Policy Analytics Generation Using Action Probabilistic Logic Programs

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

1 Introduction

Action probabilistic logic programs (*ap*-programs for short) [15] are a class of the extensively studied family of probabilistic logic programs [14, 21, 22]. *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 [10]. *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

[1] Action atoms only represent the fact that an action is taken, and not the action itself; we assume that effects and preconditions are generally not known.

G.I. Simari (✉)
Department of Computer Science, University of Oxford, Oxford OX1 3QD, UK
e-mail: gerardo.simari@cs.ox.ac.uk

J.P. Dickerson
Gates-Hillman Center, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA
e-mail: dickerson@cs.cmu.edu

A. Sliva
College of Computer and Information Science, Northeastern University, Boston, MA 02115, USA
e-mail: asliva@ccs.neu.edu

V.S. Subrahmanian
Department of Computer Science, University of Maryland College Park, College Park, MD 20742, USA
e-mail: vs@cs.umd.edu

V.S. Subrahmanian (ed.), *Handbook of Computational Approaches to Counterterrorism*,
DOI 10.1007/978-1-4614-5311-6_23,

$$
\begin{aligned}
r_1 &: \texttt{clashCas}(jk \prec 1) : [0 \triangleright 85 \prec 0 \triangleright 91] \leftarrow \texttt{socStrife}(pak \prec 1) \wedge \texttt{allianceNSAG}(1) \triangleright \\
r_2 &: \texttt{clashCas}(jk \prec 1) : [0 \prec 0 \triangleright 03] \leftarrow \texttt{socStrife}(pak \prec 0) \wedge \texttt{allianceNSAG}(0) \triangleright \\
r_3 &: \texttt{clashCas}(jk \prec 1) : [0 \triangleright 42 \prec 0 \triangleright 48] \leftarrow \texttt{socStrife}(pak \prec 0) \wedge \texttt{allianceNSAG}(1) \triangleright \\
r_4 &: \texttt{murder}(jk \prec 1) : [0 \triangleright 52 \prec 0 \triangleright 58] \leftarrow \texttt{trainCamp}(pak \prec 1) \wedge \texttt{relOrg}(1) \triangleright \\
r_5 &: \texttt{murder}(jk \prec 1) : [0 \prec 0 \triangleright 03] \leftarrow \texttt{trainCamp}(pak \prec 0) \wedge \texttt{relOrg}(0) \triangleright \\
r_6 &: \texttt{murder}(jk \prec 1) : [0 \triangleright 20 \prec 0 \triangleright 26] \leftarrow \texttt{trainCamp}(pak \prec 0) \wedge \texttt{relOrg}(1) \triangleright \\
r_7 &: \texttt{fedayeenAtt}(jk \prec 1) : [0 \triangleright 59 \prec 0 \triangleright 65] \leftarrow \texttt{govMilSupp}(pak \prec 1) \wedge \texttt{relOrg}(1) \triangleright \\
r_8 &: \texttt{fedayeenAtt}(jk \prec 1) : [0 \prec 0 \triangleright 03] \leftarrow \texttt{govMilSupp}(pak \prec 0) \wedge \texttt{relOrg}(0) \triangleright \\
r_9 &: \texttt{fedayeenAtt}(jk \prec 1) : [0 \triangleright 22 \prec 0 \triangleright 28] \leftarrow \texttt{govMilSupp}(pak \prec 0) \wedge \texttt{relOrg}(1) \triangleright
\end{aligned}
$$

Fig. 1 A small set of rules modeling Lashkar-e-Taiba

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 $[\ell, u]$."

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, Fig. 1 shows a small set of rules automatically extracted from data gathered in the *Computational Modeling of Terrorism* (CMOT) project[2] about Lashkar-e-Taiba's past (referred to from now on as LeT), where predicates correspond to rules in the data set and in general a value of 0 indicates that the action is not performed or the condition does not hold.[3] Rule 1 says that LeT engages in clashes in Jammu and Kashmir (J&K, from now on), inflicting casualties in security forces (action `clashCas`), with probability between 0.85 and 0.91 whenever there is social strife in Pakistan (condition `socStrife`), and LeT engages in alliances with non-state armed groups (condition `allianceNSAG1`). Rules 2 and 3, also about such clashes, refer to the probabilities when these conditions are slightly altered. The rest of the rules refer to murders committed in J&K, and Fedayeen attacks carried out in J&K, involving the following conditions: `trainCamp`, which means that LeT maintains training camps; `relOrg`, referring to the condition of LeT as a religious group, and `govMilSupp`, referring to the Pakistani government giving military support. *ap*-programs have been extensively (and successively) used by terrorism analysts to make predictions about terror group actions [10, 19]. The analysis of LeT from which these rules were taken is described in depth in [20].

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 LeT engaging in clashes causing casualties is below some percentage) is achieved, given some constraints on what is feasible. The *basic abductive query answering problem* (BAQA) deals

[2] http://www.umiacs.umd.edu/research/LCCD/projects/let.jsp

[3] Note that variables can have more than two possible values; therefore, even though `murder`(1) is equivalent to ¬`murder`(0) because `murder` is a binary variable, this does not hold in general.

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.

We will also 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 describe a heuristic algorithm based on probability density estimation techniques that can be used to tackle CBQA with large instances, and then present parallel algorithms capable of solving these problems faster. Finally, we describe a prototype implementation and experimental results showing that our algorithms scale well, and achieve results that are useful in practice.

2 Preliminaries

We now overview the syntax and semantics of *ap*-programs from [15].

2.1 Syntax

We assume the existence of a logical alphabet that consists of a finite set $\mathscr{L}_{cons}$ of constant symbols, a finite set $\mathscr{L}_{pred}$ of predicate symbols (each with an associated arity) and an infinite set $\mathscr{L}_{var}$ of variable symbols; function symbols are not allowed. Terms, atoms, and literals are defined in the usual way [17]. We assume that $\mathscr{L}_{pred}$ is partitioned into disjoint sets: $\mathscr{L}_{act}$ of *action symbols* and $\mathscr{L}_{sta}$ of *state* symbols. Thus, if $t_1, \ldots, t_n$ are terms, and p is an n-ary action (resp. state) symbol, then $p(t_1, \ldots, t_n)$, is called an *action (resp. state) atom*.

