Parallel Abductive Query Answering in Probabilistic Logic Programs

GERARDO I. SIMARI, JOHN P. DICKERSON, AMY SLIVA, and V. S. SUBRAHMANIAN,
University of Maryland College Park

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". Given an *ap*-program, we are interested in 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 then go on to provide an exact (exponential time) algorithm to solve it, followed by more efficient algorithms for specific subclasses of the problem. We also develop appropriate heuristics to solve BAQA efficiently.

The second problem, called the Cost-based Query Answering (CBQA) problem checks to see if there is some way of achieving a desired action (or set of actions) with a probability exceeding a threshold, given certain costs. 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 the results for the basic version of this problem. We also develop the first algorithms for parallel evaluation of CBQA. We conclude with an extensive report on experimental evaluations performed over prototype implementations of the algorithms developed for both BAQA and CBQA, showing that our parallel algorithms work well in practice.

Categories and Subject Descriptors: I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Uncertainty, "fuzzy," and probabilistic reasoning*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Probabilistic reasoning, imprecise probabilities

ACM Reference Format:
Simari, G. I., Dickerson, J. P., Sliva, A., and Subrahmanian, V. S. 2013. Parallel abductive query answering in probabilistic logic programs. *ACM Trans. Comput. Logic* 14, 2, Article 12 (June 2013), 39 pages.
DOI:http://dx.doi.org/10.1145/2480759.2480764

The authors gratefully acknowledge funding support for this work provided to the Lab for Computational Cultural Dynamics (LCCD) by the Army Research Office under grants W911NF0910206 and W911NF1110344, the Army Research Lab under grant W911NF0920072, the Air Force Office of Scientific Research under grant FA95500610405 and the National Science Foundation under grants 0540216 and SES0826886.
G. I. Simari is currently affiliated with the University of Oxford. J. P. Dickerson is currently affiliated with Carnegie Mellon University. A. Silva is currently affiliated with Northeastern University.
Authors' addresses: G. I. Simari, Department of Computer Science, Wolfson Building, Parks Road, University of Oxford, Oxford OX1 3QD, UK; J. P. Dickerson, 9219 Gates-Hillmman Center, Carnegie Mellon University, Pittsburgh, PA 15213; A. Sliva, College of Computer and Information Science, 256 West Village H, Northeastern University, Boston, MA 02115; V. S. Subrahmanian, Department of Computer Science, University of Maryland College Park, College Park, MD 20742; email: vs@cs.umd.edu.

DOI:http://dx.doi.org/10.1145/2480759.2480764

1. INTRODUCTION

Action probabilistic logic programs (*ap*-programs for short) [Khuller et al. 2007] are a class of the extensively studied family of probabilistic logic programs (PLPs) [Kern-Isberner and Lukasiewicz 2004; Ng and Subrahmanian 1992; 1993]. *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 [Giles 2008]. *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 such applications, it is essential to avoid making probabilistic independence assumptions, since the approach involves finding out what probabilistic relationships exist and then exploiting these findings in the forecasting effort. For instance, Figure 1 shows a small set of rules automatically extracted from data [Asal et al. 2008] about Hezbollah's past, where predicates correspond to rules in the data set and in general a value of zero indicates that the action is not performed or the condition does not hold[2]. Rule 1 says that Hezbollah uses kidnappings as an organizational strategy with probability between 0.5 and 0.56 whenever no political support is provided to it by a foreign state (`forstpolsup`), and the severity of inter-organizational conflict (`intersev1`) 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 when 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 the second states that the probability rises to between 0.71 and 0.77 when electoral politics is 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 extensively (and successively) used by terrorism analysts to make predictions about terror group actions [Giles 2008; Mannes et al. 2008b].

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 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 paper, we first briefly recall *ap*-programs and then formulate BAQA theoretically. We then develop a host of complexity results for BAQA under varying

[1]Action atoms represent only 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.
[2]Note that variables in general can have more than two possible values; therefore, even though `kidnap(1)` is equivalent to $\neg$`kidnap(0)` because `kidnap` is a binary variable, this does not hold in general.

assumptions. We then describe both exact and heuristic algorithms to solve the BAQA problem. We briefly describe a prototype implementation and experiments showing that our algorithm is feasible to use even when the *ap*-program contains hundreds of rules.

Following this, we take the problem one step further by reasoning about how the entity being modeled reacts to our efforts. We are interested in identifying the best course of action on our part, given some additional inputs regarding the cost of exerting influence in the environment and how desirable certain outcomes are; this is called the *cost-based query answering problem* (CBQA). 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. We then develop the first parallel algorithms for abduction in probabilistic logics. Finally, we describe a prototype implementation and experimental results showing that our parallel algorithm scales well in practice and achieves results that are useful in practice.

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 [Christiansen 2008; Poole 1997, 1993]. We are aware of no work to date on abduction in possible worlds-based probabilistic logic systems such as those of Hailperin [1984], Nilsson [1986], and Fagin et al. [1990] where independence assumptions are not made. Last, but not least, we are not aware of any parallel implementations of abduction even in the setting with probabilistic independence assumptions, or scalability results that match the results in this paper on real-world data and applications.

2. PRELIMINARIES

We now overview the syntax and semantics of *ap*-programs from Khuller et al. [2007].

2.1. Syntax

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 [Lloyd 1987]. 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 2.1 (*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 2.2 (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.

In the following, we will use $\mathcal{APF}$ to denote the set of all possible *ap*-formulas.

Definition 2.3 (*World/State*). A *world* is any finite set of ground action atoms. A *state* is any finite set of ground state atoms.

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. $\texttt{tlethciv}(1) : [0.71, 0.77] \leftarrow \texttt{elecpol}(1) \wedge \texttt{intersev2}(c)$.

Fig. 1. A small set of rules modeling Hezbollah.

It is assumed that all actions in a 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 assume here that it is not possible to directly reason about the effects of actions. One reason for this is that in many applications (e.g., counterterrorism), there are many, many variables, and the effects of our actions are not well understood. We now define *ap*-rules.

Definition 2.4 (ap-*Rule*). If F is an action formula, $B_1, \dots, B_n$ are state atoms, and μ is an *ap*-annotation, then $F : \mu \leftarrow B_1 \wedge \dots \wedge B_m$ is called an ap-*rule*. If this rule is named r, then $Head(r)$ denotes $F : \mu$ and $Body(r)$ denotes $B_1 \wedge \dots \wedge B_n$.

Intuitively, the rule we have specified says that if $B_1, \dots, B_m$ 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 2.5 (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 1 shows a small portion of an *ap*-program we derived automatically to model Hezbollah's actions. On the average, we have derived *ap*-programs consisting of approximately 11,500 *ap*-rules per terror group.

Henceforth, we use $Heads(\Pi)$ 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 $sta(\Pi)$ (resp., $act(\Pi)$) to denote the set of all state (resp., action) atoms that appear in Π.

Example 2.6 (*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:

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

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 [Lloyd 1987].

Definition 2.7 (*Satisfaction of a Rule Body*). Let Π be an *ap*-program and s a state. We say that s *satisfies* the body of a rule $F : \mu \leftarrow B_1 \wedge \dots \wedge B_m$ if and only if $\{B_1, \dots, B_M\} \subseteq s$.

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

Definition 2.8 (*Satisfaction of an Action Formula*). 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 both F_1 and F_2;

— if $F \equiv F_1 \vee F_2$, for action formulas $F_1, F_2 \in \mathit{formulas}(B_{\mathcal{L}_{act}})$, then w satisfies either F_1 or F_2;
— if $F \equiv \neg F'$, for some action formula $F' \in \mathit{formulas}(B_{\mathcal{L}_{act}})$, then w does not satisfy F'.

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

Definition 2.9 (*Reduction of an* ap-*Program*). Let Π be an *ap*-program and s a state. The *reduction of* Π *w.r.t.* s, denoted Π_s, is the set $\{F : \mu \leftarrow Body \mid s$ 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 Hailperin [1984], Nilsson [1986], and Fagin et al. [1990]. 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 p_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: $\ell \leq \sum_{w_i \in \mathcal{W} \wedge w_i \models F} p_i \leq u$.
(2) $LC(\Pi, s)$ contains the constraint $\sum_{w_i \in \mathcal{W}} p_i = 1$.
(3) All variables are non-negative.
(4) $LC(\Pi, s)$ contains only the constraints described in (1)–(3).

While [Khuller et al. 2007] provide a more formal model theory for *ap*-programs, we merely provide the following definition. Π_s is *consistent* iff $LC(\Pi, s)$ is solvable over $\mathbb{R}$.

Definition 2.10 (*Entailment of an* ap-*Formula*). 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** $\sum_{w_i \in \mathcal{W} \wedge w_i \models F} p_i$ **subject to** $LC(\Pi, s)$.
$u' =$ **maximize** $\sum_{w_i \in \mathcal{W} \wedge w_i \models F} p_i$ **subject to** $LC(\Pi, s)$.

Note that, even though Definition 2.10 defines entailment for reduced programs (i.e., w.r.t. a state), the definition contemplates general programs, since we are only interested in sets of rules for which there exists a state that make them relevant.

The quantity ℓ' in this definition is the smallest possible probability of F, given that the facts in Π are true. In the same vein, u' is the largest such probability. If the $[\ell', u']$ interval is contained in $[\ell, u]$, then $F : [\ell, u]$ is definitely entailed by Π. We will show in Example 2.11 an example of both $LC(\Pi, s)$ and entailment of an annotated action formula.

2.3. A Comparison to Other Probabilistic Approaches

We now compare the power of *ap*-programs with other probabilistic formalisms. First, the following example shows that *ap*-programs are well-suited for representing uncertainty at the level of probability distributions.

Example 2.11 (*Multiple Probability Distributions/Entailment*). Consider *ap*-program Π from Figure 1 and state s_2 from Figure 2. The set of possible worlds contains the following elements: $w_0 = \{\}$, $w_1 = \{\texttt{kidnap(1)}\}$, $w_2 = \{\texttt{tlethciv(1)}\}$, 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.[3]

Note that representing a set of distributions is not possible in many other approaches to probabilistic reasoning, such as Bayesian networks [Pearl 1988], Poole's Independent Choice Logic [Poole 1997] and related formalisms such as Poole [1993]. 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 Ng and Subrahmanian [1993] when probabilistic logic programs were first introduced. Two key extensions are required.

