Cost-Based Query Answering in Action Probabilistic Logic Programs

Gerardo I. Simari, John P. Dickerson, and V.S. Subrahmanian

Department of Computer Science and UMIACS
University of Maryland College Park
College Park, MD 20742, USA
{gisimari,jdicker1,vs}@cs.umd.edu

Abstract. Action-probabilistic logic programs (*ap*-programs), a class of probabilistic logic programs, have been applied 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". Given an *ap*-program, we have addressed the problem of deciding if there is a way to change the environment (subject to some constraints) so that the probability that entity E takes some action (or combination of actions) is maximized. In this work we tackle a related problem, in which we are interested in reasoning about the expected reactions of the entity being modeled when the environment is changed. Therefore, rather than merely deciding if there is a way to obtain the desired outcome, we wish to find the *best* way to do so, given costs of possible outcomes. This is called the Cost-based Query Answering Problem (CBQA). We first formally define and study an exact (intractable) approach to CBQA, and then go on to propose a more efficient algorithm for a specific subclass of *ap*-programs that builds on past work in a basic version of this problem.

1 Introduction

Action probabilistic logic programs (*ap*-programs for short) [10] are a class of probabilistic logic programs (PLPs) [14,15,9]. *ap*-programs have been used extensively to model and reason about the behavior of groups; for instance – an application for reasoning about terror groups based on *ap*-programs has users from over 12 US government entities [7]. *ap*-programs use a two sorted logic where there are "state" predicate symbols and "action" predicate symbols, where 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*. These programs 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 such applications, it is essential to avoid making probabilistic independence assumptions as the goal is to *discover* probabilistic dependencies and then exploit these findings for forecasting. For instance, Figure 1 shows a small set of rules automatically extracted from data [1] about Hezbollah's past. Rule 1 says that Hezbollah uses kidnappings as an organizational strategy with probability between 0.5 and 0.56 whenever no political support was provided to it by a foreign state (`forstpolsup`), and the severity of inter-organizational conflict involving it (`intersev1`) is at level "*c*". Rules 2 and 3 state that kidnappings will be used as a strategy with 80-86% probability when 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`). Rule 4 states that this action will be taken with probability 0.49 to 0.55 whenever the organization advocates democratic practices, while Rule 5 states that the probability rises to between 0.71 and 0.77 when 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 [7,13].

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

Fig. 1. A small set of rules modeling Hezbollah

In [21], we explored the problem of determining 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. In this paper, we take the problem one step further by adding the desire to reason about how the entity being modeled reacts to our efforts. We are therefore interested in finding what the *best* course of action on our part is given some additional input regarding how desirable certain outcomes are; this is called the *cost-based query answering problem* (CBQA). In the following, we first briefly recall the basics of *ap*-programs and then present CBQA formally. We then investigate an approach to solving this problem exactly based on Markov Decision Processes, showing that this approach quickly becomes infeasible in practice. Afterwards, we describe a novel heuristic algorithm based on probability density estimation techniques that can be used to tackle CBQA with much larger instances. Finally, we describe a prototype implementation and experimental results showing that our heuristic algorithm scales well in practice.

A brief note on related work: almost all past work on abduction in such settings has been devised under various independence assumptions [19,18,5]. Apart from our first proposal [21], we are aware of no work to date on abduction in possible worlds-based probabilistic logic systems such as those of [8], [16], and [6] where independence assumptions are not made.

2 Preliminaries

2.1 Syntax

We assume the existence of a logical alphabet consisting 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 [12]. We assume $\mathcal{L}_{pred}$ is partitioned into disjoint sets: $\mathcal{L}_{act}$ of *action symbols* and $\mathcal{L}_{sta}$ of *state* symbols. 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 an *action (resp. state) atom.*

Definition 1 (Action formula). *(i) A (ground) action atom is a (ground) action formula; (ii) 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.*

We will use $\mathcal{APF}$ to denote the (infinite) set of all possible *ap*-formulas.

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 rule specified above says that if $B_1, \ldots, B_m$ are all true in a given state, then there is a probability in the interval μ that the action combination F is performed by the entity modeled by the *ap*-rule.

Definition 5 (*ap*-program). *An* action probabilistic logic program *(*ap*-program for short) is a finite set of* ap*-rules. An* ap*-program Π' s.t. $\Pi' \subseteq \Pi$ is called a* subprogram *of Π.*

Figure 1 shows a small part of an *ap*-program derived automatically from data about Hezbollah. 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, *sta*(Π) (resp., *act*(Π)) denotes the set of all state (resp., action) atoms in Π.