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

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

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

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

In the following, we will use APF to denote the set of all possible *ap*-formulas.

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

It is assumed that all actions in 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., counter-terrorism), there are many, many variables, and the effects of our actions are not well understood. We now define *ap*-rules.

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

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

Definition 5 (*ap*-program). An *action probabilistic logic program* (*ap*-program for short) is a finite set of *ap*-rules. An *ap*-program Π' 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 LeT'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 1 (Worlds and states). Coming back to the *ap*-program in Fig. 1, the following are examples of worlds:

$$\{\texttt{clashCas}(jk, 1)\}, \{\texttt{clashCas}(jk, 1), \texttt{fedayeenAtt}(jk, 1)\}, \{\}$$

The following are examples of states:

$$\{\texttt{socStrife}(pak, 0), \texttt{trainCamp}(pak, 1)\},$$
$$\{\texttt{socStrife}(pak, 1), \texttt{relOrg}(1)\}. \qquad \blacksquare$$

2.2 Semantics of ap-Programs

We use $\mathscr{W}$ to denote the set of all possible worlds, and $\mathscr{S}$ to denote the set of all possible states. It is clear what it means for a state to satisfy the body of a rule [17].

Definition 6 (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 $F : \mu \leftarrow B_1 \wedge \ldots \wedge B_m$ if and only if $\{B_1, \ldots, B_M\} \subseteq s$.

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

Definition 7 (Satisfaction of an action formula). 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 $a \in B_{\mathscr{L}_{act}}$, then $a \in w$;
- If $F \equiv F_1 \wedge F_2$, for formulas $F_1, F_2 \in formulas(B_{\mathscr{L}_{act}})$, then w satisfies F_1 and w satisfies F_2;
- If $F \equiv F_1 \vee F_2$, for formulas $F_1, F_2 \in formulas(B_{\mathscr{L}_{act}})$, then w satisfies F_1 or w satisfies F_2;
- If $F \equiv \neg F'$, for some formula $F' \in formulas(B_{\mathscr{L}_{act}})$, then w does not satisfy F'.

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

Definition 8 (Reduction of an *ap*-program). 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 [9, 11, 23]. 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 $\mathscr{L}_{act}$ 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 $\ell \leq \sum_{w_i \in \mathscr{W} \wedge w_i \models F} p_i \leq u$.
2. $LC(\Pi, s)$ contains the constraint $\sum_{w_i \in \mathscr{W}} p_i = 1$.
3. All variables are non-negative.
4. $LC(\Pi, s)$ contains only the constraints described in (1)–(3).

While [15] provide a more formal model theory for *ap*-programs, we merely provide the definition below. Π_s is *consistent* iff $LC(\Pi, s)$ is solvable over $\mathbb{R}$.

Definition 9 (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** $\sum_{w_i \in \mathscr{W} \wedge w_i \models F} p_i$ **subject to** $LC(\Pi, s)$.
- $u' =$ **maximize** $\sum_{w_i \in \mathscr{W} \wedge w_i \models F} p_i$ **subject to** $LC(\Pi, s)$.

Note that, even though Definition 9 defines entailment for reduced programs (i.e., w.r.t. a state), the definition contemplates general programs, since given an arbitrary set of rules there always exists a state that makes it relevant.

The quantity ℓ' in the above definitions 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.

3 Abductive Queries to Probabilistic Logic Programs

The first kind of queries that we will study are called *basic abductive queries*, and the associated problem is called the *Basic Abductive Query Answering Problem* (BAQA for short). 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. Reachability expresses the changes that we can make in the environment; for instance, we might be able to relieve social strife in Pakistan and influence the Pakistani government to not provide military support to LeT, but perhaps influencing LeT to not be a religious organization is out of realm of possibilities.

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 now, we will assume that *reach* is provided and can be queried in polynomial time. However, in order to develop practical algorithms, later we will investigate one way in which *reach* can be specified (called *reachability constraints*).

Example 2 (Reachability between states). Suppose, for simplicity, that the only state predicate symbols are those that appear in the rules of Fig. 1, and consider the set of states in Fig. 2. Then, some examples of reachability are the following: $reach(s_4, s_1)$, $reach(s_1, s_4)$, $reach(s_1, s_3)$, $reach(s_2, s_3)$, $reach(s_3, s_2)$, $reach(s_2, s_5)$ $reach(s_3, s_5)$,, $\neg reach(s_5, s_2)$, and $\neg reach(s_5, s_3)$. ■

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.
A solution to a BAQA instance is a sequence of states that ends in a state for which the corresponding subprogram entails the probabilistic goal. Therefore, such a solution corresponds to a *policy* that can be implemented to try and bring about the goal by carrying out the actions prescribed by the sequence.

$s_1 = \{\texttt{socStrife}(pak, 1) \texttt{allianceNSAG}(1) \texttt{trainCamp}(pak, 0) \texttt{relOrg}(1) \texttt{govMilSupp}(0)\}$
$s_2 = \{\texttt{socStrife}(pak, 1) \texttt{allianceNSAG}(1) \texttt{trainCamp}(pak, 0) \texttt{relOrg}(0) \texttt{govMilSupp}(0)\}$
$s_3 = \{\texttt{socStrife}(pak, 0) \texttt{allianceNSAG}(0) \texttt{trainCamp}(pak, 0) \texttt{relOrg}(0) \texttt{govMilSupp}(0)\}$
$s_4 = \{\texttt{socStrife}(pak, 0) \texttt{allianceNSAG}(1) \texttt{trainCamp}(pak, 0) \texttt{relOrg}(1) \texttt{govMilSupp}(0)\}$
$s_5 = \{\texttt{socStrife}(pak, 1) \texttt{allianceNSAG}(1) \texttt{trainCamp}(pak, 1) \texttt{relOrg}(1) \texttt{govMilSupp}(1)\}$

Fig. 2 A small set of possible states

Example 3 (Solution to BAQA). Consider once again the program in the running example and the set of states from Fig. 2. If the goal is $clashCas(jk, 1) : [0, 0.3]$ (we want the probability that LeT engages in clashes in J&K causing casualties to be at most 0.3) and the current state is s_4, then the problem is solvable because Example 2 shows that state s_3 can be reached from s_4, and $\Pi_{s_3} \models clashCas(jk, 1) : [0, 0.3]$. ■

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 social strife in Pakistan may only succeed with some probability). We will address this problem in Sect. 4.