(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 [Khuller et al. 2007] 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 Ng and Subrahmanian [1993], on which *ap*-programs are directly based, included variables as part of their language. This extension was 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 Ng and Subrahmanian [1993]).

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:

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

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

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

Fig. 2. A small set of possible states.

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. BASIC ABDUCTIVE QUERIES TO PROBABILISTIC LOGIC PROGRAMS

Suppose s is a (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 assume the existence of a reachability predicate *reach* specifying *direct* reachability from one state to another. *reach** is the reflexive transitive closure of *reach* and *unReach* is its complement.

For the purposes of the presentation of the theoretical analysis of the problem in this section, we will assume that *reach* is provided and can be queried in polynomial time. However, in order to develop practical algorithms, in Section 3.1.2 we will investigate one way in which *reach* can be specified (by means of so-called *reachability constraints*), as well as ways in which knowledge of action effects and preconditions can be encoded into this predicate[4].

Example 3.1 (*Reachability between States*). Suppose, for simplicity, 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, 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 3.2 (*Solution to BAQA*). Consider once again the program in the running example and the set of states from Figure 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 3.1 shows that state s_1 can be reached from s_4, and $\Pi_{s_1} \models kidnap(1) : [0, 0.6]$.

[4]Furthermore, note that there is an intrinsic relationship between state reachability and consistency of *ap*-programs; this is because states for which the relevant subprogram is inconsistent should never be reachable.

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 Section 4. The following proposition shows the intractability of BAQA in the general case.

PROPOSITION 3.3. *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 $O(2^{|\Delta TM_P|})$, 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 this 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 TM_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. First, we reproduce a lemma (first introduced in [Chvtal 1983] and used in [Fagin et al. 1990]) 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 nonzero value; this lemma will be used to prove Proposition 3.6.

LEMMA 3.4 ([CHVTAL 1983; FAGIN ET AL. 1990]). *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 present our results on the complexity of BAQA under simplifying assumptions.

COROLLARY 3.5. *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 3.3 only makes use of a constant-sized $|\mathcal{L}_{act}|$. □

PROPOSITION 3.6. *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 3.4. 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 nonzero values to a polynomial number of 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 the 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 $\Pi_s \models F : [\varepsilon, 1]$ 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_s will only entail $F : [0, 1]$, and therefore the problem will not be solvable. □

These 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.
— *(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.

3.1. Algorithms for BAQA

In this section, we leverage this intuition to first develop a naïve 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.

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

Definition 3.7 (*Subprogram Reachability Graph*). Let Π be a ground *ap*-program and *reach* be a reachability predicate. The *subprogram graph* is defined as a pair $G = \left(2^{Heads(\Pi)}, E\right)$, 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 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 before, which can of course be implemented in a wide variety of ways. This algorithm is clearly sound and complete.

3.1.1. Answering Threshold Goals. A *threshold goal* is an annotated action formula of the form $F : [0, u]$ or $F : [\ell, 1]$. We now devise a better algorithm for BAQA when only threshold goals are considered. This is a reasonable approach, since threshold goals can be used to require that certain formulas (actions) should only be entailed with

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

Fig. 3. A naïve algorithm for solving BAQA based on the traversal of the relevant subprogram reachability graph induced by *reach*.

a certain maximum probability (upper bound) or should be entailed with at least a certain minimum probability (lower bound). We start by inducing equivalence classes on subprograms that limit the search space, helping address problem (P1).

Definition 3.8 (*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 $F : [\ell, u]$, written $\Pi_1 \sim_{F:\,[\ell,u]} \Pi_2$, iff $\Pi_1 \models F : [\ell, u] \Leftrightarrow \Pi_2 \models F : [\ell, u]$. Furthermore, states s_1 and s_2 are *equivalent* given $F : [\ell, u]$, written $s_1 \sim_{F:[\ell,u]} s_2$, iff $reach(s_1, s_2)$, $reach(s_2, s_1)$, and $\Pi_{s_1} \sim_{F:[\ell,u]} \Pi_{s_2}$.

Intuitively, subprograms Π_1, Π_2 are equivalent w.r.t. $F : [\ell, u]$ whenever they both entail (or do not entail) the annotated formula in question. For clarity, when the probability interval is evident from context, we will omit it from the notation.

Example 3.9 (*Equivalence of* ap-*Programs*). Let Π be the *ap*-program from Figure 1, formula F = kidnap(1); $[0, 0.56]$, $\Pi_1 = \{r_1\}$, $\Pi_2 = \{r_2, r_3\}$ $\Pi_3 = \{r_1, r_4\}$ $\Pi_4 = \{r_1, r_5\}$, 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 2 and the reachability predicate from Example 3.1. 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 identifies sufficient conditions for probabilistic entailment of threshold goals.

LEMMA 3.10 (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, $\Pi \models G : [\ell_G, u_G]$.*

In the following, we will refer to the rules characterized by Lemma 3.10 as *entailing* rules for G. Note that recognizing entailing rules as such depends on the way they are written; even though an annotated formula $F : [\ell, u]$ is semantically equivalent to $\neg F : [1 - u, 1 - \ell]$, one such rule may be entailing while the other is not, leading the application of Lemma 3.10 to fail in this case.

The following corollary to Lemma 3.10 specifies sufficient conditions for the identification of equivalence classes.

```
**Algorithm 2:** simpleAnnBAQA($\Pi, s, G : [\ell_G, u_G]$)
(1) If Lemma 3.10 is applicable, return *true* if there exists a consistent subprogram
    $\Pi' \subseteq \Pi$ such that:
    (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 $reach^*(s, s')$.
(2) **Active rule set initialization** $\left[\textbf{\textit{active}}(\Pi, G : [\ell_G, u_G])\right]$: Initialize this set by selecting
    rules of the form $r : F : [\ell_r, u_r] \leftarrow s_1 \wedge \ldots \wedge s_n$ such that $F \wedge G \not\models \bot$;
    **Passive rule set initialization** $\left[\textbf{\textit{passive}}(\Pi, G : [\ell_G, u_G])\right]$: Initialize this set with the
    rules in $\Pi$ not identified as active;
    **Conflicting rule set initialization** $\left[\textbf{\textit{conf}}(\Pi, G : [\ell_G, u_G])\right]$:
    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 the set
        *conf*($\Pi, G : [\ell_G, u_G]$).
(3) **Candidate active rule set:** Let *candAct*($\Pi, G : [\ell_G, u_G]$) = *active*($\Pi, G : [\ell_G, u_G]$) $\setminus$
    *conf*($\Pi, G : [\ell_G, u_G]$);
(4) 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*($\Pi$).
(5) 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 no pair $\{r_1, r_2\} \subseteq \Pi'$ belongs to *inc*($\Pi$) and:
    *// Iterate favoring subsets of* $\Pi$ *that contain rules in candAct*($\Pi, G : [\ell_G, u_G]$)
    (a) $\Pi' \models G : [\ell_G, u_G]$;
    (b) There exists state $s'$ for which $\Pi_{s'} = \Pi'$ such that $reach^*(s, s')$
        *// Not all subprograms are feasible;* e.g., *if two rules have the same state, one cannot
        be chosen without the other.*
(6) Return *false*;
```

Fig. 4. An algorithm to solve BAQA assuming a threshold goal.

COROLLARY 3.11 (SUFFICIENT CONDITION FOR EQUIVALENCE OF *ap*-PROGRAMS). *Let* Π *be an* ap*-program and G be an annotated action formula. Consider two consistent subprograms* $\Pi', \Pi'' \subseteq \Pi$ *such that* $\Pi' = \Pi'_a \cup \Pi'_p$ *(resp.,* $\Pi'' = \Pi''_a \cup \Pi''_p$*), where* Π'_a *and* Π''_a *are sets of entailing rules for G. Then,* $\Pi' \sim_G \Pi''$.

The algorithm in Figure 4 first tries to leverage Lemma 3.10 and Corollary 3.11, and only proceeds if this is not possible. The way in which the algorithm partitions Π is partly based on Corollary 3.11, and the algorithm then applies a heuristic way of traversing possible subsets of Π based on favoring subsets with rules whose heads share models with the goal (and thus may have a higher chance of satisfying it). The following result proves that this algorithm correctly computes solutions to our problem.

PROPOSITION 3.12 (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 simpleAnnBAQA correctly computes a solution to* BAQA*. Its worst case running time is in* $O\left(2^{|\Pi|} + 2^{|\mathcal{L}_{sta}|} + 2^{|\mathcal{L}_{act}|}\right)$.

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 3.13 (*simpleAnnBAQA over the Running Example*). Suppose Π is the *ap*-program of Figure 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 3.2,

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

note that $\Pi_{s_{curr}} = \{r_2, r_5\}$ and that clearly $\Pi_{s_{curr}} \not\models \texttt{kidnap}(1) : [0, 0.6]$. The first step checks for the applicability of Lemma 3.10; 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 3.1,

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

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. Finally, we illustrate Step 2 with this setup; in this case, the identification of active rules is simple, 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.

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.

3.1.2. An Improved BAQA Algorithm. Up to now, we have been assuming that reachability (or unreachability) was simply determined by querying a predicate; we now explore how we can leverage a syntactic specification of this predicate, showing that in this case we can come up with some good heuristics to solve BAQA.

