Parameterized Logic Programs where Computing Meets Learning

Taisuke SATO[⋆]

Dept. of Computer Science
Tokyo Institute of Technology
2-12-1 Ôokayama Meguro-ku Tokyo Japan 152

Abstract. In this paper, we describe recent attempts to incorporate learning into logic programs as a step toward adaptive software that can learn from an environment. Although there are a variety of types of learning, we focus on parameter learning of logic programs, one for statistical learning by the EM algorithm and the other for reinforcement learning by learning automatons. Both attempts are not full-fledged yet, but in the former case, thanks to the general framework and an efficient EM learning algorithm combined with a tabulated search, we have obtained very promising results that open up the prospect of modeling complex symbolic-statistical phenomena.

1 Introduction

We start by assuming that reproducing intelligence in a computer constitutes a great challenge to human intelligence in the 21st century. We on the other hand recall that the assumption held by AI researchers in the late seventies was such that it would be achieved by writing a huge program with a huge knowledge base (though no one knew how large it would be). The first assumption is taken undebatable in this paper but the second one, once thought to be undebatable, raises a serious question in light of the fact that OS, software comprising tens of millions of codes, is far short of being intelligent. We must admit that the size of a program has little to do with its intelligence. It is also recognized that intelligence is something very complex and the most feasible way of building a very complex object comprised of tens of millions of components is to write a program. So we are trapped in a kind of dilemma that writing a huge program may not be a solution to building intelligence, but we seem to have no way other than that (at least at the moment).

One way out is to note that programs can be smarter if they are born with the ability of learning, and it might be possible, instead of writing a huge complete program from the beginning, to let them learn how to behave more intelligently. Think of the following. Why is a program called a program? Because it specifies things to happen beforehand. And people have been taking it for granted that programs never change spontaneously regardless of how many times they are

⋆ email: sato@mi.cs.titech.ac.jp

used. Aside from the positive side, the negative side of this property is well-known; once an error occurs, the same error recurs indefinitely many times under the same condition. This stubbornness, "once built, no change," of programs exhibits a striking contrast to human beings who grow with time and learn from mistakes. After learning, we expect that something changes for the better, but programs lack any means of learning as they are designed to be just symbolic constructs for defining recursive functions, mathematically speaking.

The lack of learning ability is a common feature of deductive symbolic systems in general, and programs in particular, but there are well-established symbolic systems that have a close relationship to learning. For instance, there exist stochastic formal languages such as hidden Markov models (HMMs) [15].[1] HMMs are used in many fields from speech recognition to natural language processing to bioinformatics as a versatile modeling tool, and learning their parameters is a key step in their applications. Probabilistic context free grammars (PCFGs) [30, 9],[2] an extension of HMMs, have also statistical parameters learnable from linguistic data [1]. Turning to knowledge representation, we notice (discrete) Bayesian networks, a graphical representation of a finite joint distribution,[3] [4] are used to represent knowledge about uncertainty in the real world at propositional level [13, 2], and there is a standard way of statistically learning their parameters. Unfortunately, all these symbolic systems do not work as a program as they don't have a means of expressing control and data structures.

In the following,[5] we propose to integrate logic programs with parameter learning in hopes that they supply new building blocks for AI [16, 17, 7, 21, 22]. Resulting systems have the ability of expressing programs and the ability of learning at the same time. They can compute as they are logic programs and can learn parameters as well. There exist a couple of problems with this approach though. The most basic one is semantics. Notice that the basic principle

[1] A hidden Markov model is a stochastic finite automaton in which a transition is made probabilistically and a probabilistically chosen alphabet is output on each transition. The state transition is supposedly not observable from the outside.

[2] A probabilistic context free grammar is a CFG with probabilities assigned to each production rule. If a nonterminal A has N production rules $\{A \rightarrow \alpha_i \mid 1 \leq i \leq N\}$, probability p_i is assigned to each rule $A \rightarrow \alpha_i$ $(1 \leq i \leq N)$ in such a way that $\sum_{i=1}^{N} p_i = 1$. The probability of a sentence s is the sum of probabilities of each (leftmost) derivation of s. The latter is the product of probabilities of rules used in the derivation.

[3] By a joint distribution, we mean a joint probability density function [3].

[4] A Bayesian network is a graphical representation of a joint distribution $P(X_1 = x_1, \ldots, X_N = x_N)$ by a directed acyclic graph where each node is a random variable. A conditional probability table (CPT) representing $P(X_i = x_i \mid \quad_i = \quad_i)$ $(1 \leq i \leq N)$ is associated with each node X_i where $\quad_i$ represents the parent nodes $(1 \leq i \leq N)$ and $\quad_i$ are their values. When X_i has no parent, i.e. a topmost node in the graph, the table is just a marginal distribution $P(X_i = x_i)$. The whole joint distribution is given by $\prod_{i=1}^{N} P(X_i = x_i \mid \quad_i = \quad_i)$.

[5] The content of Section 2 and Section 3 is based on [22]. Related work is omitted due to space limitations.

of logic is that "nothing is connected unless otherwise specified by axioms" while the general rule of thumb in statistics is that "everything is connected unless otherwise specified by independence assumptions." This fundamental difference is carried over to semantics in such a way that logic has a compositional semantics, i.e. the meaning of $A \wedge B$ is a function of the meaning of A and that of B, but probability is not compositional, i.e. $P(A \wedge B)$ is not a function of $P(A)$ and $P(B)$. We are going to synthesize a new semantics by mixing these somewhat conflicting semantics in the next section for a class of *parameterized logic programs*, definite clause programs with a parameterized distribution over facts.

The new semantics is called *distribution semantics* [16]. It considers a parameterized logic program as defining a joint distribution (of infinite dimensions), and subsumes the standard least model semantics and the above mentioned symbolic systems, HMMs, PCFGs and Bayesian networks [17, 22]. In the following, after having established distribution semantics for parameterized logic programs in Section 2, we apply it to symbolic-statistical modeling [17, 18] in Section 3 and show that three basic tasks, i.e.

Task-1: computing probabilities
Task-2: finding out the most likely computation path
Task-3: learning parameters from data

are solved efficiently[7, 20–22]. Furthermore, in the case of PCFGs, we experimentally discovered that our learning algorithm called *the graphical EM algorithm* outperforms the Inside-Outside algorithm [1], the standard parameter learning algorithm for PCFGs, by orders of magnitudes [22]. In Section 4, we investigate another direction of combining logic programming and learning by incorporating reinforcement learning. Reinforcement learning is a method of online training by reward and penalty [5]. We show that logic programs incorporating *learning automatons*, simple reinforcement learning devices [12, 14], can be trained to behave desirably for our purpose.

The reader is supposed to be familiar with the basics of logic programming [8, 4], probability theory [3], Bayesian networks [13, 2] stochastic grammars [15, 9], reinforcement learning [5] and learning automatons [12].

2 Semantic framework

In this section, we define *distribution semantics*. Although it was already explained in various places [16, 17, 21, 22], we repeat the definition for the sake of self-containedness. First of all, our program is a definite clause program $DB = F \cup R$ in a first-order language $\mathcal{L}$ with countably many constant symbols, function symbols and predicate symbols where F is a set of unit clauses and R, a set of non-unit clauses. To avoid mathematical complications, we pretend that DB consists of countably many ground clauses and no clause head in R appears in F. Our intention is that non-unit definite clauses represent eternal laws in the universe whereas unit clauses represent probabilistic facts which happen to be

true or happen to be false. So we introduce a probability measure P_F over the set of ground atoms in F and extend it to a probability measure P_{DB} over the set of Herbrand interpretations for $\mathcal{L}$.