The BAQA problem can be shown to be intractable both in the general case as well as in constrained subcases; for a formal treatment of the complexity of BAQA, we refer the reader to [26–28]. It turns out that the complexity of this problem is caused by two factors; specifically, we need to address the following two problems:

(P1) Find a subprogram Π' of Π such that when the body of all rules in that subprogram is deleted, the resulting subprogram entails the goal, and
(P2) Decide if there exists a state s' such that $\Pi' = \Pi_s$ and s is reachable from the initial state.

We now present algorithms and techniques for addressing these problems.

3.1 Algorithms for BAQA over Threshold Queries

In this section, we leverage the above intuition that BAQA can be decomposed into two subproblems to develop algorithms for a special case of BAQA: answering *threshold queries*. These queries are over goals 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). This is a reasonable approach, since threshold goals can be used to require that certain formulas (actions) should only be entailed with a certain maximum probability (upper bound) or should be entailed with at least a certain minimum probability (lower bound). We start by inducing equivalence classes on subprograms that limit the search space, helping address problem (P1).

Definition 10 (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, sub-programs 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.

Algorithm 1: simpleAnnBAQA($\Pi, s, G : [\ell_G, u_G]$)

1. 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** $\big[\boldsymbol{active}(\Pi, G : [\ell_G, u_G])\big]$**:** 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 $\big[\boldsymbol{passive}(\Pi, G : [\ell_G, u_G])\big]$**:** Initialize this set with the rules in Π not identified as active;
   Conflicting rule set initialization $\big[\boldsymbol{conf}(\Pi, G : [\ell_G, u_G])\big]$**:**
   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 Π 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. 3 An algorithm to solve BAQA assuming a threshold goal

Example 4 (Equivalence of ap-programs). Let Π be the *ap*-program from Fig. 1, and formula $F = \texttt{clashCas}(jk, 1) : [0.4, 1]$. Intuitively, the definition of equivalence between *ap*-programs w.r.t. an action formula states that rules that don't influence the probability of the formula are immaterial. Therefore, we can conclude, for instance, that $\{r_1, r_4, r_7\} \sim_F \{r_1, r_6, r_9\}$, since the rules that change in these two sets do not influence the probability with which F is entailed. Equivalence of states given a formula is analogous. ∎

Relation $\sim$, both between states and between subprograms, is clearly an equivalence relation. The algorithm in Fig. 3 first tries to identify consistent subprograms that contain rules that clearly entail the goal (and are easily identifiable). If this is not possible, the algorithm continues by trying to leverage the presence of equivalence classes in the input *ap*-program Π. We now present an example reviewing how this algorithm works.

Example 5 (simpleAnnBAQA over the running example). Suppose Π is the *ap*-program of Fig. 1, the goal is $\texttt{clashCas}(jk, 1) : [0, 0.3]$ (abbreviated with $G : [0, 0.3]$ from now on) and the state is s_4 from Fig. 2; note that $\Pi_{s_4} = \{r_3, r_6, r_9\}$ and that clearly $\Pi_{s_4} \not\models \texttt{clashCas}(jk, 1) : [0, 0.3]$. The first step checks for possibilities to leverage subprogram equivalence; clearly, rule r_2 satisfies the condition, and we thus only need to verify that some subprogram containing it is reachable. Assuming the same reachability predicate outlined in Example 2, state s_3 is reachable from s_4; this corresponds to choosing subprogram $\Pi' = \{r_2, r_5, r_8\}$.

Finally, to illustrate Step 2 of the algorithm, which looks for rules whose heads involve formulas related to the goal, note that in this case this is simple since all the heads of rules in Π are atomic—therefore $passive(\Pi_{s_4}, G : [0, 0.3]) = \emptyset$, and the set of active rules contains all the rules in Π. ■

Next, we will explore a particular way in which *reach* can be expressed, and how this can be leveraged to solve the reachability problem (P2). The key is that the reachability predicate will be expressed through *reachability constraints*:

Definition 11 (Reachability constraint). Let F and G be first-order formulas over $\mathscr{L}_{sta}$ and $\mathscr{L}_{var}$, 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 $F \not\rightsquigarrow G$; 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 $\mathscr{S}$.

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

$$rc_1 : \texttt{relOrg}(1) \not\rightsquigarrow \texttt{relOrg}(0)$$

$$rc_2 : \texttt{govMilSupp}(pak, 1) \not\rightsquigarrow \texttt{trainCamp}(pak, 0)$$

$$rc_3 : \texttt{allianceNSAG}(1) \not\rightsquigarrow \texttt{socialStrife}(0)$$

Constraint rc_1, for instance, states that we are not capable of influencing the group being modeled to not be a religious organization. ■

Algorithm *simpleAnnBAQA-Heur-RC* (Fig. 4) is optimistic and assumes that Step 1a of *simpleAnnBAQA* will yield at least one entailing formula for the goal; furthermore, it takes advantage of the structure added by the presence of reachability constraints. After checking for the simple necessary entailment condition, the algorithm continues by executing the steps of *simpleAnnBAQA* that compute the sets $active(\Pi, G : [\ell_G, u_G])$, $passive(\Pi, G : [\ell_G, u_G])$, $candAct(\Pi, G : [\ell_G, u_G])$,

Algorithm 2: simpleAnnBAQA-Heur-RC($\Pi, s, G : [\ell_G, u_G], RC$)

1. If the condition in Step 1a of *simpleAnnBAQA* does not hold, return *false*;
2. Execute Steps 2, 3, and 4 of *simpleAnnBAQA*;
3. let *goalState*, *goalStateAct*, *goalStateConf*, and *goalStateInf* be logical formulas over the sets $\mathscr{L}_{sta}$ and $\mathscr{L}_{var}$;
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 $\left[u_G = 1, F \models G, \text{and } \ell_G \leq \ell_F\right]$ or $\left[\ell_G = 0, G \models F, \text{and } u_G \geq u_F\right]$
   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. 4 A heuristic algorithm, based on simple sufficient conditions of entailment, 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