Definition 3.14 (*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 $F \not\hookrightarrow G$; we call F the antecedent and G the consequent of the constraint, and its semantics is:

$$unReach(s_1, s_2) \Leftrightarrow s_1 \models F \;\text{ and }\; s_2 \models G,$$

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 3.15 (*Reachability Constraints*). Consider again the setting and *ap*-program from Figure 1. The following are examples of reachability constraints:

$$\begin{aligned} rc_1 &: \;\; \texttt{forstpolsup}(1) \not\hookrightarrow \texttt{intersev1}(c) \\ rc_2 &: \;\; \texttt{extsup}(1) \not\hookrightarrow \texttt{intersev1}(c) \\ rc_3 &: \;\; (\texttt{intersev1}(c) \vee \texttt{intersev2}(c)) \wedge \texttt{demorg}(0) \not\hookrightarrow \texttt{demorg}(1) \end{aligned}$$

To illustrate how this kind of constraint can be used to represent action preconditions, suppose we wish to represent the fact that action $\texttt{kidnap}(1)$ cannot be taken whenever $\texttt{demorg}(1)$ is true. This can be represented with the constraint:

$$\texttt{demorg}(1) \not\hookrightarrow \texttt{kidnap\_performed}(1),$$

```
Algorithm 3: simpleAnnBAQA-Heur-RC($\Pi, s, G : [\ell_G, u_G], RC$)
(1) If Lemma 3.10 is not applicable, return false;
(2) Execute Steps 2, 3, and 4 of simpleAnnBAQA;
(3) let goalState, goalStateAct, goalStateConf, and goalStateInf be logical formulas over
    $\mathcal{L}_{sta}$ and $\mathcal{L}_{var}$;
(4) initialize goalState to null, goalStateAct to $\bot$, and goalStateConf, goalStateInc to $\top$;
(5) for each rule $r_i \in candAct(\Pi, G : [\ell_G, u_G])$ with $Head(r_i) = F : [\ell_F, u_F]$ do
        if $[u_G = 1, F \models G$, and $\ell_G \leq \ell_F]$ or $[\ell_G = 0, G \models F$, and $u_G \geq u_F]$
            then set $goalStateAct := goalStateAct \vee Body(r_i)$;
(6) for each rule $r_i \in conf(\Pi, G : [\ell_G, u_G])$ do
        set $goalStateConf := goalStateConf \wedge \neg Body(r_i)$;
(7) for each pair of rules $(r_i, r_j) \in inc(\Pi)$ do
        set $goalStateInc := goalStateInc \wedge \neg(Body(r_i) \wedge Body(r_j))$;
(8) set $goalState := goalStateAct \wedge goalStateConf \wedge goalStateInc$;
    // goalState describes the states for which the corresponding set of relevant rules satisfy
    the input goal
(9) return $decideReachability(s, goalState, RC)$;
```

Fig. 5. A heuristic algorithm based on Lemma 3.10 to solve BAQA assuming that the goal is an *ap*-formula of the form either $G : [0, u]$ or $G : [\ell, 1]$ and that the state reachability predicate *reach* is specified as a set *RC* of reachability constraints.

where kidnap_performed(1) is an environment atom expressing that action kidnap(1) was taken.[5] Knowledge of action effects can clearly be represented with constraints built in a similar manner.

Algorithm *simpleAnnBAQA-Heur-RC* (Figure 5) is optimistic and assumes that Lemma 3.10 will yield at least one entailing formula for the goal; furthermore, it 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 solution states must satisfy (under the optimistic assumption); 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 5, the formula describes the fact that at least one of the states that make relevant entailing rules (as described in Algorithm *simpleAnnBAQA*) must be part of the solution; similarly, Step 6 builds a formula ensuring that none of the conflicting active rules can be relevant if the problem is to have a solution. Finally, Step 7 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 further constraints on the solution space under the assumptions being made by the algorithm. 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, and (ii) one based on a hill climbing strategy, whose pseudocode can be found in Figure 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

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

```
Algorithm 4: decideReachability-SAT($s, goalState, RC$)
(1) let $Reachable$ be a formula initialized to $formula(s)$;
(2) set Boolean variable $done := (Reachable \wedge goalState \not\models \bot)$;
(3) while not $done$ do
        set $Reachable_{old} := Reachable$;
        let $RC_{curr} \subseteq RC$ be the set of constraints $F_i \not\twoheadrightarrow G_i$ such that $Reachable \models F_i$;
        set $Reachable := \Big(\bigwedge_{F_i \not\twoheadrightarrow G_i \in RC_{curr}} \neg G_i\Big)$;
        set $done := \Big((Reachable \wedge goalState) \not\models \bot\Big) \vee \Big(Reachable \models Reachable_{old}\Big)$;
(4) return $(Reachable \wedge goalState \not\models \bot)$;
```

Fig. 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 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, that is, no new reachable states were discovered.

PROPOSITION 3.16. *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 $reach^*(s, s')$.*

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

Example 3.17. Consider the *ap*-program from Figure 1, along with constraint rc_1 from Example 3.15. As we saw in Example 3.13, if the goal is `kidnap(1)` : $[0, 0.6]$ 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 either $\{r_1\}$ or $\{r_1, r_4\}$ should be made relevant, which yields the following *goalState* formula:

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

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

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

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* takes a different approach. Rather than characterize the states that are eventually reachable from s and seeing if this 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

```
Algorithm 5: decideReachability-HillClimb($s, goalState, RC$)
(1) if $s \models goalState$ then return $true$;
(2) let $g$ be a state such that $g \models goalState$;
(3) set Boolean variable $done := false$;
(4) let $currState$ be a state initialized to $s$;
(5) while not $done$ do
        let $diff^+$ be the set of atoms $a_i$ such that $g \models a_i$ and $curr \not\models a_i$;
        let $diff^-$ be the 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 $F_i \not\mapsto G_i$ such that $currState \models F_i$;
        for each constraint $F_i \not\mapsto G_i \in RC_{curr}$ do
            set $Curr := \left(\bigwedge_{F_i \not\mapsto G_i \in RC \wedge currState \models F_i} \neg G_i\right)$;
        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 $currState := currState'$
(6) return $Curr \models goalState$;
```

Fig. 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.

and false in g and vice versa).[6] 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 led 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 is impossible for it to change atoms that are not part of $diff^+$ or $diff^-$. It is therefore a fast, but incomplete, heuristic algorithm. It is, however, sound, as the following proposition proves.

PROPOSITION 3.18. *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, that is, if it returns* true *then there exists a state s' such that $s' \models goalState$ and $reach^*(s, s')$.*

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

Example 3.19. Consider the same setup from Example 3.17. 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 the current state is:

$$s_0 = \{\texttt{forstpolsup}(1), \texttt{intersev1}(c), \texttt{intersev2}(1), \texttt{elecpol}(c), \texttt{extsup}(1), \texttt{demorg}(1)\}$$

[6]Note that focusing on a single state also allows it 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 3.10.

Algorithm	Reference	Comment
subProgramSearchBAQA	Alg.1, Fig.3, p. 9	Naïve approach: traverses the entire reachability graph checking for solution
simpleAnnBAQA	Alg.2, Fig.4, p. 10	Works on goals with $[0, u]$ or $[\ell, 1]$ annotations; leverages subprogram equiv. and heuristics for rule selection.
simpleAnnBAQA-Heur-RC	Alg.3, Fig.5, p. 12	Builds on top of Alg.2 to construct a formula describing all states eventually reachable from s_0.
decideReachabilitySAT	Alg.4, Fig.6, p. 13	Correctly decides eventual reachability given the formula built in Alg.3
simpleAnnBAQA-HillClimb	Alg.5, Fig.7, p. 14	Heuristic algorithm for deciding eventual reachability given the formula built by Alg.3

Fig. 8. A summary of the algorithms for BAQA.

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

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

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

Suppose the algorithm chooses to make forstpolsup(1) false from *diff*$^-$ and forstpolsup(0) true from *diff*$^+$ in the next step. This, however, does not satisfy *Curr*, which at this point is $\neg$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 intersev1(c) is not part of *diff*$^+ \cup$ *diff*$^-$, it can never change in *currState* and thus *Curr* can never be satisfied.

Example 3.20. Consider now the following *ap*-program:

$p \wedge q$	$: [0.3, 0.5]$	$\leftarrow a \wedge b.$
p	$: [0.3, 0.8]$	$\leftarrow a \wedge d.$
p	$: [0.1, 0.2]$	$\leftarrow a \wedge b \wedge c.$
q	$: [0.5, 0.9]$	$\leftarrow b.$
$s \wedge \neg p$	$: [0.8, 0.95]$	$\leftarrow b \wedge d.$

where $\mathcal{L}_{act} = \{p, q\}$ and $\mathcal{L}_{sta} = \{a, b, c, d\}$. Let the set of reachability constraints be:

$$\{c_1 : d \not\leftrightarrow a \wedge b, c_2 : b \not\leftrightarrow a, c_3 : a \not\leftrightarrow b, c_4 : \neg c \not\leftrightarrow d\}.$$

Suppose the *goalState* formula corresponding to goal $p : [0.25, 1]$ is:

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

and suppose the current state is $s_0 = \{\neg a, \neg b, c, d\}$. Suppose the hill climbing algorithm chooses state $g = \{a, b, \neg c, \neg d\}$ as the goal. Then, *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, that is, $a \wedge b$. 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. Figure 8 provides a summary of the algorithms discussed in this section, including brief comments based on the results presented earlier.

4. COST-BASED ABDUCTIVE QUERY ANSWERING

We now expand on the basic query answering problem we have described and assume that there are costs associated with transforming the current state into another state, and also an associated probability of success of this transformation; for instance, the fact that we may try to reduce foreign state political support for Hezbollah may only succeed with some probability. To model this, we use three functions:

Definition 4.1. 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 $cost_T : \mathcal{S} \times \mathcal{S} \to [0,\infty)$, with $cost_T(s,s') = \frac{cost(s')}{T(s,s')}$ whenever $T(s,s') \neq 0$, and ∞ otherwise[7].

The rationale behind this definition is that transitions with high probability of occurring are considered to be "easy," and therefore have a low associated cost.

Example 4.2 (*Transition Probabilities*). Suppose the only state predicate symbols are those that appear in the rules of Figure 1, and consider the set of states in Figure 2. 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 we have considered. 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 $seq = \langle s_0, s_1, \ldots, s_n\rangle$ can be defined in the following manner:

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

Note that Equation (1) is only one possible way of computing the cost of transitions through a sequence; the only hard requirement is that the function must be monotonic (the costs could, for instance, be additive instead of multiplicative). One way in which cost functions can be specified is in terms of *reward functions*.

Definition 4.3 (*Reward Functions*). An *action reward function* is a partial function $R : \mathit{APF} \to [0,1]$. 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 $E_{\Pi,R} : \mathcal{S} \to [0,\infty)$, defined as:

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

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. In this article, 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 paper, we assume that all transition cost functions are defined in terms of a canonical cost function.