Example 1 (Worlds and states). Coming back to the *ap*-program in Figure 1, the following are examples of worlds: $\{\texttt{kidnap}(1)\}, \{\texttt{kidnap}(1), \texttt{tlethciv}(1)\}, \{\}$. The following are examples of states: $\{\texttt{forstpolsup}(0), \texttt{elecpol}(0)\}$, $\{\texttt{demorg}(1)\}$, and $\{\texttt{extsup}(1), \texttt{elecpol}(1)\}$.

2.2 Semantics of *ap*-Programs

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 [12].

Definition 6 (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 7 (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: (i) if $F \equiv a$, for some atom $a \in B_{\mathcal{L}_{act}}$, then $a \in w$; (ii) 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; (iii) 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; (iv) if $F \equiv \neg F'$, for action formula $F' \in formulas(B_{\mathcal{L}_{act}})$, then w does not satisfy F'.*

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

Definition 8 (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.*

The semantics of *ap*-programs uses possible worlds in the spirit of [6,8,16]. 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 $1 - 3$.

While [10] provides a more formal model theory for *ap*-programs, we merely provide the definition below. Π_s is *consistent* iff $LC(\Pi, s)$ is solvable over the reals, $\mathbb{R}$.

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:*
ℓ' = ***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*** *$LC(\Pi, s)$.*

The following is an example of both $LC(\Pi, s)$ and entailment of an *ap*-formula.

Example 2 (Multiple probability distributions given $LC(\Pi, s)$ and entailment). Consider *ap*-program Π from Figure 1 and state s_2 from Figure 2. The set of all possible worlds is: $w_0 = \{\}$, $w_1 = \{\texttt{kidnap}(1)\}$, $w_2 = \{\texttt{tlethciv}(1)\}$, and $w_3 = \{\texttt{kidnap}(1), \texttt{tlethciv}(1)\}$. Suppose p_i denotes the probability of world w_i. $LC(\Pi, s_2)$ then consists of the following constraints:

$$
\begin{array}{|l|}
\hline
s_1 = \{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(0), \texttt{elecpol}(1), \texttt{extsup}(0), \texttt{demorg}(0)\} \\
s_2 = \{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(0), \texttt{elecpol}(0), \texttt{extsup}(0), \texttt{demorg}(1)\} \\
s_3 = \{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(0), \texttt{elecpol}(0), \texttt{extsup}(0), \texttt{demorg}(0)\} \\
s_4 = \{\texttt{forstpolsup}(1), \texttt{intersev1}(c), \texttt{intersev2}(c), \texttt{elecpol}(1), \texttt{extsup}(1), \texttt{demorg}(0)\} \\
s_5 = \{\texttt{forstpolsup}(0), \texttt{intersev1}(c), \texttt{intersev2}(c), \texttt{elecpol}(0), \texttt{extsup}(1), \texttt{demorg}(0)\} \\
\hline
\end{array}
$$

Fig. 2. A small set of possible states

$$
\begin{aligned}
&0.5 \leq p_1 + p_3 \leq 0.56 \\
&0.49 \leq p_2 + p_3 \leq 0.55 \\
&p_0 + p_1 + p_2 + p_3 = 1
\end{aligned}
$$

One possible solution to this set of constraints is $p_0 = 0$, $p_1 = 0.51$, $p_2 = 0.05$, and $p_3 = 0.44$; in this case, there are other possible distributions that are also solutions. Consider formula $\texttt{kidnap}(1) \wedge \texttt{tlethciv}(1)$, which is satisfied only by world w_3. This formula is entailed with probability in $[0, 0.55]$, meaning that one cannot assign a probability greater than 0.55 to this formula[1].

3 The Cost-Bounded Query Answering Problem

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. We are interested in finding a new state s' such that $\Pi_{s'}$ entails $G : [\ell, u]$. However, s' must be *reachable* from s. In this paper, we assume 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 Hezbollah may only succeed with some probability. To model this, we will make use of three functions:

Definition 10. *A* transition function *is any function* $T : \mathcal{S} \times \mathcal{S} \rightarrow [0, 1]$*, and a* cost function *is any function cost* $: \mathcal{S} \rightarrow [0, \infty)$. *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*[2].