$conf(\Pi, G : [\ell_G, u_G])$, 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. Though there are many possible ways to implement this subroutine, here we will explore a SAT-based algorithm, which is presented in Fig. 5. This algorithm is simple: if the current state does not satisfy *goalState*, it starts by initializing formula *Reachable* which will be used to represent the set of eventually reachable states at each step. The initial formula describes state s, and the algorithm then proceeds to select all the constraints whose antecedents are entailed by *Reachable*. Once we have this set, *Reachable* is updated to the conjunction of the negations of all the consequents of constraints in the set. We are done if either *Reachable* at this point models *goalState*, or the old version of

Algorithm 3: 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 $F_i \not\rightarrow G_i$ such that $Reachable \models F_i$;
   set $Reachable := \left(\bigwedge_{F_i \not\rightarrow G_i \in RC_{curr}} \neg G_i\right)$;
   set *done:*= $((Reachable \wedge goalState) \not\models \bot) \vee (Reachable \models Reachable_{old})$;
4. return $(Reachable \wedge goalState \not\models \bot)$;

Fig. 5 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; a formula is derived describing the set of all possible states eventually reachable from the initial one

Reachable is modeled by the new one, i.e., no new reachable states were discovered. The following is an example of how $decideReachability - SAT$ works.

Example 7. Consider the *ap*-program from Fig. 1, along with constraint rc_3 from Example 6. As we saw in Example 5, if the goal is $\texttt{clashCas}(jk, 1) : [0, 0.3]$ and the current state is s_4 from Fig. 2, then rule r_2 needs to be made relevant, while r_1 and r_3 should not be relevant, and the rest do not influence the outcome. This yields the following *goalState* formula:

$$\texttt{socStrife}(pak, 0) \wedge \texttt{allianceNSAG}(0) \wedge \neg \left(\bigvee_{i=1,3} Body(r_i) \right)$$

Reachable starts out with $formula(s_4)$ (that is, the conjunction of all atoms in the state) and, as $Reachable \models \texttt{allianceNSAG}(1)$, it gets updated to:

$$\neg \texttt{socStrife}(pak, 0)$$

which is mutually unsatisfiable with *goalState*. In the next iteration, however, as *Reachable* does not entail the antecedent of rc_3, 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*. ∎

4 Cost-Based Abductive Query Answering

In this section, we expand on the basic query answering problem described above 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; e.g., the fact that we may try to reduce social strife in Pakistan may only succeed with some probability. To model this, we use three functions:

Definition 12. A *transition function* is any function $T : \mathscr{S} \times \mathscr{S} \to [0, 1]$, and a *cost function* is any function $cost : \mathscr{S} \to [0, 1]$. A *transition cost function*, defined w.r.t. a transition function T and some cost function $cost$, is a function $cost_T : \mathscr{S} \times \mathscr{S} \to [0, \infty)$, with $cost_T(s, s') = \frac{cost(s')}{T(s,s')}$ whenever $T(s, s') \neq 0$, and ∞ otherwise.[4]

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

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:

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

Note that Eq. 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 13 (Reward functions). An *action reward function* is a partial function $R : APF \to [0, 1]$. 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 $E_{\Pi,R} : \mathscr{S} \to [0, \infty)$, defined as:

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

Reward functions are used to represent how desirable it is, from the reasoning agent's point of view, for a given annotated action formula to be entailed in a given state by the model being used. Here, 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. From now on, we assume that all transition cost functions are defined in terms of a canonical cost function.

Example 8. An example of an entailment-based reward function is as follows. Consider state s_2 from Fig. 2, and annotated formulas $F_1 = \texttt{clashCas}(jk, 1) \wedge \texttt{murder}(jk, 1) : [0.5, 1]$, $F_2 = \texttt{clashCas}(jk, 1) \wedge \texttt{murder}(jk, 1) : [0, 3]$, $F_3 = \texttt{fedayeenAtt}(jk, 1) : [0, 0.05]$.

Suppose we have action reward function R such that: $R(F_1) = 0.1$, $R(F_2) = 0.85$, and $R(F_3) = 0.7$. This function represents that subprograms that entail F_2 are much more preferable than those that entail F_1, and that F_3 is also a desirable formula to entail. ∎

[4] 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.

Definition 14. A *cost based query* is a four-tuple $\langle G : [\ell, u], s, cost_T, k\rangle$, where $G : [\ell, u]$ is an *ap*-formula, $s \in \mathscr{S}$, $cost_T$ is a cost function, and $k \in \mathbb{R}^+ \cup \{0\}$.

CBQA Problem. Given *ap*-program Π and cost-based query $\langle G : [\ell, u], s, cost_T, k\rangle$, return "Yes" if and only if there exists a state s' and sequence of states $seq = \langle s, s_1, \ldots, s'\rangle$ such that $cost^*_{seq}(s, s') \leq k$, and $\Pi_{s'} \models G : [\ell, u]$; the answer is "No" otherwise.

The main difference between the BAQA problem presented above and CBQA is that in BAQA there is no notion of cost, and we are only interested in the *existence of some sequence* of states leading to a state that entails the *ap*-formula. Even though we are still interested in sequences, solutions now have associated values depending on the transitions they attempt and the desirability of the states they traverse. Since CBQA is a generalization of BAQA, the same intractability results hold here as well [28]. In the following, we investigate an algorithm for CBQA when the cost function is defined in terms of entailment-based reward functions; we will focus on a tractable approach to finding solutions, albeit not optimal ones.

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) [2]. 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 (Fig. 6) 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 Sect. 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,[5] starting out as the uniform distribution.