Example 4.4. An example of an entailment-based reward function is as follows. Consider state s_2 from Figure 2, and annotated formulas $F_1 = \texttt{kidnap(1)} \wedge \texttt{tlethciv(1)}$:

[7]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.

$[0, 0.6]$, $F_2 = \texttt{kidnap}(1) : [0, 0.05]$, 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 4.3, $E_{\Pi,R}(s_2) = 0.2 + 0.14, 1 = 0.34$. Assuming $T(s_1, s_2) = 0.93$ as in Example 4.2, we have $cost_T(s_1, s_2) = \frac{0.34}{0.93} \approx 0.365$.

Definition 4.5. 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.

The main difference between the BAQA problem presented before 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 4.6. Consider 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), the current state is s_4, and $k = 3$. 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 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 3.3, which stated that the BAQA problem is EXPTIME-complete.

PROPOSITION 4.7. *CBQA is EXPTIME-complete.*

PROOF. Direct consequence of Proposition 3.3, by observing that BAQA is a special case of CBQA where the transition function 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). □

It follows directly from Corollary 3.5 and Proposition 3.6 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 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.

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

4.1. An Exact Algorithm for CBQA

We show that any CBQA problem can be mapped to a *Markov Decision Process* [Bellman 1957; Puterman 1994] problem. An instance of an MDP consists of: a finite set S of environment *states*; a finite set A of *actions*; a *transition function* $T : S \times A \rightarrow \Pi(S)$ specifying the probability of arriving at every possible state given that a certain action is taken in a given state; and a *reward function* $R : S \times A \rightarrow \mathbb{R}$ specifying the expected immediate reward gained by taking an action in a state. The objective is to compute a policy $\pi : S \rightarrow A$ 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.[8] 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 $\langle s_0, s_1, \ldots, s_k \rangle$ is the result of *following* a policy π if $\pi(s_i) = a_{i+1}$, where $0 \le i < k$ and $a_{i+1} \cong s_{i+1}$.

PROPOSITION 4.8. *Let* $O = \left(\Pi, \mathcal{S}, s_0, G : [\ell, u], cost, T, E_{\Pi,R}, k\right)$ *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.

[8]Note that here actions refer to the point of view of the reasoning agent, who desires to act over the environment in order to influence the agent being modeled by the *ap*-program.

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

$$\pi(s) = \arg\max_{a} \left(R_{MDP}(s,a) + \max_{a'} \left(\sum_{s' \in S} T_{MDP}(s,a,s') \cdot Q(s',a') \right) \right) \tag{6}$$

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 (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 [Bellman 1957] iterates over the entire state space a number of times that is polynomial in $|S|$, $|A|$, β, and B, where β is the discount factor and B is an upper bound on the number of bits that are needed to represent any numerator or denominator of β [Littman 1996]. 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 given 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 [Boutilier et al. 2000; Tsitsiklis and van Roy 1996], similar techniques for *action aggregation* have not been developed.

```
Algorithm 6: DE_CBQA($\Pi, G : [\ell, u], s_0, T, h, k, numIter, giveUp$)
 1. Initialize set of states $\mathcal{S}_G$:= $getGoalStates(\Pi, G : [\ell, u])$;
 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. 9. An algorithm for CBQA based on probability density estimation.

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) [de Bonet et al. 1996; Pelikan et al. 2002]. 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 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 9) 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 3 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,[9] starting out as 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).

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

An attractive feature of DE_CBQA is that it is an anytime algorithm, that is, 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 will show how the algorithm works shortly.

We first show how the probability distribution P in the DE_CBQA algorithm can be represented, and how this affects the algorithm.

4.2.1. Representing the Probability Distribution via a Probability Vector. One of the simplest ways in which we can represent a probability distribution over sequences of states is by means of a *probability vector* that has one component per possible state in the sequence. The basic idea in this approach is for the element at position i in the vector to represent the proportion of "good" samples seen so far that contained state i. One of the main drawbacks of this approach is its lack of consideration for any dependencies among the probabilities being represented. However, as we will see in the results of our experimental evaluation in Section 6, the results obtained are of reasonable quality. The following is an example of the probability vector approach.

Example 4.9. 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 $\texttt{kidnap}(1) : [0, 0.6]$; 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, computed according to Equation (1) 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 vector P with four components (the starting state is not included) that starts out as $(0.25, 0.25, 0.25, 0.25)$, representing the probabilities that s_1, s_2, s_3, and s_5 will be sampled, respectively.

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 $10^{21.71}$ (once again, cf. Equation (1) to see how these values were obtained). 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 $(0.1, 0.5, 0.5, 0.9)$ (it doesn't reduce the probability of s_1 to zero, nor does it push that of p_5 all the way to one). Thus, the algorithm has seen that s_5 seems to be a good state to include, since both of the "good" sequences involved it. 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.

4.2.2. Representing the Probability Distribution via a Bayesian Network. While the probability vector 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, suppose 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 reasonable to believe that real instances of CBQA will exist 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 [Pearl 1988], a directed acyclic graph modeling conditional dependencies among random variables. In our case, each node in the network 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 [Cooper and Herskovits 1992]. 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 (and, of course, independencies) between states, their ordering, and position. The following is an example of the Bayesian Network approach based on Example 4.9.

Example 4.10. Suppose we have the same setup as in Example 4.9; this time, the probability distribution over possible sequences is a Bayesian network that has three nodes (for now, since we are sampling sequences of length three). The prior probability distribution on each of these nodes is the uniform distribution over its possible values, that is, all possible states excluding the initial state. As before, suppose we start by sampling the same sequences as in the first round: $H = \{\langle s_4, s_5, s_3\rangle, \langle s_4, s_5, s_2\rangle, \langle s_4, s_1, s_3\rangle\}$.

Now, the algorithm can update the prior probability table for the first position in the sequence to heavily favor sampling state s_5. Even though this is similar to what happened in the probability vector case, the representation is now rich enough to state that what it has learned is that starting with s_5 yields good results, whereas the probability vector could only represent that its participation in sequences lead to favorable results (without really having good evidence to support this). Similarly, based on this round of sampling, the Bayesian network could be updated to represent the fact that s_3 and s_2 should have higher chances of being sampled for the second position given that s_5 was sampled for the first.

Even though this is a simple example, it clearly illustrates the difference in representational power between the two approaches. In Section 6, we present the results of our experimental evaluation of the DE_CBQA algorithm under both approaches for representing the probability distribution, comparing it first to an exact solver and then investigating its scalability. The computational cost and the accuracy of both approaches will also be compared.

```
Algorithm 7: PAR_getGoalStates($\Pi, G : [\ell, u], N, giveUp$)
 1. Initialize set of states $S_G := \emptyset$;
 2. Initialize set of rules $HeurRules := \emptyset$;
 3. Execute Steps 2, 3, and 4 of simpleAnnBAQA;
 4. for $r_i \in candAct(\Pi, G : [\ell_G, u_G])$ with $Head(r_i) = F : [\ell_F, u_F]$ do
 5.     if $[u_G = 1, F \models G$, and $\ell_G \leq \ell_F]$ or $[\ell_G = 0, G \models F$, and $u_G \geq u_F]$ then
 6.         add $r_i$ to HeurRules;
 7. BatchSize := $\left\lceil \frac{\lfloor 2^{\mathcal{L}_{sta}} \rfloor}{N} \right\rceil$;
 8. while !giveUp do
 9.     for parallel processes $n := 0$ to $N - 1$ do
10.         foreach $s_i \in 2^{\mathcal{L}_{sta}}$ where $i := (BatchSize * n)$ to $[(BatchSize * n) + BatchSize - 1]$ do
11.             if $\Pi_{s_i} \models HeurRules$ and $HeurRules \models G : [\ell_G, u_G]$ then
12.                 add $s_i$ to $S_G$;
13. return $S_G$;
```

Fig. 10. A parallel algorithm for finding entailing states for the CBQA problem.

5. PARALLEL SOLUTIONS FOR ABDUCTIVE QUERY ANSWERING

In the previous section, we presented algorithms for answering both basic and cost-based abductive queries, along with several heuristic approaches to improve the tractability of these computations. However, we can make further gains in scalability and computation time by identifying portions of these problems to compute in parallel. In this section, we present two explicitly parallel algorithms for solving CBQA problems. One algorithm will search for potential entailing states in parallel, allowing us to either examine more possible states, or to improve the running time of finding an entailing state. In addition, the iterative sampling for CBQA can be made more effective by parallelizing the sampling process, allowing for a more comprehensive search over the possible paths to goal states.

5.1. Parallel Selection of Entailing States

Recall the DE_CBQA algorithm in Figure 9 and the *getGoalStates* function invoked in line 1; this function returns entailing states, that is, states s s.t. $\Pi_s \models \Pi'$. In practice, as we will see in Section 6, the large search space makes it intractable to find all such states, and so the number of goal states returned must be limited by the user. The implementation of *getGoalStates* that we developed for our experimental evaluation iteratively goes through potential goal states until one is found; the heuristic methods shown in Algorithm *simpleAnnBAQA-Heur-RC* (Figure 5) are used to make quick (sound, but not complete) entailment checks.

Rather than looking at potential goal states in sequence, we can parallelize this procedure. Figure 10 contains a distributed version of *getGoalStates* called *PAR_getGoalStates* that will divide the state space and check for entailing states in parallel over N processors.

The DE_CBQA algorithm can now be run with *PAR_getGoalStates* in line 1. With this method, the user can specify some termination condition *giveUp* (e.g., the number of goal states to find, the amount of search time, etc.) for the concurrent search for entailing states. In lines 9 and 10, we divide the state space $2^{\mathcal{L}_{sta}}$ across N processors, and iterate through each batch in parallel to find entailing states until the *giveUp* condition is true.

If the size of S_G is still limited to a single goal state, then *PAR_getGoalStates* can provide a direct speedup of the original method, using the distributed computation to more quickly identify an entailing state. However, we can also take advantage of the

