compared to that used in $\mathsf{CONS}(\Pi, s)$ [KMN+07a] is that we now have a new set $\mathcal{W}^{\mathcal{Q}}$ that contains $2^{|\mathcal{Q}|}$ worlds, each of which is an *abstraction* of worlds in the old set $\mathcal{W}$ (each $w' \in \mathcal{W}^{\mathcal{Q}}$ represents $2^{|\mathcal{W}|-|\mathcal{W}^{\mathcal{Q}}|}$ worlds from the

original set). The constraints are then defined over the set of all possible worlds of interest $\mathcal{W}^{\mathcal{Q}}$ as follows:

1. If $F_i : [\ell, u] \in \Pi_s$, F_i is a *conjunction*, and $red(F_1, \mathcal{Q}) \neq \top$ then

$$\ell \leq \left(\sum_{w_j \models red(F,\mathcal{Q})} p_j \right) - q_0^i \leq u$$

   is in $\mathsf{CONS}_0(\Pi, s, \mathcal{Q})$.

2. otherwise (*i.e.*, F_i is a *disjunction* or $red(F_1, \mathcal{Q}) = \top$), we have that

$$\ell \leq \left(\sum_{w_j \models red(F,\mathcal{Q})} p_j \right) + q_0^i \leq u$$

   is in $\mathsf{CONS}_0(\Pi, s, \mathcal{Q})$.

3. For each variable q_0^i introduced in the constraints of type (1) and (2), the set $\mathsf{CONS}_0(\Pi, s, \mathcal{Q})$ contains the constraint $q_0^i \geq 0$.

4. $\sum_{w_i} p_i = 1$ is in $\mathsf{CONS}_0(\Pi, s, \mathcal{Q})$.

The probability of each world in $\mathcal{W}^{\mathcal{Q}}$ represents the summation of probabilities of worlds in the original set of constraints. The q_0^i variables introduced in the constraints of type (1) and (2) (referred to as *auxiliary variables* from now on) serve the purpose of compensating for the loss of granularity of this new set of worlds; for conjunctions they appear as negative values, since the reduced formula has more satisfying worlds than the original, and for disjunctions the opposite holds. In the case of conjunctions, the q_0^i values represent the summation

$$\sum_{w_i \models red(F_i,\mathcal{Q}) \wedge \neg comp(F_i,\mathcal{Q})} p_i \tag{5.1}$$

1.	$a \vee b \vee c$	$: [0.85, 0.95]$	$\leftarrow$	.
2.	$b \wedge c$	$: [0.4, 0.55]$	$\leftarrow$	.
3.	$a \vee b \vee d$	$: [0.6, 0.78]$	$\leftarrow$	.
4.	$a \wedge c \wedge d$	$: [0.15, 0.3]$	$\leftarrow$	.

$$
\begin{array}{rll}
0.85 \leq & p_1 + p_2 + p_3 + q_0^1 & \leq 0.95 \\
0.4 \leq & p_2 + p_3 - q_0^2 & \leq 0.55 \\
0.6 \leq & p_1 + p_2 + p_3 + q_0^3 & \leq 0.78 \\
0.15 \leq & p_1 + p_3 - q_0^4 & \leq 0.3 \\
& q_0^1, q_0^2, q_0^3, q_0^4 & \geq 0 \\
& p_0 + p_1 + p_2 + p_3 & = 1
\end{array}
$$

Figure 5.1: A simple *ap*-program and its corresponding set of linear constraints $\mathsf{CONS}(\Pi, \emptyset, \{a, b\})$.

from the original constraints, while in the case of disjunctions they represent the summation

$$\sum_{w_i \models comp(F_i, \mathcal{Q}) \wedge \neg red(F_i, \mathcal{Q})} p_i \tag{5.2}$$

In the following, we will use $corr(v)$ (for *correction* formula) to denote the formula associated with auxiliary variable v, which initially has the values just described. Now, this is only a first approximation towards obtaining a set of constraints that adequately reflects all the restrictions provided by the full set used in [KMN+07a]. In order to clarify what we mean by this, we first present an example of how to obtain $\mathsf{CONS}_0(\Pi, s, \mathcal{Q})$.

Example 16. *Suppose we have the basic* ap*-program* Π *of Figure 5.1, and let* $\mathcal{Q} = \{a, b\}$*. While the original set of worlds* $\mathcal{W}$ *included 16 worlds, the reduced set* $\mathcal{W}^{\mathcal{Q}}$ *contains only 4. As discussed, each of these worlds represents an abstraction of worlds from the original set; for instance, world* $\{a\}$ *represents worlds* $\{a\}, \{a, c\}, \{a, d\}$*, and* $\{a, c, d\}$*. This reflects the fact that the reasoning agent is only interested in the atoms in* $\mathcal{Q}$*, and therefore needs not differentiate among the worlds that only differ in atoms that are not in this set. As we will see later on, the*

goal is to have each variable in the linear program represent the sum *of the variables from the original formulation.*

The set of constraints $\mathsf{CONS}(\Pi, s, \mathcal{Q})$*, for which we use the enumeration of worlds* $w_0 = \{\}$, $w_1 = \{a\}$, $w_2 = \{b\}$, *and* $w_3 = \{a, b\}$, *is shown in Figure 5.1. Here, for instance,* $corr(q_0^2) = b \wedge \neg c$.

5.3.1 Refining the Set of Constraints

In Example 16, it can clearly be seen that the values of variables q_0^1 and q_0^3 are not independent of each other, since their corresponding formulas share models, *i.e.*, $(c \wedge \neg(a \vee b)) \wedge (d \wedge \neg(a \vee b)) \not\models \bot$. In this case, there is one world, $\{c, d\}$ that is a model of both correction formulas. This means that the initial set of constraints does not adequately represent the original restrictions on the probabilities that can be assigned to each world, and we should take this into account in order to refine the set of constraints. In order to address this issue, we introduce the RefineCONS algorithm (Figure 5.2), which replaces the variables that give rise to these situations with new variables, and sets their associated formulas accordingly. The *for* loop on line 15 states that each of the auxiliary variables must be bounded from above by the upper bounds of formulas in the heads of rules in Π that are satisfied by their associated formulas. This process can add redundant constraints (since more than one formula could be satisfied, and then only the lowest upper bound would make a difference) but we do not include the simple checks to avoid this in order to keep the presentation clear. The following example shows the set of constraints from the previous example after applying this algorithm.

Example 17. *When we apply the RefineCONS algorithm to the set of constraints obtained in Example 16, we obtain the following result.*

```
Algorithm 6:   RefineCONS(Π, CONS)
1.  CONS' := copy of CONS;
2.  i := 1;
3.  while CONS' has two auxiliary variables u_{k1}, v_{k2} such that
          corr(u_{k1}) ∧ corr(v_{k2}) ⊭ ⊥ {
4.      replace u_{k1} in CONS' with a new variable u_i;
5.      set corr(u_i) = corr(u_{k1}) ∧ ¬corr(v_{k2});
6.      i := i + 1;
7.      replace v_{k2} in CONS' with a new variable v_i;
8.      set corr(v_i) = corr(v_{k2}) ∧ ¬corr(u_{k1});
9.      i := i + 1;
10.     introduce a new variable r_i in CONS' (with coeff. +1)
          wherever u_{k1} and v_{k2} appeared;
11.     set corr(r_i) = (corr(u_{k1}) ∧ corr(v_{k2}));
12.     add constraint r_i ≥ 0 to CONS';
13.     i := i + 1;
14.  }
15.  for each auxiliary variable v and formula F in Π such that
          corr(v) ∧ F ⊭ ⊥ and
          F : [L, U] is the head of a rule in Π {
16.      add constraint 0 ≤ v ≤ U to CONS'
17.  }
18.  return CONS';
```

Figure 5.2: The RefineCONS algorithm.

(1). $0.85 \leq p_1 + p_2 + p_3 + q_1^1 + r_2 \leq 0.95$

(2). $0.4 \leq p_2 + p_3 - q_3^2 - r_4 \leq 0.55$

(3). $0.6 \leq p_1 + p_2 + p_3 + q_1^3 + r_2 \leq 0.78$

(4). $0.15 \leq p_1 + p_3 - q_3^4 - r_4 \leq 0.3$

(5). $q_1^1, q_3^2, q_1^3, q_3^4, r_2, r_4 \geq 0$

(6). $p_0 + p_1 + p_2 + p_3 = 1$

(7). $0 \leq q_1^1 \leq 0.95$

(8). $0 \leq r_2 \leq 0.95$

(9). $0 \leq r_2 \leq 0.78$

(10). $0 \leq q_3^2 \leq 0.95$

(11). $0 \leq q_3^2 \leq 0.78$

(12). $0 \leq r_4 \leq 0.95$

(13). $0 \leq r_4 \leq 0.78$

(14). $0 \leq q_3^4 \leq 0.95$

(15). $0 \leq q_3^4 \leq 0.55$

(16). $0 \leq q_3^4 \leq 0.78$

Here, for instance, $corr(q_3^2) = (b \wedge \neg c) \wedge \neg(a \wedge \neg(c \wedge d))$, constraint 11 states that, because $(a \vee b \vee d) \wedge corr(q_3^2) \not\models \bot$, the upper bound of rule 3 applies to q_3^2. As

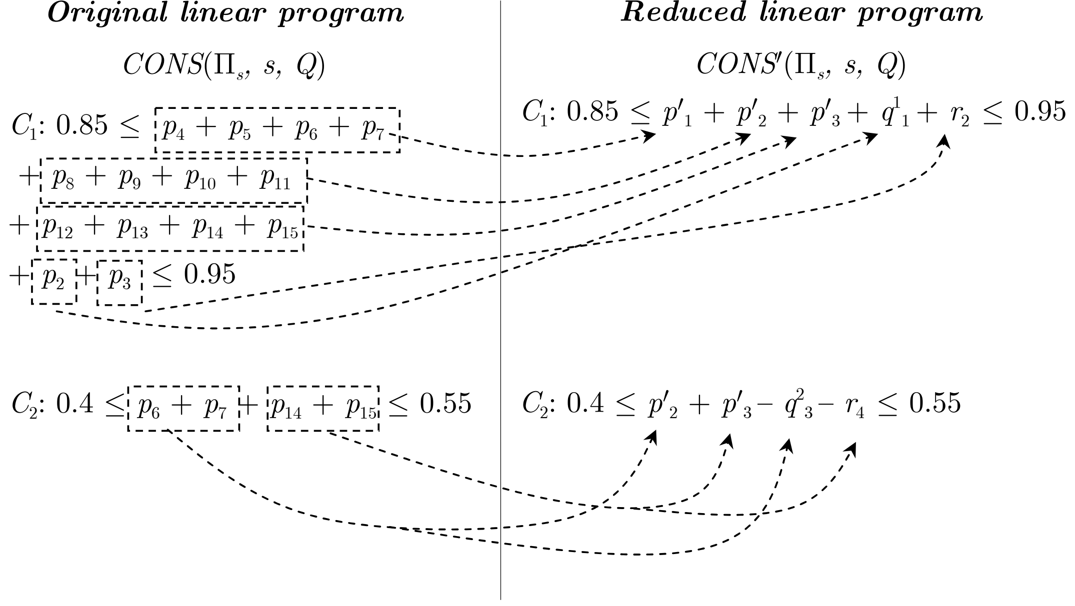


Figure 5.3: Part of the set of constraints after applying the RefineCONS algorithm to the set of constraints obtained in Example 16. Here, we show one constraint arising from a rule whose head contains a disjunction and one arising from a rule containing a conjunction (C_1 and C_2, respectively) in order to demonstrate the two possible scenarios that occur.

discussed above, some of the constraints indicating upper bounds (constraints 7 to 16) are redundant and their insertion can be easily avoided by the algorithm. This is the case with constraints 8, 10, 12, 14, and 16.

The following proposition states that the running time of the *RefineCONS* algorithm makes it intractable even for relatively small instances.

Proposition 8. *Given a state s, a set of actions of interest $\mathcal{Q}$, an* ap-*program* Π *containing m rules where the set of action atoms $\mathcal{A}$ has n elements, the worst case computation time for RefineCONS*$(\Pi, \mathsf{CONS}(\Pi, s, \mathcal{Q}))$ *is in* $O\left(2^{n+m}\right)$.

Proof. We start by analyzing the number of times that the `while` loop in line 3 is executed. In the worst case, the sets of models of the auxiliary variables are such that their *pairwise intersections* are all non-empty. This leads to a number of auxiliary variables that is exponential in the number of constraints. In Figure 5.4, we show

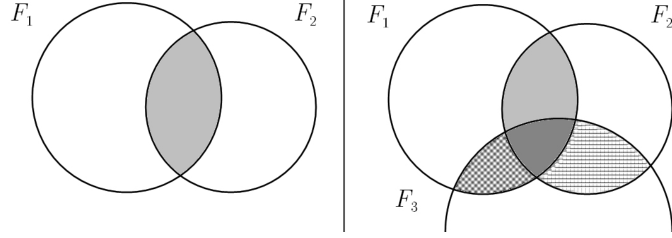


Figure 5.4: A Venn diagram representation of the models of formulas. On the left, two formulas share at least one model; the set of shared models is shown in gray. On the right, a new formula is introduced, which shares models with all three sets from the left (*i.e.*, only F_1, only F_2, and both) generating a total of seven possible subsets.

how this situation can arise; on the left, we have two formulas whose sets of models intersect (shown in gray), while on the right we show the worst possible scenario when considering one more formula, *i.e.*, that the new formula's models intersect with all three parts of the original diagram. This clearly leads to $2^m - 1$ auxiliary variables in the worst case, since m is the number of constraints in $\mathsf{CONS}_0(\Pi, s, \mathcal{Q})$.

The second source of complexity is in evaluating whether the formulas associated with a given pair of auxiliary variables are consistent or not (*i.e.*, whether their sets of models intersect or not). Note that correction formulas contain atoms from the *original* set $\mathcal{A}$, and therefore performing the satisfiability checks required to verify existence of common models will require time in $O(2^n)$. Even though all rules have basic formulas in their heads, note that correction formulas are not basic in general. □

The impact on the running time of the RefineCONS of the size of the *ap*-program (both in number of rules and action atoms) will be verified experimentally in Section 5.4. As hinted in the proof above, we can perform a *single* step of refinement in polynomial time.

Proposition 9. *Given a state s, a set of actions of interest $\mathcal{Q}$, an* ap*-program* Π *containing m rules, and if the set of action atoms $\mathcal{A}$ has n elements, the worst case computation time for refining the correction formulas in* $\mathsf{CONS}_0(\Pi, s, \mathcal{Q})$ *(*i.e.*, performing the first step of refinement) as computed by RefineCONS* $(\Pi, \mathsf{CONS}(\Pi, s, \mathcal{Q}))$ *is in $O(m^2 n)$.*

Proof. Let $corr(q_i), corr(q_j)$ be two correction formulas corresponding to auxiliary variables in $\mathsf{CONS}_0(\Pi, s, \mathcal{Q})$; we must check for each possible pair of such correction formulas whether or not $corr(q_i) \wedge corr(q_j) \models \bot$. There are three possible cases for the type of formulas of F_i and F_j that gave rise to these formulas. In the following, we will denote with $atoms(F)$ the set of atoms occurring in basic formula F.

1. *F_i and F_j are both conjunctions.* In this case, we have $corr(q_i) \equiv red(F_i) \wedge \neg comp(F_i)$, and $corr(q_j) \equiv red(F_j) \wedge \neg comp(F_j)$. The conjunction of correction formulas is *always* satisfiable, since atoms made true by $red(F_i)$ cannot conflict with those made true by $red(F_j)$. Likewise, atoms made false by $\neg comp(F_i)$ cannot conflict with those made true by $\neg comp(F_j)$. Furthermore, a model satisfying the conjunction can be found in time in $O(n)$.

2. *F_i is a conjunction and F_j is a disjunction* (the opposite case is analogous). We now have that $corr(q_i) \equiv red(F_i) \wedge \neg comp(F_i)$ and $corr(q_j) \equiv comp(F_j) \wedge \neg red(F_j)$. In order for the conjunction to be satisfiable, it must be the case that $atoms(red(q_i)) \cap atoms(red(q_j)) = \emptyset$. If this is the case, a model can be found in time in $O(n)$.

3. *F_i and F_j are both disjunctions.* In this case, we have $corr(q_i) \equiv comp(F_i) \wedge \neg red(F_i)$, and $corr(q_j) \equiv comp(F_j) \wedge \neg red(F_j)$. Similar to the fist case, the conjunction of correction formulas is *always* satisfiable, since atoms made true

by $comp(F_i)$ cannot conflict with those made true by $comp(F_j)$. Also as before, atoms made false by $\neg red(F_i)$ cannot conflict with those made true by $\neg red(F_j)$. Finally, a model satisfying the conjunction can be found in time in $O(n)$.

Checking all pairs of correction formulas in $\mathsf{CONS}_0(\Pi, s, \mathcal{Q})$ involves performing $m^2/2$ such steps, yielding a running time in $O(m^2 n)$. □

We now provide an example of how Proposition 9 can be applied.

Example 18. *Consider once again the* ap*-program from Figure 5.1, where we have the following correction formulas:* $corr(q_0^1) \equiv c \wedge \neg(a \vee b)$*,* $corr(q_0^2) \equiv b \wedge \neg c$*,* $corr(q_0^3) \equiv d \wedge \neg(a \vee b)$*, and* $corr(q_0^4) \equiv a \wedge \neg(c \wedge d)$*. According to Proposition 9,* q_0^1 *and* q_0^3 *(both corresponding to disjunctions) definitely share at least one model. This model can be obtained by finding a satisfying interpretation for* $comp(a \vee b \vee c, \mathcal{Q}) \wedge comp(a \vee b \vee d, \mathcal{Q})$ *(in this case,* $c \wedge d$*), which is guaranteed to be possible in* $O(n)$ *time. On the other hand, using Case 2, we quickly can determine that* q_0^1 *and* q_0^4 *are such that* $corr(q_0^1) \wedge corr(q_0^4) \models \perp$*, since it holds that* $atoms(red(q_0^1)) \cap atoms(red(q_0^4)) = \{c\} \neq \emptyset$*. This saves us from running a SAT check on the original conjunction:* $c \wedge \neg(a \vee b) \wedge a \wedge \neg(c \wedge d)$*.*

The above discussion and Propositions 8 and 9 leads us to the following corollary:

Corollary 1. *The* best case *computation time for the RefineCONS algorithm occurs when all pairs of auxiliary variables have mutually inconsistent correction formulas after one refinement step, which yields a running time in* $O(m^2 n)$*.*

The following result links the results obtained from solving the output of RefineCONS with those obtained from solving the original set of constraints used

in [KMN+07a] for the corresponding worlds. We use $Vars(C)$ to denote the set of variables occurring in a set C of constraints, and $val(V, S)$ to denote the value assigned to variable V by a solution S.

Theorem 5. *Let* Π *be an* ap-*program,* s *be a state,* $\mathcal{Q}$ *be a subset of all possible action atoms,* $C = \mathsf{CONS}(\Pi, s)$ *be the original set of constraints (obtained by considering the entire set of action atoms), and* C^* *be the set of constraints returned by* $RefineCONS(\Pi, C)$*. Then, there exists a mapping* $\mu : Vars(C^*) \to 2^{Vars(C)}$ *such that:*

1. *If* $V_1, V_2 \in Vars(C^*)$*,* $V_1 \neq V_2$*, then* $\mu(V_i) \cap \mu(V_2) = \emptyset$
2. *For every solution* S *of* C *there exists a solution* S^* *of* C^* *such that for every* $V_i^* \in S^*$*,* $val(V_i^*, S^*) = \sum_{V_j \in \mu(V_i^*)} val(V_j, S)$*.*

Proof. We first establish how mapping μ is obtained. For any non-auxiliary variable p_j^* corresponding to a world of interest $w_j^* \models red(F, Q)$, we have that

$$\mu(p_j^*) = \left\{p_i \mid w_i = w_j^* \cup w_r \text{ where } w_r \in \mathcal{P}\left(\mathcal{W} - \mathcal{W}^{\mathcal{Q}}\right)\right\}$$

and, for any auxiliary variable q_j^* we have

$$\mu(q_j^*) = \left\{p_i \mid w_i \in \mathcal{W} \text{ and } w_i \models corr(q_j^*)\right\}.$$

Note that this mapping is the by-product of how $\mathsf{CONS}(\Pi, s, \mathcal{Q})$ is defined.

We must first prove that condition (1) holds. For non-auxiliary variables, it is clear from the definition of the mapping that no two images can intersect, since their elements are constructed from disjoint worlds in $\mathcal{W}^{\mathcal{Q}}$. For auxiliary variables, the algorithm RefineCONS guarantees that the images of two different variables do

not intersect, since the `while` loop in line 2 exits only when, for all pairs of auxiliary variables, their corresponding correction formulas do not share models.

In order to prove condition 2, consider a solution S of C and an assignment of values S^* to variables of C^* satisfying the conditions in the theorem. We must now prove that S^* so defined is in fact a solution for C^*; in order to do this, we will consider constraints arising from disjunctions and conjunctions in turn:

- Let c_i^* be a constraint arising from a rule whose head is a *disjunction*. From the definition of $\mathsf{CONS}(\Pi, s, \mathcal{Q})$ and the operations performed by the RefineCONS algorithm, all variables in the constraint appear with a coefficient of 1 (positive). Therefore, from condition (1) it follows that c_i^* is simply a rewriting of its corresponding constraint in C, where p_j^* replaces the block of variables $\mu(q_j^*)$.

- Let c_i^* be a constraint arising from a rule whose head is a *conjunction*. In this case we will have variables with a -1 coefficient. As before, it is possible to see c_i^* as a rewriting of its corresponding constraint in C. To see why, consider the set of non-auxiliary variables in c_i^*; the value assigned by S^* to the sum of all such variables can be separated into two positive values, $s_{i,1}^*$ and $s_{i,2}^*$, such that $s_{i,1}^* = \sum_{w_j \in \mathcal{W}, w_j \models red(F,\mathcal{Q}) \wedge \neg comp(F,\mathcal{Q})} p_j$ and $s_{i,2}^* = \sum_{w_j \in \mathcal{W}, w_j \models comp(F,\mathcal{Q}) \wedge \neg red(F,\mathcal{Q})} p_j$ (note that by definition we have that $red(F, \mathcal{Q}) \wedge comp(F, \mathcal{Q}) \models \perp$). Therefore, $s_{i,2}^* = val(q_0^i)$ (as stated in Equation 5.1) and, since RefineCONS guarantees that q_0^i will be replaced by a set of variables whose correction formulas do not share models and with a union of models equal to that of q_0^i's correction formula, this means that the constraint is indeed a rewriting of its corresponding one in C, since $s_{i,2}^*$ is effectively added and then subtracted.

□

The following result is an immediate consequence of the above theorem and establishes the correctness of the RefineCONS algorithm.

Corollary 2 (Correctness of RefineCONS)**.** *Algorithm RefineCONS is correct,* i.e., *minimizing/maximizing with respect to a non-auxiliary variable q_i in the set of constraints $C^* = RefineCONS(\Pi, s, \mathcal{Q})$ yields the same value as that obtained by minimizing/maximizing the sum of variables associated with the set of worlds abstracted by world $w_i \in \mathcal{W}^{\mathcal{Q}}$.*

5.3.2 Refinement Algorithms based on Random Sampling

In the previous section we saw that the applicability of the RefineCONS algorithm is limited due to the intensive computations that it must carry out in order to ensure that the constraints are fully refined. In this section, we will begin by presenting a Monte Carlo algorithm that alleviates these computations, at the expense of not being able to guarantee full refinement.

The basic Monte Carlo refinement algorithm is described in Figure 5.5. The algorithm follows the same basic approach as RefineCONS, except that auxiliary variables are now refined based on randomly selected models instead of exhaustive verification of satisfiability of conjunctions of pairs of correction formulas. The `while` loop in line 4 uses a subroutine `terminationCond` as a condition; we assume that this subroutine is designed by the user to decide when enough refining attempts have been made (for instance, number of models tried, number of refinements made, time elapsed, etc). The following is an example of how this algorithm works.

```
Algorithm 7:  MonteCarloRefineCONS(Π, CONS)
1.   CONS′ := copy of CONS;
2.   i := 1;  m := 0;
3.   VariableAssignment test;
4.   while terminationCond(Π, CONS, i, m) = false {
5.       test = randomly generate a world wrt full set of atoms;
6.       m := m + 1
7.       for each pair of auxiliary variables v_{k1} ≠ v_{k2} such that
                test ⊨ corr(v_{k1}) ∧ corr(v_{k2})
8.          replace v_{k1} in CONS′ with a new variable u_i;
9.          set corr(v_i) := corr(v_{k1}) ∧ ¬corr(v_{k2});
10.         i := i + 1;
11.         replace v_{k2} in CONS′ with a new variable v_i;
12.         set corr(v_i) := corr(v_{k2}) ∧ ¬corr(v_{k1});
13.         i := i + 1;
14.         introduce a new variable r_i in CONS′ (with coeff. +1)
               wherever v_{k1} and v_{k2} appeared;
15.         add constraint r_i ≥ 0 to CONS′;
16.         i := i + 1;
17.      }
18.  }
19.  for each auxiliary variable v and formula F in Π such that
            corr(v) ∧ F ⊭ ⊥ and
            F : [L, U] is the head of a rule in Π {
20.      add constraint 0 ≤ v ≤ U to CONS′
21.  }
22.  return CONS′;
```

Figure 5.5: The basic MonteCarloRefineCONS algorithm.

1.	$a \vee b \vee c$	: $[0.85, 0.95]$	$\leftarrow$	.
2.	$b \wedge c$	: $[0.4, 0.55]$	$\leftarrow$	.
3.	$a \vee b \vee d$	: $[0.6, 0.78]$	$\leftarrow$	.
4.	$a \wedge c \wedge d$	: $[0.15, 0.3]$	$\leftarrow$	.
5.	$e \wedge f \wedge g$	: $[0.35, 0.42]$	$\leftarrow$	.
6.	$h \wedge i \wedge j \wedge k$	: $[0.90, 0.96]$	$\leftarrow$	.

Figure 5.6: The *ap*-program from Figure 5.1 with two additional rules.

Example 19. *Let* Π *be the* ap*-program from Figure 5.1, with the addition of two rules, as shown in Figure 5.6. Note that the new rules do not share atoms with*

the original ones; in particular, this means that the correction formulas associated with their auxiliary variables are the same as the heads of these rules. As shown in Example 17, we need to identify two cases in which pairs of correction formulas have overlapping models, these are (q_0^1, q_0^3) *and* (q_0^2, q_0^4)*. The first pair involves finding a model for* $c \wedge \neg(a \vee b) \wedge d$ *(which has a single model,* $\{c, d\}$*) and* $b \wedge \neg c \wedge a \wedge \neg(c \wedge d)$ *(which has two models,* $\{a, b\}$ *and* $\{a, b, d\}$*).*

Given this setup, suppose MonteCarloRefineCONS selects 100 randomly generated models and tests all pairs of auxiliary variables for the presence of an overlap. Suppose the algorithm generates the following 5 models first:

$$m_1 = (1, 0, 0, 1, 0, 0, 0, 1, 1, 0)$$
$$m_2 = (1, 0, 0, 0, 1, 0, 1, 1, 1, 0)$$
$$m_3 = (1, 0, 0, 1, 0, 1, 0, 0, 1, 1)$$
$$m_4 = (1, 1, 0, 1, 1, 1, 0, 0, 0, 0)$$
$$m_5 = (1, 0, 1, 1, 0, 0, 0, 1, 1, 0)$$

For models m_1 *through* m_5*, the algorithm cycles through all pairs of auxiliary variables, checking if they satisfy the corresponding conjunction of corrections formulas. For the first three, no overlaps are found. Next,* m_4 *is found to satisfy the conjunction of* q_0^2 *and* q_0^4*'s correction formulas, and the algorithm therefore introduces a new variable,* r_4*, with correction formula* $b \wedge \neg c \wedge a \wedge \neg(c \wedge d)$*;* q_0^2 *is replaced by a new variable* q_1^2 *such that* $corr(q_1^2) = b \wedge \neg c \wedge \neg(b \wedge \neg c \wedge a \wedge \neg(c \wedge d))$*. The other variable,* q_0^4 *is updated analogously. This process is carried out for the 95 remaining models.*

In the following, we assume that `terminationCond` *simply checks whether the number of samples has reached a certain number* p*.*

Proposition 10. *Let s be a state, $\mathcal{Q}$ be a set of actions of interest,* Π *be an* ap-*program containing m rules where the set of action atoms $\mathcal{A}$ has n elements. The worst case computation time of MonteCarloRefineCONS*$(\Pi, \mathsf{CONS}(\Pi, s, \mathcal{Q}))$ *is in* $O\left(2^{2m} np\right)$*, where p is the number of possible worlds sampled.*

Proof. Each interpretation generated by the algorithm in line 5 is used to evaluate the conjunction of two correction formulas in the `for` loop in line 7; each such evaluation can be performed in time in $O(n)$. Since the algorithm will generate p sample interpretations and, as we have shown in Proposition 8, the number of auxiliary variables that can be generated is bounded by above by 2^m, the running time is in $O\left(2^{2m} np\right)$ since all possible pairs of auxiliary variables are examined in the `for` loop in line 7. □

As Proposition 10 shows, the Monte Carlo algorithm reduces the computation time of the exact algorithm by a factor 2^n by only looking at a subset of all the possible interpretations that could lie at the intersection of the sets of models of the correction formulas. However, its running time is still exponential in the number of rules in the program, since all possible overlaps are checked for each interpretation that is generated. However, as in the case of the exact algorithm, we stress that this is the worst case running time that occurs when all possible overlaps in the sets of models of formulas are non-empty.

Proposition 11. *Let* Π *be an* ap-*program containing m rules where the set of action atoms $\mathcal{A}$ has n elements. The probability that the MonteCarloRefineCONS algorithm finds an interpretation that satisfies two correction formulas at once is* bounded from below *by:*

$$1 - \left(\frac{2^n - 1}{2^n}\right)^p$$

where p is the number of possible worlds sampled. The probability that MonteCarloRefineCONS finds all overlaps is then bounded from below by:

$$\left(1 - \left(\frac{2^n - 1}{2^n}\right)^p\right)^{2^m - 1}$$

Proof. For the first part of the proof, note that the worst case is that the two correction formulas in question only share a *single* interpretation. Therefore, the probability of failure when uniformly sampling an interpretation over the set of n atoms is $\frac{2^n-1}{2^n}$ in each case, and $\left(\frac{2^n-1}{2^n}\right)^p$ of failing p times. Thus, one minus this expression yields the probability of success. The second part simply states the probability of finding all possible (worst case) $2^m - 1$ overlaps. □

The probability of success of MonteCarloRefineCONS given by Proposition 11 are very loose lower bounds, since it assumes that the conjunctions in each case have a *single satisfying interpretation.* It should be noted, however, that obtaining a tighter bound is difficult, since knowing the number of satisfying interpretations in order to obtain one would, in particular, involve knowing that the conjunction is satisfiable! However, the probability of success using this strategy is likely to be lower than it could possibly be, since the sampling is being performed over the entire set of atoms, which is not always necessary. Furthermore, the algorithm is not taking advantage of easy checks that it could perform. The basic Monte Carlo algorithm can therefore possibly be enhanced by considering the following two heuristics:

Small number of variables check. Before any sampling is done, we can perform the same checks done by the exact algorithm, but only carrying them out if the total number of atoms occurring between the possible overlapping sets of models is small enough for an exhaustive verification, *i.e.*, a SAT check on the conjunction

of corresponding correction formulas. This enhances the accuracy of the final result by identifying easy refinements that could otherwise be missed by the random generation of models.