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

Algorithm 4: DE_CBQA($\Pi, G : [\ell, u], s_0, T, h, k, numIter, giveUp$)

1. Initialize set of states $\mathscr{S}_G$:= $getGoalStates(\Pi, G : [\ell, u])$;
2. test all transitions (s_0, s_G), for $s_G \in \mathscr{S}_G$; calculate $cost^*_{seq}(s_0, s_G)$ for each;
3. let ϕ_{best} be the two-state sequence that has the lowest cost, denoted c_{best};
4. let $\mathscr{S}' = \mathscr{S} - \mathscr{S}_G - \{s_0\}$; set $j := 2$;
5. initialize probability distribution P over $\mathscr{S}'$ s.t. $P(s) = \frac{1}{|\mathscr{S}'|}$ for each $s \in \mathscr{S}'$;
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 $s_G \in \mathscr{S}_G$;
10. rank each sequence ϕ with $cost^*_{seq}(s_0, \phi(j))$;
11. pick the sequence in H with the lowest cost c^*, call it ϕ^*;
12. if $c^* < c_{best}$ then $\phi_{best} := \phi^*$; $c_{best} := c^*$;
13. P:= generate new distribution based on H;
14. return ϕ_{best};

Fig. 6 An algorithm for CBQA based on probability density estimation

The *getGoalStates* function called in line 1 performs two tasks: first, it identifies subprograms Π' of Π such that $\Pi' \models G : [\ell, u]$; second, it identifies states s such that $\Pi_s = \Pi'$, for some Π' found in the first step. All such states are then labeled as *goal states*, since any sequence of states from s_0 to any goal state is a candidate solution.

The `while` loop in lines 6–13 then performs the main search; *giveUp* is a predicate given by parameter which simply tells us when the algorithm should stop (it can be based on total number of samples, time elapsed, etc.). The value j represents the length of the sequence of states currently considered, and *numIter* is a parameter indicating how many iterations we wish to perform for each length. Line 9 performs the sampling of sequences, while line 10 assigns a score to each based on the transition cost function. After updating the score of the best solution found up to now, line 13 updates the probabilistic model P being used by keeping only the best solutions found during the last sampling phase. The algorithm finally returns the best solution it found (if any). An attractive feature of DE_CBQA is that it is an anytime algorithm, i.e., once it finds a solution, given more time it may be able to refine it into a better one while always being able to return the best so far. We now discuss one way in which the probability distribution P in the DE_CBQA algorithm can be represented.

Representing the Probability Distribution via a Bayesian Network

It is reasonable to believe that, in real-world instances of CBQA, states and actions are not in general conditionally independent; as such, it is critical to explore an

approach to maintaining our probability distribution that is capable of handling these cases. One such method is the Bayesian belief network [24], 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 can use a heuristic local search algorithm to perceive graph structure; for instance, a slightly modified K2 search algorithm with a fixed ordering based on the sampled sequences to emphasize speed of structure learning [6]. 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.

In Sect. 6, we present the results of our experimental evaluation of the DE_CBQA algorithm using this approach, comparing to a baseline algorithm that uses a much simpler representation.

5 Parallel Solutions for Abductive Query Answering

In the previous sections, 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.

Algorithm 5: 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 $\left[u_G = 1, F \models G, \text{and } \ell_G \leq \ell_F\right]$ or $\left[\ell_G = 0, G \models F, \text{and } u_G \geq u_F\right]$ then
6. Add r_i to $HeurRules$;
7. $BatchSize := \left\lceil \frac{|2^{\mathscr{L}_{sta}}|}{N} \right\rceil$;
8. while !$giveUp$ do
9. for parallel processes $n := 0$ to $N - 1$ do
10. foreach $s_i \in 2^{\mathscr{L}_{sta}}$ 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. 7 A parallel algorithm for finding entailing states for the CBQA problem

5.1 Parallel Selection of Entailing States

Recall the DE_CBQA algorithm in Fig. 6 and the *getGoalStates* function invoked in line 1; this function returns entailing states, i.e., states s s.t. $\Pi_s \models \Pi'$. In practice, as we will see in Sect. 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* (Fig. 4) 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 7 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 $2^{\mathscr{L}_{sta}}$ 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 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.

Algorithm 6: ParSampleAsynch_DE_CBQA($\Pi, G:[\ell,u], s_0, T, h, numIter, giveUp, N$)
1. Initialize set of states $\mathscr{S}_G$:= $getGoalStates(\Pi, G:[\ell,u])$;
2. test all transitions (s_0, s_G), for $s_G \in \mathscr{S}_G$; calculate $cost^*_{seq}(s_0, s_G)$ for each;
3. foreach parallel process $n := 0$ to $N-1$ do
4. let ϕ_{best} be the two-state sequence that has the lowest cost, denoted c_{best};
5. let $\mathscr{S}' = \mathscr{S} - \mathscr{S}_G - \{s_0\}$; set $j := 2$;
6. $P :=$ new uniform probability distribution over $sequences(\mathscr{S}')$;
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 $s_G \in \mathscr{S}_G$;
11. rank each sequence ϕ with $cost^*_{seq}(s_0, \phi(j))$;
12. pick the sequence in H_n with the lowest cost c^*, call it ϕ^*;
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 ϕ_{n-best} to H_{total};
16. return sequence ϕ_{best} in H_{total} with lowest cost c_{best};

Fig. 8 An asynchronous parallel algorithm for CBQA using iterative distributive sampling

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 Fig. 8 we present a distributed algorithm, *ParSampleAsynch_DE_CBQA*, that will divide the iterative sampling of state paths across n processors. 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 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 above, 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 Sects. 4 and 5; for reasons of space, we cannot include experimental results for BAQA (we refer the reader to [26] and [28] for a full set of empirical results). Also, 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.

We conducted experiments using a prototype JAVA implementation consisting of roughly 2,500 lines of code. 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 uniform randomly selected conjunction or disjunction connectors and random negation. The head is nontrivial; it is guaranteed to have at least one variable 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.

6.1 Empirical Evaluation of Algorithms for CBQA

We carried out the following experiments on an Intel Core2 Q6600 processor running at 2.4 GHz with 8 GB of memory available; all runs were performed on Windows 7 Ultimate 64-bit OS, and made use of a single core.

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.

A Baseline Algorithm. In the following, we will as a baseline a straightforward representation for the probability distribution: a mapping of states to the proportion of "good" sampled sequences that contain that state. We will refer to this method as the *naïve probability vector* approach. While this representation is neither memory nor computationally intensive, it ignores any subtle relationships that may exist between individual states or their ordering in the overall sequence. 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 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.