```
Algorithm 8: ParSample_DE_CBQA($\Pi, G : [\ell, u], s_0, T, h, numIter, giveUp, N$)
 1. Initialize set of states $\mathcal{S}_G$:= 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. $P :=$ new uniform probability distribution over sequences($\mathcal{S}'$);
 6. BatchSize := $\lceil \frac{h}{N} \rceil$;
 7. while !giveUp do
 8.    $j := j + 1$;
 9.    for $i = 1$ to numIter do
10.      for parallel processes $n := 0$ to $N - 1$ do
11.         foreach sample $k$ from $i := (BatchSize * n)$ to $[(BatchSize * n) + BatchSize - 1]$ do
12.           if $k > h$ then END;
13.           randomly sample (using $P$) a sequence $\phi$ of states of length $j$ starting at $s_0$
                 and ending at some $s_G \in \mathcal{S}_G$;
14.           add $\phi$ to $H$
15.      rank each sequence $\phi$ with $cost^*_{seq}(s_0, \phi(j))$;
16.      pick the sequence in $H$ with the lowest cost $c^*$, call it $\phi^*$;
17.      if $c^* < c_{best}$ then $\phi_{best}$:= $\phi^*$; $c_{best}$:= $c^*$;
18.    $P$:= generate new distribution based on $H$;
19. return $\phi_{best}$;
```

Fig. 11. A synchronized parallel algorithm for CBQA using iterative distributive sampling.

parallelization to find a larger number of goal states to test in the DE_CBQA algorithm, rather than simply looking at the first state found.

5.2. Parallel Sampling of State Paths

The sampling method in the DE_CBQA algorithm allows the user to specify the number of possible paths to examine to reach a particular goal state. In practice, the space of possible paths from the initial state to a goal state can be very large, and random sampling may not reliably be able to find a low-cost option within a tractable computation time. In Figure 11 we present a distributed algorithm, *ParSample_DE_CBQA*, that will divide the iterative sampling of state paths across N processors.

Lines 1–8 of *ParSample_DE_CBQA* are identical to the serial version of this algorithm. However, beginning in line 9 we divide the number of sequences per iteration into batches for parallel sampling. Working in batches of size N (the number of available processors) we distribute the sampling of h paths from s_0 to each goal state. After each batch completes, we compare the costs of the sampled paths and record the optimal sequence. This result is updated after each iteration of distributed samples, making this an anytime parallel algorithm over N processors. *ParSample_DE_CBQA* also synchronizes the state sequences probability distribution with results from all of the parallel samples, and each node will use the improved distribution in the next iteration of sampling.

In some cases, this synchronization may require prohibitive amounts of communication overhead in comparison to the computation time necessary for sampling sequences. As an alternative, *ParSampleAsynch_DE_CBQA* (Figure 12) is another distributed algorithm for solving the CBQA problem that does not synchronize the probability distributions. In *ParSampleAsynch_DE_CBQA*, each of the N parallel nodes performs a separate round of iterative sampling, maintaining its own sequence probability distribution and returning the best sequence resulting from these samples. Then, in line 16, we return the overall ϕ_{best} sequence from each of the distributed samples. While this asynchronous computation is not the same as increasing the