Example 20. *Suppose we have the same program used in Example 19, and that we are only willing to perform exact SAT checks for formulas with up to three atoms. In this case, we can quickly determine that q_0^1 and q_0^2's correction formulas do not share models since $c \wedge \neg(a \vee b) \wedge b \wedge \neg c$ is not satisfiable. All other pairs of corrections formulas involve a number atoms that is above our threshold, and will therefore not be checked in this way.*

Targeted sampling. A data structure can be maintained for storing the pairs of auxiliary variables that we have proved do not share models. Such a data structure can be an array of auxiliary variables, where each variable v_i has a set of associated auxiliary variables v_j such that $corr(v_i) \wedge corr(v_j) \models \perp$. These sets are all initially empty, and then can be updated either by the checks discussed above, or when a refinement is otherwise identified (such as when a randomly generated model satisfies two or more variables' correction formulas). When such an event occurs, the two variables involved are replaced by the new three, where the conjunction variable is associated with the union of the two other variables' sets.

When such a data structure is maintained, *targeted sampling* can be performed, *i.e.*, generation of models specifically tailored towards certain pairs. This means that a pair of variables can be selected out of the possible ones remaining, and models can be generated involving only the atoms appearing in these variables' correction formulas, thus enhancing the chances of finding a model for their conjunction if one exists. Another advantage of the use of such a data structure is in being able to

determine how many pairs remain to be tested for refinement, and in particular if none remain.

Example 21. *Coming back to Example 19, consider the set of the four interpretations that the algorithm generated and failed to find an overlap. Note that if* m_1 *does not satisfy any of the pairs of correction formulas,* m_3 *won't either, since for this particular example, the two interpretations are "equivalent" since they only vary in assignments to irrelevant atoms* e, f, g, h, i, j, k*. This shows how MonteCarloRefineCONS can waste effort when sampling. If we only consider the atoms that are involved in, say,* $corr(q_0^1)$ *and* $corr(q_0^3$*, then the sampling will only focus on relevant atoms* a, b, c, d*, and generate interpretations that are guaranteed to be different.*

The algorithm in Figure 5.7 shows how these two heuristics can be combined and added to the basic idea of the Monte Carlo algorithm. The first `for` loop applies the *small number of variables* check by doing a full SAT check on all pairs of variables that only share up to *satThresh* variables; this parameter indicates the maximum number of variables for which the user is willing to perform a SAT check. It should be noted that, every time an overlap is found, a set of disjoint pairs is updated to record that certain pairs of variables are known to have non-overlapping correction formulas. The `while` loop in the second part of the algorithm focuses on the pairs of variables that are not in this set, and performs up to *testThresh* tries to find a satisfying variable assignment using the Monte Carlo method described above. As was argued before, the samples are taken over the reduced set composed only of the variables shared between the pair being tested, increasing the chances of finding a satisfying interpretation if one exists. The algorithm selects for this purpose a random number of pairs over which it will evenly divide the total number of samples it will take. The subroutine *terminationCond* that is used in line 18 simply checks

whether or not a certain termination condition is true (*e.g.*, time elapsed since a pair not in *disjPairSet* was found, etc). The line that reads "`update CONS' as above`" refers to the manipulation of the correction formulas and variables done in the first `for` loop when a pair of variables is found to be overlapping. Finally, the last `for` loop performs the same operations as the last steps of the exact algorithm in order to add the corresponding upper bounds on the values that the auxiliary variables can take. We will now show an example of how this algorithm works.

Example 22. *Consider once again the* ap*-program from Example 19, and suppose* $satThresh = 3$ *and* $testThresh = 100$*. The first part of the algorithm (shown shaded in Figure 5.7) will check to see if there are any pairs of correction formulas whose conjunction involves at most 3 atoms. As shown in Example 21, only the pair* (q_0^1, q_0^2) *meet this requirement and, since the corresponding conjunction is not satisfiable, the algorithm updates disjPairSet reflecting that this pair of correction formulas does not share models.*

*For the second part of the algorithm, it is determined that 15 pairs of auxiliary variables are initially possible; however, the small number variables check brings this to 14. Suppose the algorithm randomly determines that it will dedicate 20 samples per randomly selected pair (*i.e., *perPair* $= 20$*). Suppose it begins by randomly choosing the pair* (q_0^1, q_0^4)*; since* $c \wedge \neg(a \vee b) \wedge a \wedge \neg(c \wedge d)$ *is not satisfiable, all 20 samples produced will fail to find an overlap in this case. Suppose the pair* (q_0^1, q_0^3) *is then randomly chosen. For this pair, the algorithm has 20 chances of finding the model* $\{c, d\}$ *which proves that these two variables' correction formulas share models. Note that these chances are much better than the ones MonteCarloRefineCONS has of finding this particular model (see Example 19). If* $\{c, d\}$ *is chosen before using up all 20 possibilities, the algorithm will update the structures as usual, and save the*

Algorithm 8: HeuristicRefineCONS($\Pi, \mathcal{Q}, satThresh, testThresh$)

```
1.  CONS' := copy of CONS; i := 1;
2.  Set of variable pairs disjPairSet = new Set;
3.  for each pair of auxiliary variables v_{k_1} ≠ v_{k_2} {
4.      if |allVars(corr(v_{k_1}), corr(v_{k_2}))| ≤ satThresh
           and corr(v_{k_1}) ∧ corr(v_{k_2}) ⊭ ⊥ {
5.         replace v_{k_1} in CONS' with a new variable v_i;
6.         set corr(v_i) := corr(v_{k_1}) ∧ ¬corr(v_{k_2}); i := i + 1;
7.         replace v_{k_2} in CONS' with a new variable v_i;
8.         set corr(v_i) := corr(v_{k_2}) ∧ ¬corr(v_{k_1}); i := i + 1;
9.         introduce a new variable r_i in CONS' (coeff. +1)
              wherever v_{k_1} and v_{k_2} appeared;
10.        add constraint r_i ≥ 0 to CONS'; i := i + 1;
11.        update disjPairSet;
12.     }
13. }
14. VariableAssignment test;
15. Integer numPairs := compute # possible aux. var. pairs;
16. Integer perPair := testThresh/U(1, numPairs);
17. Integer totalSamples = 0;
18. while totalSamples < testThresh and terminationCond() {
19.    randomly select pair of aux. vars v_{k_1} ≠ v_{k_2} such that
          (v_{k_1}, v_{k_2}) ∉ disjPairSet {
20.    m := 0;
21.    while m < perPair and totalSamples < testThresh and
          (v_{k_1}, v_{k_2}) ∉ disjPairSet {
22.       test = randomly generate a world w.r.t.
23.          allVars (corr(v_{k_1}), corr(v_{k_2}));
24.       m := m + 1; totalSamples := totalSamples + 1;
25.       if test ⊨ corr(v_{k_1}) ∧ corr(v_{k_2}){
26.         update CONS' as above; update disjPairSet;
27.       }
28.    }
29. }
30. for each auxiliary variable v and formula F in Π such that
          corr(v) ∧ F ⊭ ⊥ and
          F : [L, U] is the head of a rule in Π {
31.     add constraint 0 ≤ v ≤ U to CONS'
32. }
33. return CONS';
```

Figure 5.7: An algorithm based on MonteCarloRefineCONS that uses the *small number of variables* and *targeted sampling* heuristics. The shaded part of the pseudocode corresponds to the small number of variables check.

remaining ones for future pairs. The process continues in this manner until all 100 samples are exhausted or until all pairs have been proved to be disjoint.

Analyzing the running time of the HeuristicRefineCONS algorithm is not an easy task, since it is composed of two distinct parts. However, the following two propositions give us some information regarding the computational cost in two particular cases. The first proposition involves the case in which only the exact checks in lines 3-13 are performed.

Proposition 12. *Let s be a state, $\mathcal{Q}$ be a set of actions of interest,* Π *be an* ap-*program containing m rules where the set of action atoms $\mathcal{A}$ has n elements. Suppose all possible pairs of correction formulas have in each case a combined set of atoms C such that $|C| \leq t$, then heuristicRefineCONS*$(\Pi, s, \mathcal{Q}, t, p)$ *runs in time in* $O(2^{2m+t})$.

Proof. Each SAT check in this case takes time in $O(2^t)$, and in the worst case there are $2^m - 1$ auxiliary variables. Therefore, checking all possible pairs takes time in $O(2^{2m+t})$. □

The next proposition explores a possible use of Proposition 12.

Proposition 13. *Let s be a state, $\mathcal{Q}$ be a set of actions of interest, and* Π *be an* ap-*program containing m rules where the set of action atoms $\mathcal{A}$ has n elements. Then, the worst case running time of heuristicRefineCONS*$(\Pi, s, \mathcal{Q}, 0, p)$ *is in* $O(np)$.

Proof. Since the *satThresh* parameter is zero, the exact part of the algorithm (lines 3-13) is never executed. For the rest of the algorithm, in the worst case the interpretations generated are always of length $O(n)$. Since p such interpretations are generated, and each can be evaluated in time in $O(n)$, the total running time is in $O(np)$. □

Propositions 12 and 13 illustrate two extreme cases; it is expected that in general a value of *satThresh* can be chosen such that the cost of the exact part of the algorithm does not render it too expensive. We will compare the performance of these two algorithms experimentally in Section 5.4.

Before concluding this section, we would like to briefly discuss how our randomized approach is related to the well known randomized algorithms for solving the satisfiability problem (SAT). Much work has been developed towards this end—Papadimitriou [Pap91] analyzed a *random walk*-style algorithm that, starting from a randomly selected interpretation, chooses variables at random belonging to unsatisfied clauses to "flip" in the hopes that this operation brings it closer to a satisfying assignment. The interesting result was that this procedure is very efficient for 2-SAT formulas with n variables, requiring only $O(n^2)$ flips to reach a satisfying interpretation, if one exists, with high probability. However, this result does not extend to k-SAT for $k > 2$, in which case an exponential number of flips is required in the general case (see, for instance, [WS02]). There has been much work since towards extending this algorithm [SLM92, SKC94, Sch99]); however, these algorithms suffer in the general case from exponentially high mixing times of the Markov chains used in the random walks described above [2].

Even though all of this work is closely related to the problem of refining the set of auxiliary variables in our domain, these algorithms are not directly applicable. The main reason for this is that we have a whole *set* of correction formulas, which in the worst case is of exponential size. This is why our algorithms are not based on random walks, opting instead to perform a rather simpler sampling of interpretations in an effort to at least find a subset of the existing overlaps among sets of models.

[2]The term *mixing time* refers to the number of steps the random walk must take in order to reach its stationary distribution; see [MT06] for a complete treatment.

```
Algorithm 9:  FindFocusedMPW(Π, s, Q)
1.  let W_Q be the set of worlds according to Q;
2.  obtain CONS_0(Π, s, X, Q);
3.  CONS' := RefineCONS(CONS_0(Π, s, X, Q));
4.  let currlow := 0;  currlowWorld := null;
5.  for each w_i^Q ∈ P(W^Q) {
6.      compute low(w_i^Q) w.r.t. CONS';
7.      if (low(w_i^Q) > currlow) {
8.          currlow := low(w_i^Q);
9.          currlowWorld := w_i^Q;
10.     }
11. }
12. return currlowWorld;
```

Figure 5.8: The FindFocusedMPW algorithm.

However, we will dedicate future work to investigate ways in which we may be able to take advantage of the many advances that have been made in applying random walk-style procedures to finding satisfying interpretations.

5.3.3 Finding Most Probable Worlds of Interest

The FindFocusedMPW algorithm shown in Figure 5.8 correctly computes the most probable world as long as step 6 correctly computes the result of minimizing p_i subject to the constraints in $\mathsf{CONS}_0(\Pi, s, \mathcal{Q})$. This can be done by minimizing this variable subject to the constraints computed by the RefineCONS algorithm, and in this case, the world returned by FindFocusedMPW is guaranteed to be the most probable world. However, if the Monte Carlo or Heuristic algorithms are used, this is not guaranteed.

The FindFocusedMPW algorithm yields a significant improvement in performance with respect to the basic Most Probable Worlds algorithm described in [KMN+07a], since the size of the linear constraints is much smaller. When

$n = |\mathcal{W}|$, the savings in number of worlds (and number of LPs solved) when using algorithm FindFocusedMPW is given by a factor of $2^{n-|\mathcal{Q}|}$. However, as we have seen, there is an extra cost associated with this algorithm, since the RefineCONS subroutine incurs additional costs when trying to separate all the auxiliary variables into an adequate set.

5.4 Experimental Evaluation

We developed a prototype implementation of the FindFocusedMPW algorithm using ExactRefineCONS, MonteCarloRefineCONS, and HeuristicRefineCONS constraint refinement methods described in this chapter. The implementation consisted of about 3,000 lines of Java code run on a Linux computing cluster comprised of 64 8-core, 8-processor nodes with between 10GB and 20GB of RAM; the cluster was not used for parallel computations, but for concurrent independent runs. The linear constraints for finding the most probable world were solved using the QSopt [ACDM09] linear programming solver library, and the logical formula manipulation code from the COBA belief revision system [DLST07] and SAT4J [sat09] satisfaction library were used for the SAT checks in the refinement methods.

To test the FindFocusedMPW algorithm with both the full refinement approach and the approximation algorithms, we conducted a series of experiments on *ap*-programs. Recall that an *ap*-program consists of a set of *ap*-rules, and that a state determines which of these *ap*-rules are applicable. For instance, we noticed that even though our *ap*-programs characterizing the behaviors of Hezbollah and Hamas [MMP+08a, MMP+08b] consist of several thousand rules each, in any given real world state, only a handful of rules (under 50) are usually applicable. As a

consequence, we generated sets of *applicable rules* randomly, rather than first generating an *ap*-program and then generating a state. All numbers reported below for times/number of splits are the result of averaging between 10 and 20 runs. For the experiments on small numbers of atoms, *satThresh* was set to 50% of the number of atoms of interest, and the number of samples (for Monte Carlo) as well as *testThresh* (for the heuristic) were set at 10% of the possible worlds. For the experiments on large numbers of atoms, *satThresh* was fixed at 10, and both number of samples and *testThresh* were fixed at $1,000$.

Experiment 1: *Running Times for Small Instances.* In the first experiment, we compared the running time of the FindFocusedMPW algorithm using the ExactRefineCONS to find an exact solution to using the Monte Carlo and Heuristic approaches. The number of action atoms was varied between 5 and 13, the size of the formulas was fixed at 5, and the number of applicable rules in the programs were varied between 4 and 10, in steps of two. In each case, the number of atoms of interest was set to 50% of the total for each run. For the approximation algorithms, we report runs with sample sizes of 10%, 20%, and 30% of the number of possible worlds. Figure 5.9 shows the running time results. There are a number of interesting observations we can make by looking at Figure 5.9:

- First of all, by observing the graphs on the left, we can see that the Monte Carlo algorithm does not scale well: for 4 applicable rules, its performance is already worse than the exact algorithm when 30% sampling is used; however, performance remains much better than the exact algorithm with 10% sampling, regardless of the number of rules or atoms in the program. Even though this seems to indicate that Monte Carlo is worse than Exact when sampling is

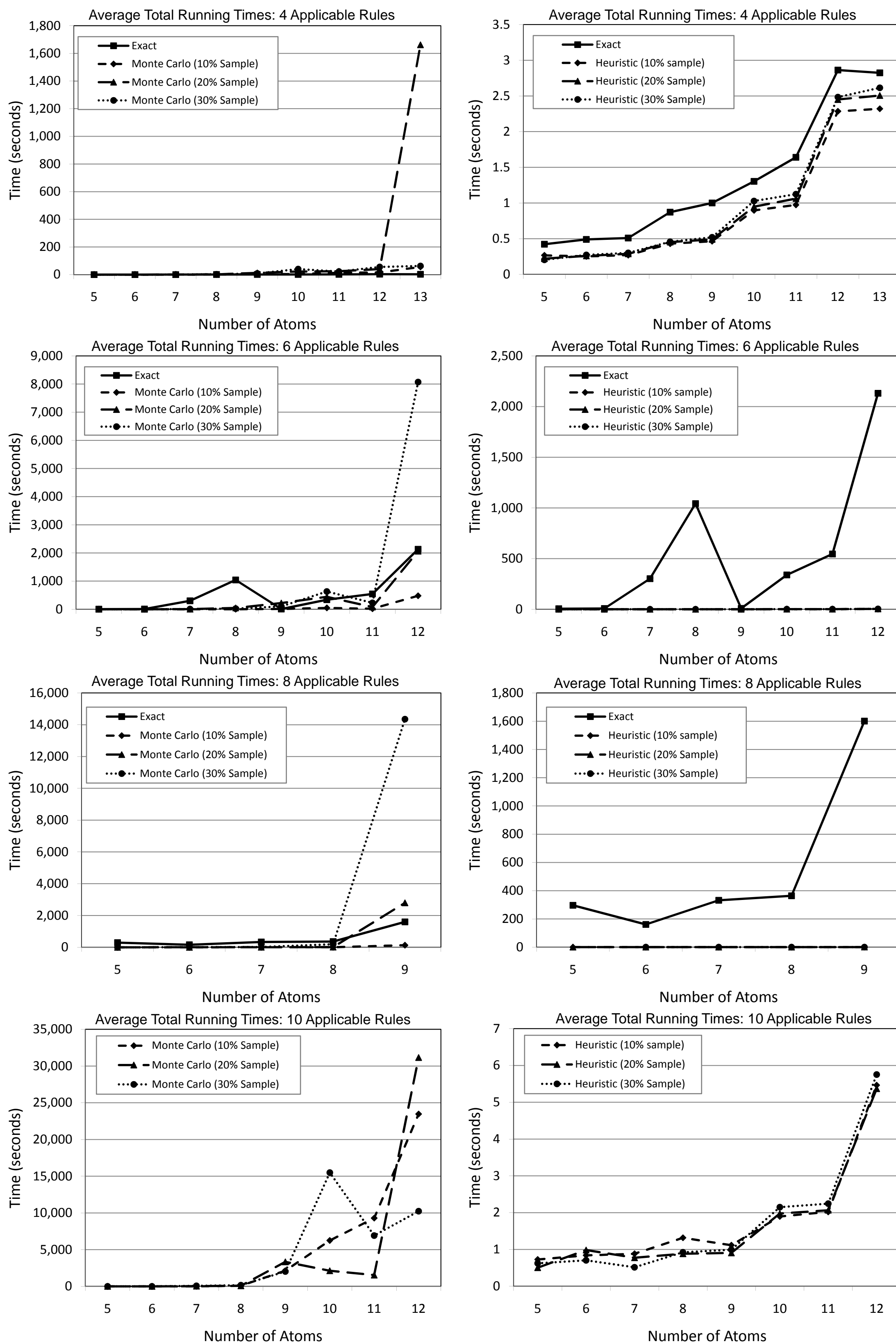


Figure 5.9: Comparison of average running times for small numbers of atoms, varying number of atoms and number of rules (*Left*: Monte Carlo, *Right*: Heuristic).

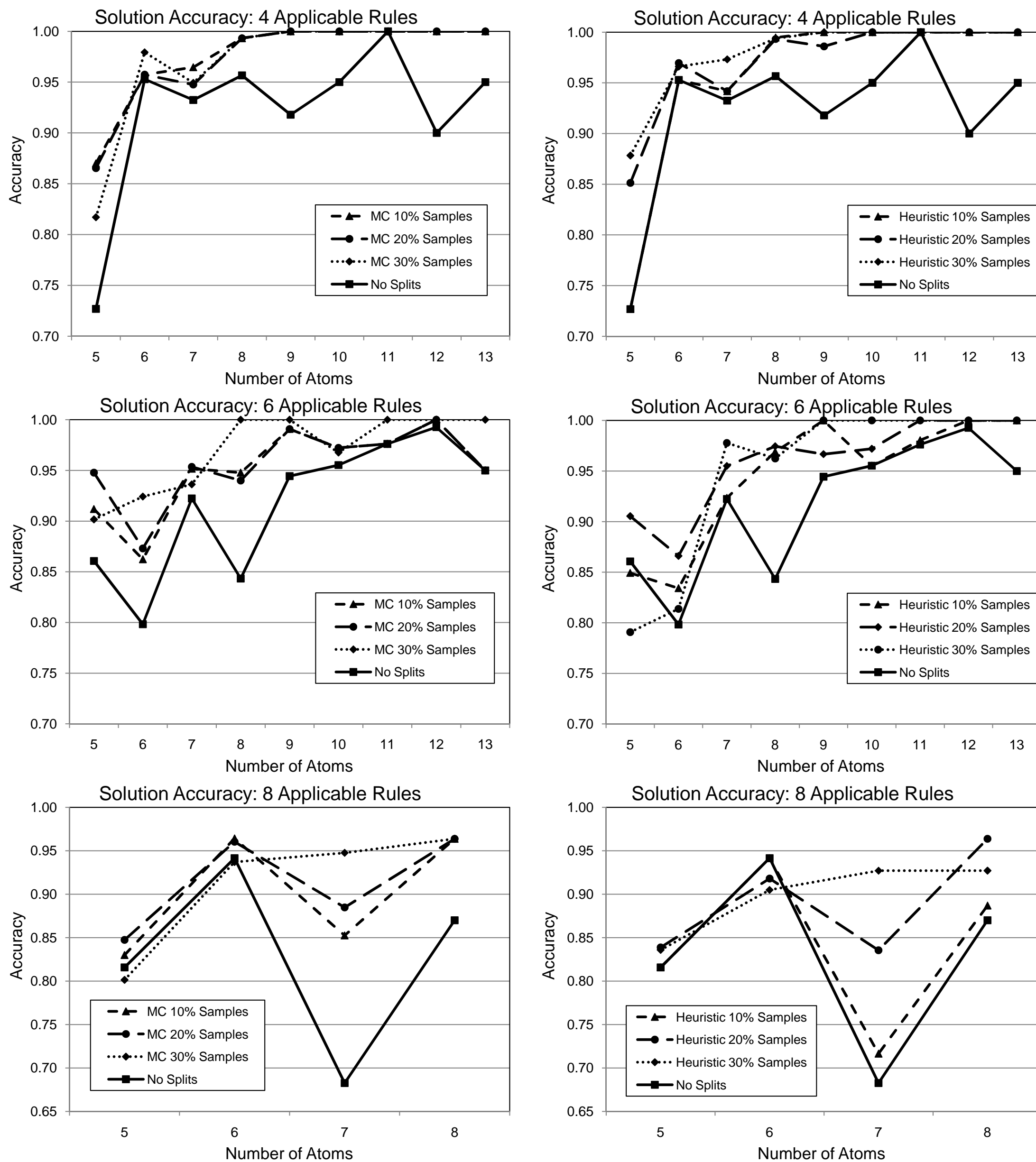


Figure 5.10: Accuracy of the approximation algorithms, varying number of atoms and number of rules (*Left*: Monte Carlo, *Right*: Heuristic).

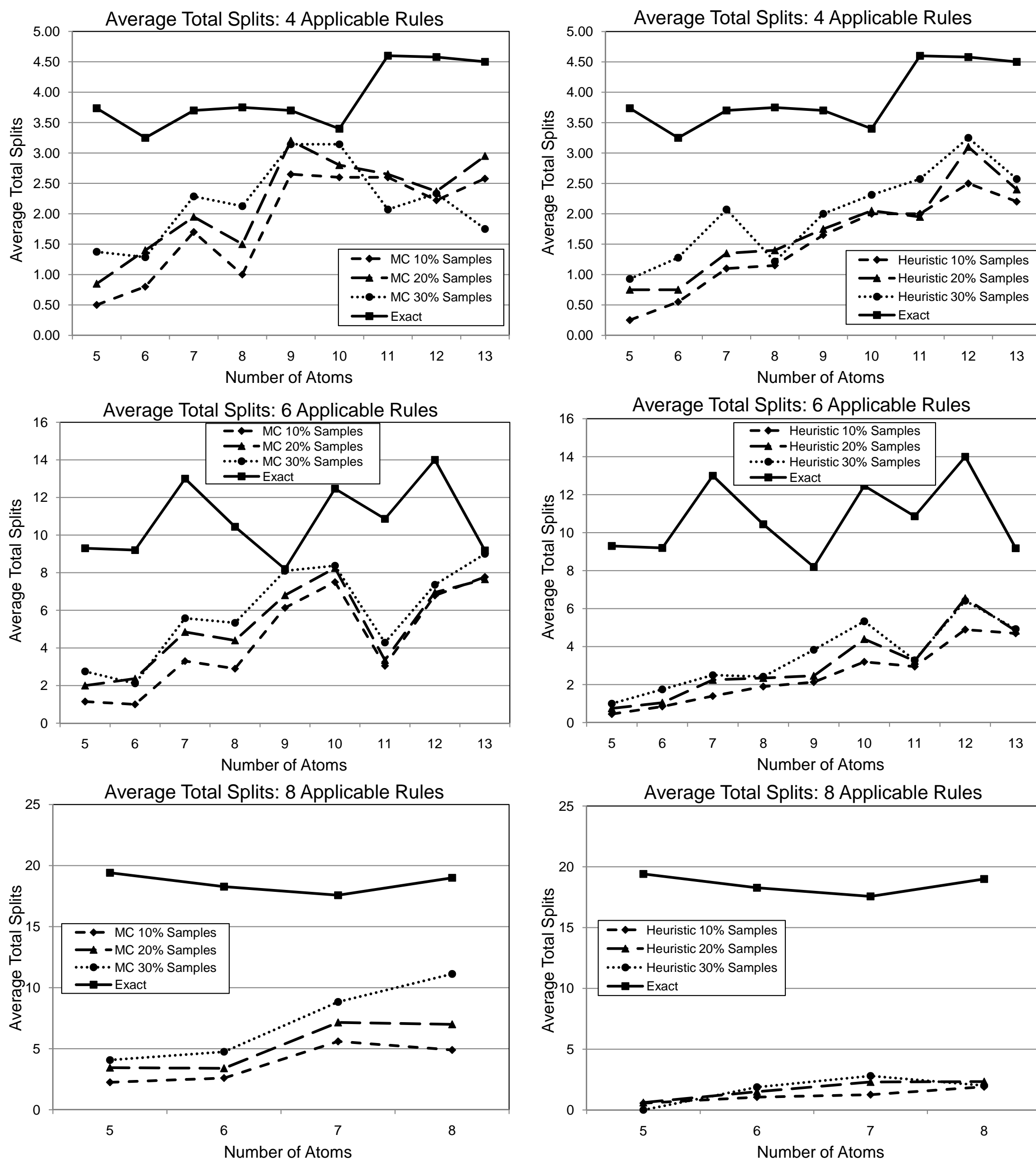


Figure 5.11: Number of "splits" found by the different algorithms, for small numbers atoms, varying number of atoms and number of rules (*Left*: Monte Carlo, *Right*: Heuristic).

higher than 10%, we will see in Experiment 3 that this is not the case, since it is capable of running on large numbers of atoms, whereas Exact is not.

- In contrast, the graphs on the right show that the Heuristic algorithm is quite efficient. For 4 applicable rules, its performance is comparable to that of the Exact algorithm, but this is due to the fact that at this instance size the Exact algorithm is efficient, and Heuristic is likely investing much of its time carrying out the exact (small number of variables) checks. The situation changes drastically for programs with 6 and 8 applicable rules, where the curve corresponding to the Heuristic algorithm can barely be seen since it is dwarfed by the time taken by the Exact algorithm.

- Finally, we report at the bottom of Figure 5.9 runs for 10 applicable rules, but only for the approximation algorithms, since the Exact algorithm cannot be evaluated at this size of instance. Once again, we see that the Heuristic algorithm is much more efficient, regardless of sampling size and size of the instance.

Experiment 2: *Accuracy for Small Instances.* For the same runs as the ones reported for Experiment 1, we evaluated the accuracy of the results obtained by the approximation algorithms, where accuracy was measured as one minus the absolute value of the difference between the approximation algorithms' answers and those yielded by the Exact algorithm. The same parameter variations as for Experiment 1 were used; however, the runs for programs with 10 applicable rules could not be carried out since Exact cannot run on instances of that size. The results are reported in Figure 5.10; the first observation that can be made is that all answers are at least 70% accurate as compared with the result yielded by the Exact algorithm. These

graphs include a fourth curve (called "No Splits" in the figure) that corresponds to the accuracy of a baseline algorithm that *does not make any effort to refine the set of constraints*. As we can see, the approximation algorithms yield more accurate results than the baseline (except for two isolated cases), and often surpass the 95% mark. Another interesting observation we can make is that the Heuristic algorithm does not sacrifice accuracy with respect to Monte Carlo, even though it is much more efficient. As a complement to this set of runs, we also measured the number of "splits" (cases in which correction formulas were found to share models) found by each algorithm; the results are reported in Figure 5.11. This is an interesting measurement to make, since it is not clear how accuracy of the result is correlated with the number of splits performed. The results we obtained show that Heuristic found a smaller number of splits than Monte Carlo (this makes sense at this size of instance, given the way that Heuristic divides up its samples among possible pairs of variables). This observation seems to indicate that there is a "threshold" number of splits that must be found in order to be reasonably accurate, since as we mentioned before both algorithms have very similar accuracy at this size of instance.

Experiment 3: *Running Times for Small Number of Rules and Large Number of Action Atoms.* To better evaluate the performance of the approximation algorithms, we performed a series of runs over programs with large numbers of action atoms, varying the number of applicable rules over three values (8, 10, and 12). The results are reported on the left hand side of Figure 5.12. Beyond some fluctuations that appear due to the random generation of *ap*-programs, the three graphs show a tendency of increasing running times when the size of the programs increase, both in terms of number of action atoms and number of applicable rules. Note that the Monte Carlo algorithm is affected much more by the variation in size, whereas the

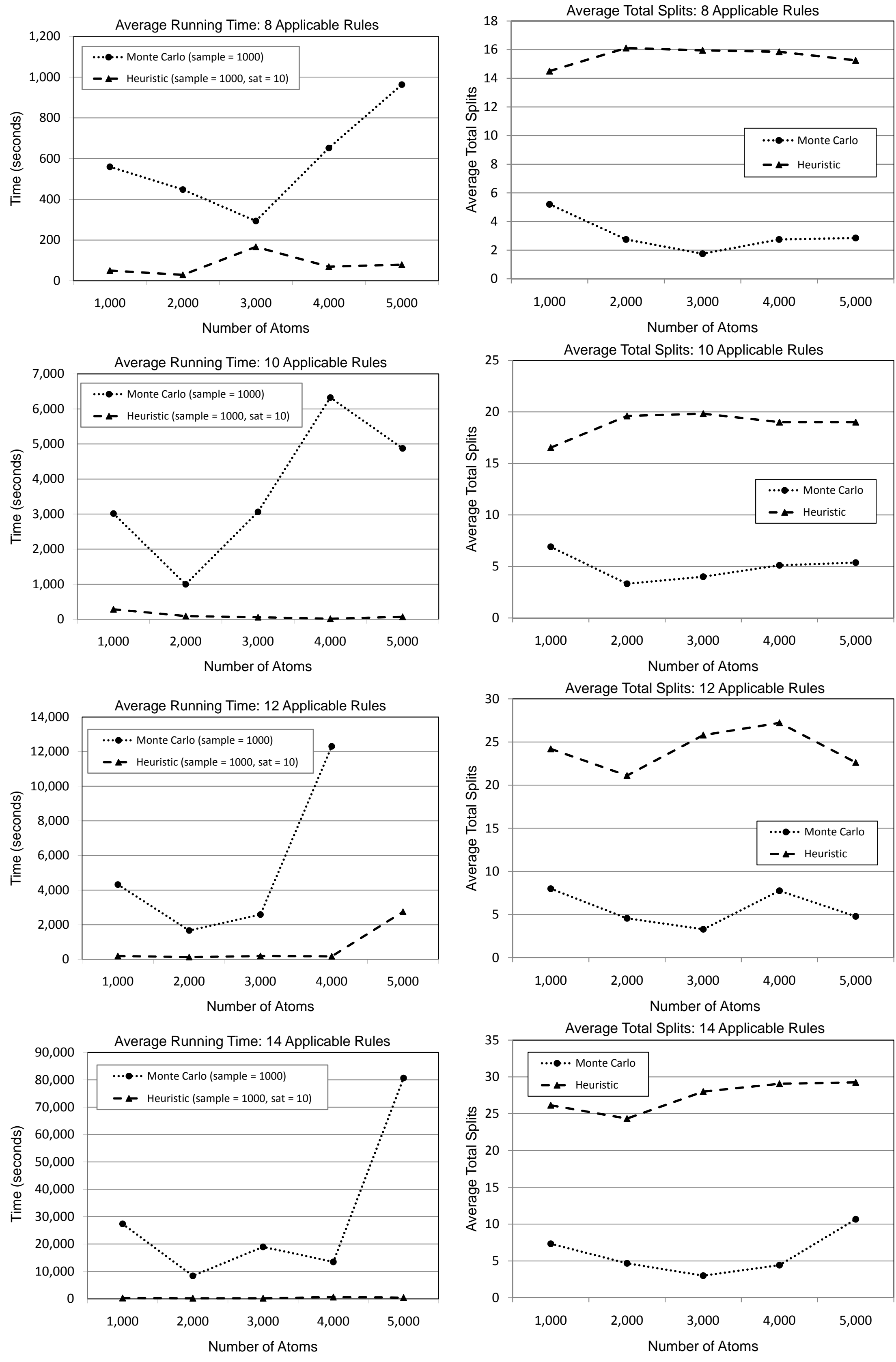


Figure 5.12: Comparison of average running times (left) and number of splits found (right) for large numbers of atoms, varying number of atoms and number of rules.