Let Ω_F be the set of Herbrand interpretations for the set of ground atoms in F and fix an enumeration $A_1, A_2, \ldots$ of ground atoms in F.[6] A Herbrand interpretation has a one-to-one correspondence to an infinite series $\langle x_1, x_2, \ldots \rangle$ of 1s and 0s by stipulating that $x_i = 1$ $(i = 1, 2, \ldots)$ (resp. $= 0$) if and only if A_i is true (resp. false). So Ω_F is identified with the direct product $\prod_{i=1}^{\infty} \{0,1\}_i$ of $\{0,1\}$s. The existence of a probability measure P_F over Ω_F is not self-evident but actually it is constructed freely from a collection of finite joint distributions $P_F^{(n)}(A_1 = x_1, \ldots, A_n = x_n)$ $(n = 1, 2, \ldots, x_i \in \{0,1\}, 1 \leq i \leq n)$ such that

$$\begin{cases} 0 \leq P_F^{(n)}(A_1 = x_1, \ldots, A_n = x_n) \leq 1 \\ \sum_{x_1, \ldots, x_n} P_F^{(n)}(A_1 = x_1, \ldots, A_n = x_n) = 1 \\ \sum_{x_{n+1}} P_F^{(n+1)}(A_1 = x_1, \ldots, A_{n+1} = x_{n+1}) \\ \qquad\qquad = P_F^{(n)}(A_1 = x_1, \ldots, A_n = x_n) \end{cases} \tag{1}$$

It is proved [3] that if $P_F^{(n)}(\cdot)$s satisfy the three conditions of (1), there exists a σ-additive probability measure P_F such that for $(n = 1, 2, \ldots, x_i \in \{0,1\}, 1 \leq i \leq n)$,

$$P_F(A_1 = x_1, \ldots, A_n = x_n) = P_F^{(n)}(A_1 = x_1, \ldots, A_n = x_n).$$

$P_F^{(n)}(\cdot)$s are presentable as an infinite binary tree like Figure 1. In the tree, $p_1, p_{21}, p_{22}, \ldots$ $(0 \leq p_1, p_{21}, p_{22}, \ldots \leq 1)$ are free parameters, and the tree specifies $P_F(A_1 = 1, A_2 = 0) = p_1(1 - p_{21})$ and so on.

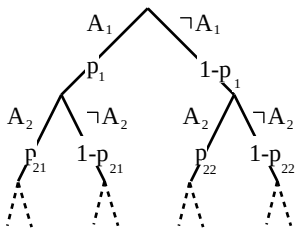


Fig. 1. Making a collection of finite distributions

Conversely, Figure 1 describes a general method of constructing $P_F^{(n)}$ but for practical reason, we assume that each probabilistic atom in F independently represents a probabilistic choice from a set of finitely many alternatives. So we introduce atoms of the form `msw(`i`,`n`,`v`)` which simulates a multi-ary random

[6] we assume that F contains countably many ground atoms.

switch whose name is i and whose outcome is v on trial n as a generalization of primitive probabilistic events such as coin tossing and rolling a dice. We henceforth assume that P_F is specified in the following way.

1. F consists of probabilistic atoms $\mathtt{msw}(i,n,v)$. The arguments i and n are arbitrary ground terms. We assume that a finite set V_i of ground terms is associated with each i, and $v \in V_i$ holds.
2. Write V_i as $\{v_1, v_2, \ldots, v_m\}$ $(m = |V_i|)$. Then, one of the ground atoms $\{\mathtt{msw}(i,n,v_1), \mathtt{msw}(i,n,v_2), \ldots, \mathtt{msw}(i,n,v_m)\}$ becomes exclusively true (takes value 1) on each trial. With each i and $(v \in V_i)$, a *parameter* $\theta_{i,v} \in [0,1]$ such that $\sum_{v \in V_i} \theta_{i,v} = 1$ is associated. $\theta_{i,v}$ is the probability of $\mathtt{msw}(i,\cdot,v)$ being true.
3. For each ground terms i, i', n, n', $v \in V_i$ and $v' \in V_{i'}$, random variable $\mathtt{msw}(i,n,v)$ is independent of $\mathtt{msw}(i',n',v')$ if $n \neq n'$ or $i \neq i'$.

Speaking more formally, we introduce a family of parameterized finite distribution $P_{(i,n)}$ such that

$$P_{(i,n)}(\mathtt{msw}(i,n,v_1) = x_1, \ldots, \mathtt{msw}(i,n,v_m) = x_m \mid \theta_{i,v_1}, \ldots, \theta_{i,v_m}) \stackrel{\text{def}}{=} \begin{cases} \theta_{i,v_1}^{x_1} \cdots \theta_{i,v_m}^{x_m} & \text{if } \sum_{k=1}^{m} x_k = 1 \\ 0 & \text{o.w.} \end{cases} \tag{2}$$

where $m = |V_i|$, $x_k \in \{0,1\}$ $(1 \leq k \leq m)$, and define P_F as the infinite-dimensional product measure

$$P_F \stackrel{\text{def}}{=} \prod_{i,n} P_{(i,n)}.$$

Based on P_F, another probability measure P_{DB} over the set of Herbrand interpretations Ω_{DB} for $\mathcal{L}$ is constructed as an extension of P_F by making use of the least Herbrand model semantics of logic programs [16]. Think of a Herbrand interpretation $\nu \in \Omega_F$. It defines a set F_ν of true atoms in F, and hence defines the least Herbrand model $M_{DB}(\nu)$ of $F_\nu \cup R$. $M_{DB}(\nu)$ determines all truth values of ground atoms in $\mathcal{L}$. A sampling ν from P_F thus determines all truth values of ground atoms. In other words, every ground atom in $\mathcal{L}$ becomes a random variable in this way. We formalize this idea. Enumerate *all ground atoms in* $\mathcal{L}$ and put it as $A_1, A_2, \ldots$ Then introduce a collection of finite disrtibutions $P_{DB}^{(n)}(A_1 = x_1, \ldots, A_n = x_n)$ $(n = 1, 2, \ldots)$ by

$$[A_1^{x_1} \wedge \cdots \wedge A_n^{x_n}]_F \stackrel{\text{def}}{=} \{\nu \in \Omega_F \mid M_{DB}(\nu) \models A_1^{x_1} \wedge \cdots \wedge A_n^{x_n}\}$$
$$P_{DB}^{(n)}(A_1 = x_1, \ldots, A_n = x_n) \stackrel{\text{def}}{=} P_F([A_1^{x_1} \wedge \cdots \wedge A_n^{x_n}]_F)$$

where $A^x = A$ if $x = 1$ and $A^x = \neg A$ if $x = 0$. Note that $[A_1^{x_1} \wedge \cdots \wedge A_n^{x_n}]_F$ is P_F-measurable. Since $P_{DB}^{(n)}$ $(n = 1, 2, \ldots)$ satisfies (1), there exists a probability measure P_{DB} over Ω_{DB}, an extension of P_F, such that

$$P_{DB}(A_1 = x_1, \ldots, A_n = x_n) = P_F(A_1 = x_1, \ldots, A_n = x_n)$$

for every finite sequence of atoms $A_1, \ldots, A_n$ in F and for every binary vector $\langle x_1, \ldots, x_n \rangle$ $(x_i \in \{0,1\},\ 1 \leq i \leq n)$. We consider $P_{DB}(\cdot)$ as a joint distribution of countably infinite dimensions and define the *denotation* of the program $DB = F \cup R$ w.r.t. P_F as P_{DB} (*distribution semantics*). DB then becomes a random vector of infinite dimensions having the distribution P_{DB} whose sample realization is a Herbrand interpretation for $\mathcal{L}$.