```
Algorithm 9: ParSampleAsynch_DE_CBQA($\Pi, G : [\ell, u], s_0, T, h, numIter, giveUp, N$)
 1. Initialize set of states $\mathcal{S}_G$:= 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. foreach parallel process $n := 0$ to $N - 1$ do
 4.     let $\phi_{best}$ be the two-state sequence that has the lowest cost, denoted $c_{best}$;
 5.     let $\mathcal{S}' = \mathcal{S} - \mathcal{S}_G - \{s_0\}$;     set $j := 2$;
 6.     $P$ := new uniform probability distribution over sequences($\mathcal{S}'$);
 7.     while !giveUp do
 8.         $j := j + 1$;
 9.         for $i = 1$ to numIter do
10.             randomly sample (using $P$) a set $H_n$ of $h$ sequences of states of length $j$ starting
                        at $s_0$ and ending at some $s_G \in \mathcal{S}_G$;
11.             rank each sequence $\phi$ with $cost^*_{seq}(s_0, \phi(j))$;
12.             pick the sequence in $H_n$ with the lowest cost $c^*$, call it $\phi^*$;
13.             if $c^* < c_{best}$ then $\phi_{n-best}$:= $\phi^*$; $c_{n-best}$:= $c^*$;
14.             $P$:= generate new distribution based on $H_n$;
15.     add $\phi_{n-best}$ to $H_{total}$;
16. return sequence $\phi_{best}$ in $H_{total}$ with lowest cost $c_{best}$;
```

Fig. 12. An asynchronous parallel algorithm for CBQA using iterative distributive sampling.

number of samples by a factor of N, as we are not using all samples to update the probability distribution, it does facilitate better coverage of the possible sequences. Because of this, we are more likely to find better sequences, and may be able to achieve this result with a fewer number of samples per iteration. We can of course also use the parallel version of *getGoalStates*, described before, along with either concurrent iterative sampling algorithm to further improve performance and results.

6. EXPERIMENTAL RESULTS

In this section, we will report on a series of experimental evaluations that we carried out on the algorithms presented in Sections 3, 4, and 5. 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[10]. 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.

6.1. Empirical Evaluation of Serial Algorithms for BAQA

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

[10]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.

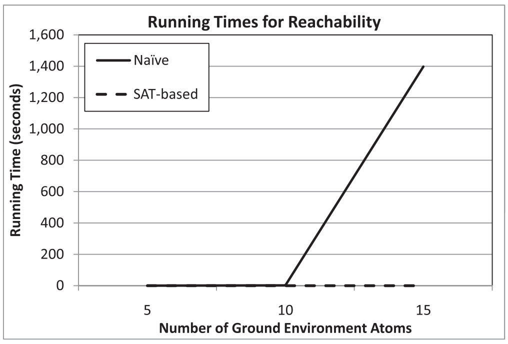

Fig. 13. Varying number of ground state atoms for programs with 5 rules, 25 ground action atoms, 5 reachability constraints, and atomic queries.

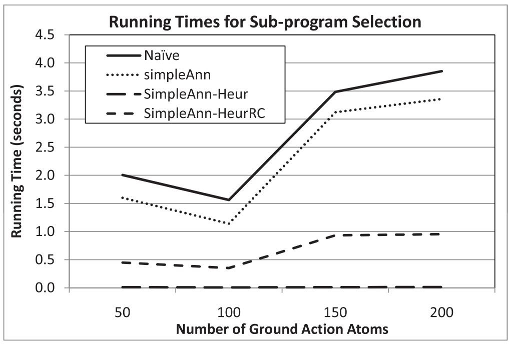

Fig. 14. Varying the number of ground action atoms for *ap*-programs with 5 rules, 5 ground state atoms, and non-atomic queries.

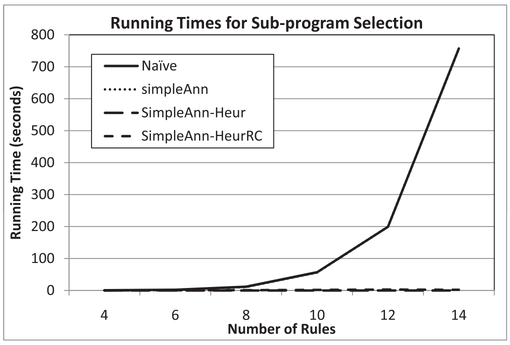

Fig. 15. Varying number of rules; 25 ground action atoms, 5 ground state atoms, and atomic queries.

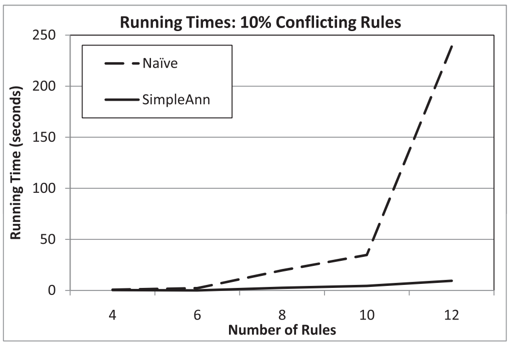

Fig. 16. Varying number of rules (with 10% of them goal-conflicting); 25 ground action atoms, 5 ground state atoms, and atomic queries.

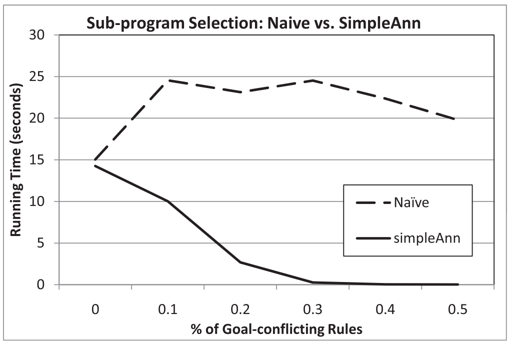

Fig. 17. Varying the percentage of rules that are in conflict with the goal; *ap*-programs with 10 rules.

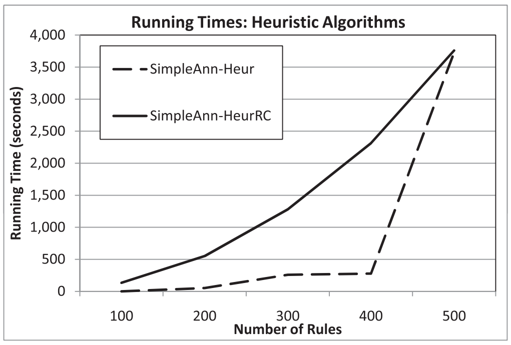

Fig. 18. Varying number of rules (larger *ap*-programs); 25 ground action atoms, 5 state atoms, and 5 reachability constraints.

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 reachability constraints, number and size of clauses in rule heads and reachability constraints, number of rules, and number and size of clauses in goals. 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, e.g., the number of ground state atoms in Figure 14, or the number of ground action atoms in Figure 15).

Each *ap*-program used in the experiments consists of a set of randomly generated *ap*-rules, each with a randomly generated head and body. The head consists of either one or two clauses of length at most two variables each, with conjunction, disjunction, and negation connectors chosen uniformly at random. The head is nontrivial; it is guaranteed to have at least one atom in at least one clause. Each *ap*-rule's body is generated by randomly selecting a conjunction of two atoms. The goal *ap*-formula is generated in a similar fashion, but with randomly generated upper and lower bounds. When experiments require a threshold goal, either the upper bound is set to 1 or the lower bound is set to 0.

No. of State Atoms. In Figure 13 we show the running times of the different approaches to deciding reachability; the naïve approach becomes intractable very quickly, while the (still exact) SAT-based algorithm approach has negligible cost for these runs.

No. of Action Atoms. Figure 14 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 naïve 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 15 reports the running times of the *SimpleAnn* rule selection algorithms, where *SimpleAnn-Heur* refers to the optimistic application of the *entailing rules* heuristic used in algorithm SimpleAnn (based on Lemma 3.10). We can see that both the naïve 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 naïve 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 16 and 17. The former shows the same experiment as Figure 15 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 naïve algorithm.

The last set of experiments are presented in Figure 18, which shows the running times for the SimpleAnn heuristic step (that is, assuming the algorithm only tries to apply the entailing rules heuristic and returns *false* otherwise) and the SimpleAnnBAQA-Heur-RC algorithm for larger programs. It is interesting to see the different shapes of the curves: as programs get larger, the SAT formulas associated with SimpleAnnBAQA-Heur-RC 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. An interesting topic for future work is to extend this experimental study to investigate which parameters have the most influence over the precision of our heuristics.

6.2. Empirical Evaluation of Serial Algorithms for CBQA

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 performed on Windows 7 Ultimate 64-bit OS, and made use of a single core.

First, we compare the running 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. As we have discussed, 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.

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 4.1, 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 running time 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 $O(|S_{MDP}|^2 \cdot |A_{MDP}|)$ 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

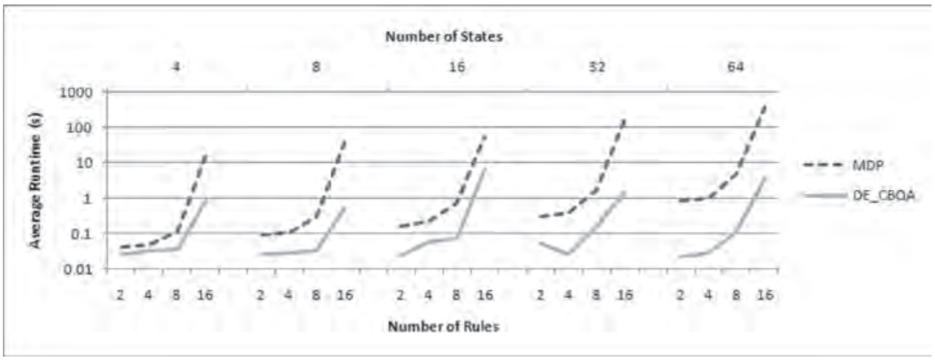

Fig. 19. Log-scale running time comparison of MDP and DE_CBQA, shown with increasing state size (top axis) for each of 2, 4, 8, and 16 rules (bottom axis). Note the sharp jump in running time as the number of rules increases compared to the gradual upward trend as the number of states rises.

multiple runs of the MDP and DE_CBQA algorithms. We varied the discount factor γ and maximum error ϵ for the MDP[11], while exploring different completion predicates, maximum and minimum sequence lengths, and number of iterations per sequence length for DE_CBQA. We now provide an overview of the results we obtained.

Figure 19 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 running times between the two algorithms increases with the number of states, with DE_CBQA maintaining nearly constant running 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 3). 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[12]; 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 running 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 in Figure 9) did not affect sequence cost; however, increasing the number of samples per iteration (line 9 in Figure 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 in Figure 9) did not benefit the final result.

Finally, we tried using Policy Iteration [Tseng 1990] instead of Value Iteration to solve the MDP; however, this method was either slower than Value Iteration or, if

[11]Given γ and ϵ, one can calculate an error threshold that guarantees an optimal policy [Williams and Baird 1994].

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

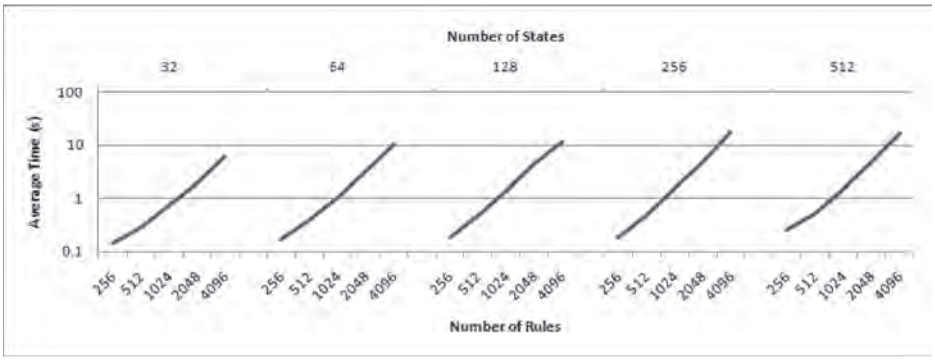

Fig. 20. Log-scale running time as DE_CBQA scales with respect to number of states (top axis) and number of rules (bottom axis). Note the addition of extra rules slows down algorithm execution time much more significantly than a similar increase in state space size.

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

Fig. 21. *Towards the limits of our current implementation.* Timing results taken by maximizing an individual parameter (the numbers in bold font). The size of the state space was limited by system memory in this implementation.

faster, forced to use such a low discount factor γ and error limit ϵ that following the resulting policy often yielded a *worse* sequence than DE_CBQA's recommendation, at a slower speed!

Scaling the Heuristic DE_CBQA Algorithm. As we have seen, 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 the same small change to the algorithm that we used in Section 6.1: whenever no goal states are found with the fast heuristics (line 1), it fails to return an answer; that is, it takes a pessimistic approach (which can also be seen as an optimistic application of the heuristic).

Figure 20 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 running 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 running time of the DE_CBQA algorithm is only as related to the size of the state space as its probability distribution requires. 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 [Khuller et al. 2007]. For these cases, DE_CBQA scales quite well. Finally, Figure 21 provides timing results for extreme values of each individual parameter; in each case, the other parameters were kept at more manageable values.

Toward Better Sampling. While the probability vector approach is neither memory nor computationally intensive, it ignores any subtle relationships that may exist