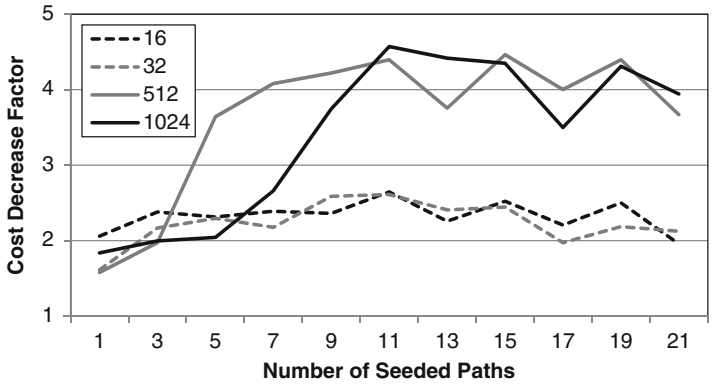

Fig. 9 Varying the number of seeded paths with a small (e.g., 16 or 32) number of states versus a larger (e.g., 512 or 1,024) state space

We compare the Bayesian method (implemented with WEKA [12]) against the 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 9 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 below.

Figure 10 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 Fig. 9, small instances of the problem stay roughly constant. Regardless of state space size, we see an increase in result quality of two to three over the naïve probability vector.

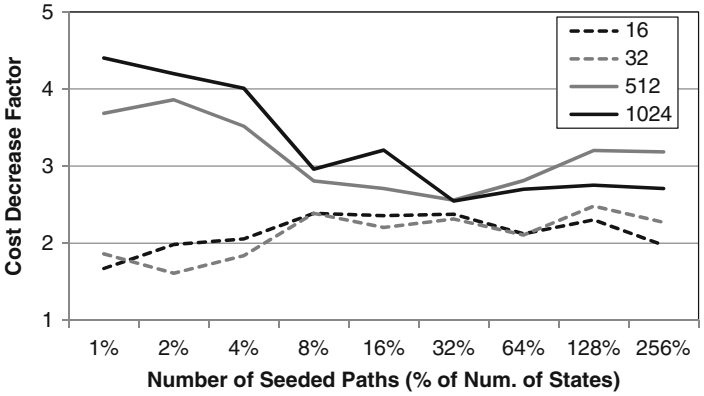

Fig. 10 Varying the number of seeded paths (and thus the level of conditional dependence in the world) as a percentage of the total number of states

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. Although the memory requirements of both algorithms increase linearly in the size of the number of states sampled, the Bayesian method is consistently slower than the vector method. This is due to a similar increase in the *runtime* complexity of the Bayesian method. The vector method represents probabilities as a simple mapping of states to real numbers; as such, an implementation with a constant lookup time data structure provides extremely fast sampling with a small memory footprint. For the more informed Bayesian variant, this relationship is based both on the number of initial iterations over the state space prior to the formation of the sampling structure and the maximum length of a sampled sequence. The Bayesian graph has as many nodes as there are states in a sampled sequence; furthermore, each of these nodes maintains knowledge of all unique states corresponding to a particular position in the sequence. Learning the structure of the network, storing the graph, and sampling from it are all dependent on the number of sampled states and sequence length. Thankfully, we can apply reasonable bounds to the number of samples, opting instead to instantiate multiple Bayesian networks over a smaller sample set.

When we include the additional cost of searching for entailing goal states (Line 1 of the DE_CBQA algorithm), both the naïve probability vector and informed Bayesian network methods scale similarly. We use the same fail-fast pessimistic approach to the heuristic goal search described earlier. Figure 11 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 runtime than the number of states. We see that the algorithm scales gracefully to large state/action spaces. As we mentioned above, in our experience, real-world instances of CBQA tend to contain significantly *fewer* rules than states and actions [15]; as such, in these cases DE_CBQA scales quite well.

Fig. 11 Run time comparison as DE_CBQA scales with respect to number of states (*top axis*) and number of rules (*bottom axis*). Note the similarity in run time between the Bayesian and vector probability models

6.2 Empirical Evaluation of Parallel Algorithms for ***CBQA***

We implemented the 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. Sect. 6, p. 532).

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 naive 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. Two experiments were performed 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

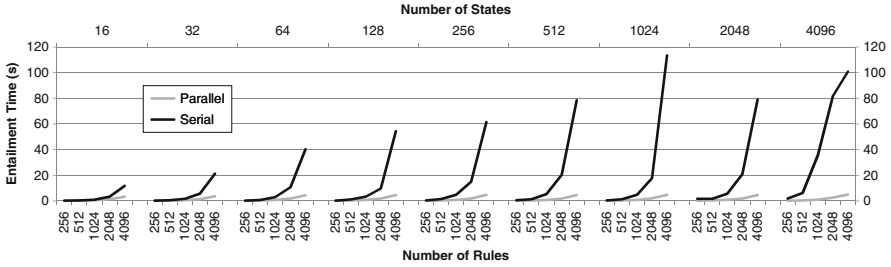

Fig. 12 Running time for both the parallel *PAR_getGoalStates* and serial *getGoalStates* methods to find up to 50 entailing states

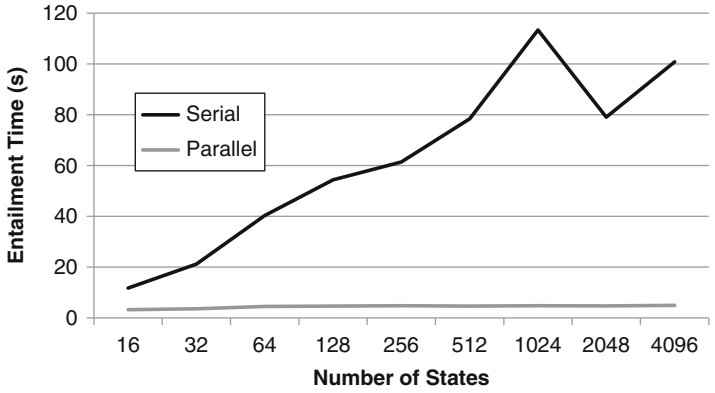

Fig. 13 Running time for both the parallel *PAR_getGoalStates* and serial *getGoalStates* methods to find entailing states. The number of states was varied between 16 and 4,096, and the number of rules held constant at 4,096

(Fig. 12). For 4,096 states and rules, the parallel entailment method requires 4.96 s, whereas serial selection is 20 times slower, taking 100.8 s. Furthermore, as shown in Figs. 13 and 14, 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, especially for large state-spaces, the parallel method provides significant overall savings.