Distribution semantics subsumes the least Herbrand semantics in logic programming as a special case in which P_F places all probability mass on one interpretation, i.e. making a specific set of ground atoms in F always true. On the contrary, it is also possible to define a uniform distribution over F (like the one over the unit interval $[0,1]$) in which every Herbrand interpretation for F is equally probable. Since the cardinality of interpretations is just that of real numbers (Ω_{DB} is isomorphic to Cantor set), each probability of an $M_{DB}(\nu)$ is 0 in this case. What we are interested in however is distributions between these extremes which better reflect our observations in the real world such as a corpus. The characteristics of distribution semantics are summarized as follows.

- It is applicable to any parameterized logic programs. The underlying first-order language is allowed to contain countably many symbols be they functions or predicates, and programs can be arbitrary.
- Since it is based on the least model semantics, both sides of the iff definition of a predicate[7] $p(x) \leftrightarrow \exists y_1(x = t_1 \wedge W_1) \vee \cdots \vee \exists y_n(x = t_n \wedge W_n)$ unconditionally coincide as a random variable for any ground instantiation $p(t)$.
- Unfold/fold transformation [25] preserves the denotation of parameterized logic programs.

Owing to P_{DB}, we may regard every ground atom (and closed formula) as a random variable on Ω_{DB}. We apply our semantics in two directions, one for symbolic-statistical modeling described in Section 3, and the other for reinforcement learning described in Section 4.

Before proceeding, we explain how to execute a parameterized logic program. Suppose $DB = F \cup R$ is a parameterized logic program such that $F = \{\texttt{msw(}i\texttt{,}n\texttt{,}v\texttt{)}\}$ has a distribution $P_F = \prod_{i,n} P_{(i,n)}$ mentioned before, and also suppose a goal $\leftarrow G$ is given. We execute $\leftarrow G$ w.r.t. DB by a special SLD interpreter whose only difference from the usual one is an action taken for a goal `msw(I,N,V)`. When it is called with ground `I` $= i$ and `N` $= n$, two cases occur.[8]

- `msw(`i`,`n`,V)` is called for the first time. The interpreter chooses some ground term v from V_i with probability $\theta_{i,v}$, instantiates `V` to v and returns successfully.

[7] Here x is a vector of new variables of length equal to the arity of p, $p(t_i) \leftarrow W_i$ $(1 \leq i \leq n, 0 \leq n)$, an enumeration of clauses about p in DB, and each y_i, a vector of variables occurring in $p(t_i) \leftarrow W_i$.

[8] If either `I` or `N` is non-ground, the execution of `msw(`i`,`n`,V)` is unspecified.

– `msw(`i`,`n`,V)` was called before, and the returned value was v. If `V` is a free variable, the interpreter instantiates `V` to v and returns successfully. If `V` is bound to some non-free variable v', the interpreter tries to unify v' with v and and returns successfully if the unification is successful, or fails otherwise.

Execution of this type is called *sampling execution* because it corresponds to sampling from P_{DB}.

3 Statistical learning and parameterized logic programs

3.1 Blood type example

Parameterized logic programs define probabilities of ground atoms. Our aim here is to express our observations of symbolic-statistical phenomena such as observations of blood types by ground atoms, and write a parameterized logic program for them and adjust their parameters so that the distribution defined by the program closely approximates to their empirical distribution. Let's take `ABO` blood types. Possible types are `'A'`, `'B'`, `'O'` and `'AB'`. We declaratively model by a parameterized logic program $DB_1 = F_1 \cup R_1$ in Figure 2 how these blood types are determined from inherited blood type genes $\{\mathtt{a}, \mathtt{b}, \mathtt{o}\}$. Note that we employ Prolog conventions [23]. So strings beginning with a upper-case letter are variables. Quoted atoms are constants. The underscore "_" is an anonymous variable. Apparently clauses in DB_1 are direct translation of ge-

$$R_1 = \left\{ \begin{array}{l} \texttt{btype('A'):- (gtype(a,a) ; gtype(a,o) ; gtype(o,a)).} \\ \texttt{btype('B'):- (gtype(b,b) ; gtype(b,o) ; gtype(o,b)).} \\ \texttt{btype('O'):- gtype(o,o).} \\ \texttt{btype('AB'):- (gtype(a,b) ; gtype(b,a)).} \\ \texttt{gtype(X,Y):- gene(fa,X),gene(mo,Y).} \\ \texttt{gene(P,G):- msw(gene,P,G).} \end{array} \right.$$

$$F_1 = \{\texttt{msw(gene,P,a)}, \texttt{msw(gene,P,b)}, \texttt{msw(gene,P,o)}\}$$

Fig. 2. ABO blood type program DB_1

netic knowledge about blood types. `btype('A'):- (gtype(a,a) ; gtype(a,o) ; gtype(o,a))` for instance says one's blood type is `'A'` if the inherited genes are $\langle \mathtt{a}, \mathtt{a} \rangle$, $\langle \mathtt{a}, \mathtt{o} \rangle$ or $\langle \mathtt{o}, \mathtt{a} \rangle$.[9] `gene(fa,X)` (resp. `gene(mo,Y)`) means one inherits a blood type gene `X` from the father (resp. `Y` from the mother). `msw(gene,P,G)` represents an event that `G`, gene, is probabilistically chosen to be inherited from `P`, a parent.

[9] The left gene (resp. the right gene) is supposed to come from the father (resp. from the mother).

P_{F_1}, a joint distribution over the set of $\{\texttt{msw(gene,}t\texttt{,}g\texttt{)} \mid t \in \{\texttt{fa,mo}\}, g \in \{\texttt{a,b,o}\}\}$, is given as the product $P_{F_1} = P_{\texttt{fa}} \cdot P_{\texttt{mo}}$.

$$P_t(\texttt{msw(gene,}t\texttt{,a)} = x, \texttt{msw(gene,}t\texttt{,b)} = y, \texttt{msw(gene,}t\texttt{,o)} = z \mid \theta_a, \theta_b, \theta_o) \stackrel{\text{def}}{=} \theta_a^x \theta_b^y \theta_c^z$$

Here $t \in \{\texttt{fa,mo}\}$ and one of $\{x, y, z\}$ is 1 and the remaining two are 0. θ_a is the probability of inheriting gene `a` from a parent and so on. When $t \notin \{\texttt{fa,mo}\}$, we put $P_{F_t}(\cdot, \cdot, \cdot \mid, \theta_a, \theta_b, \theta_o) = 0$.