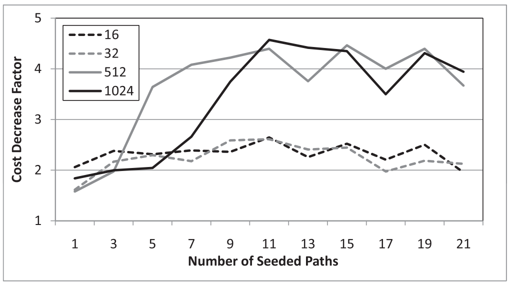

Fig. 22. Varying the number of seeded paths with a small (e.g., 16 or 32) number of states versus a larger (e.g., 512 or 1024) state space.

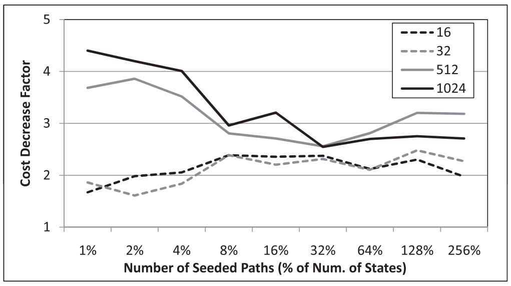

Fig. 23. Varying the number of seeded paths (and thus the level of conditional dependence in the world) as a percentage of the total number of states.

between individual states or their ordering in the overall sequence. 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 [Hall et al. 2009]) 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 22 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 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 on.

Figure 23 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 22, 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.

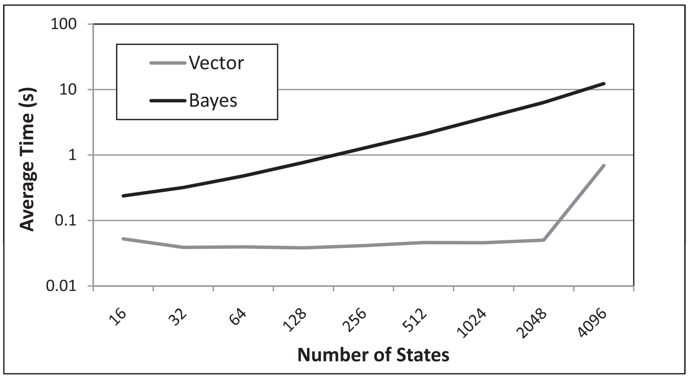

Fig. 24. Log-scale running time as both the Bayesian and vector-based DE_CBQA algorithms scale. Note the linear increase in Bayesian running time caused by structure learning, storage, and sampling overhead.

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 24 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 *running time* 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. Fortunately, 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 25 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 on overall running time than the number of states. We see that the algorithm scales gracefully to large state/action spaces. As we mentioned before, in our experience, real-world instances of CBQA tend to contain significantly fewer rules than states and actions [Khuller et al. 2007]; as such, in these cases DE_CBQA scales quite well.

6.3. Empirical Evaluation of Parallel Algorithms for CBQA

We implemented all three parallel algorithms using the Java Remote Method Invocation interface for distributed computation, and tested them on 15 nodes of a compute cluster, each with four 3.4 GHz cores and 8 GB RAM; the *ap*-programs and goals were generated in the same way as in the serial experiments (cf. Section 6.1).

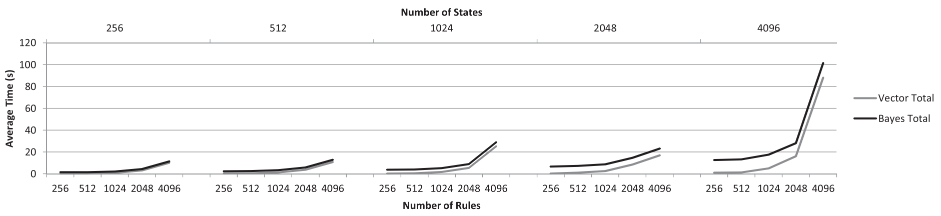

Fig. 25. Running time comparison as DE_CBQA scales with respect to number of states (top axis) and number of rules (bottom axis). Note the similarity in running time between the Bayesian and vector probability models.

First, we compare the running time of the serial and parallel methods for finding entailing states to use in the DE_CBQA computation and demonstrate how *PAR_getGoalStates* can provide significant savings overall. We then compare the performance of the parallel algorithms for iterative sampling in DE_CBQA. Unfortunately, due to the synchronization and communication overhead associated with our particular implementation, the *ParSample_DE_CBQA* algorithm is quite intractable in practice, often taking 5 times the amount of running time for DE_CBQA using the naïve vector distribution, and up to 35 times the running time for the Bayesian network distribution. The asynchronous version of this algorithm, *ParSampleAsynch_DE_CBQA*, is however able to concurrently run the DE_CBQA algorithm with only minimal impact from the communication required to initialize the problem and obtain the overall best sequence.

Second, we compare the quality of the sequences returned by the asynchronous parallel sampling algorithm and the serial DE_CBQA computation. Using 1,024 samples per iteration as a baseline for the serial algorithm, we run both algorithms over large rule and state spaces, varying the number of parallel samples per iteration. Because the parallel method takes distinct samples in parallel, it is able to explore more of the state space and find better sequences with a fewer number of samples.

Parallel methods for finding entailing states. We performed two experiments to determine the effectiveness of *PAR_getGoalStates* as compared to the serial *getGoalStates* method. The default size of S_G (the set of goal states) in *getGoalStates* is either the total number of possible states or 50, whichever value is smaller. The first experiment uses this same cap of 50 entailing states, varying the number of states between 16 and 4,096 and the number of rules from 256 to 4,096. The parallel algorithm effectively divides the state space to find goal states concurrently, consistently running much more efficiently than the serial version (Figure 26). For 4,096 states and rules, the parallel entailment method requires 4.96 seconds, whereas serial selection is 20 times slower, taking 100.8 seconds. Furthermore, as shown in Figures 27 and 28, the computation time required by the parallel algorithm increases only very slowly as the number of states and rules increase, indicating that this method will scale to a much larger number of states and larger programs. Because the entailment time is often a significant portion of the DE_CBQA algorithm's running time, especially for large state-spaces, the parallel method provides significant overall savings.

Parallel Iterative Sampling. As discussed before, the communication and synchronization overhead required for the ParSample_DE_CBQA algorithm is far too costly in practice to make this method useful. However, empirical tests showed that the performance of the asynchronous parallel algorithm is very good with respect to the serial DE_CBQA algorithm. In Figure 29, the running time of ParSampleAsynch_DE_CBQA is compared with that of the serial version for both the vector and Bayesian distribution methods. For the parallel computations, we performed 60 concurrent rounds of

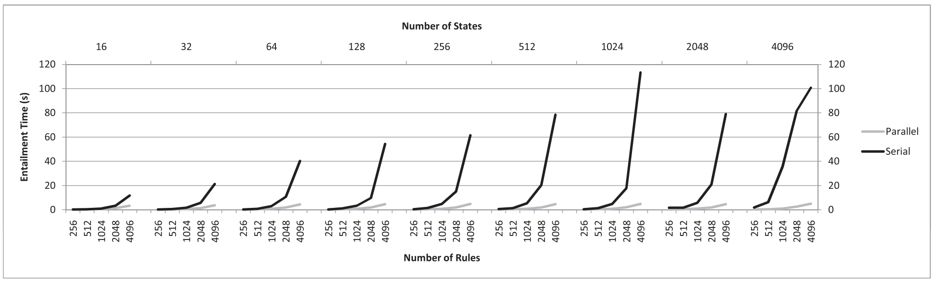

Fig. 26. Running time for both the parallel *PAR_getGoalStates* and serial *getGoalStates* methods to find up to 50 entailing states.

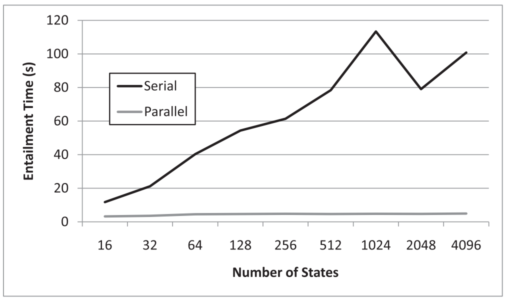

Fig. 27. Running time for both the parallel *PAR_getGoalStates* and serial *getGoalStates* methods to find entailing states. The number of states was varied between 16 and 4,096, and the number of rules held constant at 4,096.

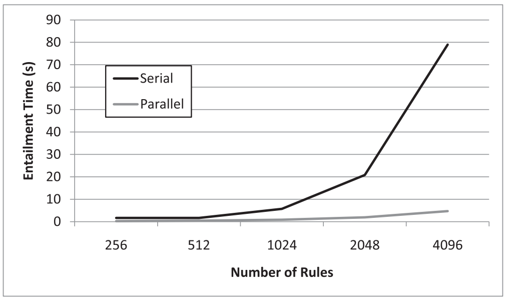

Fig. 28. Running time for both the parallel *PAR_getGoalStates* and serial *getGoalStates* methods to find entailing states. The number of rules was varied between 256 and 4,096, and the number of states held constant at 2,048.

the DE_CBQA iterative sampling, using all 4 cores on each of 15 nodes of the compute cluster. When using the probability vector representation, the communication required to set up the remote computations and combine the final results still dominates the computation even in the asynchronous sampling; in many cases the parallel version takes at least twice as long. However, this difference is much smaller in the case of the Bayesian algorithm. A two-sample *t*-test at the 95% confidence level indicates with a very high p-value of 0.8881 that there is no significant difference between the running times of the parallel and serial algorithms with the Bayesian distribution.

Even though we are not synchronizing the updated probability distributions, the ParSampleAsynch_DE_CBQA algorithm is capable of computing multiple concurrent rounds of sampling, providing potentially greater coverage of the possible state sequences. This expanded sampling ability is able to provide better quality (i.e., lower cost) result sequences than the standard serial version. Figure 30 compares the average cost ratio of sequences found by the serial and parallel sampling algorithms, where the cost decrease factor is defined as $\frac{\text{Cost of sequence found by serial}}{\text{Cost of sequence found by parallel}}$. In this experiment, we used 1,024 serial samples as our baseline, and varied the number of parallel samples from 32 to 1,024 for a large number of states (2^{12}) and rules (1,024) using both the naïve vector and Bayesian net distributions. As before, this experiment utilizes all 4 cores on each of 15 nodes in a cluster. With as few as 64 samples taken by each of these 60 processors, both the vector and Bayes versions of ParSampleAsynch_DE_CBQA achieve a quality almost at parity with 1,024 serial samples, with a cost decrease

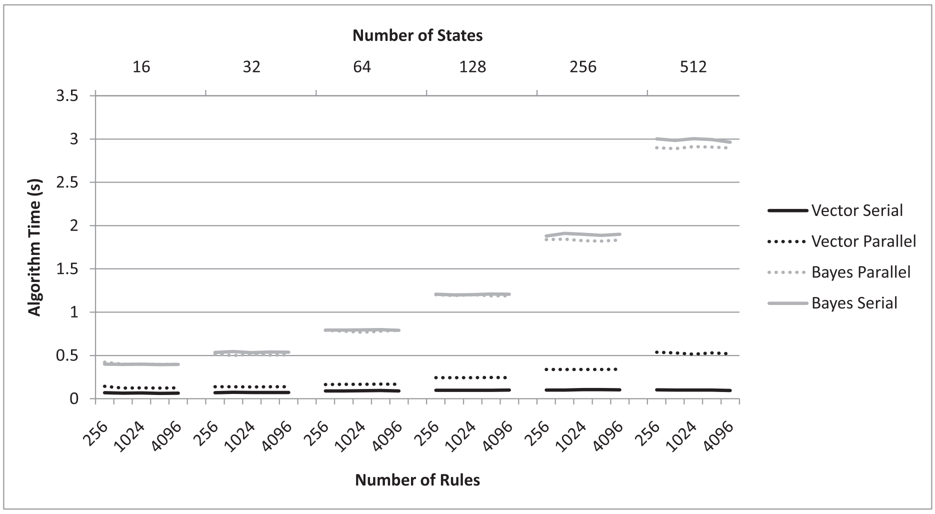

Fig. 29. Running time comparison of ParSampleAsynch_DE_CBQA and serial DE_CBQA using both the vector and Bayesian distributions.

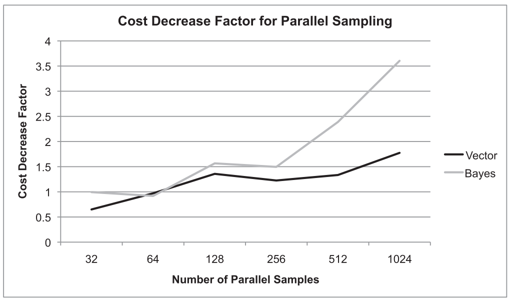

Fig. 30. Cost comparison of algorithms ParSampleAsynch_DE_CBQA and serial DE_CBQA using the vector and Bayesian distributions. The number of serial samples per iteration were held constant while the parallel samples per iteration were varied. The speedup factor measures the ratio of the serial best sequence cost to the parallel best sequence cost.

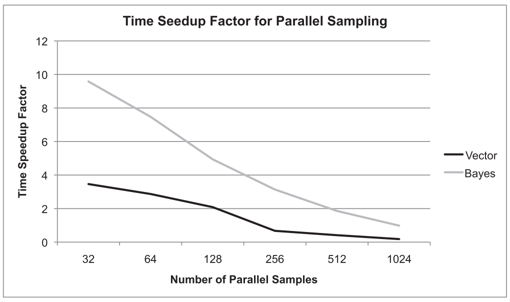

Fig. 31. Running time speedup of algorithm ParSampleAsynch_DE_CBQA versus serial DE_CBQA using both the vector and Bayesian distributions. The number of serial samples per iteration were held constant while the parallel samples per iteration were varied. The speedup factor measures the ratio of the serial running time to the parallel running time.

factor of about 0.971 and 0.918 respectively. At 128 samples, both algorithms surpass the quality of the serial DE_CBQA and find sequences with much lower costs.

The overall efficiency gains of the ParSampleAsynch_DE_CBQA algorithm are illustrated in Figure 31. Using the same parameters as the quality experiments just described, we also compared the running times of these algorithms. Not only does this method provide improved quality in the same amount of time for the default number of 1,024 samples—a 3.6 cost decrease factor for the Bayesian distribution (Figure 30)—but we can achieve greater quality in a shorter period of time. For example, taking 128 parallel samples and using the Bayesian distribution, ParSampleAsynch_DE_CBQA requires only about $\frac{1}{5}$ the time as DE_CBQA for 1,024 samples (16.26 and 3.29 seconds, respectively), but is able to find sequences with an average cost decrease factor of 1.57.