Example 3. Suppose that the only state predicate symbols are those that appear in the rules of Figure 1, and consider the set of states in Figure 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

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

[2] 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.

intermediate transformations through state the sequence of states $seq = \langle s_0, s_1, \ldots, s_n\rangle$ is defined:

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

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

Definition 11 (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:*

$$E_{\Pi,R}(s) = \sum_{F:[\ell,u] \in dom(R) \wedge \Pi_s \models F:[\ell,u]} R(F : [\ell, u]) \tag{2}$$

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 (this is the intuition behind Equation 2; other ways of defining this are also possible). *In this paper, 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 ∞ otherwise, for each state s. In the rest of this paper, we assume that all transition cost functions are defined in terms of a canonical cost function.

Example 4. An example of an entailment-based reward function is as follows. Consider state s_2 from Figure 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 2 in Definition 11, $E_{\Pi,R}(s_2) = 0.2 + 0.14 = 0.34$. Assuming $T(s_1, s_2) = 0.93$ as in Example 3, we have $cost_T(s_1, s_2) = \frac{1}{0.34} * \frac{1}{0.93} \approx 3.162$.

Definition 12. *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 an *ap*-program Π and a 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.

In [21], a related problem called the *Basic Probabilistic Logic Abduction Problem* (Basic PLAP) is proposed; the main difference is that in Basic PLAP 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 5. Consider once again the program in the running example and the set of states from Figure 2. Suppose 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, $k = 3$. 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 1 in [21], which states that the Basic PLAP problem is EXPTIME-complete. Unfortunately, due to space constraints proofs will be omitted.

Proposition 1. *CBQA is EXPTIME-complete.*

Furthermore, we can show that CBQA is NP-hard whenever one of two simplifying assumptions hold: (1) the cardinality of the set of ground action atoms is bounded by a constant, and (2) the cardinality of the set of ground state atoms is bounded by a constant. These results are interesting because they show that the complexity of CBQA is caused by having to solve two independent problems: (P1) Finding a subprogram $\Pi' \subseteq \Pi$ such that when the body of all rules in Π' 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 within the cost budget allowed.

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.

4 CBQA Algorithms for Threshold Queries

A *threshold goal* is an annotated action formula of the form $F : [0, u]$ or $F : [\ell, 1]$; this kind of 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). In this paper, we only give algorithms for such queries.

4.1 An Exact Algorithm for CBQA

We show that any CBQA problem can be mapped to a *Markov Decision Process* [2,20] 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:

$$T_{MDP}(s,a,s') = \begin{cases} 0 & \text{if } a \ncong s', \\ T(s,s') & \text{otherwise;} \end{cases} \tag{3}$$

$$T_{MDP}(s,a,s) = 1 - T(s,a,s')\text{for } a \cong s'; \tag{4}$$

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:

$$R(s,a) = \begin{cases} -1 * cost_T(s,s') & \text{for state } s' \in \mathcal{S} \text{ such that } a \cong s', \\ 1 & \text{for states } s' \in \mathcal{S} \text{ such that } \Pi_{s'} \models G : [\ell, u]. \end{cases} \tag{5}$$

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 $\langle s_0, s_1, \dots, s_k \rangle$ 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 2. *Let $O = (\Pi, \mathcal{S}, s_0, G : [\ell, u], cost, T, E_{\Pi,R}, k)$ be an instance of a CBQA problem 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.*

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 [2] 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 [11]. 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 [4,23], similar techniques for *action aggregation* have not been developed.

4.2 A Heuristic Algorithm Based on Iterative Sampling

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) [3,17]. 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 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 3) 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 [21] 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[3], which is initialized with the uniform distribution.

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 6. Consider once again the *ap*-program from Figure 1, and the states from Figure 2. Suppose that we have the following inputs. The goal is `kidnap(1)` $: [0, 0.6]$; the

[3] In an actual implementation, the probability distribution should be represented implicitly, as storing a probability for an exponential number of states would be intractable.

```
Algorithm 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 $\phi_{best}$ 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. initialize probability distribution $P$ over $\mathcal{S}'$ s.t. $P(s) = \frac{1}{|\mathcal{S}'|}$ for each $s \in \mathcal{S}'$;
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 $\phi$ with $cost^*_{seq}(s_0, \phi(j))$;
11.        pick the sequence in $H$ with the lowest cost $c^*$, call it $\phi^*$;
12.        if $c^* < c_{best}$ then $\phi_{best}$:= $\phi^*$; $c_{best}$:= $c^*$;
13.        $P$:= generate new distribution based on $H$;
14. return $\phi_{best}$;
```

Fig. 3. An algorithm for CBQA based on probability density estimation

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 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.