Suppose we observed blood types. We represent such observations by atoms chosen from $obs(DB_1) = \{\texttt{btype('A')}, \texttt{btype('B')}, \texttt{btype('O')}, \texttt{btype('AB')}\}$. Our concern then is to estimate hidden parameters $\theta_a, \theta_b, \theta_o$ from the observed atoms. First we reduce an observed atom to a disjunction of conjunction of `msw` atoms by unfolding [25]. Take `btype('A')` for instance. $\leftarrow$ `btype('A')` is unfolded by $comp(R_1)$ [8, 4] into $S_1 \vee S_2 \vee S_3$ such that

$$comp(R_1) \vdash \texttt{btype('A')} \leftrightarrow S_1 \vee S_2 \vee S_3$$

where

$$S_1 = \texttt{msw(gene,fa,a)} \wedge \texttt{msw(gene,mo,a)}$$
$$S_2 = \texttt{msw(gene,fa,a)} \wedge \texttt{msw(gene,mo,o)}$$
$$S_3 = \texttt{msw(gene,fa,o)} \wedge \texttt{msw(gene,mo,a)}$$

Each S_i is called an *explanation* for `btype('A')` and $\{S_1, S_2, S_3\}$ is called the *support set* of `btype('A')`.[10] Taking into account that S_1, S_2 and S_3 are mutually exclusive and `msw` atoms are independent, P_{DB_1}(`btype('A')` is calculated as

$$\begin{aligned} P_{DB_1}(\texttt{btype('A')} \mid \theta_a, \theta_b, \theta_o) &= P_{F_b}(S_1) + P_{F_b}(S_2) + P_{F_b}(S_3) \\ &= \theta_a^2 + 2\theta_a\theta_o \end{aligned}$$

Hence, the values of θ_a, θ_b and θ_o are determined as the maximizers of $\theta_a^2 + 2\theta_a\theta_o$ (maximum likelihood (ML) estimation). When there are multiple observations, say T observations $G_1, \ldots, G_T$, all we need to do is to maximize of the likelihood $\prod_{t=1}^{T} P_{DB_1}(G_t \mid \theta_a, \theta_b, \theta_o)$, and this optimization problem is solvable by the EM algorithm [10], an iterative algorithm for ML estimation. We face a fundamental problem here however because the EM algorithm in statistics has been defined only for numerically represented distributions and no EM algorithm is available for parameterized logic programs. So we have to derive a new one. Fortunately, thanks to the rigorous mathematical foundation of distribution semantics, it is straightforward to derive a (naive) EM algorithm for parameterized logic programs [16, 6, 22].

Given T observations $\mathcal{G} = \langle G_1, \ldots, G_T \rangle$ of observable atoms and a parameterized logic program DB that models the distribution of observable atoms, the

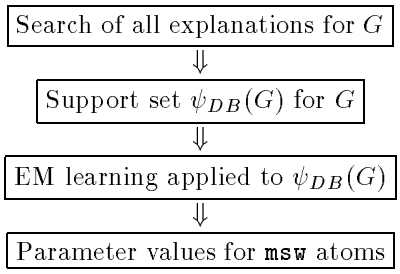
[10] An *explanation* S for a goal G w.r.t. a parameterized logic program $DB = F \cup R$ is a minimal conjunction S ($\subseteq F$) of `msw` atoms such that $S, R \vdash G$. The *support set* $\psi_{DB}(G)$ of G is the set of all explanations for G.

parameter set $\boldsymbol{\theta}$ which (locally) maximizes the likelihood $\prod_{t=1}^{T} P_{DB_1}(G_t \mid \boldsymbol{\theta})$ is computed by an EM algorithm *learn-naive*$(DB, \mathcal{G})$ below [16, 6]. Here $\psi_{DB}(G_t)$ $(1 \leq t \leq T)$ is the support set of G_t and $\boldsymbol{\theta}$ is the set of parameters associated with `msw` atoms appearing in some $\psi_{DB}(G_t)$. $\sigma_{i,v}(S)$ is $\left|\{\ n \mid \texttt{msw(}i\texttt{,}n\texttt{,}v\texttt{)} \in S\}\right|$, the number of how many times an atom of the form `msw(`i`,`n`,`v`)` appears in an explanation S.

---

procedure *learn-naive*$(DB, \mathcal{G})$ **begin**
Initialize $\boldsymbol{\theta}$ with appropriate values and ε with a small positive number ;
$\lambda^{(0)} := \sum_{t=1}^{T} \ln P_{DB}(G_t \mid \boldsymbol{\theta})$; % Compute the log-likelihood.
repeat
  foreach $i \in I, v \in V_i$ **do**
    $$\eta[i,v] := \sum_{t=1}^{T} \frac{1}{P_{DB}(G_t \mid \boldsymbol{\theta})} \sum_{S \in \psi_{DB}(G_t)} P_F(S \mid \boldsymbol{\theta}) \sigma_{i,v}(S);$$
  foreach $i \in I, v \in V_i$ **do**
    $\theta_{i,v} := \dfrac{\eta[i,v]}{\sum_{v' \in V_i} \eta[i,v']}$; % Update the parameters.
  $m := m + 1$;
  $\lambda^{(m)} := \sum_{t=1}^{T} \ln P_{DB}(G_t \mid \boldsymbol{\theta})$ % Compute the log-likelihood again.
until $\lambda^{(m)} - \lambda^{(m-1)} < \varepsilon$ % Terminate if converged.
end.

Our modeling process by a parameterized logic program DB of an observation G is summarized as follows (the case of multiple observations is analogous).



3.2 OLDT and the graphical EM algorithm

Symbolic-statistical modeling by parameterized logic programs has been formulated, but the real problem is whether it scales up or not. This is because for example the first step in the modeling process includes the search of all explanations for G, and it can be prohibitively time consuming. If there are exponentially many explanations for G as is the case with HMMs, search by backtracking would take also exponential time. We circumvent this difficulty by

employing OLDT search [26]. OLDT is a complete refutation procedure for definite clause programs which reuses previously computed results saved in a table. Because it avoids computing the same goal twice, the search of all explanations by OLDT often can be done in polynomial time. In case of HMMs, search time is $O(N^2L)$ (N the number of states, L the length of an input string) and in case of PCFGs, it is $O(N^3L^3)$ (N the number of non-terminals, L the length of an input sentence).

The adoption of OLDT search yields a very favorable side effect on achieving **Task-1**, **Task-2** and **Task-3** in Section 1.[11] Since OLDT search shares subrefutations, corresponding partial explanations are factored out, which makes it possible to represent the support set $\psi_{DB}(G)$ compactly as a hierarchical graph called a *support graph* reflecting the sharing structure between explanations for G. Look at *learn-naive*$(DB,\mathcal{G})$, especially the computation of $P_{DB}(G_t \mid \boldsymbol{\theta})$ and $\sum_{S \in \psi_{DB}(G_t)} P_F(S \mid \boldsymbol{\theta})\sigma_{i,v}(S)$. The computation of $P_{DB}(G_t \mid \boldsymbol{\theta}) = \sum_{S \in \psi_{DB}(G_t)} P_F(S \mid \boldsymbol{\theta})$ takes time proportional to the number of explanations for G_t if it is implemented faithfully to this formula. We reorganize this computation by introducing *inside probabilities* for logic programs as a generalization of the backward algorithm for HMMs [15] and the inside probabilities for PCFGs [1] in order to share subcomputations in $\sum_{S \in \psi_{DB}(G_t)} P_F(S \mid \boldsymbol{\theta})$. They are just a probability $P_{DB}(A \mid \boldsymbol{\theta})$ but recursively computed from the set of iff definitions of predicates and efficiently perform **Task-1** in time proportional to the size of a support graph.