7. RELATED WORK

Abduction has been extensively studied in diagnosis [Console and Torasso 1991; Peng and Reggia 1990], reasoning with non-monotonic logics [Eiter and Gottlob 1995; Eiter et al. 1997b], probabilistic reasoning [Bhatnagar and Kanal 1993; Josang 2008; Pearl 1991; Poole 1993, 1997], argumentation [Kohlas et al. 2002], planning [Eshghi 1988; Shanahan 2000], and temporal reasoning [Eshghi 1988]; furthermore, it has been combined quite naturally with different variants of logic programs [Baldoni et al. 1997; Christiansen 2008; Denecker and Kakas 2002; Eiter et al. 1997a; Kakas et al. 2000]. 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*. An 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 aspects are defined has led to many different models.

David Poole and others combined probabilistic and non-monotonic reasoning, leading to the development of Probabilistic Horn Abduction [Poole 1993], and eventually the Independent Choice Logic [Poole 1997]. Christiansen [2008] addresses 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 [Poole 1997] or [Christiansen 2008]) or are based on the class of graphical models including Bayesian Networks (BNs). 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 represents *independence*. Joint probability distributions can therefore be obtained from the graph, and abductive reasoning is carried out by applying Bayes's theorem given these joint distributions and a set of 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). The main difference between graphical model-based work and our work 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 this paper). This would cause states to be potentially exponentially larger than those in this paper and would also exponentially increase their number.

8. CONCLUSIONS

There are many applications where we need to reason about the behaviors of actors about whom we can learn probabilistic rules of behavior. Examples of such applications include the modeling of terror groups [Mannes et al. 2008a, 2008b], or the modeling of animal groups (e.g., groups of gorillas that exhibit behaviors such as avoidance of other gorilla groups, attacks on other gorilla groups, and so forth) [Bryson et al. 2007]. The US Treasury, for instance, is interested in modeling behaviors of investor groups to learn their attitudes towards risk under different conditions. Governments are interested in the impact of policies on groups (e.g., farmers). In many cases, we would like to influence these behaviors by understanding what actions we can take to ensure that the probability that a desired outcome occurs exceeds some threshold. This is further complicated by the fact that groups do not take actions "one at a time", but these actions are often correlated and planned and, furthermore, the effects of these actions are not well understood.

We have formulated these problems via the Basic Abductive Query Answering (BAQA) and Cost-based Query Answering (CBQA) problems. We have studied the computational complexity of both these algorithms and developed exact algorithms, as well as heuristic algorithms that are relatively fast and sound, though not complete. We have developed innovative algorithms that maintain and update probability distributions as they run, allowing better estimation of solutions while reducing running times. Finally, a further important contribution is the first parallel algorithm for abduction in probabilistic logic. We present two parallel algorithms for CBQA and show that one of them works very well in practice.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers of this manuscript for their insightful comments, which helped improve this work.

REFERENCES

Asal, V., Carter, J., and Wilkenfeld, J. 2008. Ethnopolitical violence and terrorism in the Middle East. In *Peace and Conflict 2008*, J. Hewitt, J. Wilkenfeld, and T. Gurr Eds., Paradigm.

Baldoni, M., Giordano, L., Martelli, A., and Patti, V. 1997. An abductive proof procedure for reasoning about actions in modal logic programming. In *Selected Papers from the Workshop on Non-Monotonic Extensions of Logic Programming (NMELP'96)*. Springer, 132–150.

Bellman, R. 1957. A Markovian decision process. *J. Math. Mech. 6*.

Bhatnagar, R. and Kanal, L. 1993. Structural and probabilistic knowledge for abductive reasoning. *IEEE Trans. Pattern Anal. Mach. Intell. 15,* 3, 233–245.

Boutilier, C., Dearden, R., and Goldszmidt, M. 2000. Stochastic dynamic programming with factored representations. *Artif. Intell. 121,* 1–2, 49–107.

Bryson, J. J., Ando, Y., and Lehmann, H. 2007. Agent-based modelling as scientific method: A case study analysing primate social behaviour. *Philos. Trans. R. Soc. London, Ser. B 362,* 1485, 1685–1698.

Christiansen, H. 2008. Implementing probabilistic abductive logic programming with constraint handling rules. In *Constraint Handling Rules*, T. Schrijvers and T. W. Frühwirth Eds., Lecture Notes in Computer Science Series, vol. 5388, Springer, 85–118.

Chvtal, V. 1983. *Linear Programming*. W.H.Freeman, New York.

Console, L. and Torasso, P. 1991. A spectrum of logical definitions of model-based diagnosis. *Comput. Intell. 7,* 3, 133–141 .

Cooper, G. and Herskovits, E. 1992. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning 9,* 4, 309–347.

de Bonet, J. S., Isbell, C. L. Jr., and Viola, P. A. 1996. MIMIC: Finding optima by estimating probability densities. In *Proceedings of the Conference on Advances in Neural Information Processing Systems (NIPS'96)*. MIT Press, 424–430.

Denecker, M. and Kakas, A. C. 2002. Abduction in logic programming. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, Springer, 402–436.

Eiter, T. and Gottlob, G. 1995. The complexity of logic-based abduction. *J. ACM 42,* 1, 3–42.

Eiter, T., Gottlob, G., and Leone, N. 1997a. Abduction from logic programs: Semantics and complexity. *Theor. Comput. Sci. 189,* 1–2, 129–177.

Eiter, T., Gottlob, G., and Leone, N. 1997b. Semantics and complexity of abduction from default theories. *Artif. Intell. 90*, 90–1.

Eshghi, K. 1988. Abductive planning with event calculus. In *Proceedings of the International Conference on Logic Programming*. 562–579.

Fagin, R., Halpern, J. Y., and Megiddo, N. 1990. A logic for reasoning about probabilities. *Inf. Comput. 87,* 1/2, 78–128.

Giles, J. 2008. Can conflict forecasts predict violence hotspots? *New Scientist* 2647.

Hailperin, T. 1984. Probability logic. *Notre Dame Journal of Formal Logic 25*, 3, 198–212.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. 2009. The WEKA data mining software: An update. *ACM SIGKDD Explor. Newsletter 11,* 1, 10–18.

Josang, A. 2008. Abductive reasoning with uncertainty. In *Proceedings of the Conference on Information Processing and Management of Uncertainty*, L. Magdalena, M. Ojeda-Aciego, and J. L. Verdegay Eds., 9–16.

Kakas, A., Michael, A., and Mourlas, C. 2000. ACLP: Abductive constraint logic programming. *J. Logic Program. 44*, 129–177(49).

Kern-Isberner, G. and Lukasiewicz, T. 2004. Combining probabilistic logic programming with the power of maximum entropy. *Artif. Intell. 157,* 1–2, 139–202.

Khuller, S., Martinez, M. V., Nau, D. S., Sliva, A., Simari, G. I., and Subrahmanian, V. S. 2007. Computing most probable worlds of action probabilistic logic programs: Scalable estimation for $10^{30,000}$ worlds. *Ann. Math. Artif. Intell. 51,* 2–4, 295–331.

Kohlas, J., Berzati, D., and Haenni, R. 2002. Probabilistic argumentation systems and abduction. *Ann. Math. Artif. Intell. 34,* 1–3, 177–195.

Littman, M. L. 1996. Algorithms for sequential decision making. Ph.D. thesis, Department of Computer Science, Brown University, Providence, RI.

Lloyd, J. W. 1987. *Foundations of Logic Programming* 2nd Ed. Springer.

Mannes, A., Michael, M., Pate, A., Sliva, A., Subrahmanian, V. S., and Wilkenfeld, J. 2008a. Stochastic opponent modeling agents: A case study with Hamas. In *Proceedings of the International Conference on Computer and Communication Devices*.

Mannes, A., Michael, M., Pate, A., Sliva, A., Subrahmanian, V. S., and Wilkenfeld, J. 2008b. Stochastic opponent modelling agents: A case study with Hezbollah. In *Proceedings of the 1st International Workshop on Social Computing, Behavioral Modeling, and Prediction*. H. Liu and J. Salerno Eds.

Ng, R. T. and Subrahmanian, V. S. 1992. Probabilistic logic programming. *Inf. Comput. 101,* 2, 150–201.

Ng, R. T. and Subrahmanian, V. S. 1993. A semantical framework for supporting subjective and conditional probabilities in deductive databases. *J. Autom. Reason. 10,* 2, 191–235.

Nilsson, N. 1986. Probabilistic logic. *Artif. Intell. 28*, 71–87.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco.

Pearl, J. 1991. Probabilistic and qualitative abduction. In *Proceedings of the AAAI Spring Symposium on Abduction*. AAAI Press, Stanford, CA, 155–158.

Pelikan, M., Goldberg, D. E., and Lobo, F. G. 2002. A survey of optimization by building and using probabilistic models. *Comput. Optim. Appl. 21,* 1, 5–20.

Peng, Y. and Reggia, J. A. 1990. *Abductive Inference Models for Diagnostic Problem-Solving*. Springer.

Poole, D. 1993. Probabilistic Horn abduction and Bayesian networks. *Artif. Intell. 64,* 1, 81–129.

Poole, D. 1997. The independent choice logic for modelling multiple agents under uncertainty. *Artif. Intell. 94,* 1–2, 7–56.

Puterman, M. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons.

Shanahan, M. 2000. An abductive event calculus planner. *J. Logic Program. 44*, 207–239.

Tseng, P. 1990. Solving H-horizon, stationary Markov decision problems in time proportional to log(H). *Oper. Res. Lett. 9,* 5, 287–297.

Tsitsiklis, J. and van Roy, B. 1996. Feature-based methods for large scale dynamic programming. *Machine Learning 22,* 1/2/3, 59–94.

Williams, R. and Baird, L. 1994. Tight performance bounds on greedy policies based on imperfect value functions. In *Proceedings of the 10th Yale Workshop on Adaptive and Learning Systems*.

Received December 2010; revised October 2011; accepted March 2012