Parallel iterative sampling. As discussed above, 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 Fig. 15, the running time of ParSampleAsynch_DE_CBQA is compared with the serial version for both the vector and Bayesian distribution methods. For the parallel computations, we performed 60

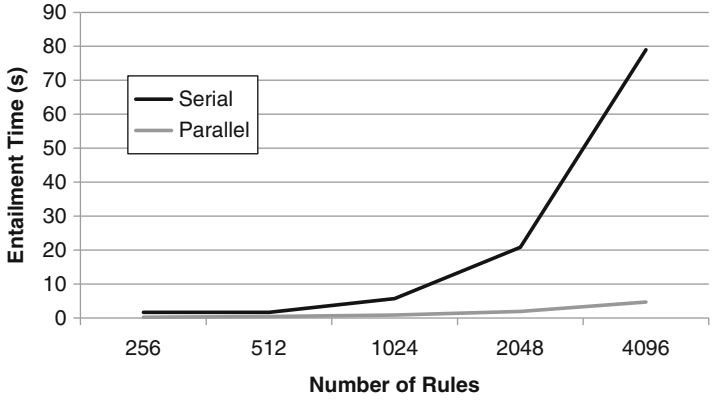

Fig. 14 Running time for both the parallel *PAR_getGoalStates* and serial *getGoalStates* methods to find entailing states. The number of rules was varied between 256 and 4,096, and the number of states held constant at 2,048

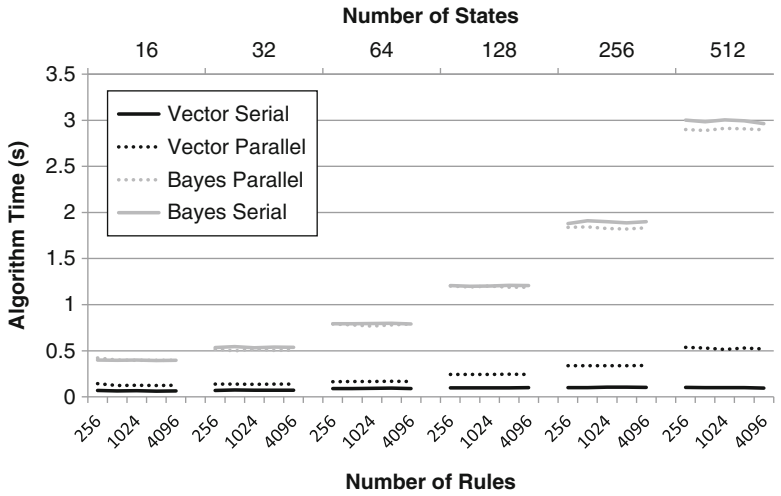

Fig. 15 Running time comparison of ParSampleAsynch_DE_CBQA and serial DE_CBQA using both the vector and Bayesian distributions

concurrent rounds of 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.

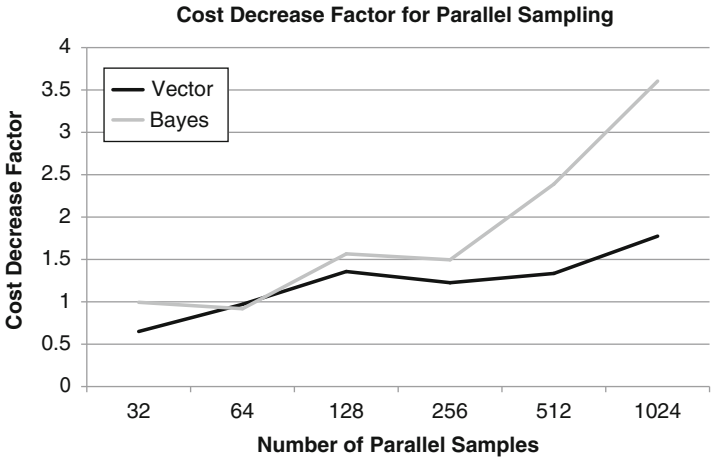

Fig. 16 Cost comparison of ParSampleAsynch_DE_CBQA and serial DE_CBQA using the vector and Bayesian distributions. The number of serial samples per iteration were held constant while the parallel samples per iteration were varied. The speedup factor measures the ratio of the serial best sequence cost to the parallel best sequence cost

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 16 compares the average cost ratio of sequences found by the serial and parallel sampling algorithms, where the cost decrease factor is defined as $\frac{\text{Costofsequencefoundbyserial}}{\text{Costofsequencefoundbyparallel}}$. 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 naive vector and Bayesian net distributions. As above, 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 factor of 0.970690347 and 0.918439618 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 Fig. 17. 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 (Fig. 16)—but we can achieve greater quality in a shorter period of time. For example, taking 128 parallel samples and using the Bayesian distribution,

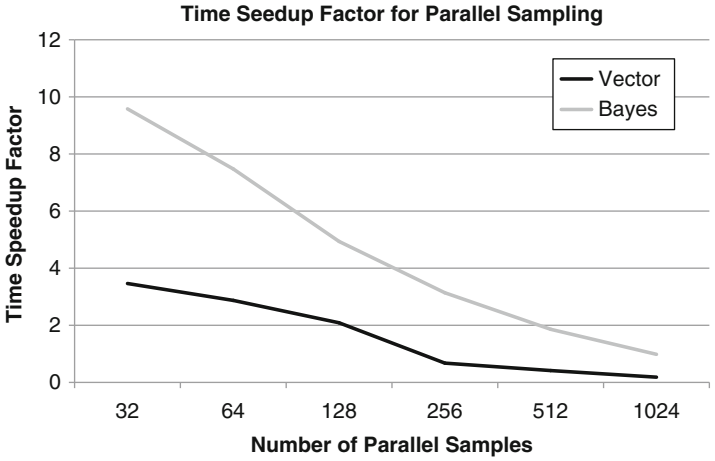

Fig. 17 Running time speedup of ParSampleAsynch_DE_CBQA versus serial DE_CBQA using both the vector and Bayesian distributions. The number of serial samples per iteration were held constant while the parallel samples per iteration were varied. The speedup factor measures the ratio of the serial running time to the parallel running time

ParSampleAsynch_DE_CBQA requires only about $\frac{1}{5}$ the time as DE_CBQA for 1,024 samples (16.26 and 3.29 s, 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 [5], reasoning with non-monotonic logics [7], probabilistic reasoning [13, 25], argumentation [16], planning [8], and temporal reasoning [8]; furthermore, it has been combined quite naturally with different variants of logic programs [1, 4]. 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 lead to many different models.