The same optimization can be done for the computation of $\sum_{S \in \psi_{DB}(G_t)} P_F(S \mid \boldsymbol{\theta})\sigma_{i,v}(S)$ by introducing *outside probabilities* for logic programs as a generalization of the forward algorithm for HMMs [15] and the outside probabilities for PCFGs [1]. They represent the probability of "context" in which a specific atom occurs. Inside probabilities and outside probabilities are recursively computed from a support graph and the computation takes time only proportional to the size of the graph. Thus a new EM algorithm called the *graphical EM algorithm* incorporating inside probabilities and outside probabilities that works on support graphs has been proposed in [7]. We omit the description of the graphical EM algorithm though (see [22] for the detailed description). When combined with OLDT search, it is shown [7, 22] that

$$\text{OLDT search} + (\text{support graphs}) + \text{the graphical EM algorithm}$$

is as efficient as specialized EM algorithms developed domain-dependently (the Baum-Welch algorithm for HMMs, the Inside-Outside algorithm for PCFGs and the one for singly connected Bayesian networks) complexity-wise.[12] So **Task-3** is efficiently carried out in our framework.

It is also easy to design an algorithm for **Task-2** that finds out the most likely explanation in time linear in the size of a support graph [20], which generalizes

[11] Detailed explanations are found in [7, 22].

[12] This is a bit surprising as the graphical EM algorithm is a single algorithm generally designed for definite clause programs (satisfying certain conditions [7, 22]) allowing recursion and infinite domains.

the Viterbi algorithm for HMMs [15]. We therefore can say that the introduction of support graphs (constructed from OLDT search) and that of inside probabilities and outside probabilities for logic programs computed from a support graph efficiently solve the three tasks laid in Section 1.

3.3 Learning experiments

It is appropriate here to report some performance data about the graphical EM algorithm. We conducted learning experiments to compare the performance of the graphical EM algorithm with that of the Inside-Outside algorithm, the standard parameter learning algorithm for PCFGs, by using a real corpus and a PCFG developed for it.

We used ATR corpus [28] containing 10,995 sentences whose minimum length, average length and maximum length are respectively 2, 9.97 and 49. The grammar, G_{ATR}, is a hand crafted CFG comprising 860 rules (172 nonterminals and 441 terminals) [27] developed for ATR corpus. G_{ATR}, being not in Chomsky normal form, was translated into CFG G^*_{ATR} in Chomsky normal form for the Inside-Outside algorithm to be applicable.[13] G^*_{ATR} contains 2,105 rules (196 nonterminals and 441 terminals).

We created subgroups S_L ($1 \leq L \leq 25$) of sentences with length L and $L+1$ by randomly choosing 100 sentences for each L from the corpus. Support graphs for S_Ls w.r.t. G_{ATR} and G^*_{ATR} were generated by Tomita (Generalized LR) parser by the way. After these preparations, for each S_L, we compared time per iteration for the Inside-Outside algorithm to update the parameters of G^*_{ATR}[14] with that for the graphical EM algorithm to update the parameters of G_{ATR} and G^*_{ATR}. The results are shown in Figure 3.

A vertical axis shows the required time. A horizontal axis is L, the length parameter of learned sentences. A curve labeled I-O in the left graph[15] is drawn by the Inside-Outside algorithm. It is cubic as $O(N^3L^3)$ predicts. The curves labeled gEM in the right graphs are drawn by the graphical EM algorithm. One with a comment "original" is for G_{ATR}. As seen from the graph, the graphical EM algorithm runs almost 850 times faster than the Inside-Outside algorithm at length 10 (the average sentence length in the ATR corpus is 9.97). The other one with "Chomsky NF" is the curve obtained by applying the graphical EM algorithm to G^*_{ATR}.[16] The graphical EM algorithm still runs 720 times faster than the the Inside-Outside algorithm. Further more, a closer inspection reveals that update time by the graphical EM algorithm is almost linearly dependent

[13] The Inside-Outside algorithms requires a CFG to be in Chomsky normal form while the graphical EM algorithm accepts every form of production rules.

[14] We used a somewhat improved version of the Inside-Outside algorithm that avoids apparently redundant computations. For instance $p \cdot \sum_x q(x)$ is immediately evaluated as 0 if $p = 0$.

[15] The right graph is an enlarged version of the left one.

[16] We would like to emphasize again that the graphical EM algorithm does not require a CFG in Chomsky normal form. This comparison is only made to measure time for updating parameters of a common PCFG.

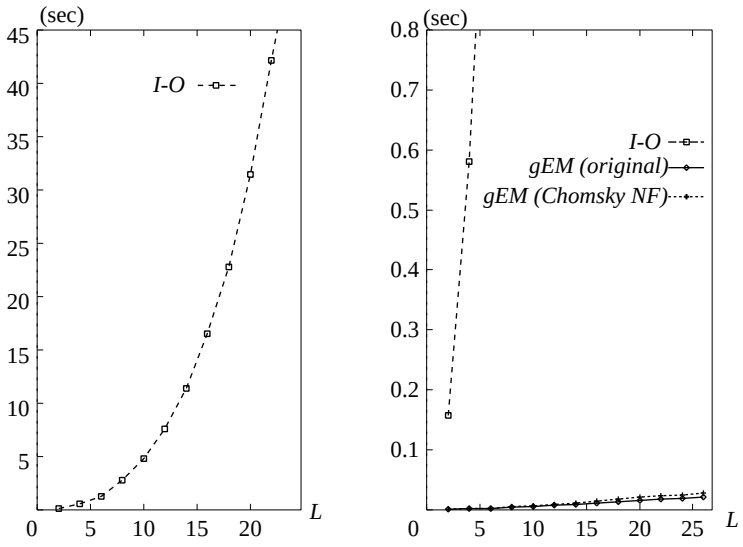


Fig. 3. The Inside-Outside algorithm vs. the graphical EM algorithm

on L, not on L^3. The conclusion is that although the graphical EM algorithm has the same time complexity as the Inside-Outside algorithm, the difference in their performances is enormous when applied to real data, and the almost linear dependency of the graphical EM on the sentence length suggests that it can cope with the learning of a complex stochastic grammar applied to a bigger corpus with longer sentences.

3.4 PRISM

As an implementation of distribution semantics, a symbolic-statistical modeling language PRISM (URL = `http://sato-www.cs.titech.ac.jp/prism/`) has been developed [16–18]. It is intended for modeling complex symbolic-statistical phenomena such as discourse interpretation in natural language processing, consumer behavior, gene inheritance interacting with complicated social rules [18]. As a programming language, it is an extension of Prolog with built-in predicates including `msw` predicate and other special predicates for manipulating `msw` atoms and their parameters.

A PRISM program is comprised of three parts, one for directives, one for modeling and one for utilities. The directive part contains declarations telling the system what `msw` atoms will be used in the program. The modeling part is a non-unit definite clause program like DB_1 that defines the denotation of the program containing `msw` atoms. The last part, the utility part, is an arbitrary Prolog program which refers to predicates defined in the modeling part. We can use there `learn` built-in predicate to carry out EM learning by *learn-naive*(DB,$\mathcal{G}$) from

observed atoms. There are three modes of execution. The sampling execution corresponds to a random sampling drawn from the distribution defined by the modeling part. The second one computes the probability of a given atom. The third one returns the support set for a given goal. These modes of execution are available through built-in predicates. Currently the implementation of the graphical EM algorithm and the simplified OLDT search mechanism is underway.