Heuristic algorithm barely lifts up from the x-axis except for the largest programs (14 applicable rules and 5,000 atoms).

Experiment 4: *Number of Splits for Small Number of Rules and Large Number of Action Atoms.* Using the same runs as those for Experiment 3, we analyzed the number of splits found by the two algorithms; the results are reported on the right hand side of Figure 5.12. The main conclusion we can obtain from this experiment is that the Heuristic algorithm always finds more splits than Monte Carlo does, which is in line with the hypothesis that we formulated when presenting the Heuristic algorithm above. In conjunction with the conclusions of Experiment 2, this means that the results obtained by Heuristic will in general be of greater quality than those obtained by Monte Carlo.

Experiment 5: *Running Times for Large Instances.* Finally, given the performance of the Heuristic algorithms in the previous experiments, we decided to run an experiment for a larger numbers of applicable rules (60), varying the number of atoms in a similar fashion as for the above experiments. The results are reported in Figure 5.13. As we can see, the Heuristic algorithm remains scalable even at this number of applicable rules, taking a little over 500 seconds (about 8.3 minutes) to refine the set of constraints for 7,000 atoms.

5.5 Concluding Remarks

Before concluding this chapter, we believe a note is in order to clarify the similarities and differences between our two approaches to most probable world computations. While both Chapters 4 and 5 address the problem of finding *most probable worlds*, the main difference between the two is that the latter focuses on

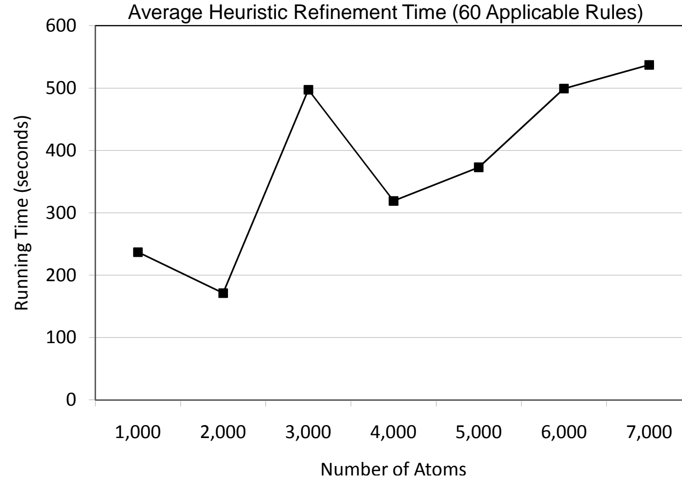


Figure 5.13: Average time taken by HeuristicRefineCONS to refine the set of constraints associated with an *ap*-program with 60 applicable rules, varying the number of atoms.

finding most probable worlds *of interest*. Suppose we consider an *ap*-program Π containing k ground action atoms in its associated language. Then the algorithms in Chapter 4 would consider 2^k worlds and try to find the most probable world. In contrast, in Chapter 5, we assume we have an additional quantity: a subset of k' ground action atoms in which the user is interested. Therefore, in Chapter 5, the worlds of interest correspond to subsets of these k' ground action atoms. A most probable world drawn from this class of "worlds of interest" can vary dramatically from the most probable worlds of Chapter 4 and moreover, they are of great interest. For instance, a user using the STOP system mentioned in Chapter 1 may only be interested in certain actions (*e.g.*, transnational attacks by a given group instead of the whole suite of actions). This, for instance, may define the mission that a European or American defense analyst might have with respect to certain groups' actions. In this case, he is only interested in worlds defined with respect to the subset of actions of interest to him. The methods described in Chapter 5 show how we can seamlessly address this problem by building upon the work in Chapter 4.

In summary, in this chapter we have developed methods to obtain an alternative linear program by taking into account the set of action atoms that a specific user might be interested in. This has a twofold effect: (1) it allows the user to obtain probability values that are the product of only discerning among worlds of interest, instead of receiving as an answer a single world with an associated probability, as in Chapter 4; and (2) the linear programs involved in most probable world of interest computations are smaller than those obtained in Chapter 4 and often, but not always, lead to a fast solution. The SemiHOP algorithm presented in Chapter 4 also proposes a reduction in number of variables in the resulting linear program by defining equivalence classes of worlds. However, that algorithm bases the equivalence classes on co-occurrence of worlds in constraints, whereas the equivalence classes that arise here are purely a consequence of the input provided by the user through the set $\mathcal{Q}$ of actions that are interesting to him.

We have developed the RefineCONS algorithm, and two approaches based on Monte Carlo sampling in order tackle the scalability issues that affect the exact algorithm. We report on extensive experiments showing that the latter work efficiently in practice, and that two heuristics can be used to greatly enhance the performance of the basic Monte Carlo approach.

Chapter 6

Abductive Inference in Action Probabilistic Logic Programs

As we discussed in Chapter 1, Action-probabilistic logic programs (*ap*-programs) are a class of probabilistic logic programs that have been extensively used during the last few years for modeling behaviors of entities. Rules in *ap*-programs have the form "If the environment in which entity E operates satisfies certain conditions, then the probability that E will take some action A is between L and U". In this chapter, we are interested in a the problem of given an *ap*-programs, trying to *change the environment*, subject to some constraints, so that the probability that entity E takes some action (or combination of actions) is maximized. This is called the Basic Abductive Query Answering Problem (BAQA). We first formally define and study the complexity of BAQA and several variants of it. We then provide an exact (exponential) algorithm to solve BAQA, followed by more efficient algorithms for specific subclasses of BAQA. We also develop appropriate heuristics to solve BAQA efficiently.

6.1 Introduction

Action probabilistic logic programs (*ap*-programs for short) [KMN+07a] are a class of the extensively studied family of probabilistic logic programs (PLPs) [NS92, NS93, KIL04]. We now provide a brief recap of the introduction from Chapters 1 and 2.

Action probabilistic logic programs have been used extensively to model and reason about the behavior of groups and an application for reasoning about terror groups based on *ap*-programs has users from over 12 US government entities [Gil08]. *ap*-programs use a two sorted logic where there are "state" predicate symbols and "action" predicate symbols[1] and can be used to represent behaviors of arbitrary entities (ranging from users of web sites to institutional investors in the finance sector to corporate behavior) because they consist of rules of the form "if a conjunction C of atoms is true in a given state S, then entity E (the entity whose behavior is being modeled) will take action A with a probability in the interval $[L, U]$."

In this kind of applications, it is essential to avoid making probabilistic independence assumptions, since the approach involves *finding out* what probabilistic relationships exist and then exploit these findings in the forecasting effort. For instance, Figure 6.1 shows a small set of rules automatically extracted from data [ACW08] about Hezbollah's past. Rule 1 says that Hezbollah uses kidnappings as an organizational strategy with probability between 0.5 and 0.56 in years in which no political support was provided to it by a foreign state (`forstpolsup`), and the severity of inter-organizational conflict involving (`intersev1`) it is at level

[1] Action atoms only represent the fact that an action is taken, and not the action itself; they are therefore quite different from actions in domains such as AI planning or reasoning about actions, in which effects, preconditions, and postconditions are part of the specification. We assume that effects and preconditions are generally not known, though later on we show how to represent the information we may have about them.

r_1. $\texttt{kidnap}(1) : [0.50, 0.56] \leftarrow \texttt{forstpolsup}(0) \wedge \texttt{intersev1}(c)$.
r_2. $\texttt{kidnap}(1) : [0.80, 0.86] \leftarrow \texttt{extsup}(1) \wedge \texttt{demorg}(0)$.
r_3. $\texttt{kidnap}(1) : [0.80, 0.86] \leftarrow \texttt{extsup}(1) \wedge \texttt{elecpol}(0)$.
r_4. $\texttt{tlethciv}(1) : [0.49, 0.55] \leftarrow \texttt{demorg}(1)$.
r_5. $\texttt{tlethciv}(1) : [0.71, 0.77] \leftarrow \texttt{elecpol}(1) \wedge \texttt{intersev2}(c)$.

Figure 6.1: A small set of rules modeling Hezbollah.

"c". Rules 2 and 3, also about kidnappings, state that this action will be performed with probability between 0.8 and 0.86 in years in which no external support is solicited by the organization (`extsup`) and either the organization does not advocate democratic practices (`demorg`) or electoral politics is not used as a strategy (`elecpol`). Similarly, Rules 4 and 5 refer to the action "civilian targets chosen based on ethnicity" (`tlethciv`). The first one states that this action will be taken with probability 0.49 to 0.55 in years in which the organization advocates democratic practices, while the second states that the probability rises to between 0.71 and 0.77 in years in which electoral politics are used as a strategy and the severity of inter-organizational conflict (with the organization with which the second highest level of conflict occurred) was not negligible" (`intersev2`). *ap*-programs have been used extensively by terrorism analysts to make predictions about terror group actions [Gil08, MMP$^+$08a].

Suppose, rather than predicting what action(s) a group would take in a given situation or environment, we want to determine what *we can do* in order to induce a given behavior by the group. For example, a policy maker might want to understand what we can do so that a given goal (*e.g.*, the probability of Hezbollah using kidnappings as a strategy is below some percentage) is achieved, given some constraints on what is feasible. The *basic abductive query answering problem* (BAQA) deals with finding how to *reach* a new (feasible) state from the current state such

$s_1 = \{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(0), \texttt{elecpol}(1), \texttt{extsup}(0), \texttt{demorg}(0)\}$
$s_2 = \{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(0), \texttt{elecpol}(0), \texttt{extsup}(0), \texttt{demorg}(1)\}$
$s_3 = \{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(0), \texttt{elecpol}(0), \texttt{extsup}(0), \texttt{demorg}(0)\}$
$s_4 = \{\texttt{forstpolsup}(1), \texttt{intersev1}(c), \texttt{intersev2}(c), \texttt{elecpol}(1), \texttt{extsup}(1), \texttt{demorg}(0)\}$
$s_5 = \{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(c), \texttt{elecpol}(0), \texttt{extsup}(1), \texttt{demorg}(0)\}$

Figure 6.2: A small set of possible states

that the *ap*-program associated with the group and the new state jointly entail that the goal will be true within a given probability interval.

In this chapter, we first briefly recall *ap*-programs and then formulate BAQA theoretically. We then develop a host of complexity results for BAQA under varying assumptions. We then describe both exact and heuristic algorithms to solve the BAQA problem. We also describe a prototype implementation and experiments showing that our algorithm is feasible to use even when the *ap*-program contains hundreds of rules. A brief note on related work before we begin; almost all past work on abduction in such settings have been devised under various independence assumptions [Poo97, Poo93b, Chr08]. We are aware of no work to date on abduction in possible worlds-based probabilistic logic systems such as those of [Hai84], [Nil86], and [FHM90] where independence assumptions are not made.

6.2 Basic Abductive Queries to Probabilistic Logic Programs

Suppose s is a state (the current state), G is a goal (an action formula), and $[\ell, u] \subseteq [0, 1]$ is a probability interval. The BAQA problem tries to find a new state s' such that $\Pi_{s'}$ entails $G : [\ell, u]$. However, s' must be *reachable* from s. For this, we merely assume the existence of a reachability predicate *reach* specifying *direct*

reachability from one state to another. The reflexive transitive closure of *reach* is denoted with *reach**, and *unReach* is its complement.

A Note about Reachability. We will investigate, in Section 6.3 below, one way in which *reach* can be specified, as well as ways in which knowledge of action effects and preconditions can be encoded into this predicate. It should be noted that specifying the reachability predicate may not always be an easy task for the user. However, an *iterative* process will likely help the user refine his input to *reach* when presented with solutions that he realizes are not actually possible. The same situation could arise with respect to goals, since the user may realize after being presented with a solution that his goal was underspecified. The need for such an iterative process underscores the importance of fast algorithms for BAQA, which is the main goal of this chapter.

The following is a simple example of a reachability predicate based on the running example introduced above.

Example 23. *Suppose, for simplicity, that the only state predicate symbols are those that appear in the rules of Figure 6.1, and consider the set of states in Figure 6.2. Then, some examples of reachability are the following:* $reach(s_1, s_2)$, $reach(s_1, s_3)$, $reach(s_2, s_1)$, $reach(s_4, s_1)$, $\neg reach(s_2, s_5)$, *and* $\neg reach(s_3, s_5)$. *Note that, if state* s_5 *is reachable, then the* ap-*program is inconsistent, since both rules 1 and 2 are relevant in that state.*

We can now state the BAQA problem formally:

BAQA Problem.

Input: An *ap*-program Π, a state s, a reachability predicate *reach* and a ground *ap*-formula $G : [\ell, u]$.

Output: "Yes" if there exists a state s' such that $reach^*(s, s')$ and $\Pi_{s'} \models G : [\ell, u]$, and "No" otherwise.

Example 24 (Solution to BAQA)**.** *Consider once again the program in the running example and the set of states from Figure 6.2. If the goal is $kidnap(1) : [0, 0.6]$ (we want the probability of Hezbollah using kidnappings to be at most 0.6) and the current state is s_4, then the problem is solvable because Example 23 shows that state s_1 can be reached from s_4, and $\Pi_{s_1} \models kidnap(1) : [0, 0.6]$.*

There may be costs associated with transforming the current state s into another state s', and also an associated probability of success of this transformation (*e.g.*, the fact that we may try to reduce foreign state political support for Hezbollah may only succeed with some probability). We will formulate this problem formally and present algorithms for solving it in Chapter 7

The following proposition shows the intractability of the BAQA problem in the general case.

Proposition 14. *The BAQA problem is EXPTIME-complete.*

Proof. Suppose we are given an instance of BAQA consisting of an *ap*-program Π, a goal $G : [\ell_G, u_G]$, a reachability predicate *reach*, and an initial state s_0. We first point out that any such instance of BAQA can be solved in time exponential in the size of the input by straightforward search through the space of all possible states, testing all possible subsets of Π, and solving the linear programs associated with each one of these possible subsets in order to test for entailment.

In order to show completeness, let P be an arbitrary problem in *EXPTIME* and TM_P be a deterministic Turing machine that decides P for any input x in time in $O(2^{|x|})$. We will provide a polynomial-time transformation from a description

ΔTM_P of TM_P and x to an instance of BAQA such that TM_P accepts x if and only if the associated BAQA instance returns *true*. We start by describing a state space $\mathcal{S}$ that mimics the space of all possible configurations of TM_P over x, allowing for two special states s_0 and s^*. Since the size of $\mathcal{S}$ is clearly in $O\left(2^{|\Delta TM_P|}\right)$, we can encode it by means of a set $\mathcal{L}_{sta}$ of size in $O(|\Delta TM_P|)$. Now, we will specify the *reach* predicate by making $reach(s_0, s_1)$ true for the state s_1 corresponding to the initial configuration of TM_P over x, and $reach(s_f, s^*)$ true for any state s_f that corresponds to an accepting configuration. Finally, we will make $reach(s_i, s_j)$ true for any states $s_i, s_j \in \mathcal{S}$ such that the transition rules in ΔTM_P state that the configuration associated with s_j can be reached directly from the configuration associated with s_i; $reach(s_i, s_j)$ is false for all pairs of states s_i, s_j that do not fall under any of the preceding cases. Finally, let Π consist of the single rule $F : [\varepsilon, 1] \leftarrow s^*$, where F is an arbitrary satisfiable formula over an arbitrary set $\mathcal{L}_{act}$ and $\varepsilon \in (0, 1]$, s_0 be the initial state, and let the goal be $F : [\varepsilon, 1]$.

Given the above construction, it is clear that the only way in which the BAQA instance can be solvable is if s^* is reachable from s_0, and this is possible if and only if MT_P accepts x. Since the transformation was done in polynomial time, the statement follows. □

Moreover, this problem is likely to be intractable even under simplifying assumptions, as shown in the following two results.

Corollary 3. *Let $\mathcal{L}_{act}$ be such that $|\mathcal{L}_{act}| \leq c'$ for some constant $c' \in \mathbb{N}$; the BAQA problem under this assumption is EXPTIME-complete.*

Proof. The proof is immediate by observing that the proof of Proposition 14 only makes use of a constant-sized $|\mathcal{L}_{act}|$. □

Proposition 15. *Let $\mathcal{L}_{sta}$ be such that $|\mathcal{L}_{sta}| \leq c'$ for some constant $c' \in \mathbb{N}$; the BAQA problem under this assumption is NP-complete.*

Proof. Membership in *NP* can be shown by applying Lemma 1 (cf. Chapter 4, Page 57). For any "yes" instance of the problem, the witness will consist of a proof of reachability (of polynomial size given the hypothesis $|\mathcal{L}_{sta}| \leq c'$), a set of rules $\Pi' \subseteq \Pi$, and an assignment of non-zero values to a polynomial number variables in the associated linear program. This witness can clearly be verified in polynomial time.

We will prove *NP*-hardness by reduction from SAT. Let F be a boolean formula that is the input to SAT instance; we then need to obtain, in polynomial time, an instance of BAQA such that it has a solution if and only if F is satisfiable. Let Π be an *ap*-program consisting of a single rule $F : [\varepsilon, 1] \leftarrow s$, for some $\varepsilon > 0$; furthermore, let s be the initial state and $F : [\varepsilon, 1]$ be the goal formula.

If F is satisfiable, clearly $\Pi \models F : [\varepsilon, 1]$ and, since the initial state makes the only rule in Π relevant, the problem has a solution. On the other hand, if F is not satisfiable, then Π will only entail $F : [0, 1]$, and therefore the problem will not be solvable. □

The above results reveal that the complexity of BAQA is caused by two factors. Specifically, we need to address the following two problems:

(P1) Find a subprogram Π' of Π such that when the body of all rules in that subprogram is deleted, the resulting subprogram entails the goal, and

(P2) Decide if there exists a state s' such that $\Pi' = \Pi_s$ and s is reachable from the initial state.

In the following, we will present algorithms and techniques for addressing these problems.

6.3 Algorithms for BAQA

In this section, we leverage the above intuition to first develop a naive algorithm for BAQA, then develop a more efficient algorithm for BAQA under the assumption that all goals are of the form $F : [0, u]$ (ensure that F's probability is less than or equal to u) or $F : [\ell, 1]$ (ensure that F's probability is at least ℓ). Finally, we develop a heuristic algorithm.

Naive Algorithm for BAQA. Before presenting a simple approach to solving BAQA exactly, we first define the concept of a subprogram graph.

Definition 20 (Subprogram reachability graph)**.** *Let* Π *be a ground* ap*-program and reach be a reachability predicate. The* subprogram graph *is defined as* $G = \left(2^{Heads(\Pi)}, E\right)$*, where* $(\Pi_1, \Pi_2) \in E$ *if and only if there exist states* s_1, s_2 *such that* Π_1 *(resp.* Π_2*) is the reduction of a subprogram relevant in* s_1 *(resp.* s_2*), and reach*$^*(s_1, s_2)$*.*

Figure 6.3 uses this graph to present a general template for solving BAQA. For instance, the subroutine *isSolution* called in line 3 simply checks if the *ap*-program that is being considered satisfies the goal; this check will depend on the specific problem that is being solved. The other generic subroutine is *getNextSubprogram*, called in line 5. This function is based on a traversal of the graph defined above, which can of course be implemented in a wide variety of ways. This algorithm is clearly sound and complete.

```
Algorithm 10:  subProgramSearchBAQA(Π, s, G : [ℓ_G, u_G], unReach)
1. curr:= Π_s; done:= false;
2. while not done do
3.   if isSolution(curr, G : [ℓ_G, u_G]) then
4.    return yes;
5.   curr:= getNextSubprogram(curr, unReach);
6.   if curr = null then
7.    done:= true;
8. return no;
```

Figure 6.3: A naive algorithm for solving BAQA based on the traversal of the relevant subprogram reachability graph induced by *reach*.

Answering Threshold Goals

A *threshold goal* is an annotated action formula of the form $F : [0, u]$ or $F : [\ell, 1]$. In this section, we study how we can devise a better algorithm for BAQA when only threshold goals are considered. This is a reasonable approach, since threshold goals can be used to express the desire that certain formulas (actions) should only be entailed with a certain maximum probability (upper bound) or should be entailed with at least a certain minimum probability (lower bound). The tradeoff lies in the fact that we lose the capacity to express both desires at once. We start by inducing equivalence classes on subprograms that limit the search space, helping address problem (P1).

Definition 21 (Equivalence of *ap*-programs)**.** *Let* Π *be a ground* ap-*program and* F *be a ground action formula. We say that subprograms* $\Pi_1, \Pi_2 \subseteq \Pi$ *are* equivalent *given* F*, written* $\Pi_1 \sim_F \Pi_2$*, iff* $\Pi_1 \models F : [\ell, u] \Leftrightarrow \Pi_2 \models F : [\ell, u]$ *for any* $\ell, u \in [0, 1]$*. Furthermore, states* s_1 *and* s_2 *are* equivalent *given* F*, written* $s_1 \sim_F s_2$*, iff* $reach(s_1, s_2)$*,* $reach(s_2, s_1)$*, and* $\Pi_{s_1} \sim_F \Pi_{s_2}$*.*

Example 25 (Equivalence of *ap*-programs)**.** *Let* Π *be the* ap-*program from Figure 6.1, formula* $F =$ kidnap(1)*,* $\Pi_1 = \{r_1\}$*,* $\Pi_2 = \{r_2, r_3\}$ $\Pi_3 = \{r_1, r_4\}$ $\Pi_4 =$

$\{r_1, r_5\}$, *and* $\Pi_5 = \{r_2, r_3, r_5\}$. *Here,* $\Pi_1 \sim_F \Pi_3$, $\Pi_1 \sim_F \Pi_4$, $\Pi_3 \sim_F \Pi_4$, *and* $\Pi_2 \sim_F \Pi_5$. *For instance, we can see that* $\Pi_1 \sim_F \Pi_3$ *because the probability with which* `kidnap`(1) *is entailed is given by rule* r_1*; rule* r_4 *is immaterial in this case. Clearly,* $\Pi_1 \not\sim_F \Pi_2$ *since* F *is entailed with different probabilities in each case.*

Next, consider the states from Figure 6.2 and the reachability predicate from Example 23. Since we have that $reach(s_1, s_2)$, $reach(s_2, s_1)$, Π_1 *is relevant in* s_1*, and* Π_3 *is relevant in* s_2*, we can conclude that* $s_1 \sim_F s_2$*.*

Relation $\sim$, both between states and between subprograms, is clearly an equivalence relation. The following lemma specifies a way to construct equivalence classes.

Lemma 2 (Sufficient condition for equivalence of *ap*-programs)**.** *Let* Π *be an* ap-*program and* G *be an action formula. Consider two subprograms* $\Pi', \Pi'' \subseteq \Pi$ *such that* $\Pi' = \Pi_a \cup \Pi'_p$ *(resp.,* $\Pi'' = \Pi_a \cup \Pi''_p$*), where* Π_a *is a set of rules whose heads have formulas* F *such that* $F \wedge G \not\models \perp$ *and* Π'_p *(resp.,* Π''_p*) contains rules whose heads have formulas* H *such that* $H \wedge G \models \perp$*. Then,* $\Pi' \sim_G \Pi''$*.*

Lemma 3 (Sufficient conditions for entailment)**.** *Let* Π *be a consistent* ap-*program and* $G : [\ell_G, u_G]$ *be a threshold goal. If there exists a rule* $r \in \Pi$ *such that* $Head(r) = F : [\ell_F, u_F]$ *and: either (1) if* $u_G = 1$, $F \models G$, *and* $\ell_G \leq \ell_F$*; or (2) if* $\ell_G = 0$, $G \models F$, *and* $u_G \geq u_F$*; then,* $\Pi \models G : [\ell_G, u_G]$*.*

The algorithm in Figure 6.4 first tries to leverage Lemma 3 and only proceeds if this is not possible. The way in which the algorithm partitions Π is partly based on Lemma 2; the following result proves that it correctly computes solutions to our problem.

Proposition 16 (Correctness and complexity of simpleAnnBAQA)**.** *Given an* ap-*program* Π*, a state* $s \in \mathcal{S}$*, and an annotated action formula* $G : [\ell, u]$*, Algorithm*

Algorithm 11: simpleAnnBAQA($\Pi, s, G : [\ell_G, u_G]$)

1. Select rules of the form $r : F : [\ell_r, u_r] \leftarrow s_1 \wedge \ldots \wedge s_n$ such that $F \wedge G \not\models \bot$; call all such rules *active rules*, and the complement set *passive rules*. We denote the former with *active*($\Pi, G : [\ell_G, u_G]$) and the latter with *passive*($\Pi, G : [\ell_G, u_G]$).
2. If Lemma 3 is applicable, return *true* if there exists a consistent $\Pi' \subseteq$ *candAct*($\Pi, G : [\ell_G, u_G]$) $\cup$ *passive*($\Pi, G : [\ell_G, u_G]$) s.t.:
   (a) If $u_G = 1$, then at least one rule $r \in \Pi'$ must have head $F : [\ell_F, u_F]$ such that $F \models G$ and $\ell_G \leq \ell_F$; otherwise (*i.e.*, $\ell_G = 0$), at least one rule $r \in \Pi'$ must have head $F : [\ell_F, u_F]$ such that $G \models F$ and $u_G \geq u_F$;
   (b) State s' for which $\Pi_{s'} = \Pi'$ is such that *reach*$^*(s, s')$.
3. Otherwise, for each rule $r_i : F : [\ell_r, u_r] \leftarrow s_1 \wedge \ldots \wedge s_n$ do:
   (a) If $\ell_G = 0$, $F \models G$, and $\ell_r > u_G$ then add r_i to set *conf*($\Pi, G : [\ell_G, u_G]$)
   (b) Otherwise (*i.e.*, $u_G = 1$), if $G \models F$ and $u_r < \ell_G$ then add r_i to set *conf*($\Pi, G : [\ell_G, u_G]$).
4. Let *candAct*($\Pi, G : [\ell_G, u_G]$) =
   active($\Pi, G : [\ell_G, u_G]$) $\setminus$ *conf*($\Pi, G : [\ell_G, u_G]$);
5. Consider the set *candAct*($\Pi, G : [\ell_G, u_G]$) $\cup$ *passive*($\Pi, G : [\ell_G, u_G]$) and, for each pair of rules $r_i : F_i : [\ell_{r_i}, u_{r_i}] \leftarrow s_1^i \wedge \ldots \wedge s_n^i$ and $r_j : F_j : [\ell_{r_j}, u_{r_j}] \leftarrow s_1^j \wedge \ldots \wedge s_m^j$ such that $F_i : [\ell_{r_i}, u_{r_i}]$ and $F_j : [\ell_{r_j}, u_{r_j}]$ are mutually inconsistent, add the pair (r_i, r_j) to a set called *inc*(Π).
6. Return *true* if there exists a set of rules $\Pi' \subseteq$ *candAct*($\Pi, G : [\ell_G, u_G]$) $\cup$ *passive*($\Pi, G : [\ell_G, u_G]$) such that $\Pi' \cap$ *candAct*($\Pi, G : [\ell_G, u_G]$) $\neq \emptyset$, no pair $\{r_1, r_2\} \subseteq \Pi'$ belongs to *inc*(Π), and:
   (a) $\Pi' \models G : [\ell_G, u_G]$;
   (b) $\exists$ state s' for which $\Pi_{s'} = \Pi'$ such that *reach*$^*(s, s')^2$.
7. If Step 6 is not possible, return *false*;

Figure 6.4: An algorithm to solve BAQA assuming a threshold goal.

simpleAnnBAQA correctly computes a solution to BAQA. Its worst case running time is in $O\left(2^{|\Pi|} + 2^{|\mathcal{L}_{sta}|} + 2^{|\mathcal{L}_{act}|}\right)$.

Proof. The construction of the sets in Steps 1 and 3 are designed to partition Π into rules that must play a part in the entailment of the goal (*candAct*), those that cannot play such a part (*conf*), and those that are irrelevant (*passive*). We have that $candAct = active \setminus conf$, since rules in *active* are the ones that have heads that are mutually satisfiable with the goal. It remains to see then that *conf* is correctly computed. To show that this is the case, consider first the case of a goal with a $[0, u_G]$ annotaton (Step 3a). Here, the algorithm marks as conflicting every rule with head F such that $F \models G$ and its lower bound annotation ℓ_r is such that $\ell_r > u_G$. Since all models of F are models of G, there will exist a linear constraint in LC stating that the sum of their probabilities must exceed ℓ_r; this implies that the sum of the probabilities of worlds satisfying G also exceeds ℓ_r, and therefore also u_G, which is inconsistent with the goal's annotation. Therefore, this rule should not be made relevant. Consider now a goal with a $[\ell_G, 1]$ annotation (Step 3b); now, conflicting rules are characterized as those with heads F such that $G \models F$ and upper bound annotation u_r such that $u_r > \ell_G$. Similar to the previous case, since all models of G are models of F, and there will exist a linear constraint in LC stating that the sum of the probabilities of models of F should not exceed u_r, this means that the sum of models of G cannot exceed u_r either, which is inconsistent with the $[\ell_G, 1]$ annotation since $u_r < \ell_G$. Therefore, such rules should not be made relevant. In Step 4, the algorithm takes the active rules that are not conflicting and calls them "candidate active". The final step of the setup is Step 5, which identifies pairs of rules that are mutually inconsistent.

In Step 6, we characterize the composition of the result; it is clear from our analysis above that all rules in *any* result must belong to $candAct(\Pi, G : [\ell_G, u_G]) \cup passive(\Pi, G : [\ell_G, u_G])$, and no pair can be mutually inconsistent. The two following

requirements simply make sure that the selected subprogram entails the goal and that the associated state is eventually reachable. Finally, if such a result cannot be obtained, then *false* must be returned.

To prove the worst case running time result, it is enough to note that satisfiability checks require time exponential in the number of atoms in the formulas, that we may need to search through all possible subsets of rules in $candAct \cup passive$ (of which there are $O(2^{|\Pi|})$), and that the other step that dominates computation time is the search for a reachable subprogram. The proof of Proposition 14 then yields the necessary result. □

Note that, if we assume that the number of atoms that can appear in action formulas in the heads of rules is bounded by a constant, then the term exponential in $|\mathcal{L}_{act}|$ will not be present in the running time of the algorithm. We now present an example of how this algorithm works.

Example 26 (simpleAnnBAQA over the running example). *Suppose* Π *is the* ap-*program of Figure 6.1, the goal is* $\texttt{kidnap}(1) : [0, 0.6]$ *(abbreviated with* $G : [0, 0.6]$ *from now on) and the state is that of Example 24,* $s_{curr} = \{\texttt{forstpolsup}(1),$ $\texttt{intersev1}(c), \texttt{intersev2}(c), \texttt{elecpol}(1), \texttt{extsup}(1), \texttt{demorg}(0)\}$*;*
note that $\Pi_{s_{curr}} = \{r_2, r_5\}$ *and that clearly* $\Pi_{s_{curr}} \not\models \texttt{kidnap}(1) : [0, 0.6]$*. Step 1 of simpleAnnBAQA is simple in this case, since all the heads of rules in* Π *are atomic – therefore* $passive(\Pi_{s_{curr}}, G : [0, 0.6]) = \emptyset$*, and the set of active rules contains all the rules in* Π*. The following step checks for the applicability of Lemma 3; clearly rule* r_1 *satisfies the conditions and we only need to verify that some subprogram containing it is reachable. Assuming the same reachability predicate outlined in Example 23,* $s_1 =$ $\{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(0), \texttt{elecpol}(1), \texttt{extsup}(0), \texttt{demorg}(0)\}$ *is*

reachable from s_{curr}; this corresponds to choosing subprogram $\Pi' = \{r_1\}$. The only other possibilities are to make both r_1 and one of r_4 or r_5 relevant.