Next, we present the results of our experimental evaluation of this algorithm.

5 Empirical Evaluation

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 $|S|$, where the element at position i represents the proportion of "good" samples that contained state i. This representation does not scale as $|S|$ increases; our implementation thus only keeps track of the states we have visited, implicitly assigning proportion 0 to all nonvisited states. 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.

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 $O(|S_{MDP}|^2 \cdot |A_{MDP}|)$ time. From the transformation discussed in Section 4.1, we see that $|A_{MDP}| = |S_{MDP}|$; furthermore, since $|S_{MDP}|$ is exponentially larger than the number of state atoms found in Π, 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 a space-constrained overview of the results here.

Figure 4 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 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 [21]). 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 1. Typically, the recommended sequences' costs are close[5]; however, in

[4] Given γ and ϵ, one can calculate an error threshold that gaurantees an optimal policy [24].

[5] In terms of relative error, $\eta = \frac{|v-v'|}{|v|}$, for true cost v (MDP) and approx. cost v' (DE_CBQA).

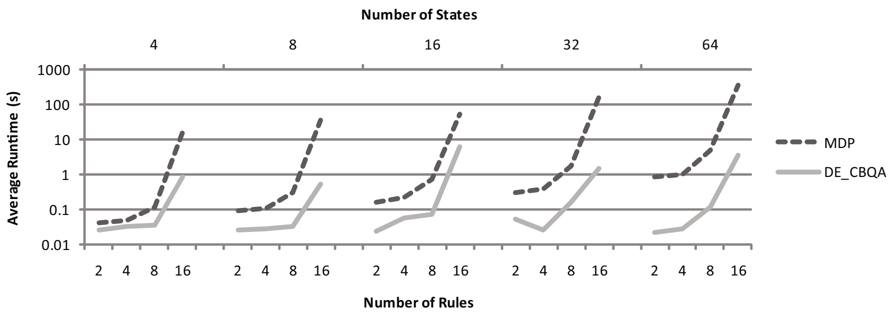

Fig. 4. Log-scale run time comparison of MDP and DE_CBQA, shown with increasing state size (top axis) for each of 2, 4, 8, and 16 rules (bottom axis). Note the sharp jump in run time as the number of rules increases compared to the gradual upward trend as the number of states rises.

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 [22] 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 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 5 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 barely 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 not related to the size of the state space at all. Following these intuitions, we see that the algorithm scales gracefully

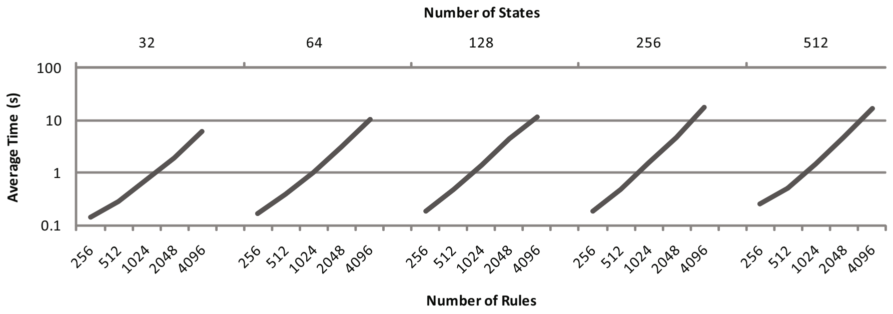

Fig. 5. Log-scale run time as DE_CBQA scales with respect to number of states (top axis) and number of rules (bottom axis). Note the addition of extra rules slows down algorithm execution time much more significantly than a similar increase in state space size.

Fig. 6. *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.

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

to larger state/action spaces. In our experience, real-world instances of CBQA tend to contain significantly *fewer* rules than states and actions [10]; as such, in these cases DE_CBQA scales quite well.

6 Conclusions and Future Work

In this paper, we introduce 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.

In the future, we will explore different formalisms used to learn the probability distribution in Line 13 of DE_CBQA. Currently, we use a simple probability vector to update weights for different states; however, such a representation assumes complete independence between any pair of states. A model that takes into account relationships between states (*e.g.*, Bayesian or neural nets) could provide a more intelligent sampling strategy. It is likely that we will see both a higher computational cost and higher quality solutions as the complexity of the formalism increases.