4 Reinforcement learning and parameterized logic programs

4.1 Reinforcement learning

We turn our attention here to on-line learning, i.e. data come one by one and learning is done each time. The idea is to use reinforcement learning [24, 29] (see [5] for a survey) to reactively adjust parameters $\theta_{i,k}$ for `msw(`i`,`n`,`v_k`)` ($1 \leq k \leq N$). Reinforcement learning is a model of learning good behavior by trial and error, by receiving reward and penalty from a random environment.[17] For instance, in Q-learning [29], one of the most popular reinforcement learning methods based on MDP (Markov Decision Process) [11], an agent is expected to learn, while randomly or systematically walking in the environment, a best action among several ones at each state to maximize its discounted or total expected reward. In this paper however, we adopt *learning automaton*s [12, 14] instead of MDP as an underlying theoretical framework because of their affinity with parameterized logic programs as a learning device for `msw` atoms.[18]

A *learning automaton* (henceforth referred to as LA) is a reactive learning device working in a random environment. On each execution, it selects an action in proportion to choice probabilities assigned to each action, and adjust them in response to the gained reward so that profitable actions are likely to be selected again. The point is that, while it just makes a random choice in the beginning, by repeatedly choosing an action and adjusting probabilities, it asymptotically converges to a stage where it only takes the most rewarding action (in average sense) [12, 14].

We embed LAs in a parameterized logic program to automatically adjust parameters of `msw` atoms. Aside from the theoretical question about convergence of their collective behavior, the learning ability of LAs added to logic programs makes them reactive to the environment. We call such programs as *reactive logic program*s. Reactive logic programs are intended to model an agent with rich background knowledge working in an uncertain world who learns from the environment how to behave optimally.

[17] By random, we mean the reward returned for taking an action is a random variable. In this paper, we assume the environment is stationary, i.e. the distribution of reward does not change with time. Also we leave out penalty as a type of response for convenience.

[18] They require neither the notion of state nor that of state change.

4.2 Learning Automaton

A learning automaton (LA) [12, 14][19] is a learning device applicable to a situation in which there are several possible actions with different probabilistic rewards, and we want to know the best one that maximizes the average reward. The most typical situation would be gambling. Imagine playing with a slot machine that has N levers (N-armed bandit). Pulling a lever gives us winnings if we are lucky. We want to know the best lever yielding the maximum average payoff while betting coins repeatedly. Which lever should we pull at nth time?

One way to solve this problem is to use an LA. Suppose there are N possible actions $\alpha_1, \ldots, \alpha_N$. We associate a choice probability $\theta_i(n)$ with each α_i ($1 \leq i \leq N$) so that α_i is chosen with probability $\theta_i(n)$ at time n. The distribution for actions at time n is expressed by a random vector $\boldsymbol{\theta}(n) = \langle \theta_1(n), \ldots, \theta_N(n) \rangle$ ($\sum_i \theta_i(n) = 1$). Suppose α_i is chosen. By executing α_i, we get a reward γ from an environment[20], with which we update $\boldsymbol{\theta}(n)$ by using L_{R-I} (Linear Reward-Inaction) scheme [12][21]

$$\theta_i(n+1) = \theta_i(n) + c_n\gamma(1 - \theta_i(n))$$
$$\theta_j(n+1) = (1 - c_n\gamma)\theta_j(n) \quad (j \neq i)$$

where c_n ($0 \leq c_n \leq 1$) is a *learning rate* at time n. The scheme says that if the reward γ is non-zero, we will give more chance of selection to action α_i. Since γ is a random variable, so are the $\theta_i(n)$s. Figure 4 illustrates an LA where edges denote actions and d_i is the average reward for action α_i ($1 \leq i \leq N$).

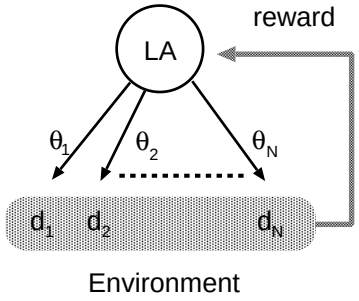


Fig. 4. A learning automaton

It is known that the L_{R-I} scheme can improve the average reward of the LA [12]. To see it, let $\boldsymbol{\theta}(n)$ be a current distribution. $M(n)$, the average reward at time n conditioned on $\boldsymbol{\theta}(n)$, is derived as

$$M(n) = E[\gamma \mid \boldsymbol{\theta}(n)]$$

[19] The type of LAs we use is called a variable-structure stochastic automaton in [12].
[20] We assume γ is normalized such that $0 \leq \gamma \leq 1$.
[21] There are other learning schemes [12]. The L_{R-I} scheme is the simplest one.

$$= \sum_{i=1}^{N} \theta_i(n) d_i \tag{3}$$

where $E[\cdot]$ denotes expectation and d_i is the average reward for taking action α_i $(1 \leq i \leq N)$. $M(n)$ is a random variable as a function of $\boldsymbol{\theta}(n)$. $E[M(n+1) \mid \boldsymbol{\theta}(n)]$, the average reward at time $n+1$ conditioned on $\boldsymbol{\theta}(n)$, then comes out as

$$\begin{aligned} E[M(n+1) \mid \boldsymbol{\theta}(n)] &= E[\sum_{i=1}^{N} \theta_i(n+1) d_i) \mid \boldsymbol{\theta}(n)] \\ &= M(n) + \frac{c_n \gamma}{2} \sum_{i,j} (d_i - d_j)^2 \theta_i(n) \theta_j(n) \qquad (4) \\ &\geq M(n) \qquad (5) \end{aligned}$$

By taking expectation of both sides in (5), we conclude

$$E[M(n+1)] \geq E[M(n)] \tag{6}$$

i.e. the average reward is increasing at every time step.

It is also important to note that $\{M(n)\}_{n=1,2,\ldots}$ forms a submartingale. Since $E[M(n)] \leq 1$ holds for all n, $\{M(n)\}$ has an a.s.(almost sure) convergence, which in turn implies, when the learning rate c_n is kept constant c, we see $\theta_j(n) \xrightarrow{\text{a.s.}} 0$ or 1 for every $\theta_j(n)$ $(1 \leq j \leq N)$. Regrettably, this does not necessarily mean that the probability $\theta_i(n)$ corresponding to the maximum average reward d_i will converge to 1 with probability 1 after infinite trials. However, it was proved that the probability of $\theta_i(n)$ not converging to 1 is made arbitrarily small by setting c small enough [12]. Also, recently it also has been proved that if we decay the learning rate c_n like $c_n = \frac{b}{n+a}$ $(a > b, 1 > b > 0)$, and if initial probabilities are positive, i.e. $\theta_j(1) > 0$ $(1 \leq j \leq N)$ and $d_i > d_j$ for $j \neq i$, we have $\theta_i(n) \xrightarrow{\text{a.s.}} 1$ [14].

4.3 LA networks

In reactive logic programming, parameters $\theta_{i,k}$ $(1 \leq k \leq N)$ associated with `msw(`i`,`n`,`$\cdot$`)` in a parameterized logic program are trained by an LA. The embedded LAs in the program as a whole constitute a tree, a dag (directed acyclic graph) or a more general graph structure. Unfortunately, unlike a single LA, not much has been known about the convergent behavior of the graph-structured LAs.[22] So we here introduce *LA networks*, the class of LAs organized as a dag and discuss their mathematical properties. Although we are aware that our reactive logic programming sometimes requires the use of a more general graph

[22] When there are multiple LAs connected with each other, the analysis becomes much harder. The reason is that for a given LA, other LAs are part of the environment and the average reward for a selected action becomes non-stationary, i.e. changes with time as the learning of the other LAs proceed, which prevents the derivation of formulas (5) which entails (6).

structures containing loops, we introduce LA networks because they are relatively expressive but yet analyzable.