In the next section we will explore ways in which *reach* can be expressed, and how different restrictions on this predicate impact the difficulty of solving BAQA.

An Improved BAQA Algorithm

In this section, we show that if we assume reachability/unreachability is specified in a syntactic manner rather than in a very general manner as presented earlier, we can come up with some good heuristics to solve BAQA.

Definition 22 (Reachability constraint). *Let F and G be first-order formulas over $\mathcal{L}_{sta}$ and $\mathcal{L}_{var}$, connectives $\wedge$, $\vee$, and $\neg$, such that the set of variables over F is equal to those over G, and all variables are assumed to be universally quantified with scope over both F and G. A* reachability constraint *is of the form $F \not\leftrightarrow G$; we call F the antecedent and G the consequent of the constraint, and its semantics is:*

$$unReach(s_1, s_2) \Leftrightarrow s_1 \models F \ \ and \ \ s_2 \models G$$

where s_1 and s_2 are states in $\mathcal{S}$.

Reachability constraints simply state that if the first formula is satisfied in a certain state, then no states that satisfy the second formula are reachable from it. We now present an example of a set of reachability constraints.

Example 27 (Reachability constraints)**.** *Consider again the setting and* ap*-program from Figure 6.1. The following are examples of reachability constraints:*

$$rc_1: \quad \texttt{forstpolsup}(1) \not\mapsto \texttt{intersev1}(c)$$

$$rc_2: \quad \texttt{extsup}(1) \not\mapsto \texttt{intersev1}(2)$$

$$rc_3: \quad \texttt{intersev1}(c) \vee \texttt{intersev2}(c)) \wedge \texttt{demorg}(0) \not\mapsto \texttt{demorg}(1)$$

Suppose that we wish to represent the fact that action kidnap(1) *cannot be taken whenever* demorg(1) *is true. This can be represented with constraint:*

$$\texttt{demorg}(1) \not\mapsto \texttt{kidnap_performed}(1)$$

where kidnap_performed(1) *is an environment atom expressing that action* kidnap(1) *was taken*[3]*. Knowledge of action effects can clearly be represented with constraints built in a similar manner.*

Algorithm *simpleAnnBAQA-Heur-RC* (Figure 6.5) takes advantage of the structure added by the presence of reachability constraints. The algorithm starts out by executing the steps of *simpleAnnBAQA* that compute the sets $active(\Pi, G : [\ell_G, u_G])$, $passive(\Pi, G : [\ell_G, u_G])$, $candAct(\Pi, G : [\ell_G, u_G])$, $conf(\Pi, G : [\ell_G, u_G])$, and $inc(\Pi)$. It then builds formulas generated by reachability constraints that any solution state must satisfy; the algorithm uses a subroutine *formula*(s) which returns a formula that is a conjunction of all the atoms in state s and the negations of those not in s. In Step 4, the formula describes the fact that at least one of the states that make relevant "candidate active" rules (as described in Algorithm *simpleAnnBAQA*) must

[3]This sort of atom is only necessary if we wish to encode knowledge of action effects and preconditions.

Algorithm 12: simpleAnnBAQA-Heur-RC($\Pi, s, G : [\ell_G, u_G], RC$)

1. execute Steps 1, 3, 4, and 5 of *simpleAnnBAQA*;
2. let *goalState*, *goalStateAct*, *goalStateConf*, and *goalStateInf* be logical formulas over $\mathcal{L}_{sta}$ and $\mathcal{L}_{var}$;
3. initialize *goalState* to null, *goalStateAct* to $\bot$, and *goalStateConf*, *goalStateInc* to $\top$;
4. for each rule $r_i \in candAct(\Pi, G : [\ell_G, u_G])$ with
   $Head(r_i) = F : [\ell_F, u_F]$ do
   if $\left[(u_G = 1) \text{ and } (F \models G \text{ and } \ell_G \leq \ell_F)\right]$ or
   $\left[(\ell_G = 0) \text{ and } (G \models F \text{ and } u_G \geq u_F)\right]$
   then set $goalStateAct := goalStateAct \vee getStateFormula(r_i)$;
5. for each rule $r_i \in conf(\Pi, G : [\ell_G, u_G])$ do
   set $goalStateConf := goalStateConf \wedge \neg getStateFormula(r_i)$;
6. for each pair of rules $(r_i, r_j) \in inc(\Pi)$ do
   set $goalStateInc := goalStateInc \wedge \neg(getStateFormula(r_i) \wedge getStateFormula(r_j))$;
7. set $goalState := goalStateAct \wedge goalStateConf \wedge goalStateInc$;
   // *goalState* describes the states that satisfy the goal
8. return $decideReachability(s, goalState, RC)$;

Figure 6.5: A heuristic algorithm based on Lemma 3 to solve BAQA assuming that the goal is an *ap*-formula of the form either $G : [0, u]$ or $G : [\ell, 1]$ and that state reachability is expressed as a set RC of reachability constraints.

be part of the solution; similarly, Step 5 builds a formula ensuring that none of the conflicting active rules can be relevant if the problem is to have a solution. Finally, Step 6 describes the constraints associated with making relevant rules that are probabilistically inconsistent. Noticeably absent are the "passive" rules from the previous algorithm; such rules impose no constraints on the solution. The last two steps put subformulas together into a conjunction of constraints, and the algorithm

must decide if there exist any states that model formula *goalState* and are eventually reachable from s.

Deciding eventual reachability, as we have seen, is one of the main problems that we set out to solve as part of BAQA. We therefore propose two possible implementations of this subroutine: *(i)* a SAT-based algorithm, presented in Figure 6.6 and *(ii)* one based on a hill climbing strategy, whose pseudocode can be found in Figure 6.7. The SAT-based algorithm is simple: if the current state does not satisfy *goalState*, it starts by initializing formula *Reachable* which will be used to represent the set of eventually reachable states at each step. The initial formula describes state s, and the algorithm then proceeds to select all the constraints whose antecedents are entailed by *Reachable*. Once we have this set, *Reachable* is updated to the conjunction of the negations of all the consequents of constraints in the set. We are done if either *Reachable* at this point models *goalState*, or the old version of *Reachable* is modeled by the new one, *i.e.*, no new reachable states were discovered.

Proposition 17. *Let s_1 be a state, goalState be a formula over states, and RC be a set of reachability constraints. Algorithm decideReachability-SAT$(s, goalState, RC)$ correctly decides if there exists a state s' such that $s' \models goalState$ and $reach^*(s, s')$.*

Proof. To prove correctness, we must prove that the algorithm returns *true* if and only if there exists a sequence of states $s_1, s_2, \ldots, s_k$ such that $reach(s_i, s_{i+1})$, for $1 \leq i < k$, and $s_k \models goalState$.

($\Rightarrow$) Suppose the algorithm returns *true*. Formula *Reachable* is built taking into account the formula associated with the initial state, and the subsequent updates made during the while loop in Line 3. *Reachable* is updated with the conjunction of the negations of the consequents of reachability constraints whose antecedents are modeled by the previous version of *Reachable*; clearly, the only models of the

updated *Reachable* are the states that are reachable from the old version. Since by hypothesis the algorithm returned *true*, we know that after a certain number of iterations *Reachable* and *goalState* are mutually satisfiable. Since reachability constraints were respected by construction at each iteration, this entails the existence of the sequence mentioned above for all states that are eventually reachable from s.

($\Leftarrow$) Suppose there exists a sequence of states $s_1, s_2, \ldots, s_k$ such that $reach(s_i, s_{i+1})$, for $1 \leq i < k$, and $s_k \models goalState$. By construction, the formula associated with s_1 and *Reachable* are mutually satisfiable by the start of the first iteration of the while loop in Line 4. Now, by hypothesis, state s_2 is reachable from s_1; since the algorithm updates *Reachable* to model all states that satisfy the reachability constraints, s_2 must be a model of *Reachabile* after the first iteration. Continuing in this manner, we can arrive at state s_k, which by the same reasoning will be a model of *Reachable*, and therefore the algorithm will return *yes* since it is also a model of *goalState*. □

The following is an example of how *decideReachability-SAT* works.

Example 28. *Consider the* ap-*program from Figure 6.1, along with constraint* rc_1 *from Example 27. As we saw in Example 26, if the goal is* $\texttt{kidnap}(1) : [0, 0.06]$ *and the current state is* $s_0 =$
$\{\texttt{forstpolsup}(1), \texttt{intersev1}(c), \texttt{intersev2}(c), \texttt{elecpol}(1), \texttt{extsup}(1), \texttt{demorg}(0)\}$*; then the either* $\{r_1\}$ *or* $\{r_1, r_4\}$ *should be made relevant, which yields the following goalState formula:*

$$\texttt{forstpolsup}(0) \wedge \texttt{intersev1}(c) \wedge \neg \left(\bigvee_{i=2,3,5} Body(r_i) \right)$$

Algorithm 13: decideReachability-SAT(s, *goalState*, *RC*)

1. let *Reachable* be a formula initialized to $formula(s)$;
2. set $done := (Reachable \wedge goalState \not\models \bot)$;
3. while not *done* do
   set $Reachable_{old} := Reachable$;
   let $RC_{curr} \subseteq RC$ be the set of constraints $F_i \not\leftrightarrow G_i$
   such that $Reachable \models F_i$;
   set $Reachable := \left(\bigwedge_{F_i \not\leftrightarrow G_i \in RC_{curr}} \neg G_i\right)$;
   set *done*:=
   $((Reachable \wedge goalState) \not\models \bot) \vee (Reachable \models Reachable_{old})$;
4. return $(Reachable \wedge goalState \not\models \bot)$;

Figure 6.6: An algorithm to decide reachability from a state s to any of the states that satisfy the formula *goalState*, where reachability is expressed as a set *RC* of reachability constraints. This version is based on deriving a formula that describes the set of all possible states eventually reachable from the initial one.

Reachable starts out with $formula(s_0)$ *and, as Reachable* $\models$ `forstpolsup(1)`*, it gets updated to:*

$$\neg\texttt{intersev1}(c)$$

which is mutually unsatisfiable with goalState. In the next iteration, however, as Reachable does not entail the antecedent of rc_1*, it gets updated to* $\top$*, which means that there are no constraints regarding the states that can be reached, and therefore the algorithm will answer* true.

Algorithm *decideReachability-HillClimb*, on the other hand, takes a different approach. Rather than characterize the states that are eventually reachable from s and seeing if such set overlaps with the models of *goalState*, it simply finds a *single* model g of *goalState* and computes the atoms that are “different” between s and g

(*i.e.*, atoms that are true in s and false in g and vice versa).[4] The algorithm then begins the hill climbing strategy by selecting atoms from these sets of differences to change in the current state, checking that the new state is in fact reachable from the old one given the constraints in *RC*. We are done whenever we find that such a change is impossible, or the change lead to a state that satisfies *goalState*. It should be noted that this algorithm is vulnerable to bad choices regarding the changes it makes to intermediate states, as well as the fact that it's impossible for it to change atoms that are not part of $diff^+$ or $diff^-$. It is therefore a heuristic algorithm that has the advantage of speed over completeness. It is, however, sound, as the following proposition proves.

Proposition 18. *Let s be a state, goalState be a formula over states, and RC be a set of reachability constraints. Algorithm decideReachability-HillClimb*$(s, goalState, RC)$ *is sound,* i.e.*, if it returns* true *then there exists a state s' such that $s' \models goalState$ and $reach^*(s, s')$.*

Proof. Apart from the trivial case in Line 1, the algorithm begins by obtaining a model g of the *goalState* formula, which describes all states that are solutions to the problem. The main work done by the algorithm is in the while loop in Line 4, which starts by computing the set of atoms that differ in *currState* (initialized to the starting state) w.r.t. g. Clearly, if *currState* can be changed into g by applying changes that satisfy all constraints in *RC*, then the algorithm will return *true*. The following operations in the while loop build a formula *Curr* that describes the set of states that can be accessed from *currState* according to the constraints. Finally, the loop ends by trying to select at random an atom to change, making sure that

[4] Note that focusing on a single state also allows is to be used in conjunction with the original *simpleAnnBAQA* algorithm, as it does not rely on a general reachability formula that can only be obtained by relying on Lemma 3.

Algorithm 14: decideReachability-HillClimb($s, goalState, RC$)

1. if $s \models goalState$ then return *true*;
2. let g be a state such that $g \models goalState$; set *done:= false*;
3. let *currState* be a state initialized to s;
4. while not *done* do

   let $diff^+$ be a set of atoms a_i such that $g \models a_i$ and $curr \not\models a_i$;

   let $diff^-$ be a set of atoms a_i such that $g \not\models a_i$ and $curr \models a_i$;

   let $RC_{curr} \subseteq RC$ be the set of constraints $F_i \not\leftrightarrow G_i$ such that $currState \models F_i$;

   for each constraint $F_i \not\leftrightarrow G_i \in RC_{curr}$ do

   set $Curr := \left(\bigwedge_{F_i \not\leftrightarrow G_i \in RC \wedge currState \models F_i} \neg G_i\right)$;

   let $currState'$ be a new state equal to *currState* except that each atom from a randomly chosen subset of $diff^+ \cup diff^-$ is made true (for +) or false (for $-$) in state $currState'$ and such that $currState' \models Curr$; if this is not possible, set *done:= true*;

   set *done:=* $currState' \models goalState$;

   set $currState := currState'$
5. return $Curr \models goalState$;

Figure 6.7: An algorithm to decide reachability from a state s to any of the states that satisfy the formula *goalState*, where reachability is expressed as a set RC of reachability constraints. This version is based on selecting a single goal state that satisfies *goalState* and performing a *hill climb* by selecting atoms that must be made true or false in order to reach it from the current one.

the change does not violate any constraints. Therefore, by construction, if the algorithm returns *true*, the sequence of intermediate values of *currState* proves that there exists a state that is eventually reachable from the starting state and satisfies *goalState*. □

The following is an example of how *decideReachability-HillClimb* works.

Example 29. *Consider the same setup from Example 28. The first step in the algorithm is to obtain a state g that satisfies goalState. Suppose we choose the following state $g =$*

$\{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(1), \texttt{elecpol}(c), \texttt{extsup}(0), \texttt{demorg}(1)\}$

Suppose that the current state is $s_0 =$

$\{\texttt{forstpolsup}(1), \texttt{intersev1}(c), \texttt{intersev2}(1), \texttt{elecpol}(c), \texttt{extsup}(1), \texttt{demorg}(1)\}$

The following are the two sets computed at the beginning of the while loop:

$$\mathit{diff}^+ = \{\texttt{forstpolsup}(0), \texttt{extsup}(0)\}$$

$$\mathit{diff}^- = \{\texttt{forstpolsup}(1), \texttt{extsup}(1)\}$$

Suppose the algorithm chooses to make $\texttt{forstpolsup}(1)$ *false from diff^-, and* $\texttt{forstpolsup}(0)$ *true from diff^+ in the next step. This, however, does not satisfy Curr, which at this point is $\neg\texttt{intersev1}(c)$. This is a case in which the algorithm will return* false *when there is actually a solution to the problem; unfortunately, as atom* $\texttt{intersev1}(c)$ *is not part of $\mathit{diff}^+ \cup \mathit{diff}^-$, it can never change in currState and thus Curr can never be satisfied.*

Example 30. *Consider now the following* ap*-program:*

$$\begin{aligned}
p \wedge q &\quad : [0.3, 0.5] \quad \leftarrow a \wedge b. \\
p &\quad : [0.3, 0.8] \quad \leftarrow a \wedge d. \\
p &\quad : [0.1, 0.2] \quad \leftarrow a \wedge b \wedge c. \\
q &\quad : [0.5, 0.9] \quad \leftarrow b. \\
s \wedge \neg p &\quad : [0.8, 0.95] \quad \leftarrow b \wedge d.
\end{aligned}$$

where $\mathcal{L}_{act} = \{p, q\}$ *and* $\mathcal{L}_{sta} = \{a, b, c, d\}$. *Let the following be the set of reachability constraints:* $\{c_1 : d \not\leftrightarrow a \wedge b, c_2 : b \not\leftrightarrow a, c_3 : a \not\leftrightarrow b, c_4 : \neg c \not\leftrightarrow d\}$. *Suppose we have the following goalState formula corresponding to goal* $p : [0.25, 1]$*:*

$$(a \wedge b) \wedge \neg(a \wedge b \wedge c) \wedge \neg(a \wedge d)$$

and that the current state is $s_0 = \{\neg a, \neg b, c, d\}$. *Suppose now that the hill climbing algorithm chooses state* $g = \{a, b, \neg c, \neg d\}$ *as the goal. Then, we have that* $diff^+ = \{a, b\}$ *and* $diff^- = \{c, d\}$. *The only antecedent that is satisfied in the current state is that of* c_1*, and therefore we can access any state that does not satisfy its consequent,* i.e., $a \wedge b$. *Suppose the algorithm chooses* $s_1 = \{\neg a, \neg b, \neg c, d\}$*; now, the only antecedent that is satisfied is that of* c_4*, and therefore we must make* d *false in the next state. Let* $s_2 = \{\neg a, \neg b, \neg c, \neg d\}$. *Now, since there is nothing stopping a transition to states in which* a *and* b *are both true, we can reach* g *and we are done. Note that this step could have been taken from* s_1 *directly, but the algorithm does not necessarily make the smallest number of transitions.*

Algorithm	**Reference**	**Comment**
subProgramSearchBAQA	Alg. 10, Fig. 6.3, p. 136	Naive approach: traverses the entire reachability graph checking for a solution.
simpleAnnBAQA	Alg. 11, Fig.6.4, p. 138	Works on goals with $[0, u]$ or $[\ell, 1]$ annotations; leverages subprogram equivalence and heuristics for rule selection.
simpleAnnBAQA-heur-RC	Alg. 12, Fig. 6.5, p. 143	Builds on top of Alg. 11 to build a formula describing all states eventually reachable from s_0
decideReachabilitySAT	Alg. 13, Fig. 6.6, p. 146	Correctly decides eventual reachability given the formula built in Alg. 12
decideReachabilityHillClimb	Alg. 14, Fig. 6.7, p. 148	Heuristic for deciding eventual reachability given the formula built in Alg. 12

Figure 6.8: A summary of the algorithms for BAQA.

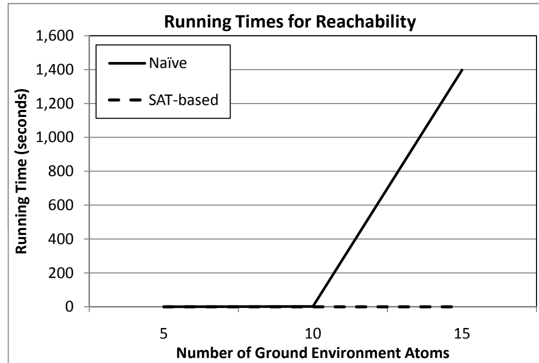


Figure 6.9: Varying number of ground state atoms for programs with 5 rules, 25 ground action atoms, 5 reachability constraints, and atomic queries.

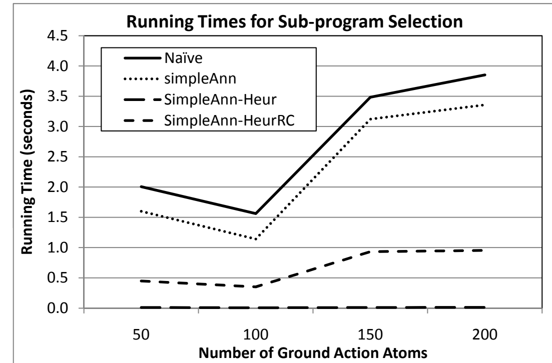


Figure 6.10: Varying the number of ground action atoms for *ap*-programs with 5 rules, 5 ground state atoms, and non-atomic queries.

6.4 Experimental Results

We conducted experiments using a prototype JAVA implementation consisting of roughly 2,500 lines of code All experiments were run on multiple multi-core Intel Xeon E5345 processors at 2.33GHz, 8GB of memory, running the Scientific Linux distribution of the GNU/Linux operating system, kernel version 2.6.9-55.0.2.ELsmp. *We note that this implementation makes use of only one processor and one core.* All numbers reported are averages over at least 20 runs to minimize experimental error; runs were performed over randomly generated *ap*-programs and goals based on the following parameters: number of ground state and action atoms, number of

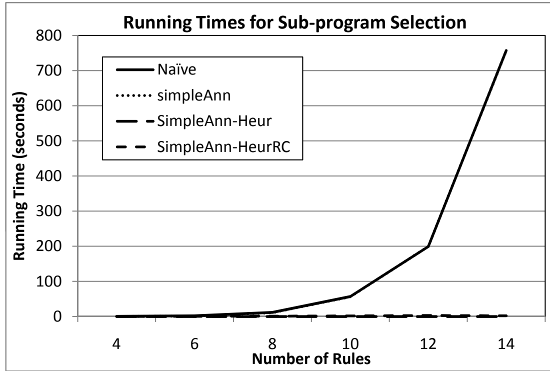


Figure 6.11: Varying number of rules; 25 ground action atoms, 5 ground state atoms, and atomic queries.

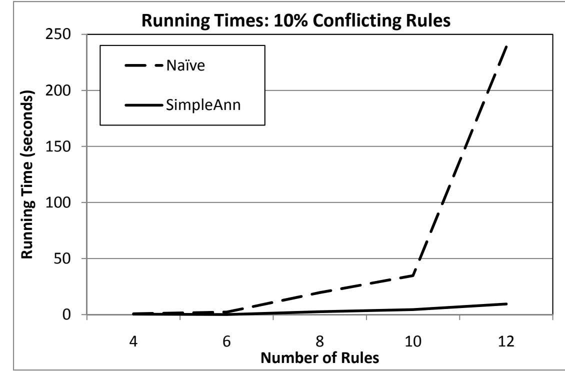


Figure 6.12: Varying number of rules (with 10% of them goal-conflicting); 25 ground action atoms, 5 ground state atoms, and atomic queries.

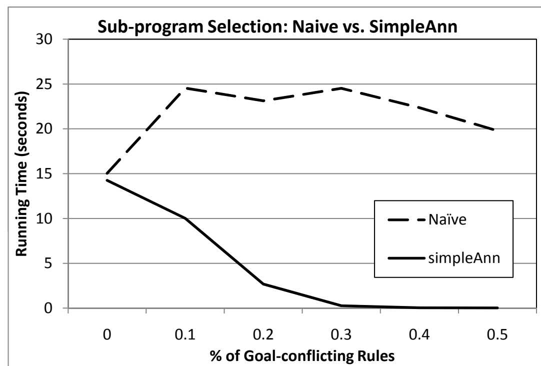


Figure 6.13: Varying the percentage of rules that are in conflict with the goal; *ap*-programs with 10 rules.

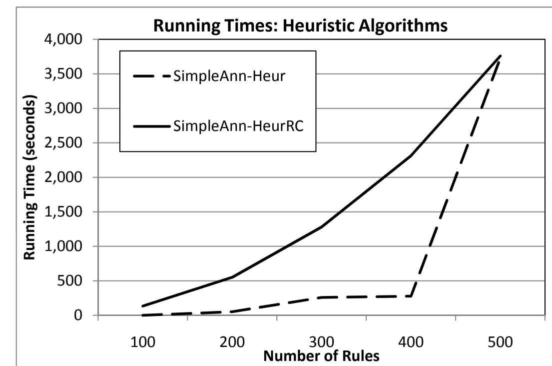


Figure 6.14: Varying number of rules (larger *ap*-programs); 25 ground action atoms, 5 state atoms, and 5 reachability constraints

reachability constraints, number and size of clauses in rule heads and reachability constraints, number of rules, and number and size of clauses in goals. Due to the vast number of parameters, we chose to vary a selection of them for the purposes of this study; since these experiments were designed to show the effects of varying certain parameters, those that were not varied in each case were kept at low values to simplify the presentation of the results (see, for instance, the number of ground state atoms in Figure 6.10, or the number of ground action atoms in Figure 6.11).

No. of State Atoms. In Figure 6.9 we show the running times of the different approaches to deciding reachability; the naive approach becomes intractable very quickly, while the (still exact) SAT-based algorithm approach has negligible cost for these runs.

No. of Action Atoms. Figure 6.10 shows the effect of varying the number of action atoms on the running times of the different approaches to solving the rule selection problem. Again we see how SimpleAnn is only slightly better than naive since conflicting rules did not arise in the randomly generated programs. The algorithms applying the (sound but not complete) heuristics exhibit a much lower running time, though are clearly affected by the increase in number of atoms due to the difficulty of satisfiability and entailment checks.

No. of Rules. Figure 6.11 reports the running times of the *SimpleAnn* rule selection algorithms, where *SimpleAnn-Heur* refers to the heuristic applied by algorithm SimpleAnn. We can see that both the naive approach and SimpleAnn quickly become intractable as the number of rules in the input program increases. For SimpleAnn, this is because the randomly generated programs do not provide it with the opportunity to apply its enhancements over the naive approach, in particular dismissing conflicting rules. To show the effect of the presence of this kind of rules, we ran another series of experiments in which a certain percentage of the rules in the input program were forced to be in probabilistic conflict with the goal; the results are shown in Figures 6.12 and 6.13. The former shows the same experiment as Figure 6.11 but with (rounded) 10% of the rules forced to be in conflict, while the latter shows the effect of increasing this percentage for programs of 10 rules. Both figures show how SimpleAnn leverages the presence of these rules, greatly reducing its running time w.r.t. that of the naive algorithm.

The last set of experiments are presented in Figure 6.14, which shows the running times for the SimpleAnn heuristic step (that is, assuming the algorithm only tries to apply the heuristic and returns *false* otherwise) and the SimpleAnn-HeurRC algorithm for larger programs. It is interesting to see the different shapes of the curves: as programs get larger, the SAT formulas associated with SimpleAnn-HeurRC become larger as well, leading to the gradual increase in the running time; on the other hand, we can see that the strategy of only focusing on certain "heuristic rules" pays off for the SimpleAnn heuristic step, but there is a spike in running time when the size grows from 400 to 500 rules. This is likely due to the appearance of more such rules, which means that the algorithm has many more subprograms to verify.

Finally, we would like to point out that all runs reported a percentage of false negatives of at most 20% for the heuristic algorithms (false positives are not possible because they are sound algorithms), and were close to zero in many cases. In future work we will extend this experimental study to investigate which parameters have the most influence over the precision of our heuristics.

6.5 Concluding Remarks

To the best of our knowledge, this is the first effort that tackles the problem of abductive reasoning in probabilistic logic programming under no independence assumptions, in the tradition of the works of [NS92] for probabilistic logic programming, and [Hai84], [Nil86], and [FHM90] for probabilistic logic in general. As we are adopting the class of *action* probabilistic logic programs from [KMN^{+}07a], it is natural to consider abductive reasoning with respect to *goals* instead of observations

(as is done sometimes when the logic programming perspective of abductive inference is adopted). In Chapter 3, we discuss the work in the literature that is most closely related to our approach.

In the next chapter, we will focus on a generalization of BAQA, which we call *cost-based abductive query answering* (or CBQA). The generalization arises from the fact that we are no longer interested in finding a reachable state for which the goal is entailed, but rather a reachable state for which the associated path from the current state has *minimal cost*. Chapter 7 will focus on the set of related issues that arise, such as how cost can be defined, and how this generalization can be used to take into account the adversary's reactions to the reasoning agent's attempts at changing the environment.

Chapter 7

Abductive Inference Taking the Adversary into Account

In this chapter, we will expand on the basic query answering problem described in Chapter 6 by assuming that there are costs associated with *transforming the current state* into another state, and also an associated probability of success of this transformation; *e.g.*, the fact that we may try to reduce foreign state political support for a certain group may only succeed with some probability. Furthermore, we are interested in associating a *reward* with each state; such a reward can be defined in many ways, but we will focus on associating a value that is related to the probabilities of certain actions being taken by the adversary (where actions undesired by the reasoning agent ensue lower rewards). To model this, we will make use of three functions, defined next:

Definition 23. *A* transition function *is any function* $T : \mathcal{S} \times \mathcal{S} \to [0, 1]$*, and a* cost function *is any function* $cost : \mathcal{S} \to [0, 1]$*. A* transition cost function*, defined w.r.t.*

a transition function T *and some cost function cost, is a function* $cost_T : \mathcal{S} \times \mathcal{S} \rightarrow [0, \infty)$*, with* $cost_T(s, s') = \frac{cost(s')}{T(s,s')}$ *whenever* $T(s, s') \neq 0$*, and* ∞ *otherwise*[1].

Example 31. *Suppose that the only state predicate symbols are those that appear in the rules of Figure 6.1 (Page 129), and consider the set of states in Figure 6.2. Then, an example of a transition function is:* $T(s_1, s_2) = 0.93$*,* $T(s_1, s_3) = 0.68$*,* $T(s_2, s_1) = 0.31$*,* $T(s_4, s_1) = 1$*,* $T(s_2, s_5) = 0$*,* $T(s_3, s_5) = 0$*, and* $T(s_i, s_j) = 0$ *for any pair* s_i, s_j *other than the ones considered above. Note that, if state* s_5 *is reachable, then the* ap-*program is inconsistent, since both rules 1 and 2 are relevant in that state.*

Function $cost_T$ describes *reachability* between any pair of states – a cost of ∞ represents an impossible transition. The cost of transforming a state s_0 into state s_n by intermediate transformations through the sequence of states $seq = \langle s_0, s_1, \ldots, s_n \rangle$ is defined:

$$cost^*_{seq}(s_0, s_n) = e^{\sum_{0 \leq i < n, s_i \in seq} cost_T(s_i, s_{i+1})} \tag{7.1}$$

One way in which cost functions can be specified is in terms of *reward functions.*

Definition 24 (Reward functions)**.** *An* action reward function *is a partial function* $R : \mathcal{APF} \rightarrow [0, 1]$*. An action reward function is* finite *if* $dom(R)$ *is finite.*

Let R *be a finite reward function and* Π *be an* ap-*program. An* entailment-based reward function *for* Π *and* R *is a function* $E_{\Pi,R} : \mathcal{S} \rightarrow [0, \infty)$*, defined as:*

$$E_{\Pi,R}(s) = \sum_{F:[\ell,u] \in dom(R) \wedge \Pi_s \models F:[\ell,u]} R(F : [\ell, u]) \tag{7.2}$$

---

[1]We assume that ∞ represents a value for which, in finite-precision arithmetic, $\frac{1}{\infty} = 0$ and $x^\infty = \infty$ when $x > 1$. The IEEE 754 floating point standard satisfies these rules.

Reward functions are used to represent how desirable it is, from the reasoning agent's point of view, for a given annotated action formula to be entailed in a given state by the model being used. *In this chapter, we will assume that all reward functions are finite.* We use this notion of reward to define a natural *canonical cost function* as $cost^{\circ}(s) = \frac{1}{E_{\Pi,R}(s)}$ when $E_{\Pi,R}(s) \neq 0$, and 1 otherwise, for each state s. In the rest of this chapter, we assume that all transition cost functions are defined in terms of a canonical cost function.

Example 32. *An example of an entailment-based reward function is as follows. Consider state s_2 from Figure 6.2, and annotated formulas $F_1 = \texttt{kidnap}(1) \wedge \texttt{tlethciv}(1) : [0, 0.60]$, $F_2 = \texttt{kidnap}(1) : [0, 0.05]$, and $F_3 = \texttt{tlethciv}(1) : [0, 0.5]$. Suppose we have action reward function R such that $R(F_1) = 0.2$, $R(F_2) = 0.54$, and $R(F_3) = 0.14$. Now, considering that $\Pi_{s_2} \models F_1$, $\Pi_{s_2} \not\models F_2$, and $\Pi_{s_2} \models F_3$, we have that, according to Equation 7.2 in Definition 24, $E_{\Pi,R}(s_2) = 0.2 + 0.14, 1 = 0.34$. Assuming $T(s_1, s_2) = 0.93$ as in Example 31, we have $cost_T(s_1, s_2) = \frac{0.34}{0.93} \approx 0.365$.*

In the next section, we will formally present the *cost-based query answering* problem.