Acknolwedgements. The authors were funded in part by AFOSR grant FA95500610405 and ARO grant W911NF0910206.

References

1. Asal, V., Carter, J., Wilkenfeld, J.: Ethnopolitical violence and terrorism in the middle east. In: Hewitt, J., Wilkenfeld, J., Gurr, T. (eds.) Peace and Conflict 2008, Paradigm (2008)
2. Bellman, R. A markovian decision process. J. Mathematics and Mechanics 6 (1957)
3. Bonet, J.S.D., Isbell Jr., C.L., Viola, P.A.: MIMIC: Finding optima by estimating probability densities. In: Proceedings of NIPS 1996, pp. 424–430. MIT Press, Cambridge (1996)
4. Boutilier, C., Dearden, R., Goldszmidt, M.: Stochastic dynamic programming with factored representations. Artificial Intelligence 121(1-2), 49–107 (2000)
5. Christiansen, H.: Implementing probabilistic abductive logic programming with constraint handling rules. In: Schrijvers, T., Frühwirth, T. (eds.) Constraint Handling Rules. LNCS (LNAI), vol. 5388, pp. 85–118. Springer, Heidelberg (2008)
6. Fagin, R., Halpern, J.Y., Megiddo, N.: A logic for reasoning about probabilities. Information and Computation 87(1/2), 78–128 (1990)
7. Giles, J.: Can conflict forecasts predict violence hotspots? New Scientist, 2647 (March 2008)
8. Hailperin, T.: Probability logic. Notre Dame J. Formal Logic 25(3), 198–212 (1984)
9. Kern-Isberner, G., Lukasiewicz, T.: Combining probabilistic logic programming with the power of maximum entropy. Artif. Intell. 157(1-2), 139–202 (2004)
10. Khuller, S., Martinez, M.V., Nau, D.S., Sliva, A., Simari, G.I., Subrahmanian, V.S.: Computing most probable worlds of action probabilistic logic programs: scalable estimation for 10^30,000 worlds. AMAI 52(2-4), 295–331 (2007)
11. Littman, M.L.: Algorithms for Sequential Decision Making. PhD thesis, Department of Computer Science, Brown University, Providence, RI (February 1996)
12. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1987)
13. Mannes, A., Michael, M., Pate, A., Sliva, A., Subrahmanian, V.S., Wilkenfeld, J.: Stochastic opponent modelling agents: A case study with Hezbollah. In: Liu, H., Salerno, J. (eds.) Proceedings of IWSCBMP (2008)
14. Ng, R.T., Subrahmanian, V.S.: Probabilistic logic programming. Information and Computation 101(2), 150–201 (1992)
15. Ng, R.T., Subrahmanian, V.S.: A semantical framework for supporting subjective and conditional probabilities in deductive databases. J. Autom. Reas. 10(2), 191–235 (1993)
16. Nilsson, N.: Probabilistic logic. Artificial Intelligence 28, 71–87 (1986)
17. Pelikan, M., Goldberg, D.E., Lobo, F.G.: A survey of optimization by building and using probabilistic models. Comput. Optim. Appl. 21(1), 5–20 (2002)
18. Poole, D.: Probabilistic horn abduction and bayesian networks. Artif. Intell. 64(1), 81–129 (1993)
19. Poole, D.: The independent choice logic for modelling multiple agents under uncertainty. Artif. Intell. 94(1-2), 7–56 (1997)
20. Puterman, M.L.: Markov decision processes: Discrete stochastic dynamic programming. John Wiley & Sons, Inc., Chichester (1994)
21. Simari, G.I., Subrahmanian, V.S.: Abductive inference in probabilistic logic programs. In: Proceedings of ICLP 2010 (Tech. Comm.) (to appear, 2010)
22. Tseng, P.: Solving H-horizon, stationary Markov decision problems in time proportional to log(H). Operations Research Letters 9(5), 287–297 (1990)
23. Tsitsiklis, J., van Roy, B.: Feature-based methods for large scale dynamic programming. Machine Learning 22(1/2/3), 59–94 (1996)
24. Williams, R., Baird, L.: Tight performance bounds on greedy policies based on imperfect value functions. In: 10th Yale Workshop on Adaptive and Learning Systems (1994)