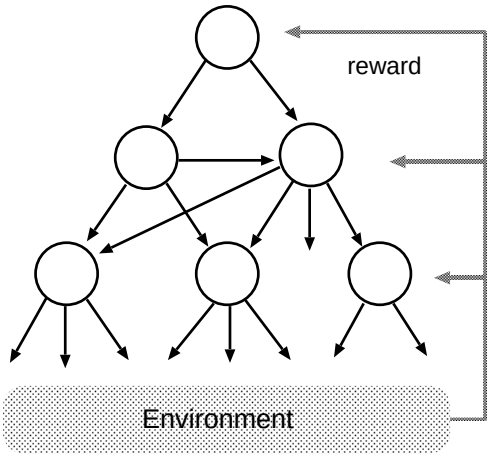


Fig. 5. A LA network

Formally, a *LA network* is a connected dag (Figure 5) such that a node and its outgoing edges comprise an LA and there is the only one node with no incoming edges called the *root* LA. A node which has no descendent node is called a *leaf* LA. Each edge is labeled by an action and by the probability associated with it (see Figure 4). Actions are chosen sequentially from the root LA to a leaf LA in an obvious way and the reward is simultaneously given to each LA on the path from the root to the leaf to update choice probabilities.

Let $\boldsymbol{\theta}(n)$ be probabilities for all edges in the graph at time n, and $M(n)$ the average reward at time n conditioned on $\boldsymbol{\theta}(n)$. When the shape of the LA network is a tree, we can prove (6) w.r.t. the root LA by setting learning rates differently for each LA and for each trial (hierarchical systems [12]). On the other hand, the need of computing different learning rates is computationally disadvantageous compared to using a common constant learning rate, for which case we can still prove (5) (and hence (6)) under the condition that the reward is binary i.e. {0,1} (proof omitted).

For LA networks which are not trees, we do not know at the moment if (5) holds w.r.t. the root LA for a constant learning rate. However, we proved in [19] that (5), the increase of the average reward, holds for arbitrary LA networks as long as a common learning rate $\frac{1}{n+a}$ with large enough a is used by all LAs at time n. That is, with this decaying learning rate, an LA network gains more reward on average every time the program is executed. What is disappointing about it is that it gives too slow convergence, experimentally. So we decided instead to use a constant learning rate set by a user.

4.4 LA library

We built a library called *LA library* so that various built-in predicates are available to manipulate LAs in a parameterized logic program. We here explain some of them. First `new_choice(`i`,`$list$`)` creates `msw` atoms $\{$`msw(`i`,`t`,`v`)` $\mid (v \in list)\}$ with an LA attached to them with equal values for vs. We use fresh t every on every execution of such `msw` atom. `new_choice(sanfrancisco, [neworleans, chicago])` for instance sets up a new LA named `sanfrancisco` that outputs equiprobably `neworleans` or `chicago`. *list* is a list containing arbitrary terms but i must be a ground term. `choice(`i`,V)` executes an LA named i with a current distribution and the output is returned in `V`. `reinforce_choice(`c`,`$list$`)` adjusts according to the L_{R-I} learning scheme choice probabilities in all LAs contained in *list* using the base line cost c (see Section 4.5).

4.5 Reinforcement condition

It must be emphasized that we are a tackling optimization problem. We use a reactive logic program expecting that by repeatedly running it, the parameters of LAs are automatically adjusted, and the most rewarding choice sequence will eventually gain the highest execution probability. In view of this, we should not apply `reinforce_choice/2` unconditionally every time computation is done, because we are not seeking for a correct answer but for a better answer that corresponds to a more rewarding choice sequence and there is no reason for giving out a reward to whatever answer is computed.

We try to obtain a better answer by comparing a new one and the old ones. Suppose we run a program and the computed answer is judged better than the old ones. We reinforce every LA involved in the computation by invoking `reinforce_choice/2`. The criterion for invocation is called a *reinforcement condition*. We can conceive of several reinforcement conditions depending on the type of problem, but in this paper, we use the following heuristic condition.[23]

$$\frac{C_s + C_a}{2} > C$$

where C is the cost of the computed answer, C_s the least cost achieved so far and C_a the average cost in the past M trials ($M = 50$ is used in this paper).

4.6 Going to the East

Now we present a very simple reactive logic program for a stochastic optimization problem. The problem is to find a route in Figure 6 from San Francisco to New York but a traveling cost between adjacent cities (cost attached to edges in Figure 6) is supposed to vary randomly for various reasons on each trial. The task is to find a route giving the least average cost.

We first define `travel/2` predicate to compute the route and cost of traveling from the current city (`Now`) to New York.

[23] Note that the condition is stated in terms of cost, so the lesser, the better.

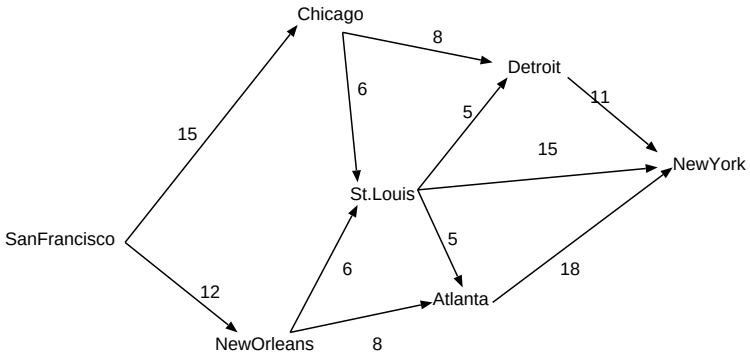


Fig. 6. Going to the East

```
% Destination = New York
 travel(newyork,0,[newyork]).
 travel(Now,Cost,[Now|Route]) :-
    choice(Now,Next),
    travel(Next,C1,Route),
    link_cost(Now,Next,C2),
    Cost is C1 + C2.
```

`link_cost/2` returns the traveling cost of adjacent two cities. It consists of a basic cost plus a random increment (uniform distribution, max 20%) determined by `random_float/2`.

```
 link_cost(Now,Next,Cost) :-
    cost(Now,Next,C1),
    random_float(0.2,C2),
    Cost is C1 * (1+C2).
```

A table of basic cost is given by unit clauses (numbers are artificial).

```
 cost(sanfrancisco, neworleans, 10).
 cost(sanfrancisco, chicago, 15).
 cost(sanfrancisco, stlouis, 12).
 ...
 cost(stlouis, newyork, 15).
 cost(atlanta, newyork, 18).
```

After 10,000 trials of `?- travel(sanfrancisco,_,Route)` with a learning rate 0.01 and the reinforcement condition previously stated, we obtained the following route. As can be seen, the recommended route (`San Francisco, New Orleans, St. Louis, New York`) gives the least average cost. How learning converges is depicted in Figure 8. It plotted the average cost in the past 50 trials. We see that the average cost decreases rapidly in the initial 1,000 trials and then slowly. Theoretically speaking however, there is no guarantee of the

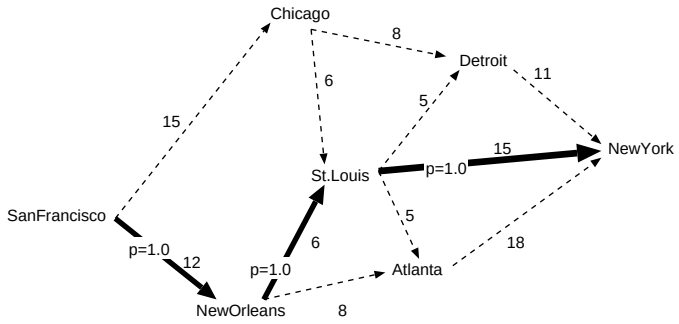


Fig. 7. A route to New York

decreasing average cost when the learning rate is kept constant in an LA network with loops, but taking into account the fact that the average cost decreases with a decaying learning rate (Section 4.3), we may reasonably expect it happens as well provided the learning rate is kept sufficiently small.