7.1 Cost-based Query Answering (CBQA)

Given the preliminary definitions above, we can now present the main problem that will be addressed in this chapter.

Definition 25. *A* cost based query *is a 4-tuple $\langle G : [\ell, u], s, cost_T, k\rangle$, where $G : [\ell, u]$ is an* ap*-formula, $s \in \mathcal{S}$, $cost_T$ is a cost function, and $k \in \mathbb{R}^+ \cup \{0\}$.*

CBQA Problem. Given *ap*-program Π and cost-based query $\langle G : [\ell, u], s, cost_T, k\rangle$, return "Yes" if and only if there exists a state s' and sequence of states $seq =$

$\langle s, s_1, \ldots, s' \rangle$ such that $cost^*_{seq}(s, s') \leq k$, and $\Pi_{s'} \models G : [\ell, u]$; the answer is "No" otherwise.

The main difference between the BAQA problem presented above and CBQA is that in BAQA there is no notion of cost, and we are only interested in the *existence of some sequence* of states leading to a state that entails the *ap*-formula.

Example 33. *Consider once again the program in the running example and the set of states from Figure 6.2. Suppose the goal is* $\texttt{kidnap}(1) : [0, 0.6]$ *(we want the probability of Hezbollah using kidnappings to be at most* 0.6*) and the current state is* s_4*,* $k = 3$*. Suppose we have a reward function* $E_{\Pi,R}$ *such that* $E_{\Pi,R}(s_1) = 0.5$*,* $E_{\Pi,R}(s_2) = 0.15$*,* $E_{\Pi,R}(s_3) = 0.5$*,* $E_{\Pi,R}(s_4) = 0.1$*,* $E_{\Pi,R}(s_5) = 0$*, and* $E_{\Pi,R}(s_i) = 0$ *for all other* $s_i \in \mathcal{S}$*. Finally, for the sake of simplicity, suppose transition function* T *states that all transitions have probability 1.*

The states that make relevant a subprogram that entails the goal are: s_1*,* s_2*,* s_3*, and* s_5*. The objective is then to find a finite sequence of states starting at* s_4 *and finishing in any other state such that the total cost of the sequence is less than* 3 *(recall that cost is defined* $cost_T(s, s') = cost^{\circ}(s')/T(s, s')$*). We can easily see that directly moving to either state* s_1 *or* s_3 *satisfies these conditions, with a cost of* 2*; moving to* s_2 *or* s_5 *does not, since the cost would be* ≈ 6.67 *and* ∞*, respectively.*

The following proposition is a direct consequence of Proposition 14, which stated that the BAQA problem is EXPTIME-complete.

Proposition 19. *CBQA is EXPTIME-complete.*

Proof. Direct consequence of Proposition 14, by observing that BAQA is a special case of CBQA where the transition function is T is such that $T(s_i, s_j) = 1$ for any

$s_i, s_j \in \mathcal{S}$, the reward function is such that $E_{\Pi,R}(s) = 1$ for all $s \in \mathcal{S}$, and $k = \infty$ (*i.e.*, a high enough value). $\square$

This same argument allows us to show, as direct consequences of Corollary 3 and Proposition 15, that CBQA is *EXPTIME*-complete and *NP*-complete whenever the cardinality of the set of ground action atoms is bounded by a constant, and the cardinality of the set of ground state atoms is bounded by a constant, respectively. Problems P1 and P2 described on Page 134 also apply to CBQA, the only difference being that in P2 we must take the cost budget (and therefore also the transition probabilities) into account.

In the next sections we will investigate algorithms for CBQA when the cost function is defined in terms of entailment-based reward functions. We will begin by presenting an exact algorithm, and then go on to investigate a more tractable approach to finding solutions, albeit not optimal ones.

7.2 An Exact Algorithm for CBQA

We show that any CBQA problem can be mapped to a *Markov Decision Process* [Bel57, Put94] problem. An instance of an MDP consists of: a finite set S of environment *states*; a finite set A of *actions*; a *transition function* $T : S \times A \to \Pi(S)$ specifying the probability of arriving at every possible state given that a certain action is taken in a given state; and a *reward function* $R : S \times A \to \mathbb{R}$ specifying the expected immediate reward gained by taking an action in a state. The objective is to compute a policy $\pi : S \to A$ specifying what action should be taken in each state – the policy should be optimal w.r.t. the expected utility obtained from executing it.

Obtaining an MDP from the Specification of a CBQA Instance. We show how any instance of a CBQA problem can be mapped to an MDP in such a way that an optimal policy for this MDP corresponds to solutions to the original CBQA problem.

State Space: The set S_{MDP} of MDP states corresponds directly to the set $\mathcal{S}$.

Actions: The set A_{MDP} of possible actions in the MDP domain corresponds to the set of all possible attempts at changing the current state. We can think of the set of actions as containing one action per state in $s \in \mathcal{S}$, which represents the change from the current state to s. We will therefore say that action a specifying that the state will be changed to s is *congruent* with s, denoted $a \cong s$.

Transition Function: The transition function T_{MDP} for the MDP can be directly obtained from the transition function T in the CBQA instance. Formally, let $s, s' \in S_{MDP}$ and $a \in A_{MDP}$; we define:

$$T_{MDP}(s,a,s') = \begin{cases} 0 & \text{if } a \ncong s', \\ T(s,s') & \text{otherwise;} \end{cases} \tag{7.3}$$

$$T_{MDP}(s,a,s) = 1 - T(s,a,s') \text{ for } a \cong s'; \tag{7.4}$$

the last case represents the fact that, when actions fail to have the desired effect, the current state is unchanged.

Reward Function: The reward function of the MDP, which describes the reward directly obtained from performing action $a \in A$ in state $s \in S$, can also be directly obtained from the CBQA instance. Let $s \in S_{MDP}$, $a \in A_{MDP}$, Π be an *ap*-program,

$G : [\ell, u]$ be the goal, and $E_{\Pi,R}$ be an entailment-based reward function:

$$R(s,a) = \begin{cases} -1 * cost_T(s,s') & \text{for state } s' \in \mathcal{S} \text{ such that } a \cong s', \\ \\ 1 & \text{for states } s' \in \mathcal{S} \text{ such that } \Pi_{s'} \models G : [\ell, u]. \end{cases} \tag{7.5}$$

To conclude, we present the following results. The first states that given an instance of CBQA, our proposed translation into an MDP is such that an optimal policy under Maximum Expected Utility (MEU) for such an MDP expresses a solution for the original instance. In the following, we say that a sequence of states $\langle s_0, s_1, \ldots, s_k \rangle$ is the result of *following* a policy π if $\pi(s_i) = a_{i+1}$, where $0 \leq i < k$ and $a_{i+1} \cong s_{i+1}$.

Proposition 20. *Let $O = (\Pi, \mathcal{S}, s_0, G : [\ell, u], cost, T, E_{\Pi,R}, k)$ be a CBQA problem instance that has a solution (output "Yes"), and $M = (S_{MDP}, A_{MDP}, T_{MDP}, R_{MDP})$ be its corresponding translation into an MDP. If π is a policy for M that is optimal w.r.t. the MEU criterion, then following π starting at state $s_0 \in S_{MDP}$ yields a sequence of states that satisfies the conditions for a solution to O.*

Proof. By hypothesis we have that π is MEU-optimal, which means that

$$\pi(s) = \arg\max_a \left(R_{MDP}(s,a) + \max_{a'} \left(\sum_{s' \in S} T_{MDP}(s,a,s') \cdot Q(s',a') \right) \right) \tag{7.6}$$

where Q is the action utility function defined as usual:

$$Q(s,a) = R_{MDP}(s,a) + \max_{a'} \left(\sum_{s' \in S} T_{MDP}(s,a,s') \cdot Q(s',a') \right)$$

By hypothesis, we have that the answer to instance O is "Yes", meaning that there exists a sequence $seq = \langle s_0, \ldots, s_n \rangle$ such that $cost^*_{seq}(s_0, s_n) \leq k$. We will prove, by induction on the length of seq, that the theorem holds.

Base case: For $|seq| = 2$, $\pi(s_0)$ must correspond to an action that takes us directly to state s' satisfying the entailment condition. Furthermore, by definition of MEU policy, it must be the action that maximizes the reward function defined in Equation 7.5. By hypothesis, it must be the case that $cost^*_{seq}(s_0, s') \leq k$; the theorem therefore holds.

Inductive step: Assume that the theorem holds whenever solution seq is such that $|seq| = k$, for some $k \in \mathbb{N}, k > 2$; we must then prove that it also holds whenever $|seq| = k+1$. Consider the set S'_0 comprised of states s'_0 such that $T(s_0, s'_0) \neq 0$ and $cost^*_{seq}(s_0, s'_0) \leq k$. Then, since by hypothesis we know that there exists a solution to O of length $k+1$, there must exist a solution of length k to some instance

$$O' = \left(\Pi, \mathcal{S} - \{s_0\}, s'_0, G : [\ell, u], cost, T, E_{\Pi,R}, k - cost^*_{seq}(s_0, s'_0)\right),$$

for some $s'_0 \in S'_0$. By the inductive hypothesis, the theorem is satisfied for O', meaning that the MEU optimal policy π for O is defined for all states in $\mathcal{S} - \{s_0\}$. Now, $\pi(s_0)$ will correspond to the action with the highest reward; clearly, the action that corresponds to state s'_0 from O' satisfies this property. □

Second, we analyze the computational cost of taking this approach. As there are numerous algorithms to solve MDPs, we only analyze the size of the MDP resulting from the translation of an instance of CBQA. The well-known Value Iteration algorithm [Bel57] iterates over the entire state space a number of times that is polynomial in $|S|$, $|A|$, β, and B, where B is an upper bound on the number of bits that are needed to represent any numerator or denominator of β [Lit96]. Now, each iteration takes time in $O(|A| \cdot |\mathcal{S}|^2)$, which is equivalent to $O(|\mathcal{S}|^3)$ since $|A| = |\mathcal{S}|$;

this means that only for very small instances will solving the corresponding MDP be feasible.

As can be seen from the above mapping, the *key point in which our problem differs* from approaches like planning under uncertainty is that finding a sequence of states that is a solution to CBQA involves executing actions in parallel which, among other things, means that the number of possible actions that can be considered in a given state is *very large*. This makes planning approaches infeasible since their computational cost is intimately tied to the number of possible actions in the domain (generally assumed to be fixed at a relatively small number). In the case of MDPs, even though state aggregation techniques have been investigated to keep the number of states being considered manageable [BDG00, TvR96], similar techniques for *action aggregation* have not been developed.

7.3 A Heuristic Algorithm based on Iterative Sampling of Solutions

Given the exponential search space, we would like to find a tractable heuristic approach. We now show how this can be done by developing an algorithm in the class of *iterated density estimation* algorithms (IDEAs) [BJV96, PGL02]. The main idea behind these algorithms is to improve on other approaches such as Hill Climbing, Simulated Annealing, and Genetic Algorithms by maintaining a *probabilistic model* characterizing the best solutions found so far. An iteration then proceeds by (1) generating new candidate solutions using the current model, (2) singling out the best out of the new samples, and (3) updating the model with the samples from Step 2. One of the main advantages of these algorithms over classical approaches

Algorithm 15: DE_CBQA($\Pi, G : [\ell, u], s_0, T, h, k, numIter, giveUp$)
1. $\mathcal{S}_G$:= $getGoalStates(\Pi, G : [\ell, u])$;
2. test all transitions (s_0, s_G), for $s_G \in \mathcal{S}_G$; calculate $cost^*_{seq}(s_0, s_G)$ for each;
3. let ϕ_{best} be the two-state sequence that has the lowest cost, denoted c_{best};
4. let $\mathcal{S}' = \mathcal{S} - \mathcal{S}_G - \{s_0\}$; set $j := 2$;
5. $P :=$ new uniform probability distribution over $sequences(\mathcal{S}')$;
6. while !*giveUp* do
7. $j := j + 1$;
8. for $i = 1$ to *numIter* do
9. randomly sample (using P) a set H of h sequences of states of length j starting at s_0 and ending at some $s_G \in \mathcal{S}_G$;
10. rank each sequence ϕ with $cost^*_{seq}(s_0, \phi(j))$;
11. pick the sequence in H with the lowest cost c^*, call it ϕ^*;
12. if $c^* < c_{best}$ then ϕ_{best}:= ϕ^*; c_{best}:= c^*;
13. P:= generate new distribution based on H;
14. return ϕ_{best};

Figure 7.1: An algorithm for CBQA based on probability density estimation.

is that the probabilistic model, a "byproduct" of the effort to find an optimum, contains a wealth of information about the problem at hand.

Algorithm DE_CBQA (Figure 7.1) follows this approach to finding a solution to our problem. The algorithm begins by identifying certain *goal states*, which are states s' such that $\Pi_{s'} \models G : [\ell, u]$; these states are pivotal, since any sequence of states from s_0 to a goal state is a candidate solution. The algorithms in Section 6.2 can be used to compute a set of goal states. Continuing with the preparation phase, the algorithm then tests how good the direct transitions from the initial state s_0 to each of the goal states is; ϕ^* now represents the current best sequence (though it might not actually be a solution). The final step before the sampling begins occurs in line 5, where we initialize a probability distribution over all states[2], starting out as the uniform distribution.

[2]In an actual implementation, the probability distribution should be represented implicitly, as storing a probability for an exponential number of states would be intractable.

The *getGoalStates* function called in line 1 performs two tasks: first, it identifies subprograms Π' of Π such that $\Pi' \models G : [\ell, u]$; second, it identifies states s such that $\Pi_s = \Pi'$, for some Π' found in the first step. All such states are then labeled as *goal states*, since any sequence of states from s_0 to any goal state is a candidate solution. The algorithms developed in Section 6.2 can be used to compute a set of goal states.

The `while` loop in lines 6-13 then performs the main search; *giveUp* is a predicate given by parameter which simply tells us when the algorithm should stop (it can be based on total number of samples, time elapsed, etc). The value j represents the length of the sequence of states currently considered, and *numIter* is a parameter indicating how many iterations we wish to perform for each length. Line 9 performs the sampling of sequences, while line 10 assigns a score to each based on the transition cost function. After updating the score of the best solution found up to now, line 13 updates the probabilistic model P being used by keeping only the best solutions found during the last sampling phase. The algorithm finally returns the best solution it found (if any). An attractive feature of DE_CBQA is that it is an anytime algorithm, *i.e.*, once it finds a solution, given more time it may be able to refine it into a better one while always being able to return the best so far. We now show an example of this algorithm at work.

Example 34. *Consider once again the* ap*-program from Figure 6.1, and the states from Figure 6.2. Suppose that we have the following inputs. The goal is* `kidnap`$(1) : [0, 0.6]$*; the transition probabilities are as follows:* $T(s_4, s_1) = 0.1$, $T(s_4, s_2) = 0.1$, $T(s_4, s_3) = 0.1$, $T(s_2, s_1) = 0.9$, $T(s_3, s_2) = 0.8$, $T(s_5, s_2) = 0.9$, $T(s_5, s_3) = 0.2$, $T(s_5, s_1) = 0.3$, $T(s_1, s_3) = 0.01$, *and* $T(s_i, s_j) = 1$ *for any pair of states* s_i, s_j *not previously mentioned; the initial state is* s_4*; the reward function* $E_{\Pi,R}$ *is defined as*

follows: $E_{\Pi,R}(s_1) = 0.5$, $E_{\Pi,R}(s_2) = 0.15$, $E_{\Pi,R}(s_3) = 0.5$, $E_{\Pi,R}(s_4) = 0.1$, *and* $E_{\Pi,R}(s_5) = 0.7$*; giveUp is a predicate that simply checks if we've sampled a total of 5 or more sequences; numIter* $= 2$*;* $h = 3$*; and* $k = 1,000$*.*

The three states that make relevant a subprogram that entails the goal are s_1, s_2*, and* s_3*. The costs of the two-state direct sequences are the following:* $cost_{seq}(s_4, s_1) \approx 10^{8.68}$, $cost_{seq}(s_4, s_2) \approx 10^{28.9}$*, and* $cost_{seq}(s_4, s_3) \approx 10^{8.68}$*; therefore,* $c_{best} = 10^{8.68}$ *and* $\phi_{best} = \langle s_4, s_3 \rangle$*. Next, since we are assuming that* s_1*-*s_5 *are the only states for the sake of brevity, the algorithm sets up a probability distribution* P *represented as a distribution over the set of states that starts out as* $(0.2, 0.2, 0.2, 0.2, 0.2)$*. Suppose we sample* $H = \{\langle s_4, s_5, s_3 \rangle, \langle s_4, s_5, s_2 \rangle, \langle s_4, s_1, s_3 \rangle\}$*. These sequences have respective costs of* $10^{9.23}$, $10^{3.21}$*, and* $10^{21.71}$*. The update step in line 13 of the algorithm will then look at the two best sequences in* H *and, depending on how it is implemented, might update* P *to* $(0.1, 0.1, 0.1, 0, 0.7)$*. Thus, the algorithm has learned that* s_4, s_5 *seems to be a good way to start. For brevity, suppose that the next iteration of samples (the last one according to giveUp) contains* $\langle s_4, s_5, s_1 \rangle$*, whose cost is* $\approx 10^{2.89}$*; it is the best seen so far, and since* $10^{2.89} < k$*, it is a valid answer.*

In Section 7.4, we present the results of our experimental evaluation of this algorithm, comparing it first to an exact solver and then investigating its scalability.

7.4 Experimental Results

In this section, we will report on a series of experimental evaluations that we carried out on the algorithms presented in Sections 6.2 and 7.1. Due to the vast number of possible parameters in these algorithms, we chose to vary a subset of them for the purposes of this study.

A note about state space size. As with ground action atoms and worlds, the number of possible states grows exponentially with the number of ground state atoms. However, the situation is made worse in the case of states since the cardinality of this set influences the number of possible *state transitions*, and therefore also the number of *sequences of states*, which is basically the search space of the problem at hand[3]. For n ground state atoms, we have 2^n states, 2^{2n} state transitions, and $\binom{2^{2n}}{k}$ possible sequences of length k without repetition. Thus, for 10 ground state atoms we have $1,024$ states, around 1 million possible state transitions, and about 10^{18} possible sequences of length 3! This number rapidly grows to about 10^{36} for 13 ground state atoms and sequences of length 5.

We carried out all experiments on an Intel Core2 Q6600 processor running at 2.4GHz with 8GB of memory available, using code written in Java 1.6; all runs were preformed on Windows 7 Ultimate 64-bit OS, and made use of a single core.

First, we compare the run time and accuracy of the MDP formulation against that of the DE_CBQA algorithm. Recall that DE_CBQA randomly selects states with respect to a probability distribution that is updated from one iteration to the next. The simplest way to represent this probability distribution is with a vector of size $|\mathcal{S}|$, where the element at position i represents the proportion of "good" samples that contained state i. This representation does not scale as $|\mathcal{S}|$ increases; our implementation thus only keeps track of the states we have visited, implicitly assigning proportion 0 to all nonvisited states. As such, the required storage for the probability distribution is proportional only to the number of states visited, not the entire state space.

[3]Another direct consequence of this is that the number of possible state transitions directly affects the size of the transition probability matrices, at least for explicit representations.

Second, we explore instances of CBQA that are beyond the scope of the exact MDP implementation, but within reach of the DE_CBQA heuristic algorithm. As discussed in Section 7.2, our problem assumes the agent being modeled can carry out actions in *parallel*. For realistic problem settings, this leads to a very large number of possible actions to be considered at every state, alongside an equally large number of states to consider. As such, the exact MDP algorithm runs in polynomial time with respect to an *exponential* number of actions and states, losing its tractability. To address this shortcoming, we apply the basic DE_CBQA algorithm to large problem instances and discuss how it scales in relation to increased rule and state spaces.

Finally, we explore a different representation of the probability distribution in the DE_CBQA algorithm based on a Bayesian network. We contrast the two implementations of the DE_CBQA algorithm in large problem instances and end with a discussion of "smarter" heuristics and their effects on both runtime and quality of result.

For all experiments, we assume an instance of the CBQA problem with *ap*-program Π and cost-based query $Q = \langle G : [\ell, u], s, cost_T, k \rangle$. The required cost, transition, and reward values for both algorithms are assigned randomly in accordance with their definitions. We assume an infinite budget for our experiments, choosing instead to compare the numeric costs associated with the sequences returned by the algorithms.

Exact MDP versus Heuristic DE_CBQA. Let S_{MDP} and A_{MDP} be the state and action spaces of the MDP corresponding to a given CBQA – each iteration of the Value Iteration algorithm requires $O\left(|S_{MDP}|^2 \cdot |A_{MDP}|\right)$ time. From the transformation discussed in Section 7.2, we see that $|A_{MDP}| = |S_{MDP}|$; furthermore, since $|S_{MDP}|$ is exponentially larger than the number of state atoms found in Π,

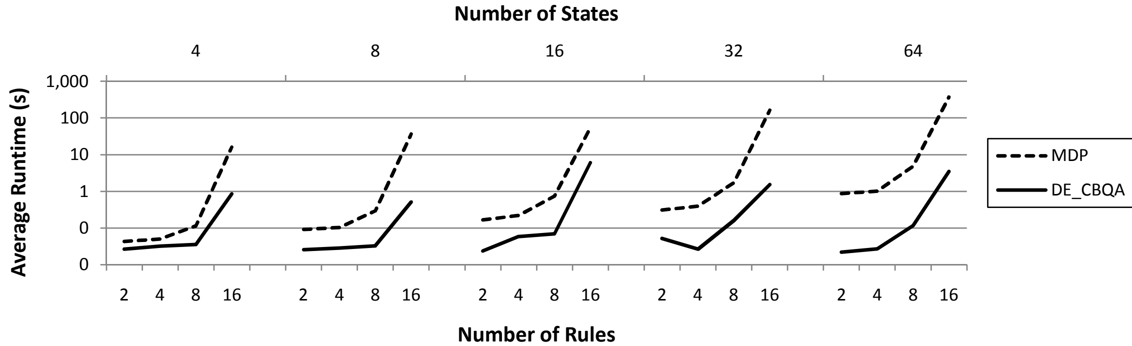


Figure 7.2: Log-scale run time comparison of MDP and DE_CBQA, shown with increasing state size (top axis) for each of 2, 4, 8, and 16 rules (bottom axis). Note the sharp jump in run time as the number of rules increases compared to the gradual upward trend as the number of states rises.

we expect running the multiple iterations of Value Iteration required to obtain an optimal policy to be intractable for all but very small instances of our problem. Our experimental results support this intuition.

For this set of experiments, we varied the number of state atoms, action atoms, and *ap*-rules in an *ap*-program Π; 10 unique *ap*-programs were created per combination of these inputs. We tested 10 randomly generated cost, transition, and reward assignments for each unique *ap*-program. Then, for each of these generations, we tested multiple runs of the MDP and DE_CBQA algorithms. We varied the discount factor γ and maximum error ϵ for the MDP[4], while exploring different completion predicates, maximum and minimum sequence lengths, and number of iterations per sequence length for DE_CBQA. We provide an overview of the results here.

Figure 7.2 compares the running time (log-scale) of both algorithms. Immediately clear is the fact that, although increasing state and rule space size slows down both algorithms, DE_CBQA consistently outperforms the standard MDP implementation. More subtle is the observation that the difference in run times between the two algorithms increases with the number of states, with DE_CBQA maintaining

[4]Given γ and ϵ, one can calculate an error threshold that guarantees an optimal policy [WB94].

nearly constant run time across small numbers of states as the MDP implementation increases noticeably. This disparity is explained at least in part by the MDP's optimality requirement; it requires an exhaustive list of *all* goal states while DE_CBQA can rely on faster heuristic search methods (see Section 6.2). As the state space increases, so too does the list of states that must be tested for entailment of the goal *ap*-formula.

We now compare the costs of sequences returned by MDP and DE_CBQA, as given by Equation 7.1. Typically, the recommended sequences' costs are close[5]; however, in rare cases, DE_CBQA performs poorly. We believe this is due to the initial probability distribution assigning mass uniformly to all states – meaning that "good" and "bad" states are equally likely to be selected, at least initially. When DE_CBQA randomly selects bad states at the start, its ability to find better, lower-cost states in future iterations is hampered. Given its low run time, one strategy for dealing with these fringe cases is executing DE_CBQA multiple times, selecting and returning the overall lowest-cost sequence over all runs. In general, increasing the number of iterations (line 8) did not affect sequence cost; however, increasing the number of samples per iteration (line 9) often resulted in a better sequence. This hints that allowing the probability mass to converge to a small number of states too quickly is not desirable, as low-cost candidates that are not immediately evident can be ignored. Furthermore, increasing the minimum and maximum sequence lengths (lines 4 and 6) did not benefit the final result.

Finally, we tried using Policy Iteration [Tse90] instead of Value Iteration to solve the MDP; however, this method was either slower than Value Iteration or, if faster, forced to use such a low discount factor γ and error limit ϵ that following the

[5] In terms of relative error, $\eta = \frac{|v - v'|}{|v|}$, for true cost v (MDP) and approx. cost v' (DE_CBQA).

Figure 7.3: Log-scale run time as DE_CBQA scales with respect to number of states (top axis) and number of rules (bottom axis). Note the addition of extra rules slows down algorithm execution time much more significantly than a similar increase in state space size.

resultant policy often yielded a *worse* sequence than DE_CBQA's recommendation – at a slower speed!

Scaling the Heuristic DE_CBQA Algorithm. The MDP formulation of CBQA quickly becomes intractable as Π becomes more complex. In this section, we discuss how DE_CBQA scales beyond the reach of MDP as the number of states, actions, and rules increase. In order to avoid a direct exponential blowup when increasing the number of rules, we made one small change to the algorithm: whenever no goal states are found with the fast heuristics (line 1), it fails to return an answer; *i.e.*, it takes a pessimistic approach.

Figure 7.3 compares an increase in number of states to a similar increase in number of rules; observe that the number of rules seems to have a larger effect on overall run time, with an increase in state space being less noticeable. This is due to two characteristics of our algorithm. First, the heuristic sampling strategy to find states that entail the goal formula visits every rule, but not every state. Second, once entailing states are found, the run time of the DE_CBQA algorithm is only as related to the size of the state space as its probability distribution requires.

States	*Actions*	*Rules*	*Time (s)*
4,096	2^{20}	4,096	35.817
64	$\mathbf{2^{25,600}}$	1,024	6.881
64	2^{20}	**16,384**	213.511

Figure 7.4: *Towards the limits of our current implementation.* Timing results taken by maximizing an individual parameter. The size of the state space was limited by system memory in this implementation.

For the basic probability vector variant implemented with a data structure that supports constant lookup, there is very little relation to the number of states. In our experience, real-world instances of CBQA tend to contain significantly *fewer* rules than states and actions [KMN+07a]. For these cases, DE_CBQA scales quite well.

Toward Better Sampling. The most straightforward representation of the probability distribution in the DE_CBQA algorithm is, as discussed earlier, a mapping of states to the proportion of "good" sampled sequences that contain that state. While this representation is neither memory nor computationally intensive, it ignores any subtle relationships that may exist between individual states or their ordering in the overall sequence. For instance, assume there is some state that is very desirable if and only if it is visited immediately after the initial state; otherwise, it is extremely undesirable. If DE_CBQA happens to choose this state initially, its naïve probability vector will be inclined to recommend the state equally at *all* locations in future sequences, including those that are undesirable.

It is our belief that real instances of CBQA will exist in similar worlds where states and actions are not conditionally independent; as such, it is critical to explore a more informed approach to maintaining our probability distribution. One such method is the Bayesian belief network [Pea88], a directed acyclic graph modeling conditional dependencies among random variables. In our case, each node in the net-

work structure represents a random variable covering all possible states for a single (ordered) position in the final sequence. For a given node, a state is assigned probability mass proportional to how likely it is to be included in a "good" sequence at the position associated with that node. These values are initially provided through uninformed sampling of the state space, while the structure of the final network is learned through standard machine learning techniques.

Since an exhaustive search for the optimal structure across all potential networks is superexponential in the number of variables — in our case, the length of the sequence — we use a heuristic local search algorithm to perceive graph structure. We use a slightly modified K2 search algorithm with a fixed ordering based on the sampled sequences to emphasize speed of structure learning [CH92]. Our intuition is that neighboring nodes in the sequence are more likely to affect each other than those farther away. Many other heuristic search algorithms exist, but a discussion of their merits is outside the scope of this paper.

Sampling from the network is accomplished in two steps. First, recall that a state's probability mass at a root node in our Bayesian network is related only to the proportion of "good" training sequences containing that state at a specific location. With this in mind, for every root node, we take a weighted sample from its prior probability distribution table. Second, we sample the conditional probability table of each child node with respect to the partial assignment provided by sampling its immediate parents. In this way, we provide a method for sampling a full path through the state space that takes into account conditional dependencies between states, their ordering, and position.

Intuitively, an informed sampling method should provide higher accuracy (*i.e.*, lower sequence costs) at a greater computational cost, especially in instances when

states and actions interact. To explore this intuition, we remove some of the randomness from our original testing suite by seeding desirable paths through the state space. This is accomplished by manipulating the cost and transition functions between states, yielding low costs for specific sequences of states and high costs otherwise. In this way, obvious conditional dependencies are introduced into the world.

We now compare the Bayesian method (implemented with WEKA [HFH+09]) against the initial naïve probability vector method. First, as a measure of result quality, we define the *cost decrease factor* to be the factor difference in the cost of the best sequence returned by the Bayesian method over that returned by the vector implementation. Higher cost decrease factors correspond to better relative Bayesian method performance. Figure 7.5 shows the cost decrease factor for very small amounts of seeded paths compared to different sizes of state spaces. For extremely small numbers of seeded paths, the Bayesian algorithm outperforms by roughly a factor of 2. This low number signifies similar performance to the vector method and is due to both DE_CBQA implementations missing the very few "carved" sequences in their initial sampling, before any probability distribution is constructed. The conditional network constructed from bad sampling is less useful; however, this problem can be easily solved by repetition of the algorithm.

Two trends, distinguished by the size of the state space, begin to form as we increase the number of seeded paths. When considering a larger number of seeded paths in larger state spaces, the Bayesian method shows its ability to discover dependencies in sampled sequences; however, when considering the same number of paths in a smaller state space, the Bayesian method continues to perform only slightly better than its vector counterpart. Carving too many (relative to the size of the state space) desirable paths essentially randomizes the transitions between

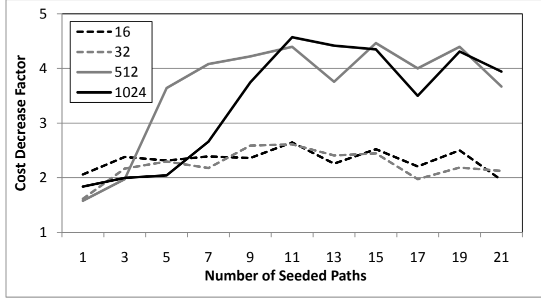


Figure 7.5: Varying the number of seeded paths with a small (*e.g.*, 16 or 32) number of states versus a larger (*e.g.*, 512 or 1024) state space.

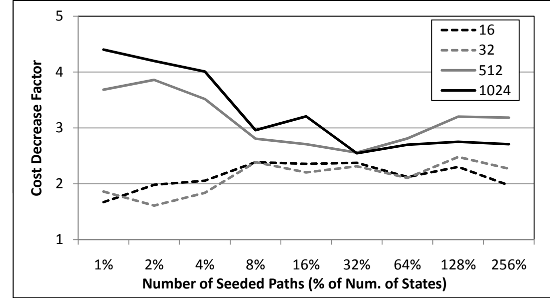


Figure 7.6: Varying the number of seeded paths (and thus the level of conditional dependence in the world) as a percentage of the total number of states.