David Poole et al. combined probabilistic and non-monotonic reasoning, leading to the development of Probabilistic Horn Abduction and eventually the Independent Choice Logic [25]. Christiansen [4] 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 [25] or [4]) 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 bigger 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 [18, 19], 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) [3]. The US Treasury, for instance, is interested in modeling behaviors of investor groups to learn their attitudes towards risk under different conditions; similarly, 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"; instead, 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 presented heuristic algorithms that are relatively fast and sound, though incomplete, and

developed innovative algorithms that maintain and update probability distributions as they run, allowing better estimation of solutions while reducing running times. A further important contribution is that of parallel algorithms for abduction in probabilistic logic.

Acknowledgements Some of the authors of this paper were funded in part by AFOSR grant FA95500610405, ARO grant W911NF0910206 and ONR grant N000140910685.

References

1. Baldoni M, Giordano L, Martelli A, Patti V (1997) An abductive proof procedure for reasoning about actions in modal logic programming. In: Selected papers from NMELP '96. Springer, London, pp 132–150
2. Bonet JSD, Isbell CL Jr, Viola PA (1996) MIMIC: finding optima by estimating probability densities. In: Proceedings of NIPS '96. MIT press, USA, pp 424–430
3. Bryson JJ, Ando Y, Lehmann H (2007) Agent-based modelling as scientific method: a case study analysing primate social behaviour. Philos Trans R Soc Lond B 362(1485):1685–1698
4. Christiansen H (2008) Implementing probabilistic abductive logic programming with constraint handling rules. In: Constraint handling rules. Springer, Berlin/New York, pp 85–118
5. Console L, Torasso P (1991) A spectrum of logical definitions of model-based diagnosis. Comput Intell 7(3):133–141
6. Cooper G, Herskovits E (1992) A Bayesian method for the induction of probabilistic networks from data. Mach Learn 9(4):309–347
7. Eiter T, Gottlob G (1995) The complexity of logic-based abduction. JACM 42(1):3–42
8. Eshghi K (1988) Abductive planning with event calculus. In: Proceedings of ICLP. MIT Press, USA, pp 562–579, ISBN 0-262-61056-6
9. Fagin R, Halpern JY, Megiddo N (1990) A logic for reasoning about probabilities. Inf Comput 87(1/2):78–128
10. Giles J (2008) Can conflict forecasts predict violence hotspots? New Sci (2647)
11. Hailperin T (1984) Probability logic. Notre Dame J Form Log 25(3):198–212
12. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten I (2009) The WEKA data mining software: an update. ACM SIGKDD Explor Newsl 11(1):10–18
13. Josang A (2008) Magdalena, L, Ojeda-Aciego, M, Verdegay, J.L. Abductive reasoning with uncertainty. In: Proceedings of the IPMU 2008, Torremolinos, Malaga, Spain. pp 9–16
14. Kern-Isberner G, Lukasiewicz T (2004) Combining probabilistic logic programming with the power of maximum entropy. Artif Intell 157(1–2):139–202
15. Khuller S, Martinez MV, Nau DS, Sliva A, Simari GI, Subrahmanian VS (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
16. Kohlas J, Berzati D, Haenni R (2002) Probabilistic argumentation systems and abduction. Ann Math Artif Intell 34(1–3):177–195
17. Lloyd JW (1987) Foundations of logic programming, 2nd edn. Springer, Berlin/New York
18. Mannes A, Michael M, Pate A, Sliva A, Subrahmanian VS, Wilkenfeld J (2008) Stochastic opponent modeling agents: a case study with Hamas. In: Proceedings of ICCCD 2008, AAAI Press, USA, ISBN 978-1-57735-389-8
19. Mannes A, Michael M, Pate A, Sliva A, Subrahmanian VS, Wilkenfeld J (2008) Stochastic opponent modelling agents: a case study with Hezbollah. In: Liu H, Salerno J (eds) Proceedings of the first international workshop on social computing, behavioral modeling, and prediction, Springer, Germany, ISBN 978-0-387-77671-2

20. Mannes A, Shakarian J, Sliva A, Subrahmanian VS (2011) A computationally-enabled analysis of Lashkar-e-Taiba attacks in Jammu and Kashmir. In: Proceedings of EISIC. IEEE Computer Society, pp 224–229, ISBN 978-0-7695-4406-9
21. Ng RT, Subrahmanian VS (1992) Probabilistic logic programming. Inf Comput 101(2): 150–201
22. Ng RT, Subrahmanian VS (1993) A semantical framework for supporting subjective and conditional probabilities in deductive databases. J Autom Reason 10(2):191–235
23. Nilsson N (1986) Probabilistic logic. Artif Intell 28:71–87
24. Pearl J (1988) Probabilistic reasoning in intelligent systems: networks of plausible inference. Morgan Kaufmann, San Francisco
25. Poole D (1997) The independent choice logic for modelling multiple agents under uncertainty. Artif Intell 94(1–2):7–56
26. Simari GI, Subrahmanian VS (2010) Abductive inference in probabilistic logic programs. In: Technical communications of ICLP'10. LIPIcs, vol 7, Schloss Dagstuhl. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik 2010, ISBN 978-3-939897-17-0, pp 192–201
27. Simari GI, Dickerson JP, Subrahmanian VS (2010) Cost-based query answering in probabilistic logic programs. In: Proceedings of SUM 2010. LNCS. Springer, Berlin, Germany
28. Simari GI, Dickerson JP, Sliva A, Subrahmanian VS (2012) Parallel abductive query answering in probabilistic logic programs. Trans Comput Log