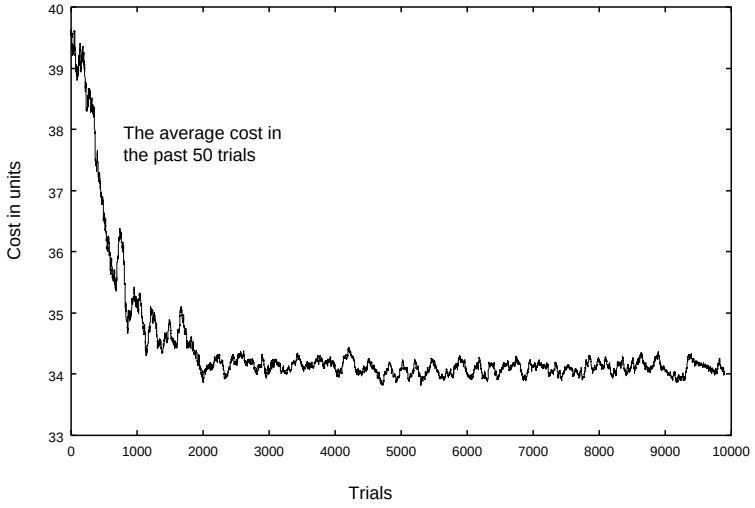


Fig. 8. Learning curve

5 Conclusion

We have presented attempts to make programs more adaptive by adding learning ability. In order to unify computing and learning at semantic level, distribution

semantics is introduced for parameterized logic programs that contain probabilistic facts with a parameterized distribution. The new semantics is a generalization of the least model semantics for definite clause programs to possible world semantics with a distribution. We then apply the semantics to statistical modeling in Section 3 and to reactive programming in Section 4. It is shown that efficient symbolic-statistical modeling is made possible by the graphical EM algorithm working on a new data structure called support graphs. The learning experiments have shown that the graphical EM algorithm combined with OLDT search is not only as competitive as existing specialized EM algorithms complexity-wise, but in the case of PCFGs, runs 850 times faster than the Inside-Outside algorithm, a rather pleasant surprise. We also applied distribution semantics to reinforcement learning of parameterized logic programs by learning automatons, and showed a simple reactive programming example applied to a stochastic search problem.

References

1. Baker, J. K., Trainable grammars for speech recognition, *Proc. of Spring Conference of the Acoustical Society of America*, pp.547–550, 1979.
2. Castillo, E., Gutierrez, J.M., and Hadi, A.S., *Expert Systems and Probabilistic Network Models*, Springer-Verlag, 1997.
3. Chow,Y.S. and Teicher, H., *Probability Theory* (3rd ed.), Springer, 1997.
4. Doets, K., *From Logic to Logic Programming*, MIT Press, Cambridge, 1994.
5. Kaelbling,L.P. and Littman,M.L., Reinforcement Learning: A Survey, *J. of Artificial Intelligence Research*, Vol.4, pp.237–285, 1996.
6. Kameya, Y., Ueda, N. and Sato, T., A graphical method for parameter learning of symbolic-statistical models, *Proc. of DS'99*, LNAI 1721, pp.264–276, 1999.
7. Kameya, Y. and Sato, T., Efficient EM learning for parameterized logic programs, *Proc. of CL2000*, LNAI 1861, pp.269–294, 2000.
8. Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, 1984.
9. Manning, C. D. and Schütze, H., *Foundations of Statistical Natural Language Processing*, The MIT Press, 1999.
10. McLachlan, G. J. and Krishnan, T., The EM Algorithm and Extensions, Wiley Interscience, 1997.
11. Monahan,G.E., A Survey of Partially Observable Markov Decision Processes: Theory, Models, and Algorithms, *Management Science* Vol.28 No.1, pp.1–16, 1982.
12. Narendra,K.S. and Thathacher,M.A.L., *Learning Automata: An Introduction*, Prentice-Hall Inc., 1989.
13. Pearl, J., *Probabilistic Reasoning in Intelligent Systems*, Morgan Kaufmann, 1988.
14. Poznyak,A.S. and Najim,K., *Learning Automata and Stochastic Optimization*, Lecture Notes in Control and Information Sciences 225, Springer, 1997.
15. Rabiner, L. R. and Juang, B., *Foundations of Speech Recognition*, Prentice-Hall, 1993.
16. Sato, T., A statistical learning method for logic programs with distribution semantics, *Proc. of ICLP'95*, pp.715-729, 1995.
17. Sato, T. and Kameya, Y., PRISM:A Language for Symbolic-Statistical Modeling, *Proc. of IJCAI'97*, pp.1330–1335, 1997.
18. Sato, T., Modeling Scientific Theories as PRISM Programs, *ECAI Workshop on Machine Discovery*, pp.37–45, 1998.

19. Sato, T., On Some Asymptotic Properties of Learning Automaton Networks, Techinical report TR99-0003, Dept. of Computer Science, Tokyo Institute of Technology, 1999.
20. Sato,T. and Kameya, Y., A Viterbi-like algorithm and EM learning for statistical abduction", *Proc. of UAI2000 Workshop on Fusion of Domain Knowledge with Data for Decision Support*, 2000.
21. Sato,T., Statistical abduction with tabulation, submitted for publication, 2000.
22. Sato,T. and Kameya, Y., Parameter Learning of Logic Programs for Symbolic-statistical Modeling, submitted for publication, 2000.
23. Sterling, L. and Shaprio, E. *The Art of Prolog*, The MIT Press, 1986.
24. Sutton,R.S., Learning to predict by the method of temporal difference, *Machine Learning*, Vol.3 No.1, pp.9–44, 1988.
25. Tamaki, H. and Sato, T., Unfold/Fold Transformation of Logic Programs, *Proc. of ICLP'84*, Uppsala, pp.127–138, 1984.
26. Tamaki, H. and Sato, T., OLD resolution with tabulation, *Proc. of ICLP'86*, London, LNCS 225, pp.84–98, 1986.
27. Tanaka, H. and Takezawa, T. and Etoh, J., Japanese grammar for speech recognition considering the MSLR method (in Japanese), *Proc. of the meeting of SIG-SLP (Spoken Language Processing)*, 97-SLP-15-25, Information Processing Society of Japan, pp.145–150, 1997.
28. Uratani, N. and Takezawa, T. and Matsuo, H. and Morita, C., ATR Integrated Speech and Language Database (in Japanese), TR-IT-0056, ATR Interpreting Telecommunications Research Laboratories, 1994.
29. Watkins,J.C.H. and Dayan,P., Q-learning, *Machine Intelligence*, Vol.8, No.3, pp.279–292, 1992.
30. Wetherell, C.S., Probabilistic languages: a review and some open questions, *Computing Surveys*, Vol.12, No.4, pp.361–379, 1980.