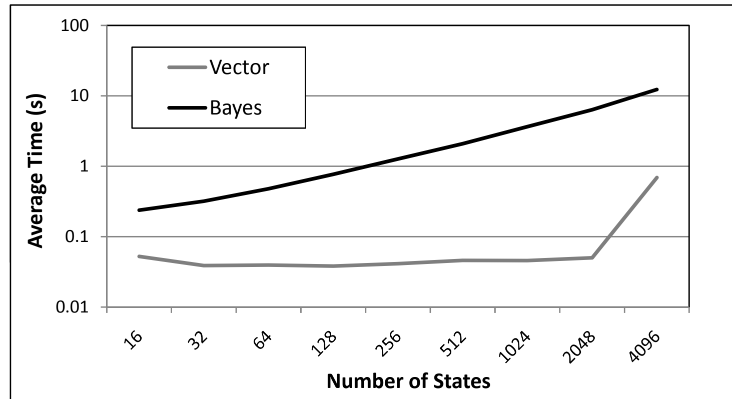


Figure 7.7: Log-scale run time as both the Bayesian and vector-based DE_CBQA algorithms scale. Note the linear increase in Bayesian runtime caused by structure learning, storage, and sampling overhead.

states; for example, 20 paths through only 16 states alters overall dependencies far more than a similar number through 1,024 states. We explore this relationship further below.

Figure 7.6 shows the quality of results as the number of seeded paths is increased significantly. We see that the Bayesian network version performs admirably in large state spaces until roughly 8%, when its performance degrades to that of the Bayesian version in a smaller state space. As in Figure 7.5, small instances of the problem stay roughly constant. Regardless of state space size, we see an increase in result quality of 2 to 3 over the naïve probability vector.

We have seen that the more informed sampling method performs well, decreasing overall sequence cost. However, as our initial intuition suggested, the increased overhead of maintaining conditional dependencies slows the DE_CBQA algorithm significantly. Figure 7.7 shows that although the memory requirements of both algorithms increase linearly in the size of the number of states sampled, the Bayesian method is consistently slower than the vector method. This is due to a similar increase in the *runtime* complexity of the Bayesian method. The vector method represents probabilities as a simple mapping of states to real numbers; as such, an implementation with a constant lookup time data structure provides extremely fast sampling with a small memory footprint. For the more informed Bayesian variant of the heuristic, this relationship is based both on the number of initial iterations over the state space prior to the formation of the sampling structure and the maximum length of a sampled sequence. The Bayesian graph has as many nodes as there are states in a sampled sequence; furthermore, each of these nodes maintains knowledge of all unique states corresponding to a particular position in the sequence. Learning the structure of the network, storing the graph, and sampling from it are all dependent on the number of sampled states and sequence length. Thankfully, we can apply reasonable bounds to the number of samples, opting instead to instantiate multiple Bayesian networks over a smaller sample set.

When we include the additional cost of searching for entailing goal states (Line 1 of the DE_CBQA algorithm), both the naïve probability vector and informed Bayesian network methods scale similarly. We use the same fail-fast pessimistic approach to the heuristic goal search described earlier. Figure 7.8 shows how both algorithms scale with respect to an increase in number of states and number of rules. As before, the number of rules has a significantly higher effect

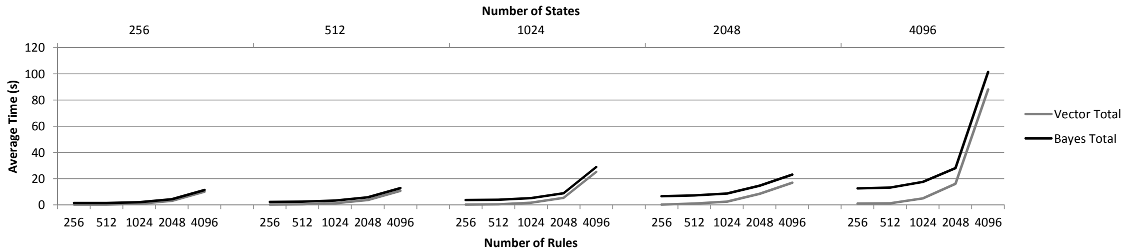


Figure 7.8: Run time comparison as DE_CBQA scales with respect to number of states (top axis) and number of rules (bottom axis). Note the similarity in run time between the Bayesian and vector probability models.

on overall runtime than the number of states. We see that the algorithm scales gracefully to large state/action spaces. As we mentioned above, in our experience, real-world instances of CBQA tend to contain significantly *fewer* rules than states and actions [KMN+07a]; as such, in these cases DE_CBQA scales quite well.

7.5 Concluding Remarks

In this chapter, we introduced the Cost-based Query Answering Problem (CBQA), and show that computing an optimal solution to this problem is computationally intractable, both in theory and in practice. We then propose a heuristic algorithm (DE_CBQA) based on iterative random sampling and show experimentally that it provides comparably accurate solutions in significantly less time. Finally, we show that DE_CBQA scales to very large problem sizes. The main goal of generalizing the BAQA problem from Chapter 6 in this way was to be able to take into account how the adversary responds in the intermediate states that must be traversed in order to reach the final state that satisfies the reasoning agent's probabilistic goal.

Chapter 8

Advanced Applications of *ap*-programs: Taking Promise Fulfillment into Account

The main problems addressed so far have been about computing the most probable sets of actions given a situation (Chapters 4 and 5), and reasoning about how the current environment can be changed in such a way that a given ground action probabilistic formula (*i.e.*, a boolean combination of concrete actions and an associated probability interval) is entailed in the new environment (Chapters 6 and 7). As we discussed in Chapter 2 when presenting the basic syntax and semantics of *ap*-programs, in all of these problems we assume that the formalism has been set up in such a way that predicates can refer to either actions taken by the agent being modeled or the environment, and that all rules been learned with respect to this setup.

In this chapter, we will study how the *ap*-program framework can be set up when certain special considerations need to be made. In particular, this chapter will provide a mechanism by which *ap*-programs can include state and action atoms as part of rules that refer to *promises made between agents*. For instance, based on

past performance of agent A on a certain kind of promise, agent C may derive the following rules:

$$\begin{aligned} &\mathit{fulfill}(P, \mathit{medium}) : [0.85, 0.92] \leftarrow \\ &\qquad (P = \mathit{promise}(A, C, fly(\mathit{From}, \mathit{To}, \mathit{DepTime}), \mathit{ArrTime})) \wedge \\ &\qquad \mathit{inNE(From)} \wedge \mathit{inMidwest(To)} \wedge \mathit{holidaySeason(DepTime)}. \end{aligned}$$

This rule intuitively says that if A made a promise to C to fly from a certain airport *From* in the Northeast to airport *To* in the Midwest, departing at time *DepTime* which is during the holiday season (we assume the time includes the date as well) and arriving at time *ArrTime*, then it is likely that the promise will be fulfilled only to a medium degree (the probability of this is between 0.85 and 0.92). Note that for this example we are assuming that degrees of fulfillment are categorized into *buckets* such as *low*, *medium*, and *high*. Even though this is not necessary, it will most likely be advantageous to reason about degrees of fulfillment at a higher level than the real numbers that are likely to occur in raw data, since the reasoning agent may only care about a few different levels of fulfillment. The user can of course define as many buckets as he sees fit for the application at hand. These rules can be derived in a fairly straightforward manner using standard learning algorithms. If an agent has access to historic records of how others have fulfilled their promises, it can derive this sort of rules and use them during negotiations to decide if it is dealing with a trustworthy party or not, or to perform more complex reasoning such as expected payoff of a given deal.

We will begin by presenting a motivating example in the next section, which we will refer to throughout this chapter.

8.1 Introduction and Motivating Example

Politicians and political parties are prime examples of people and organizations that make and break promises . Conference and journal reviewers often promise to review papers by a deadline, but may not meet the deadline, may only review part of their assigned load, or not do it at all. Airlines promise to deliver passengers and their bags by a deadline, but may miss the deadline altogether. Suppliers to manufacturing plants and/or to retail outlets make promises about when inventory and/or supplies will be delivered, but may meet their promises partially or completely.

The goal of this chapter is to develop a formal theory to *quantitatively* evaluate how well an agent has fulfilled its past promises and use that as a predictor of whether it will keep its current (as yet unfulfilled promises). An agent A can use the theory developed in this chapter to assess the likelihood that an agent B will fulfill a given promise. Our framework takes into account three important factors not considered before: *partial fulfillment* of a promise is taken into account, as is *late fulfillment*, fulfillment of a promise that is *similar to, but not identical* to the promise that was made, and combinations thereof. The toy example below, called the Store example, is used throughout the chapter.

Example 35. *Consider a Store agent and a Supplier agent. The supplier provides, among other things, shirts and balls to the store. The supplier promises to deliver 10 blue balls by time t_1, 5 green balls by time t_2, and 15 green shirts by time t_3. Here are some possible scenarios.*

(S1) He delivers 7 blue balls at time t_1 and 3 blue balls separately at time t_1.

(S2) He delivers 7 blue balls at time t_1 and 3 blue balls at time $t_1 + 1$.

(S3) He delivers 7 blue balls at time t_1, 2 at time $t_1 + 1$ and 1 at time $t_1 + 2$.

(S4) He delivers 10 blue balls at time $t_1 - 3$.

Most readers will agree that the degree of fulfillment of the supplier's promise in S1 exceeds that in S2 which in turn exceeds that in case S3. However, case S4 is less clear. Should early delivery be penalized? For example, a grocery store may want just in time *delivery as there are storage costs involved. An early delivery of fish by one supplier may cause the fish to rot if sufficient refrigeration is not available when the delivery occurs.*

In this chapter, we make the following contributions. First, we define a formal syntax for expressing promises and actions that an agent might take to keep those promises. These include promises and actions with a numerical component. We then define distance measures between actions (with the same action symbol), followed by distance measures between sets of promises and actions. Rather than define distance measures directly, we develop *axioms* that such distance measures should satisfy and we then show some example distance measures that satisfy the axioms.

We then define the concept of an *enactment mapping*, which maps actions taken by an agent to the promises that those actions were intended to contribute towards. Given an enactment mapping, we can define a degree of fulfillment of a promise w.r.t. the enactment mapping. When the enactment mapping is not known, finding the enactment mapping that maximizes the degree of fulfillment is shown to be computationally intractable. Fortunately, one can get around this complexity result easily as long as the agent taking an action specifies which promise the action is supposed to contribute towards. We prove various desirable properties of our notion of fulfillment and show that when the enactment mapping is fixed, it leads to an incremental way of updating the degree of fulfillment when new actions are performed, *i.e.*, without having to process past actions all over again. We then derive

specific methods to estimate the likelihood that a given promise will be fulfilled in the future.

In order to test our methods, we developed a prototype implementation of our system and tested it out on real US airline data where the promises pertain to on-time flight departures and arrivals. We show that our estimation methods have strong predictive power. We used our algorithms to predict how well airlines would perform (in terms of on-time flight arrivals) in 2007 based on previous years' data. Our predictions were highly accurate and took small amounts of compute time.

There is some past work on developing models of trust in agent systems. [SD07a] presents a model of decision making based on trust in simple *Offer, Accept, Reject* negotiations. Decision-making in this model integrates the utilitarian, information, and semantic views of the exchange of information, and the authors present summary measures that generalize trust, reliability, and reputation as an illustration of the model's capabilities. However, promises of the kind we discuss in this chapter are not considered. Another important difference with our approach is that these measures assume the availability of probability distributions that describe the *ideal* enactments with respect to a given commitment, expected enactments, a more general *semantic similarity* measure that allows to gauge the similarity between the commitment and its actual enactment, and a measure of how much uncertainty we expect to have given a certain commitment. Other related work is that in the area of trust and reputation in agent systems (which can include both artificial and human agents). [DB07] propose a generic method of selecting evidence that is recognized as support for trust, while [Del06] provides a recent survey of the area, focusing on Internet-based mechanisms (such as for online auctions). Game-theoretic treatments of this topic have also been developed, such as in [EFL02], but this approach

has been criticized for placing too much importance on probability while underestimating its cognitive aspects, such as in [FC01]. In this respect, our work takes a step in this direction by allowing agents to influence the measure of fulfillment according to their own preferences.

In contrast to this past work, we focus on developing a general model of promises that tries to quantitatively assess how well an agent has met its past commitments, taking into account the fact that time plays a role in whether a promise is met or not (promises often involve doing things by a deadline), that the "content" of an action sometimes (but not always) allows a promise to be replaced by a similar, but different promise (*e.g.*, delivering red balls instead of green balls), and partial fulfillment where part of a promise is kept. Our framework is one of the first to develop a unified theory around these important concepts.

8.2 Preliminaries

We start by defining the notion of *temporal expression*, which is used to denote time points: we assume that time in our model is discrete.

Definition 26 (from [DKS06])**.** *(1) Every integer is a temporal expression. (2) t_{now} is a temporal expression. (3) If t_1 and t_2 are temporal expressions, then so is $(t_1 + t_2)$.*

Symbol t_{now} represents the current time point; we assume its value gets automatically updated as time goes by in the environment.

We assume the existence of a logical alphabet that consists of a finite set $\mathcal{L}$ of constant symbols, a finite set $\mathcal{A}$ of *action symbols* (each with an associated arity),

the predicate symbols *Do* and *Promise*, and an infinite set $\mathcal{V}$ of variable symbols. A constant or variable symbol is called a *term*.

Definition 27 (action atom)**.** *If $\alpha \in \mathcal{A}$, and $t_1, ..., t_m$ $(m \geq 0)$ are terms (resp. constants), then $\alpha(t_1, ..., t_m)$ is called an* action atom *(resp. ground action atom).*

$atoms(\alpha)$ denotes the set of all possible action atoms of the form $\alpha(\ldots)$.

Definition 28 (*Do* and *Promise* atoms)**.** *Suppose A, B are agents, T is a temporal expression, and X is an action atom. Then, Do(A,B,X,T) and Promise(A,B,X,T) are called Do and Promise atoms respectively.*

Intuitively, *Do*(A,B,X,T) is read "agent A *does* X for agent B at time T" while *Promise*(A,B,X,T) is read "agent A *promised* agent B that it would do X at time T". The following example, based on the Store example above, is presented in order to illustrate these concepts.

Example 36. *Let A and B be the Supplier and Store agents, Here are some example Promise and Do atoms involving the action del(it, col, am) which states that am amount of item it of color col are delivered.*

$P_1 = Promise(B, A, del(ball, blue, 10), T_1)$,

$P_2 = Promise(B, A, del(shirt, green, 5), T_2)$,

$P_3 = Promise(B, A, del(shirt, green, 15), T_3)$,

$D_1 = Do(B, A, del(ball, blue, 7), T_1)$,

$D_2 = Do(B, A, del(ball, darkBlue, 3), T_1)$,

$D_3 = Do(B, A, del(shirt, green, 20), T_3)$

However, it might be the case that the Store agent is neutral about whether the supplier delivers blue balls or green balls, even though the supplier promised

blue balls. We therefore need a binary *replaceability* relation on action atoms in order to capture this type of situation.

Definition 29. *Let A be an agent, and $S_1 = \{\alpha_1(\vec{t_1}), ..., \alpha_l(\vec{t_l})\}$ and $S_2 = \{\beta_1(\vec{u_1}), ..., \beta_m(\vec{u_m})\}$ be two sets of action atoms. We assume each agent A has an associated relation of replaceability, denoted $S_1 \rightleftharpoons^A S_2$, read as: S_1* is replaceable by S_2 for agent A. *We only require $S \rightleftharpoons^A S$ for any set S and agent A.*

When the agent is clear from context, we will simply write $S_1 \rightleftharpoons S_2$. The above definition allows us to consider a promise to be fulfilled when the agent has taken an action that is considered good enough, even though it does not exactly fulfill the promise as stated. The store manager who thinks it is all right to replace blue balls with green ones may set $\{del(ball, blue, N_1), \ldots, del(ball, blue, N_m)\} \rightleftharpoons \{del(ball, green, M_1), \ldots, del(ball, green, M_k)\}$ iff $N_1 + \ldots + N_m = M_1 + \ldots + M_k$.[1] One reason we need the $\rightleftharpoons$ relation is because a Supplier might have made multiple promises (of 5 green balls and 3 green balls all to be delivered at time 7) and may execute multiple *Do* actions (*e.g.*, by delivering two packages each of 4 green balls at time 7) that jointly meet the promises. In order to reason about this kind of situation, we need ways of aggregating promises together. We start by defining two sets. Given agents A, B, and a temporal expression T:

- $\mathcal{U}^{Prom}_{A,B,T} = \{\langle Promise(A, B, a_i, T), \omega_i\rangle \mid 0 \leq \omega_i \leq 1$ and $Promise(A, B, a_i, T)$ is a promise atom $\}$. ω_i is any real number in the $[0, 1]$ interval called the *proportion component*.
- $\mathcal{U}^{Do}_{A,B,T} = \{Do(A, B, a_i, T) \mid a_i$ is an action atom $\}$.

[1] We do not provide an explicit syntax to express the $\rightleftharpoons$ relation here; we will address this in future work.

The following definition specifies whether it is possible to merge multiple *Promise* atoms or *Do* atoms into one.

Definition 30. *Let A and B be agents, T be a temporal expression, $S_p \subseteq \mathcal{U}^{Prom}_{A,B,T}$, and $S_d \subseteq \mathcal{U}^{Do}_{A,B,T}$.*

- *Suppose a^* is an action atom such that $\{a_i \mid \langle Promise(A,B,a_i,T),\omega_i\rangle \in S_p\} \fallingdotseq \{a^*\}$. The* promise composition *operator χ takes any subset $S_p \subseteq \mathcal{U}^{Prom}_{A,B,T}$ as input and returns the Promise atom $\chi(S_p) = Promise(A,B,a^*,T)$ if and only if $\sum_{\langle Promise(A,B,a_i,T),\omega_i\rangle \in S_p} \omega_i = 1$. Otherwise, it is undefined.*
- *If a^* is an action atom such that $\{a_i \mid Do(A,B,a_i,T) \in S_d\} \fallingdotseq \{a^*\}$, then the* do composition *operator χ takes a set $S_d \subseteq \mathcal{U}^{Do}_{A,B,T}$ as input and returns the Do atom $\chi(S_d) = Do(A,B,a^*,T)$ if and only if $\{a_i \mid Do(A,B,a_i,T) \in S_d\} \fallingdotseq \{a^*\}$. Otherwise, it is undefined.*

We let $\chi^{-1}(X)$, be the set of all sets $S \subseteq \mathcal{U}^{Do}_{A,B,T}$ or $S \subseteq \mathcal{U}^{Promise}_{A,B,T}$ such that $\chi(S) = X$. Informally, compositions and decompositions are simply ways in which to refer to "parts" of *Promise* and *Do* atoms. In the case of *Promise* atoms, decompositions are sets of pairs that include a proportion for each atom in the set, whereas in the case of *Do* atoms, a decomposition is just a set. In contrast, compositions specify a set of *Do* atoms as input and composes them, when possible, into a single *Do* atom.

The following is an example of combinations and decompositions of *Promise* and *Do* atoms.

Example 37. *Consider the Promise atoms and Do atoms in Example 36 and let:*

$P^1_1 = Promise(B,A,del(ball,blue,7),T_1)$,

$P^2_1 = Promise(B,A,del(ball,blue,3),T_1)$,

$D_3^1 = Do(B, A, del(shirt, green, 5), T_3)$,

$D_3^2 = Do(B, A, del(shirt, green, 15), T_3)$

Now, if χ_p *is a promise composition operator and* $S_p = \{\langle P_1^1, 0.7\rangle, \langle P_1^2, 0.3\rangle\}$, *we have that* $\chi_p(S_p) = P_1$. *Similarly, if* χ_d *is a do composition operator and* $S_d = \{D_3^1, D_3^2\}$, *we have that* $\chi_d(S_d) = D_3$.

We now define *event sets* and *action histories.*

Definition 31 (event sets and action histories)**.** *An* event set *is any finite set of ground Do and Promise atoms. An* action history *is a function h from* $[0, ..., t_{now}]$ *to event sets.*

An action history describes what promises were made and what actions occurred at each time point before t_{now}. We will generally be interested in *finite* action histories, *i.e.*, where $\{t \mid h(t) \neq \emptyset\}$ is finite.

8.3 A Distance Measure between Atoms

In order to determine the degree of fulfillment between promises and actions, we will develop distance functions in three phases: first between action atoms, then between *Promise* atoms and *Do* atoms, and finally between sets of *Promise* atoms and sets of *Do* atoms. Of course, these distance functions can be defined in many ways, and so we present *axioms* governing the definition of such distance functions so that application specific knowledge can play a role in our framework.

8.3.1 Distance between Two Action Atoms

We start with distance functions on action atoms by first considering two action atoms that share the same action symbol.

Definition 32. *A* distance measure *between two action atoms* $\alpha(t_1, t_2, ..., t_n)$ *and* $\alpha(s_1, s_2, ..., s_n)$*, from the point of view of agent A is a function* $\delta^A_\alpha : atoms(\alpha) \times atoms(\alpha) \to \bar{R}^+ \cup \{0\}$*. Function* δ^A_α *must satisfy the property of Weak Identity of Indiscernibles: If* $a_1 = a_2$ *then* $\delta^A_\alpha(a_1, a_2) = 0$*.*

Note that the distance measure δ^A_α is undefined when comparing atoms with different action symbols.

Example 38. *From the point of view of the Store agent, the distance between two atoms* $deliver(i_1, c_1, q_1)$ *and* $del(i_2, c_2, q_2)$ *may be* $|q_1 - q_2|$ *if and only if* $i_1 = i_2$ *and* $c_1 = c_2$*, and some very large constant* $d \gg 0$ *otherwise, indicating that the manager considers any deviation in the product to represent a large difference.*

Note that this is not a distance metric from a mathematical point of view, since symmetry and triangle inequality are not required by the definition; as we will argue in the following, these properties are not always desirable in this framework. For instance, consider actions $a_1 = del(ball, 7)$ and $a_2 = del(ball, 10)$. Here, three extra balls were delivered and we might want to set $\delta^A_{del}(a_1, a_2) = 3$. However, we might want to set $\delta^A_{del}(a_2, a_1) > \delta^A_{del}(a_1, a_2)$ because delivering three fewer balls may be less desirable. For triangle inequality, consider an order for screws with actions $a_1 = del(screws, 5mm)$, $a_2 = del(screws, 5.2mm)$, and $a_3 = del(screws, 5.4mm)$, where the first component refers to a standard sized bag of screws and the second refers to their size. If the allotted error range of the manufacturer is 0.3mm, then $\delta^A_{del}(a_1, a_2)$ and $\delta^A_{del}(a_2, a_3)$ might be 0, but $\delta^A_{del}(a_1, a_3)$ would be strictly positive.

We now present a set of axioms that describe the desired characteristics for a measure of distance between two action atoms; in the following, sharing the same action symbol. Let $a_1 = \alpha(t_1, t_2, ..., t_n)$, $a_2 = \alpha(s_1, s_2, ..., s_n)$, and $a_3 = \alpha(r_1, r_2, ..., r_n)$ be action atoms. The following definition is required before presenting the axioms.

Definition 33. *Let* $a_1 = \alpha(t_1, t_2, ..., t_n)$ *and* $a_2 = \alpha(s_1, s_2, ..., s_n)$ *be two action atoms. The* disagreement set *of the two atoms, denoted by* $disagree(a_1, a_2)$ *is the set of all triples* (t_i, s_i, i) *such that* $t_i \neq s_i$.

We now present axioms that δ_α^A should satisfy.

Axiom A1: $\delta_\alpha^A(a_1, a_2) = 0$ iff $\{a_1\} \rightleftharpoons^A \{a_2\}$.

This axiom simply states that the distance between two actions is zero if and only if the singleton sets that contain each are replaceable from the point of view of the agent.

Axiom A2: If $disagree(a_1, a_2) \subseteq disagree(a_1, a_3)$, $a_2 \not\leftrightharpoons a_3$, and $a_1 \not\leftrightharpoons a_3$, then $\delta_\alpha^A(a_1, a_2) < \delta_\alpha^A(a_1, a_3)$.

Axiom $A2$ intuitively states that if the discordances between a_1 and a_2 are a subset of those between a_1 and a_3, and a_2 is not replaceable by a_3 (*i.e.*, the remaining differences are significant), then the distance between a_1 and a_2 is strictly smaller than that between a_1 and a_3. The following axiom deals with the case in which the remaining differences are not significant from the point of view of the agent.

Axiom A3: If $a_2 \leftrightharpoons^A a_3$, then $\delta_\alpha^A(a_1, a_2) = \delta_\alpha^A(a_1, a_3)$.

According to Axiom $A3$, the distance between an atom a_1 and two others a_2 and a_3, such that a_2 is replaceable by a_3, is the same. The following example illustrates these axioms.

Example 39. *Suppose we have* $a_1 = del(bball, blue, 50)$, $a_2 = del(vball, blue, 45)$, *and* $a_3 = del(vball, white, 45)$. *Here bball may refer to a beach ball, while vball refers to a volleyball. We then have* $disagree(a_1, a_2) = \{(bball, vball, 1), (50, 45, 3)\}$ *and* $disagree(a_1, a_3) = \{(bball, vball, 1), (blue, white, 2), (50, 45, 3)\}$, *and therefore the inclusion holds. Then, if* $a_2 \not\leftrightharpoons a_3$ *we have that the difference in color is significant and therefore* $\delta_\alpha(a_1, a_2) < \delta_\alpha(a_1, a_3)$ *according to A2. However, if the difference in*

color is not significant, which would be the case if $a_2 \leftrightharpoons a_3$*, the two distances should be equal, as stated by axiom* $A3$.

8.3.2 Distance between a *Promise* and a *Do* Atom

We now deal with the problem of measuring the distance between a single *Promise* atom and a single *Do* atom, interpreted as being the *enactment* of the promise.

Definition 34. *A* distance measure *between a Promise atom* P *of the form* $Promise(B, A, a_1, T_1)$ *and either a Do atom* D *of the form* $Do(B, A, a_2, T_2)$ *or the special constant Null, from the point of view of agent* A*, is a function* $\phi^A_\alpha(P, D) \rightarrow \bar{R}^+ \cup \{0\}$*, where* a_1 *and* a_2 *are action atoms that share the same action symbol* α*.*

When clear from context, we will simply write $\phi_\alpha(P, D)$. The *Null* constant stands for the "lack of enactment", it is a key aspect of the treatment of degree of fulfillment presented in the next section. We now present axioms that constrain the value that the degree of fulfillment function can take given the various situations.

Axiom F1: $\phi^A_\alpha(Promise(B, A, a_1, T_1), Do(B, A, a_2, T_2)) \geq \delta^A_\alpha(a_1, a_2)$

This basic axiom states that the distance between a *Promise* atom and a *Do* atom cannot be less than that between the actions they refer to.

Axiom F2: $\phi^A_\alpha(Promise(B, A, a_1, T_1), Null) = \infty$

This axiom states that the distance between a *Promise* atom and the constant *Null* is infinite.

Axiom F3: If $T_1 = T_2$ then

$$\phi^A_\alpha(Promise(B, A, a_1, T_1), Do(B, A, a_2, T_2)) = \delta^A_\alpha(a_1, a_2)$$

If the action enactment was performed at the time agreed in the promise, then the distance between the promise and the enactment must be the distance between the two actions. In particular, if the action agreed upon is replaceable by the one that performed, then the distance between the promise and its enactment must be zero, according to Axiom A1.

Two key situations, early completion and late completion, are unconstrained by the axioms. This is because different scenarios can arise, both where either of these are beneficial and detrimental to the agent to which the promise was made. For instance, while early delivery of an email is most likely harmless, a manager receiving items like fish and meat that require refrigeration earlier than expected must have the appropriate storage space (*e.g.*, refrigerator space) to store it. The following proposition presents a class of functions that satisfy all of the axioms.

Proposition 21. *Let $P = Promise(B, A, a_1, T_1)$ and $D = Do(B, A, a_2, T_2)$ be two atoms such that a_1 and a_2 have action symbol α. Any ϕ function of the form:*

$$\phi^A_\alpha(P, Null) = \infty$$

$$\phi^A_\alpha(P, D) = \begin{cases} f_\ell(T_1, T_2) + k_\ell \delta^A_\alpha(a_1, a_2) & \text{if } T_2 > T_1, \\ \delta^A_\alpha(a_1, a_2) & \text{if } T_1 = T_2, \\ f_e(T_1, T_2) + k_e \delta^A_\alpha(a_1, a_2) & \text{if } T_1 > T_2, \end{cases}$$

where $f_\ell(.)$ and $f_e(.)$ are positive real functions, and k_ℓ and k_e are constants in $R^{\geq 1}$, satisfies axioms F1, F2, and F3.

Proof. We have to show that $\phi^A_\alpha(P, D)$ satisfies all three axioms F1, F2, and F3.

Satisfaction of Axiom F1. Case 2 of the function definition, for $T_1 = T_2$, trivially satisfies this axiom by definition. For cases 1 and 3, we assume that $k_\ell, k_e \geq 1$ and get:

$$\begin{aligned} \phi_\alpha^A(P, D) &= f_{\ell/e}(T_1, T_2) + k_{\ell/e}\delta_\alpha^A(a_1, a_2) \\ &\geq k_{\ell/e}\delta_\alpha^A(a_1, a_2) \geq \delta_\alpha^A(a_1, a_2) \end{aligned}$$

since $f_\ell(T_1, T_2)$ and $f_e(T_1, T_2)$ are both positive functions.

Satisfaction of axiom F2. Satisfied trivially by definition.

Satisfaction of axiom F3. Satisfied by case 2 of the function definition. □

There are many such examples of reasonable ϕ functions that fall into this category, such as one that simply fixes the weight assigned to every time unit under or over the deadline by defining $f_\ell = \ell \times |T_2 - T_1|$, $f_e = e \times |T_2 - T_1|$, for some $e, \ell \in R^+$, and $k_\ell = k_e = 1$. Of course, functions outside this class can also be defined.

8.4 A Function to Measure Degree of Fulfillment

The ϕ function presented above is the backbone of the final measure that we will present, which allows an agent to measure the *degree of fulfillment* given a set of *Promise* atoms and a set of *Do* atoms. Before introducing this function, we need the definition of an *enactment mapping*:

Definition 35. *Let A and B be agents, and $S_p = \{P_1, \ldots, P_n\}$ and $S_d = \{D_1, \ldots, D_m\}$ be sets of Promise and Do atoms, respectively. Let*

$$\Delta_p = \bigcup_{i=1}^{|S_p|} S_i \quad \textit{and} \quad \Delta_d = \bigcup_{j=1}^{|S_d|} S_j$$

for some $S_i \in \chi^{-1}(P_i)$ and $S_j \in \chi^{-1}(D_j)$.

An enactment mapping *between S_p and S_d is defined as any $M_{S_p,S_d} : \Delta_p \to \Delta_d \cup \{Null\}$ that is quasi injective,* i.e., $\forall p_i, p_j \in S_p : M_{S_p,S_d}(p_i) = M_{S_p,S_d}(p_j) \wedge M_{S_p,S_d}(p_i) \neq Null \implies p_i = p_j$.

Intuitively, an enactment mapping associates "parts" of *Promise* atoms with "parts" of *Do* atoms (or *Null*), stating that the latter "counts towards" the former. The space of all possible such mappings is given by the different possible compositions of the Δ sets given the variation in the S sets involved. The following example shows a simple enactment mapping function:

Example 40. *Consider the set of atoms from Example 37. A possible enactment mapping M is the following:*

$M(P_1^1) = D_1,\ M(P_1^2) = D_2,$

$M(P_2) = D_3^1,\ M(P_3) = D_3^2.$

In this case, promise P_1 was split into two atoms in order to map it to D_1 and D_2, as was enactment D_3, in order to map it to P_2 and P_3.

The degree of fulfillment is then defined as:

Definition 36. *Let M be an enactment mapping between two sets S_p and S_d and let $P = Promise(A, B, a, T)$. The* degree of fulfillment *of P, denoted $degFulfill_M(P)$,*

is defined as:

$$e^{-\sum_{P_i \in Dom(M_P)} \omega_i \phi^A_\alpha(P_i, M(P_i))}$$

where Dom_{M_P} *is the domain of* M *restricted to considering only atom* P*. The degree of fulfillment for* S_p*, denoted* $\mathit{degFulfill}_M(S_p)$*, is then defined as:*

$$\frac{\sum_{P_i \in S_p} \gamma^{\mathit{diff}(t_{now}, T_i)} \mathit{degFulfill}_M(P_i))}{\sum_{P_i \in S_p} \gamma^{\mathit{diff}(t_{now}, T_i)}}$$

where $\gamma \in (0, 1]$*,* T_i *is the time point associated with* P_i*, and* $\mathit{diff}(x, y) = x - y$ *if* $x > y$ *and* 0 *otherwise.*

Note that *degree of fulfillment* according to this definition is a real number in $[0, 1]$. Intuitively, this definition assumes the existence of a set of *Promise* atoms S_p and a set of *Do* atoms S_d, that represent promises made by an agent B to an agent A and what actions were carried out by B towards fulfilling such promises. In order to obtain the associated *degree of fulfillment*, A will evaluate the *distance* between the promises and the enactments by establishing a mapping from some suitable decomposition of S_p to some suitable decomposition of S_d. With such a mapping, individual degrees of fulfillment are obtained using the first part of the definition, and then each individual degree is weighted according to the time at which the promise was due as in the second part of the definition.

In Definition 36, the term $\phi^A_\alpha(P_i, M(P_i))$ refers to the distance between a given promise P_i and the action $M(P_i)$ that was performed to fulfill that promise according to enactment mapping M. Multiplying this term by the proportion ω_i of P_i gives us a weighted assessment of this distance (for cases in which the P_i's correspond to parts of an original promise). The summation over all P_i's in $Dom(M_P)$ gives us all sub-promises P_i associated with P and computes their individual fulfillments.

Taking e to the negative power of this summation weights promise P's fulfillment in a way that is inversely proportional to these distances, resulting in a value in $[0, 1]$. The following example shows a simple calculation of degree of fulfillment:

Example 41. *Let us return to Examples 36 and 37, where D_1 is now changed to $Do(B, A, del(ball, blue, 5), T_1)$ instead. Consider the functions*

$$\delta_{del}(del(I_1, C_1, X_1), del(I_2, C_2, X_2)) = |X_1 - X_2|$$

iff $I_1 = I_2$ and $C_1 = C_2$, and ∞ otherwise, and

$$\phi_{del}(Promise(B, A, a_1, T_1), Do(B, A, a_2, T_2)) = \delta^A_{del}(a_1, a_2) + |T_1 - T_2|$$

In this case, using mapping M from Example 40, we have the following individual degrees of fulfillment:

$$degFulfill_M(P_1) = e^{-(0.7*2+0.3*0)} = 0.246$$
$$degFulfill_M(P_2) = e^{-1*0} = 1$$
$$degFulfill_M(P_3) = e^{-1*0} = 1$$

Then, assuming $\gamma = 0.9$, that the different time points are at unit distance, and that $t_{now} = T_3 + 1$, we get:

$$\frac{0.9^3 * 0.246 + 0.9^2 * 1 + 0.9 * 1}{0.9^3 + 0.9^2 + 0.9} \approx \frac{1.889}{2.439} \approx 0.774$$

As we have seen, mapping M plays a major role in how the degree of fulfillment is computed, and there are many ways in which this mapping can be obtained. For instance, it can be built by the agents involved in the promises made, since they can agree on this mapping when each action described by a *Do* atom is performed. Another way would be to perform a search through the space of possible mappings

in order to obtain one that *maximizes* the degree of fulfillment that is obtained, *i.e.*, the mapping that is *most beneficial to agent* B. However, this approach has a high computational cost, as shown in the following result.

Proposition 22. *Given two sets* S_p *and* S_d *of Promise and Do atoms, respectively, and a real number* $k \in [0, 1]$*, finding an enactment mapping* M *such that* $degFulfill_M(S_p) > k$ *is NP-complete.*

Proof. We will first show membership in NP and then NP-hardness.

Membership in NP: If we are given a mapping M, checking that it is well defined (*i.e.*, that its domain and codomain are valid decompositions of S_p and S_d, respectively) can be done in polynomial time. Hence, it remains to be proven, that the size of M (where M is a relation, *i.e.*, a set of pairs) is polynomial in the size of the input, *i.e.*, $|S_p|$ and $|S_d|$. For this, it is important to observe that the number of elements in any minimal decomposition $\chi^{-1}(P)$ of any promise $P \in S_p$ is bounded by $|S_d|+1$. To prove this, assume the contrary. Hence, we have $|\chi^{-1}(P)| > |S_d|+1$, from which we can conclude that at least two elements p_1, p_2 must be mapped onto elements d_1, d_2 which are part of the decomposition of a single do atom $D \in S_d$, *i.e.*, $d_1, d_2 \in \chi^{-1}(D)$. Hence, we can merge p_1, p_2 and d_1, d_2 within their respective decompositions and arrive at a new mapping M^* that is semantically equivalent (*i.e.*, promises are fulfilled by the same *Do*'s or decompositions thereof) due to the fact that M is quasi injective and hence violates the minimality of the original decomposition $\chi^{-1}(P)$. The case where both p_1 and p_2 are mapped onto *Null* is similar to the one presented above, whereby we only merge p_1, p_2. Consequently, the size of M is bounded by $|S_p|\,(|S_d|+1)$ as had to be proven. Note that we only consider the bound on minimal mappings (minimality with respect to the decompositions). The argument above shows that this does not exclude reasonable mappings.

NP-hardness: We will reduce the problem of SUBSET-SUM (SS) with positive integers to our problem in polynomial time in order to prove NP-hardness. This corresponds to deciding, given a set S of positive integers and an integer c, if there exists $S' \subseteq S$ such that $\sum_{e_i \in S'} e_i = c$.

Given an instance of SS, we must then provide an instance of our problem such that its solution provides an answer to SS. Let $S_p = \{Promise(A, B, \alpha_0, 0)\}$, and $S_d = \{Do(A, B, \beta_j, 0) \mid j \in S\}$, where α_i and β_j are dummy action symbols of arity zero. We fix the replaceability relation $\leftrightharpoons$ such that it states that $\{\alpha_0\} \leftrightharpoons D$ if and only if $D = \{\alpha_i \mid i \in S\}$ and $\sum_{\alpha_i \in D} i = c$. Next, $\phi(\alpha_i, \beta_j) = 0$ if and only if $i = j$ and ∞ otherwise. for $i, j \in S$. Lastly, let $k = 0$.

This transformation yields the desired results, since an enactment mapping M such that $degFulfill_M(S_p) > 0$ exists if and only if S_p can be decomposed into a set of *Promise* atoms that represent a subset of S that sums to c. If this is not possible, then by Definition 36, $degFulfill_M(S_p) = 0$. Lastly, note that this reduction can be done in polynomial time. □

We conclude this section by stating some propositions that characterize the degree of fulfillment introduced in Definition 36. We first show that the overall degree of fulfillment does not depend on the reference time point t_{now}, and hence gives justification for our notation which leaves the time point t_{now} implicit with the context.

Proposition 23. *The overall degree of fulfillment, $degFulfill_M(S_p)$, is independent of the reference time point t_{now},* i.e., *evaluating $degFulfill_M(S_p)$ w.r.t. two reference time points t^1_{now} and t^2_{now} such that $\forall P_i \in S_p : T_i \leq t^1_{now}, t^2_{now}$ yields the same value.*

Proof. Let M be a fixed mapping between promise and do decompositions and let S_p be a set of promises. Let t^1_{now} and t^2_{now} be two time points such that $\forall P_i \in S_p : T_i \leq t^1_{now}, t^2_{now}$. Then we have:

$$\begin{aligned}
degFulfill_M(S_p) &= \frac{\sum_{P_i \in S_p} \gamma^{diff(t^1_{now},T_i)}\, degFulfill(P_i, S_d)}{\sum_{P_i \in S_p} \gamma^{diff(t^1_{now},T_i)}} \\
&= \frac{\sum_{P_i \in S_p} \gamma^{t^1_{now}-T_i}\, degFulfill(P_i, S_d)}{\sum_{P_i \in S_p} \gamma^{t^1_{now}-T_i}} \\
&= \frac{\gamma^{t^1_{now}} \sum_{P_i \in S_p} \gamma^{-T_i}\, degFulfill(P_i, S_d)}{\gamma^{t^1_{now}} \sum_{P_i \in S_p} \gamma^{-T_i}} \\
&= \frac{\sum_{P_i \in S_p} \gamma^{-T_i}\, degFulfill(P_i, S_d)}{\sum_{P_i \in S_p} \gamma^{-T_i}} \\
&= \frac{\gamma^{t^2_{now}}}{\gamma^{t^2_{now}}} \frac{\sum_{P_i \in S_p} \gamma^{-T_i}\, degFulfill(P_i, S_d)}{\sum_{P_i \in S_p} \gamma^{-T_i}} \\
&= \frac{\sum_{P_i \in S_p} \gamma^{t^2_{now}-T_i}\, degFulfill(P_i, S_d)}{\sum_{P_i \in S_p} \gamma^{t^2_{now}-T_i}} \\
&= \frac{\sum_{P_i \in S_p} \gamma^{diff(t^2_{now},T_i)}\, degFulfill(P_i, S_d)}{\sum_{P_i \in S_p} \gamma^{diff(t^2_{now},T_i)}}
\end{aligned}$$

□

The following result shows how the overall degree of fulfillment can be incrementally computed.

Proposition 24. *Let S_d be a set of promises and let $\rho = degFulfill_M(S_p)$ denotes its overall degree of fulfillment. Suppose $\tilde{S_p} = S_p \cup \{P\}$ and $degFulfill_M(P)$ denotes the degree of fulfillment of P according to M. Then we have:*

$$degFulfill_M(\tilde{S_p}) = \frac{\rho\tau + \gamma^{diff(t_{now},time(P))}\, degFulfill_M(P)}{\tau + \gamma^{diff(t_{now},time(P))}}$$

where τ is the denominator of the degree formula, i.e. $\tau = \sum_{P_i \in S_p} \gamma^{diff(t_{now},T_i)}$.

Proof. Starting from the definition of $\mathit{degFulfill}_M(\tilde{S}_p)$, we have the following derivation:

$$\mathit{degFulfill}_M(\tilde{S}_p) = \frac{\sum_{P_i \in \tilde{S}_p} \gamma^{\mathit{diff}(t_{now},T_i)} \mathit{degFulfill}_M(P_i))}{\sum_{P_i \in \tilde{S}_p} \gamma^{\mathit{diff}(t_{now},T_i)}}$$

$$= \frac{\sum_{P_i \in S_p} \gamma^{\mathit{diff}(t_{now},T_i)} \mathit{degFulfill}_M(P_i)) + \gamma^{\mathit{diff}(t^1_{now},\mathit{time}(P))} \mathit{degFulfill}_M(P)}{\sum_{P_i \in S_p} \gamma^{\mathit{diff}(t_{now},T_i)} + \gamma^{\mathit{diff}(t^1_{now},\mathit{time}(P))}}$$

$$= \frac{\rho\tau + \gamma^{\mathit{diff}(t_{now},\mathit{time}(P))} \mathit{degFulfill}_M(P)}{\tau + \gamma^{\mathit{diff}(t^1_{now},\mathit{time}(P))}}$$

since

$$\mathit{degFulfill}_M(S_p) = \frac{\rho}{\tau}$$

where τ is the denominator of the degree formula, *i.e.*, $\tau = \sum_{P_i \in S_p} \gamma^{\mathit{diff}(t_{now},T_i)}$. □

The following result shows that the degree of fulfillment of a single promise changes by a constant factor for different weightings of time in the particular distance measure introduced in Proposition 21.

Proposition 25. *Let* $\phi_\mu(P, D) = \ell_\mu \times |T_2 - T_1| + \delta^A_\alpha(a_1, a_2)$*, for atoms of the form* $P = \mathit{Promise}(B, A, a_1, T_1)$ *and* $D = \mathit{Do}(B, A, a_2, T_2)$*, be distance measures similar to the one defined in Proposition 21 where* $\ell_\mu \geq 0$ *are a set of real numbers to weight time delays. Let* M *be the fixed mapping between promise and do decompositions as before. Let* $\mathit{degFulfill}^{\phi_\mu}_M(P)$ *denote the degree of fulfillment for a single promise* P

with respect to the distance measure ϕ_μ. Then we have:

$$\frac{degFulfill_M^{\phi_k}(P)}{degFulfill_M^{\phi_j}(P)} = e^{-(\ell_k-\ell_j)\sum_{P_i\in D_{\{P\}}}\omega_i|time(P_i)-time(M(P_i))|}$$

Proof. We start from the definitions of $degFulfill_M^{\phi_k}(P)$ and $degFulfill_M^{\phi_j}(P)$; we then have the following derivation:

$$\frac{degFulfill_M^{\phi_k}(P)}{degFulfill_M^{\phi_j}(P)} = \frac{e^{-\sum_{P_i\in D_{\{P\}}}\omega_i\phi_k(P_i,M(P_i))}}{e^{-\sum_{P_i\in D_{\{P\}}}\omega_i\phi_j(P_i,M(P_i))}} =$$

$$= e^{-\sum_{P_i\in D_{\{P\}}}\omega_i(\phi_k(P_i,M(P_i))-\phi_j(P_i,M(P_i)))} =$$

$$= e^{-\sum_{P_i\in D_{\{P\}}}\omega_i((\ell_k-\ell_j)\times|time(P_i)-time(M(P_i))|+d-d)}$$

where $d = \delta_\alpha^A(a_1,a_2)$; continuing with the derivation:

$$e^{-\sum_{P_i\in D_{\{P\}}}\omega_i((\ell_k-\ell_j)\times|time(P_i)-time(M(P_i))|)} =$$

$$= ke^{-(\ell_k-\ell_j)\sum_{P_i\in D_{\{P\}}}\omega_i|time(P_i)-time(M(P_i))|}$$

□

The next result shows how linear changes in the distance function on actions impact the degree of fulfillment.

Proposition 26. *Let $\phi_1(P,D) = \ell\times|T_2-T_1| + \delta_\alpha^A(a_1,a_2)$ and $\phi_2(P,D) = \ell\times|T_2-T_1| + \lambda\delta_\alpha^A(a_1,a_2)$, for $P = Promise(B,A,a_1,T_1)$ and $D = Do(B,A,a_2,T_2)$, be two*

distance measures. Then we have

$$\frac{degFulfill_M^{\phi_2}(P)}{degFulfill_M^{\phi_1}(P)} = e^{-(\lambda-1)\sum_{P_i \in D_{\{P\}}} \omega_i \delta_\alpha^A(action(P_i), action(M(P_i)))}$$

Proof. We start from the definitions of $degFulfill_M^{\phi_1}(P)$ and $degFulfill_M^{\phi_2}(P)$; we then have the following derivation:

$$\frac{degFulfill_M^{\phi_2}(P)}{degFulfill_M^{\phi_1}(P)} = \frac{e^{-\sum_{P_i \in D_{\{P\}}} \omega_i \phi_2(P_i, M(P_i))}}{e^{-\sum_{P_i \in D_{\{P\}}} \omega_i \phi_1(P_i, M(P_i))}} =$$

$$= e^{-\sum_{P_i \in D_{\{P\}}} \omega_i (\phi_2(P_i, M(P_i)) - \phi_1(P_i, M(P_i)))} =$$

$$= e^{-\sum_{P_i \in D_{\{P\}}} \omega_i ((\ell-\ell)\times|time(P_i) - time(M(P_i))| + (1-\lambda)\times d)} =$$

where $d = \delta_\alpha^A(a_1, a_2)$; continuing with the derivation:

$$= e^{-\sum_{P_i \in D_{\{P\}}} \omega_i ((1-\lambda)\times \delta_\alpha^A(a_1, a_2))} =$$

$$= e^{-(1-\lambda)\sum_{P_i \in D_{\{P\}}} \omega_i \delta_\alpha^A(a_1, a_2)}$$

□

In the following section, we will preent an application of this framework for reasoning about the fulfillment of promises made by airlines.

8.5 Application and Experiments

In this section, we use the preceding results to estimate the *future behavior* of agents that made a promise in the past. We show that our fulfillment measures have a strong predictive power (unlike past papers on this topic which did not demonstrate predictive power).

In the rest of this section, we assume that mapping M used in Definition 36 above is fixed *a priori*[2]. We now discuss two different ways in which an agent can reason about the likelihood of the different outcomes that can arise in the presence of a pending promise or set of promises.

- **FFIP Strategy** (Future fulfillment is identical to the Past). Agent A decides that the likelihood that a certain promise $P^* = Promise(B,A,a,T)$ where $T > t_{now}$ will be kept by agent B at a future time T is completely determined by the experiences with past promises. Hence, in this case, we set $FFIP = degFulfill_M(S_p)$, which simply states that we expect the agent to fulfill its promises to the degree of fulfillment associated with its past promises. The agent is free to choose which promises should be included in this computation, since taking different subsets into account (taking into account the type of promise) may have an impact on how accurate the estimation is.

- **FFLT Strategy** (Future fulfillment is a Linear Trend based on the Past). Agent A evaluating agent B's promise notices that the reliability of B has changed over time. For instance, its reliability at time 1 was r_1, its reliability

[2]This assumption is made without loss of generality, and is needed in order to avoid unwanted variations in the way in which the mapping is done when changing the set of relevant promises taken from historic information. An easy way to fix a mapping is the following: when an agent performs an action, it merely states which promise that action is intended to fulfill, partially or completely.

at time 2 was r_2, and so forth. The reliability at any time t is computed using $degFulfill_M(S_p)$ as above. The agent now considers the r_i's as a time series and uses linear regression to predict the value of this time series at time T. This method allows our system to establish a more controlled way in which to penalize an agent that has broken recent promises (even though promises in the distant past were kept well) or to reward an agent that has kept its promises recently (even though it broke promises in the more distant past). Of course, this method can be easily extended to the use of other kinds of regression models such as logistic regression or higher degree polynomial regressions used commonly in statistics; we leave the study of the application of these models for future work.

8.5.1 The US Airline On-Time Performance Dataset

As an example application, we use our approach to analyze the reliability of US airlines. The dataset used in this experimental evaluation corresponds to the on-time performance data for over 117 million flights in the US, recorded over a span of 20 years. For each flight, 55 attributes are stored, including flight dates, origin and destination, departure and arrival delays, whether the flight was canceled or diverted, and information about who was responsible for delays and/or cancelations [BTS08].

We considered each flight stored in the database to represent both a promise made by the airline to the customer (of departing and arriving on time, without deviating from the agreed on departure and arrival airports) and its enactment. Therefore, we have a single action symbol *fly* of arity 3, *i.e.*, actions are of the form *fly(from, to, depTime)*, while promises and enactments have the form *Promise(A,*

Airline	FFIP	FFLT	Actual	$dist_{FFIP}$	$dist_{FFLT}$
A_1	0.933	0.924	0.924	0.009	0
A_2	0.922	0.911	0.877	0.045	0.034
A_3	0.914	0.909	0.883	0.031	0.026
A_4	0.942	0.936	0.935	0.007	0.001
A_5	0.924	0.918	0.895	0.029	0.023
A_6	0.935	0.927	0.904	0.031	0.023
A_7	0.934	0.926	0.899	0.035	0.027
A_8	0.923	0.907	0.908	0.015	0.001

Table 8.1: Predictions for 2007: all past data and linear trend

C, f, arrTime) and *Do(A, C, f, arrTime)*, respectively, where A is the airline agent, C is the customer agent, f is a *fly* atom, and *arrTime* is the arrival time promised. The information provided in the database for each flight is enough to derive these atoms.

8.5.2 Empirical Results

Out of the airlines that reported on-time performance for their flights, we chose the eight that have reported continuously from 1988 to 2007; this set includes all major US airlines active today, but we will keep their names anonymous in reporting our results.

For these preliminary evaluations, we computed degrees of fulfillment over sets of promises made throughout entire years, in order to avoid seasonal variations (such as increased delays during winter). However, each individual flight made a contribution to the final degree computed, as dictated by Definition 36. We implemented the FFIP and FFLT strategies that an individual traveler or a travel agent could adopt in order to predict the degree of fulfillment that a promise will have when made by a certain airline. Table 8.1 shows how these strategies performed when trying to predict the degree of fulfillment for the year 2007 based on information of

all flights from 1988 to 2006. All degrees of fulfillment reported in these tables were obtained using a distance function ϕ that ignored delays in departures, and used a "step" function for assigning distances regarding arrival delays. This step function is defined as follows: 0.1 for delays up to 15 minutes, 0.2 up to 30 minutes, 0.8 up to 45 minutes, 2.0 up to 60 minutes, and 10.0 for 90 minutes or more. This means, for instance, that a flight arriving 18 minutes late is considered to be fulfilled to a degree of $e^{0.2}$, which is about 0.818. A value of 0.99 was used for γ, t_{now} was set to January 2, 2008, and the unit of time granularity was set to 30 days, meaning that a flight that occurred in January of 1988 is 244 time units away, and its weight is $0.99^{244} \approx 0.086$.

Fulfillment Model Construction Time. The time taken to compute these degrees depends linearly on the number of promises, as can be deduced from Definition 36. For example, as a general indication of the time required to perform this computation, all 15.6 million flights for airline a_1 (from 1988 to 2006) were processed at a rate of about 0.18 milliseconds per promise. All computations were performed on a computer with an Intel Xeon CPU at 3.4GHz and 32GB of RAM under the Linux Operating System (2.6.9-42.0.10.ELlargesmp kernel); the database engine used was PostgreSQL version 7.4.16.

FFIP and FFLT query processing time. Most computations for the FFIP and FFLT strategies are performed during the model construction time; actual query processing involves a small number of primitive operations (looking up a value, and computing a linear function, respectively), and therefore query processing times are under 1ms.

Accuracy of Predictions. Two things to note from the results in Table 8.1 are:

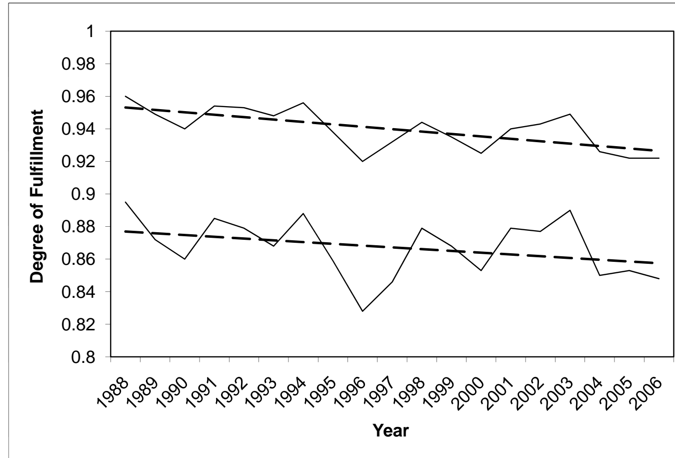


Figure 8.1: Evolution of degree of fulfillment for a single airline over time, for two different ϕ functions sensitive only to arrival delays.

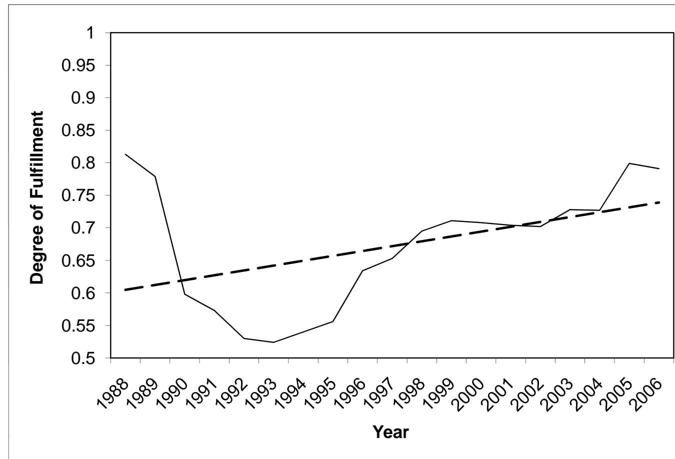


Figure 8.2: Evolution of degree of fulfillment for a single airline over time, for a ϕ function sensitive to departure delays only; note that the trend is positive, unlike those shown in Figure 8.1 (note the difference in scale in the x axis w.r.t. that figure).

1. When comparing what actually happened in terms of an airline's performance (the "Actual" column) and what FFIP and FFLT predicted, the distances between them was relatively small — under 0.03 in almost all cases.

2. When comparing FFIP and FFLT against each other, FFLT was closer to the actual degree of fulfillment in every single case. We then conclude that FFLT is the better algorithm.

Figure 8.1 presents an example of two trend analyses. The top curve shows how the degree of fulfillment (grouped in years) has evolved for a certain airline from 1988 to 2006, while the dotted line indicates the linear trend it follows. The bottom curve and dotted line correspond to the same analysis, but w.r.t. a different, “harsher” ϕ distance function. This function is similar to the one presented above, but assigns larger distances, namely: 0.1 for delays up to 5 minutes, 0.2 up to 15 minutes, 0.8 up to 25 minutes, 2.0 up to 35 minutes, and 10.0 for 45 minutes or more. For the example flight above which was 18 minutes late, this function declares a degree of fulfillment of $e^{0.8}$, which is about 0.449.

We observe that this yielded overall lower degrees of fulfillment, but the shape of the curve is more or less the same, with each inflection being more exaggerated than its counterpart for the previous function. These changes correspond to what was expected given the change in how the ϕ function was defined.

Finally, Figure 8.2 shows an example of a trend analysis for the same airline shown in Figure 8.1. The ϕ distance function used in this case is a different one, which is only sensitive to delays in departures instead of arrivals (the specific values are the same as for the first function presented above). It is interesting to observe that, depending on the perspective of the user, the same airline displays both a downward trend and an upward trend in the evolution of its degree of fulfillment over time. This shows one of the strengths of the framework, *i.e.*, that the user's preferences are taken into account in evaluating the degree of fulfillment of an agent's promises. This is in contrast to, for example, the conclusions that could be obtained by performing a traditional statistical analysis of the frequency of delays such as those presented in [Fli08].

8.6 Discussion

In this section, we would like to discuss several aspects of our framework that we want to highlight, including several limitations that the reader should be aware of.

First of all, the work in this chapter assumes that all promises considered have already been made, and therefore "agreed upon" by both parties, *i.e.*, the promise was proposed and accepted. This means that agents cannot simply make promises leaving a lot of room for possible failures (for instance, promising to land at 9AM instead of at 8:15AM), since this kind of behavior will likely not be accepted by the other agent. Furthermore, promises in this framework only involve one action, so a "complex" promise that requires several actions to be performed is actually regarded as a series of promises, each of which will have its associated degree of fulfillment. Lastly, we are focusing only on reasoning based on actions taken towards these promises, and not about beliefs regarding the capabilities of agents to fulfill the promises they have made.

We would like to discuss certain limitations that the framework exhibits. First of all, the axiomatization presented is intended to be a general set of properties that any system should exhibit. Even though this generality can be perceived as a weakness, it lays the groundwork for future research in which assumptions can be made in accordance with specific domains. Another important aspect to note is that the current presentation assumes that the reasoning agent evaluates degrees of fulfillment for *one agent at a time*. This means that, for instance, it will not reason about what the other agent did towards fulfilling its promises with other agents; if this were not the case, Axiom F2 would not always be a desirable property since no action might be preferable to actions benefiting others. Another aspect that may

be perceived as a limitation is the fact that degrees of fulfillment are real numbers. This means that it is hard to make the distinction, for instance, between complete fulfillment of a strict subset of promises versus partial fulfillment of all promises in the same set (this is similar to the limitation exhibited by customer satisfaction ratings that merge ratings in different areas into one percentage value). Finally, the distance function between actions as defined here can only be evaluated for atoms that share the same action symbol, which does not allow agents to compare promises with respect to different actions, even though this may be desirable in certain situations.

8.7 Concluding Remarks

There are numerous applications where an organization or an individual wants to estimate the likelihood that a given organization or individual will fulfill a promise. Manufacturing companies wish to make such estimates in order to assign logistics assets and to plan accordingly. Consumers would like to decide whether one airline is more reliable than another or whether one politician is more likely to honor his promises than another.

In this chapter, we have developed axioms that a notion of distance between actions, between promises and performed actions, and between sets of promises and sets of actions must satisfy. These axioms are generic and can be satisfied by many different specific distance functions. We provide an epistemic basis for these axioms and define some specific distance functions.

Based on these ideas, we propose a notion of fulfillment of promises that has many important features. In particular, it accounts for three phenomena not fully

handled in previous works. First, we develop a notion of time in studies of promises. Our axioms allow us to penalize late (or early) fulfillments of promises if we so wish, though it does not require such penalties to be imposed. Second, we develop a notion of numeric quantities in promises: delivering 50 of a promised 100 units of a given item can be considered better than nothing and has an impact on our rating of the fulfillment of that promise. Third, we develop notions of replaceability where an agent can accept actions in place of promises that are close enough (*e.g.* 50 red balls may be acceptable in place of 50 blue balls). Our framework is rich enough to support a variety of desires on the part of users to customize the notion of promise fulfillment to their needs.

We implemented two methods for using such fulfillment metrics in order to predict the likelihood of fulfillment of a promise in the future by a given agent, and tested them out on a database of flight on-time information for 8 major US airlines over the last 20 years. Our predictions, tested on the degrees of fulfillment for all flights in 2007 operated by these airlines, are highly accurate and can be computed within reasonable amounts of time.

Even though the methods used above to predict future degrees of fulfillment had a relatively low error in their estimations, it should be noted that the types of predictions that they were evaluated on were very simple since no distinctions were made regarding the *kind* of promises that were made. We argue here that the reasoning agent will be mostly interested in evaluating how likely it is that the actions involved in the promises that it is analyzing given past behavior on the *same kind* of promise. In order to perform this kind of analysis, a richer representation is needed for the *model* that the agent is building of others.

In particular, this framework can be applied in the derivation of *probabilistic logic programs* that can later be used by agents to reason about the likelihood that certain promises will be kept, or what its expected degree of fulfillment is.

Chapter 9

Conclusions

In Chapter 1, we began by introducing the general set of problems that we address in this thesis, including the intuition behind the use of action probabilistic logic programs for stochastic reasoning about the kinds of actions that can be expected from agents being modeled in a certain environment. Various applications of this kind of reasoning were discussed in detail, and we also discussed briefly how rules for *ap*-programs can be derived using straightforward data mining techniques; even though more complex algorithms could also be used, this is not in the scope of this thesis, but we considered it important to at least discuss briefly how these rules can be automatically extracted. Finally, an important point made in Chapter 1 as a starting point for this work was the need for formalisms that make no assumptions about probabilistic independence.

In Chapter 2, we presented the basic concepts and definitions pertinent to action probabilistic logic programs that would be extensively used in later chapters, including syntax, semantics, and a brief discussion about the fixpoint operator that can be used to derive *ap*-programs without action atoms in the body (which is the basis of the assumption we make, without loss of generality, that no such atoms

occur in the bodies of rules). We included examples in this chapter as well in order to enhance the presentation by illustrating the basic concepts being introduced.

Chapter 3 contains important discussions about the vast amount of literature that is related to the work in this thesis to varying degrees. In particular, we discuss probabilistic logic (with a brief recap of how it was originated in the time of Leibniz). Probabilistic logic programming, the close relative of probabilistic logic, is also discussed in detail given its close relationship to our work. An important section in this chapter discusses how action probabilistic logic programs compare to other approaches to probabilistic reasoning (such as Bayesian Networks), which includes examples illustrating the points being made. Finally, this chapter also contains discussions on work related to probabilistic abduction, trust and reputation in autonomous agents, and reasoning about adversaries.

Chapters 4 and 5 present our work on computing most probable worlds in *ap*-programs. Chapters 4 introduces this problem, and presents exact and heuristic algorithms to solve it. On the other hand, Chapter 5 treats a different version of the problem, in which we assume that the user has selected a subset of actions representing the ones he considers to be of interest. The basic idea is to redefine worlds in such a way that each world of interest subsumes a set of the original worlds so the user is not burdened by having to distinguish differences he does not consider to be relevant. We show that a new, smaller linear program can be derived from the original one, and that solutions to this new problem correspond to the sums of the probabilities of all the worlds being subsumed. This has a twofold effect: the linear program is easier to solve because it contains exponentially less variables, and the results obtained are likely to be more meaningful to the user since they more closely represent his interests. We present exact and heuristic algorithms to solve

this new version of the most probable world problem. Finally, both Chapters 4 and 5 conclude with empirical evaluations of the accuracy and scalability of our algorithms.

Chapters 6 and 7 also solve closely related problems. The main idea is to solve a dual of the most probable world problem, which asks: "how can we change the environment in such a way that certain actions are evoked with a given probability?"; we call this problem *abductive query answering*. Chapter 6 presents the basic setup, which formalizes how we can represent the reasoning agent's capabilities for changing the environment, and what it means for an abductive query to be successful; it contains a set of exact and heuristic algorithms for this basic case, as well as empirical evaluations for them. On the other hand, Chapter 7 extends these results for the cost-based (or non-basic) setup, in which we assume that the reasoning agent's capabilities to change the environment are not always successful, and have an associated cost. Furthermore, we assume that intermediate states through which the environment goes in the path to satisfying an abductive query have different rewards; these rewards, along with the costs and probabilities of success just mentioned, represent the fact that the agent being modeled must be taken into account when answering this kind of query, in the spirit of game-theoretic reasoning. Finally, this chapter also concludes with extensive experimental evaluation of the algorithms presented, both for scalability and accuracy.

The final chapter before these conclusions, Chapter 8, presents a novel framework for quantitative reasoning about fulfillment of promises made between autonomous agents; the goal of developing such a framework was to show how action probabilistic logic programs can be tailored to work in advanced applications. The result of this work is a notion of fulfillment of promises that accounts for aspects that

were not fully addressed in past work in the area. In particular, the consideration of time is central to the approach, since late or early fulfillment may be an important point to consider; second, a quantitative treatment of fulfillment is made, taking into account for instance that delivering half of what was promised may be better than nothing; and we propose the notion of promise replaceability, which allows agents to specify what they consider to be equivalent when evaluating fulfillment. In all three of these aspects, the perspective of the agents involved is central, since different agents may have different valuations in different situations. We present an empirical evaluation of this framework on real world data regarding the on-time performance of airlines over two decades time (where airlines are assumed to have made the promise to take off and land on time), showing that different perspectives have a large impact on the results obtained.

In conclusion, this thesis develops action probabilistic logic programs, a formalism that is based on classical probabilistic logic programming. The problems proposed, as well as the solutions, are novel and driven by applications to reasoning about the kinds of actions that can be expected from agents being modeled in an environment. One of the key aspects of all this work is that no probabilistic independence assumptions are made with respect to events involved, which reflects the situations most commonly encountered in the real world in the kind of applications discussed throughout this work.

Bibliography

[ACDM09] D. Applegate, W. Cook, S. Dash, and M. Mevenkamp. QSopt library. Availble at: `http://www2.isye.gatech.edu/~wcook/qsopt/`, 2009.

[ACW08] V Asal, J Carter, and J Wilkenfeld. Ethnopolitical violence and terrorism in the middle east. In J Hewitt, J Wilkenfeld, and T Gurr, editors, *Peace and Conflict 2008*, Boulder, CO, 2008. Paradigm.

[AH94] Martín Abadi and Joseph Y. Halpern. Decidability and expressiveness for first-order logics of probability. *Information and Computation*, 112(1):1–36, 1994.

[AH98] Ashraf M. Abdelbar and Sandra M. Hedetniemi. Approximating maps for belief networks is np-hard and other theorems. *Artificial Intelligence*, 102(1):21 – 38, 1998.

[AP01] Kim Allan Andersen and Daniele Pretolani. Easy cases of probabilistic satisfiability. *Annals of Mathematics and Artificial Intelligence*, 33(1):69–91, 2001.

[BBD+03] Darse Billings, Neil Burch, Aaron Davidson, Robert C. Holte, Jonathan Schaeffer, Terence Schauenberg, and Duane Szafron. Approximating game-theoretic optimal strategies for full-scale poker. In *Proceedings of IJCAI 2003*, pages 661–668, 2003.

[BC01] Bruno Bouzy and Tristan Cazenave. Computer go: An AI oriented survey. *Artificial Intelligence*, 132(1):39 – 103, 2001.

[BDG00] Craig Boutilier, Richard Dearden, and Mosiés Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121(1–2):49–107, 2000.

[BEA09] Michael Bar-Eli and Ofer H. Azar. Penalty kicks in soccer: An empirical analysis of shooting strategies and goalkeepers' preferences. *Soccer and Society*, 10(2):183–191, 2009.

[BEAR+05] Michael Bar-Eli, Ofer H. Azar, Ilana Ritov, Yael Keidar-Levin, and Galit Schein. Action bias among elite soccer goalkeepers: The case of penalty kicks. MPRA Paper 4477, University Library of Munich, Germany, 2005.

[Bel57] Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6, 1957.

[BFW08] Florian Baumann, Tim Friehe, and Michael Wedow. General ability and predictability: Evidence from penalty kicks in soccer. Social Science Research Network, available at: `http://ssrn.com/abstract=1314787`, 2008.

[BGMP97] Matteo Baldoni, Laura Giordano, Alberto Martelli, and Viviana Patti. An abductive proof procedure for reasoning about actions in modal logic programming. In *Selected papers from NMELP '96*, pages 132–150, London, UK, 1997. Springer-Verlag.

[Bha07] Yudhijit Bhattacharjee. Pentagon asks academics for help in understanding its enemies. *Science*, 316(5824):534–535, 2007.

[BJV96] Jeremy S. De Bonet, Charles Lee Isbell Jr., and Paul A. Viola. MIMIC: Finding optima by estimating probability densities. In *Proceedings of NIPS '96*, pages 424–430. MIT Press, 1996.

[BK93] R. Bhatnagar and L.N. Kanal. Structural and probabilistic knowledge for abductive reasoning. *IEEE TPNML*, 15(3):233–245, 1993.

[Boo54] George Boole. *The Laws of Thought.* Macmillan, London, 1854.

[BSS09] Matthias Broecheler, Gerardo Ignacio Simari, and V.S. Subrahmanian. Using histograms to better answer queries to probabilistic logic programs. In Patricia Hill and David Warren, editors, *Logic Programming*, volume 5649, pages 40–54. Springer Berlin / Heidelberg, 2009.

[BTS08] BTS. US DOT Bureau of Transportation Statistics: http://www.transtats.bts.gov/, 2008.

[CH92] G.F. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine learning*, 9(4):309–347, 1992.

[Chr08] Henning Christiansen. Implementing probabilistic abductive logic programming with constraint handling rules. In Tom Schrijvers and Thom W. Frühwirth, editors, *Constraint Handling Rules*, volume 5388 of *Lecture Notes in Computer Science*, pages 85–118. Springer, 2008.

[Chv83] Vašek Chvátal. *Linear Programming.* W.H.Freeman, New York, 1983.

[CLG02] P.-A. Chiappori, S. Levitt, and T. Groseclose. Testing mixed-strategy equilibria when players are heterogeneous: The case of penalty kicks in soccer. *American Economic Review*, 92(4):1138–1151, September 2002.

[CMS08] Jianzhong Chen, Stephen Muggleton, and José Santos. Learning probabilistic logic models from probabilistic examples. *Machine Learning*, 73(1):55–85, 2008.

[Col07] German Coloma. Penalty kicks in soccer: An alternative methodology for testing mixed strategy equilibria. *Journal of Sports Economics*, (5):530–545, 2007.

[CS90] E. Charniak and S. E. Shimony. Probabilistic semantics for cost based abduction. In T. S. W. Dietterich, editor, *AAAI-90*, pages 106–111,

Boston, MA, 1990. MIT Press.

[CT91] Luca Console and Pietro Torasso. A spectrum of logical definitions of model-based diagnosis. *Computational Intelligence*, 7(3):133–141, 1991.

[DB07] Pierpaolo Dondio and Stephen Barrett. Presumptive selection of trust evidence. In *AAMAS '07*, pages 1–8, New York, NY, USA, 2007. ACM.

[Del06] Chrysanthos Dellarocas. Reputation mechanisms. *Handbook on Economics and Information Systems*, 2006.

[DFE09] Gabriel Diaz, Brett Fajen, and Dennis Ehlinger. Learning to anticipate the actions of others: The goal-keeper problem. *Journal of Vision*, 9(8):608–608, 2009.

[DK02] Marc Denecker and Antonis C. Kakas. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, pages 402–436, London, UK, 2002. Springer-Verlag.

[DKS06] Jürgen Dix, Sarit Kraus, and V. S. Subrahmanian. Heterogeneous temporal probabilistic agents. *ACM Transactions on Computational Logic*, 7(1):151–198, 2006.

[dLPdB04] Silvio do Lago Pereira and Leliane Nunes de Barros. Planning with abduction: A logical framework to explore extensions to classical planning. In *SBIA*, pages 62–72, 2004.

[DLST07] James P. Delgrande, Daphne H. Liu, Torsten Schaub, and Sven Thiele. COBA 2.0: A consistency-based belief change system. In *Proceedings of ECSQARU '07*, pages 78–90, Berlin, Heidelberg, 2007. Springer-Verlag.

[DPS99] Carlos Viegas Damasio, Luis Moniz Pereira, and Terrance Swift. Coherent well-founded annotated logic programs. In *Logic Programming*

and Non-monotonic Reasoning, pages 262–276, 1999.

[DS97] Alex Dekhtyar and V. S. Subrahmanian. Hybrid probabilistic programs. In *International Conference on Logic Programming*, pages 391–405, 1997.

[EFL02] Jeffrey Ely, Drew Fudenberg, and David K. Levine. When is reputation bad? Harvard Institute of Economic Research Working Papers 1962, Harvard - Institute of Economic Research, 2002.

[EG95] Thomas Eiter and Georg Gottlob. The complexity of logic-based abduction. *Journal of the ACM*, 42(1):3–42, 1995.

[EGL97a] Thomas Eiter, Georg Gottlob, and Nicola Leone. Abduction from logic programs: Semantics and complexity. *Theoretical Computer Science*, 189(1-2):129–177, 1997.

[EGL97b] Thomas Eiter, Georg Gottlob, and Nicola Leone. Semantics and complexity of abduction from default theories. *Artificial Intelligence*, 90:90–1, 1997.

[Esh88] Kave Eshghi. Abductive planning with event calculus. In *ICLP/SLP*, pages 562–579, 1988.

[ESP99] Thomas Eiter, V.S. Subrahmanian, and George Pick. Heterogeneous active agents, I: Semantics. *Artificial Intelligence*, 108(1/2):178–255, 1999.

[FC01] Rino Falcone and Cristiano Castelfranchi. Social trust: a cognitive approach. In *Trust and deception in virtual societies*, pages 55–90, Norwell, MA, USA, 2001. Kluwer Academic Publishers.

[FH94] Alan M. Frisch and Peter Haddawy. Anytime deduction for probabilistic logic. *Artificial Intelligence*, 69:93–122, 1994.

[FHM90] Ronald Fagin, Joseph Y. Halpern, and Nimrod Megiddo. A logic for reasoning about probabilities. *Information and Computation*, 87(1/2):78–128, 1990.

[Fli08] FlightStats. FlightStats: http://www.flightstats.com/, 2008.

[FMH99] I.M. Franks, T McGarry, and T Hanvey. From notation to training: Analysis of the penalty kick. *Insight*, 2(3):24–26, 1999.

[GAKB02] Roderich Groß, Keno Albrecht, Wolfgang Kantschik, and Wolfgang Banzhaf. Evolving chess playing programs. In *GECCO 2002*, San Francisco, CA, 2002. Morgan Kaufmann.

[GG61] P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem. *Operations Research*, 9(6):849–859, 1961.

[Gil08] Jim Giles. Can conflict forecasts predict violence hotspots? *New Scientist*, (2647), March 2008.

[GMSB10] Matej Guid, Martin Možina, Aleksander Sadikov, and Ivan Bratko. Deriving concepts and strategies from chess tablebases. In H. van den Herik and Pieter Spronck, editors, *Advances in Computer Games*, volume 6048 of *Lecture Notes in Computer Science*, pages 195–207. Springer Berlin / Heidelberg, 2010.

[GS06] Andrew Gilpin and Tuomas Sandholm. A texas hold'em poker player based on automated abstraction and real-time equilibrium computation. In *Proceedings of AAMAS 2006*, pages 1453–1454, 2006.

[Hai84] T. Hailperin. Probability logic. *Notre Dame Journal of Formal Logic*, 25 (3):198–212, 1984.

[Hal90] Joseph Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46(3):311–350, December 1990. *The paper that we cite here is a revised and expanded version of this paper; it can be found at: http://www.cs.cornell.edu/home/halpern/papers/first-*

order_prob_analysis.pdf.

[HFH+09] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten. The WEKA data mining software: An update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[Hsu02] Feng-Hsiung Hsu. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, Princeton, NJ, USA, 2002.

[JHdAa90] Brigitte Jaumard, Pierre Hansen, and Marcus Poggi de Aragão. Column generation methods for probabilistic logic. In *Proceedings of the First Integer Programming and Combinatorial Optimization Conference*, pages 313–331, Waterloo, Ontario, Canada, 1990. University of Waterloo Press.

[Jos08] A. Josang. Abductive reasoning with uncertainty. In L Magdalena, M. Ojeda-Aciego, and J. L. Verdegay, editors, *Proceedings of IPMU 2008*, pages 9–16, Torremolinos, Malaga, Spain, 2008.

[Jos09] A. Josang. Probabilistic abduction using markov logic networks. In *Proceedings of PAIR 2009 (IJCAI 2009 Workshop)*, Pasadena, CA, USA, 2009.

[KBH02] J. Kohlas, D. Berzati, and R. Haenni. Probabilistic argumentation systems and abduction. *Annals of Mathematics and Artificial Intelligence*, 34(1-3):177–195, 2002.

[KIL04] Gabriele Kern-Isberner and Thomas Lukasiewicz. Combining probabilistic logic programming with the power of maximum entropy. *Artificial Intelligence*, 157(1-2):139–202, 2004.

[KL09] Kenneth Kovash and Steven D. Levitt. Professionals do not play minimax: Evidence from major league baseball and the national football league. Working Paper 15347, National Bureau of Economic Research, September 2009.

[KMM00] A.C. Kakas, A. Michael, and C. Mourlas. Aclp: Abductive constraint logic programming. *The Journal of Logic Programming*, 44:129–177(49), 2000.

[KMN+07a] Samir Khuller, Maria Vanina Martinez, Dana Nau, Gerardo Simari, Amy Sliva, and VS Subrahmanian. Computing most probable worlds of action probabilistic logic programs: Scalable estimation for $10^{30,000}$ worlds. *Annals of Mathematics and Artificial Intelligence*, 51(2–4):295–331, 2007.

[KMN+07b] Samir Khuller, Maria Vanina Martinez, Dana Nau, Gerardo Simari, Amy Sliva, and V.S. Subrahmanian. Finding most probable worlds of probabilistic logic programs. In *Proceedings of SUM 2007*, volume 4772, pages 45–59. LNCS, Springer-Verlag, 2007.

[Lit96] Michael Lederman Littman. *Algorithms for Sequential Decision Making*. PhD thesis, Department of Computer Science, Brown University, Providence, RI, February 1996.

[LKI99] Thomas Lukasiewicz and Gabriele Kern-Isberner. Probabilistic logic programming under maximum entropy. *Lecture Notes in Computer Science (Proceedings of ECSQARU 1999)*, 1638, 1999.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming, Second Edition*. Springer-Verlag, 1987.

[LLRS97] Laks V. S. Lakshmanan, Nicola Leone, Robert Ross, and Venkatramanan Siva Subrahmanian. Probview: a flexible probabilistic database system. *ACM Transactions on Database Systems*, 22:419–469, 1997.

[LS01] Laks V. S. Lakshmanan and Nematollaah Shiri. A parametric approach to deductive databases with uncertainty. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):554–570, 2001.

[LSS04] Nicola Leone, Francesco Scarcello, and V.S. Subrahmanian. Optimal models of disjunctive logic programs: Semantics, complexity, and com-

putation. *IEEE Transactions on Knowledge and Data Engineering*, 16(4):487–503, 2004.

[Luk98] Thomas Lukasiewicz. Probabilistic logic programming. In *European Conference on Artificial Intelligence*, pages 388–392, 1998.

[ML10] Joseph P. McGarrity and Brian Linnen. Pass or run: An empirical test of the matching pennies game using data from the national football league. *Southern Economic Journal*, 76(3):791 – 810, 2010.

[MMP+08a] A. Mannes, M. Michael, A. Pate, A. Sliva, Venkatramanan Siva Subrahmanian, and J. Wilkenfeld. Stochastic opponent modelling agents: A case study with Hezbollah. In Huan Liu and John Salerno, editors, *Proceedings of IWSCBMP*, 2008.

[MMP+08b] Aaron Mannes, Mary Michel, Amy Pate, Amy Sliva, Venkatramanan Siva Subrahmanian, and Jonathan Wilkenfeld. Stochastic opponent modeling agents: A case study with Hamas. In *Proceedings of ICCCD 2008*, 2008.

[Mos04] GianCarlo Moschini. Nash equilibrium in strictly competitive games: Live play in soccer. Staff general research papers, Iowa State University, Department of Economics, 2004.

[MSSS08] V. Martinez, G. Simari, A. Sliva, and Venkatramanan Siva Subrahmanian. The soma terror organization portal (STOP): Social network and analytic tools for the real-time analysis of terror groups. In Huan Liu and John Salerno, editors, *Proceedings of IWSCBMP*, 2008.

[MT06] R. Montenegro and P. Tetali. Mathematical aspects of mixing times in markov chains. *Foundations and Trends in Theoretical Computer Science.*, 1(3):237–354, 2006.

[Nau82] Dana S. Nau. The last player theorem. *Artificial Intelligence*, 18(1):53 – 65, 1982.

[Nau83] Dana S. Nau. Decision quality as a function of search depth on game trees. *Journal of the ACM*, 30(4):687–708, 1983.

[NH95] Liem Ngo and Peter Haddawy. Probabilistic logic programming and bayesian networks. In *Asian Computing Science Conference*, pages 286–300, 1995.

[Nil86] Nils Nilsson. Probabilistic logic. *Artificial Intelligence*, 28:71–87, 1986.

[NLP+10] Dana S. Nau, Mitja Lustrek, Austin Parker, Ivan Bratko, and Matjaz Gams. When is it better not to look ahead? *Artificial Intelligence*, In Press, Accepted Manuscript, 2010.

[NS91] Raymond T. Ng and V. S. Subrahmanian. A semantical framework for supporting subjective and conditional probabilities in deductive databases. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 565–580. The MIT Press, 1991.

[NS92] Raymond T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[NS93] Raymond T. Ng and V. S. Subrahmanian. A semantical framework for supporting subjective and conditional probabilities in deductive databases. volume 10, pages 191–235, 1993.

[Osb03] Martin J. Osborne. *An Introduction to Game Theory*. Oxford University Press, USA, 2003.

[Pag96] Maurice Pagnucco. *The Role of Abductive Reasoning within the Process of Belief Revision*. PhD thesis, Basser Department of Computer Science, University of Sydney, 1996.

[Pap91] Christos H. Papadimitriou. On selecting a satisfying truth assignment (extended abstract). In *SFCS '91: Proceedings of the 32nd annual symposium on Foundations of computer science*, pages 163–169, Wash-

ington, DC, USA, 1991. IEEE Computer Society.

[Pea88] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[Pea91] Judea Pearl. Probabilistic and qualitative abduction. In *AAAI Spring Symposium on Abduction*, pages 155–158, Stanford, CA, 1991. AAAI Press.

[Pei83] C.S. Peirce. A theory of probable inference. In C.S. Peirce, editor, *Studies in Logic: By Members of Johns Hopkins University*, pages 126–181. Little, Brown and Company, Boston, 1883.

[Pei40] Charles S. Peirce. *The Philosophy of Peirce: Selected Writings*. Harcourt, New York, NY, 1940.

[PGA87] D. Poole, R. Goebel, and R. Aleliunas. Theorist: A logical reasoning system for defaults and diagnosis. In N. Cercone and G. McCalla, editors, *The Knowledge Frontier*, pages 331–352. Springer, New York, 1987.

[PGL02] Martin Pelikan, David E. Goldberg, and Fernando G. Lobo. A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications*, 21(1):5–20, 2002.

[Poo88] David Poole. A logical framework for default reasoning. *Artificial Intelligence*, 36(1):27–47, 1988.

[Poo89] D. Poole. Explanation and prediction: an architecture for default and abductive reasoning. *Computational Intelligence*, 5(2):97–110, 1989.

[Poo92] David Poole. Representing diagnostic knowledge for probabilistic horn abduction. pages 467–473, 1992.

[Poo93a] David Poole. Logic programming, abduction and probability - a top-down anytime algorithm for estimating prior and posterior probabilities. *New Generation Computing*, 11(3):377–400, 1993.

[Poo93b] David Poole. Probabilistic horn abduction and bayesian networks. *Artificial Intelligence*, 64(1):81–129, 1993.

[Poo97] David Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial Intelligence*, 94(1-2):7–56, 1997.

[PR90] Yun Peng and James A. Reggia. *Abductive inference models for diagnostic problem-solving*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[Put94] M.L. Puterman. *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley & Sons, Inc. New York, NY, USA, 1994.

[Rei49] Hans Reichenbach. *The theory of probability, an inquiry into the logical and mathematical foundations of the calculus of probability. English translation by Ernest H. Hutten and Maria Reichenbach*. University of California Press, Berkeley,, 2d ed. edition, 1949.

[Rep10] Sportsmail Reporter. The perfect penalty formula has been uncovered. The Daily Mail, available at:
`http://www.dailymail.co.uk/sport/worldcup2010/article-1282109/WORLD-CUP-2010-Fabio-Capello-note-perfect-penalty-formula-uncovered.html`, 2010.

[SAM+07] V.S. Subrahmanian, M. Albanese, V. Martinez, D. Reforgiato, G.I. Simari, A. Sliva, O. Udrea, and J. Wilkenfeld. CARA: A Cultural Reasoning Architecture. *IEEE Intelligent Systems*, 22(2):12–16, 2007.

[sat09] SAT4J library. Available at: `http://www.sat4j.org/`, 2009.

[SBB+07] Jonathan Schaeffer, Neil Burch, Yngvi Bjornsson, Akihiro Kishimoto, Martin Muller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers Is Solved. *Science*, 317(5844):1518–1522, 2007.

[SBSK08] Gerardo I. Simari, Matthias Broecheler, VS Subrahmanian, and Sarit Kraus. Promises made, promises broken: A axiomatic and quantitative treatment of fulfillment. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*, pages 59–68, 2008.

[Sch99] Uwe Schöning. A probabilistic algorithm for k-sat and constraint satisfaction problems. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 410, Washington, DC, USA, 1999. IEEE Computer Society.

[SD07a] Carles Sierra and John K. Debenham. Information-based agency. In Manuela M. Veloso, editor, *IJCAI 2007*, pages 1513–1518, 2007.

[SD07b] Carles Sierra and John K. Debenham. A trust model for simple negotiation. In *DEST 2007*, pages 672–677, 2007.

[SDS10a] Gerardo Simari, John P. Dickerson, and V.S. Subrahmanian. Cost-based query answering in probabilistic logic programs. In *Proceedings of SUM 2010*. LNCS, Springer-Verlag, 2010.

[SDS10b] Gerardo I. Simari, John P. Dickerson, and VS Subrahmanian. Abductive query answering in probabilistic logic programs. Under review for publication in a journal, 2010.

[Sha83] Ehud Y. Shapiro. Logic programs with uncertainties: A tool for implementing rule-based systems. In *IJCAI*, pages 529–532, 1983.

[Sha89] Murray Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings IJCAI 89*, pages 1055–1060. Morgan Kaufmann, 1989.

[Sha00] Murray Shanahan. An abductive event calculus planner. *Journal of Logic Programming*, 44:207–239, 2000.

[SHG06] David Stern, Ralf Herbrich, and Thore Graepel. Bayesian pattern ranking for move prediction in the game of go. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 873–880, New York, NY, USA, 2006. ACM.

[SK09] Stefan Szymanski and Simon Kuper. Penalty shoot-outs: how to take the perfect spot-kick. Telegraph.co.uk, available at: `http://www.telegraph.co.uk/sport/football/european/championsleague/5940896/How-to-take-the-perfect-penalty.html`, 2009.

[SKC94] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of AAAI '94*, pages 337–343. MIT press, 1994.

[SL02] Jiefu Shi and Michael L. Littman. Abstraction methods for game theoretic poker. In *CG '00: Revised Papers from the Second International Conference on Computers and Games*, pages 333–345, London, UK, 2002. Springer-Verlag.

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. pages 440–446, 1992.

[SMSS07a] Amy Sliva, Vanina Martinez, Gerardo I. Simari., and Venkatramanan Siva Subrahmanian. Soma models of the behaviors of stakeholders in the afghan drug economy: A preliminary report. In *Proceedings of the First International Conference on Computational Cultural Dynamics (ICCCD 2007)*. ACM Press, 2007.

[SMSS07b] Amy Sliva, Vanina Martinez, Gerardo I. Simari, and VS Subrahmanian. Soma models of the behaviors of stakeholders in the afghan drug economy: A preliminary report. In Dana Nau and Jonathan Wilkenfeld, editors, *Proceedings of the First International Conference*

on Computational Cultural Dynamics (ICCCD 2007), pages 78–86. AAAI Press, 2007.

[SMSS08] Gerardo I. Simari, Vanina Martinez, Amy Sliva, and V.S. Subrahmanian. Scaling Most Probable World Computations in Probabilistic Logic Programs. In *Proceedings of the Second International Conference on Scalable Uncertainty Management (SUM 2008)*, volume 5291, pages 372–385. LNCS, Springer-Verlag, 2008.

[SMSS10] Gerardo I. Simari, Maria Vanina Martinez, Amy Sliva, and VS Subrahmanian. Focused most probable world computations in probabilistic logic programs. *Annals of Mathematics and Artificial Intelligence (To Appear)*, 2010.

[SNT96] Stephen J. J. Smith, Dana S. Nau, and Thomas A. Throop. A planning approach to declarer play in contract bridge. *Computational Intelligence*, 12:106–130, 1996.

[SS10] Gerardo I. Simari and VS Subrahmanian. Abductive inference in probabilistic logic programs. 7:192–201, July 2010.

[SSD05] Bogdan Stroe, V. S. Subrahmanian, and Sudeshna Dasgupta. Optimal status sets of heterogeneous agent programs. In *Proceedings of AAMAS '05*, pages 709–715, New York, NY, USA, 2005. ACM.

[SSMS08] Amy Sliva, VS Subrahmanian, Vanina Martinez, and Gerardo I. Simari. The soma terror organization portal (stop): Social network and analytic tools for the real-time analysis of terror groups. In *Proceedings of the First International Workshop on Social Computing, Behavioral Modeling, and Prediction*, pages 9–18. Springer US, 2008.

[SSNS06] Gerardo Simari, Amy Sliva, Dana Nau, and V. S. Subrahmanian. A stochastic language for modelling opponent agents. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 244–246, New York, NY, USA, 2006. ACM.

[Sub07] V. S. Subrahmanian. Cultural Modeling in Real Time. *Science*, 317(5844):1509–1510, 2007.

[Tar36] Alfred Tarski. Wahrscheinlichkeitslehre und mehrwertige logik. *Erkenntnis*, 5:174–175, 1935-36.

[Tse90] P. Tseng. Solving H-horizon, stationary Markov decision problems in time proportional to log(H). *Operations Research Letters*, 9(5):287–297, 1990.

[TvR96] John Tsitsiklis and Benjamin van Roy. Feature-based methods for large scale dynamic programming. *Machine Learning*, 22(1/2/3):59–94, 1996.

[vE86] M.H. van Emden. Quantitative deduction and its fixpoint theory. *Journal of Logic Programming*, 4:37–53, 1986.

[WAJ+07] Jonathan Wilkenfeld, Victor Asal, Carter Johnson, Amy Pate, and Mary Michael. The use of violence by ethnopolitical organizations in the middle east. Technical report, National Consortium for the Study of Terrorism and Responses to Terrorism, February 2007.

[WB94] R.J. Williams and L.C. Baird. Tight performance bounds on greedy policies based on imperfect value functions. In *10th Yale Workshop on Adaptive and Learning Systems*, 1994.

[WS02] Wei Wei and Bart Selman. Accelerating random walks. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, pages 216–232, London, UK, 2002. Springer-Verlag.

[WW01] Mark Walker and John Wooders. Minimax play at wimbledon. *American Economic Review*, 91(5):1521–1538, December 2001.

[YLH08] Anbu Yue, Weiru Liu, and Anthony Hunter. Measuring the ignorance and degree of satisfaction for answering queries in imprecise proba-

bilistic logic programs. In *SUM '08: Proceedings of the 2nd international conference on Scalable Uncertainty Management*, pages 386–400, Berlin, Heidelberg, 2008. Springer-Verlag.

Index

action history, 188

Algorithm

APEX, 11

HOP, 63, 69

SemiHOP, 71, 72

APEX, 79

Binary, 77

DE_CBQA, 165

decideReachabilityHillClimb, 148

decideReachabilitySAT, 146

Exact for CBQA, 160

FindFocusedMPW, 115

for BAQA (summary), 151

HeuristicRefineCONS, 112

MonteCarloRefineCONS, 105

NaiveMPW, 62

RefineCONS, 97

simpleAnnBAQA, 138

simpleAnnBAQA-Heur-RC, 143

subProgramSearchBAQA, 136

ap-program, *see* program,action probabilistic

APEX, *see* Algorithm, APEX

atom

action, 20, 185

of interest, 90

do, 185

promise, 185

state, 20

BAQA, *see* query answering, basic abductive

CBQA, *see* query answering, cost based

constraints

reachability, 141

reduced, 66

refined, 91

semi-reduced, 73

set of linear, 26

cost function, 156

DE_CBQA, *see* Algorithm, DE_CBQA

decideReachabilityHillClimb, *see* Algorithm, decideReachabilityHillClimb

decideReachabilitySAT, *see* Algorithm, decideReachabilitySAT

disagreement set, 190
distance measures, 188, 191
  axioms, 190, 191

enactment mapping, 193
entailment
  probabilistic, 25
event set, 188

FELT, 203
FFIP, 203
fixpoint operator, 26–29
formula
  action, 21
  action probabilistic, 21
  annotated, *see* formula, action probabilistic
fulfillment, degree, 193, 194

heuristic
  Binary, 74
  Iterative Sampling for CBQA, 164
  Monte Carlo Sampling, 104
    Small number of variables, 108
    Targetted sampling, 109
heuristicRefineCONS, *see* Algorithm, heuristicRefineCONS
HOP, *see* Algorithm, HOP

LP, *see* program,linear

MAROB, 19
MonteCarloRefineCONS, *see* Algorithm, MonteCarloRefineCONS
MPW, *see* world, most probable

NaiveMPW, *see* Algorithm, NaiveMPW

PLP, *see* program,probabilistic logic
problem
  constraint refinement, 96
  Most Probable World, 61
  Most Probable World of Interest, 115
program
  action probabilistic, 18, 22
    semantics, 24
    syntax, 20
  linear, 26
  probabilistic logic, 18, 23
promise, 181
promise composition, 187

query answering
  basic abductive, 127, 130, 131
query answering, cost based, 158

reduction
  of an *ap*-formula, 90
  of an *ap*-program, 23

RefineCONS, *see* Algorithm, RefineCONS

replaceability, 186

reward function, 157

rule

  action probabilistic, 22

  relevance, 23

satisfaction

  of an action formula by a world, 24

  of an *ap*-rule body by a state, 24

SemiHOP, *see* Algorithm, SemiHOP

simpleAnnBAQA, *see* Algorithm, simpleAnnBAQA

simpleAnnBAQA-Heur-RC, *see* Algorithm, simpleAnnBAQA-Heur-RC

state, 21

subProgramSearchBAQA, *see* Algorithm, subProgramSearchBAQA

threshold goal, 136

transition function, 156

world, 21

  lower/upper probability, 56

  most probable, 61