TLP **14** (6): 909–937, 2014. © Cambridge University Press 2013
doi:10.1017/S1471068413000562 First published online 4 November 2013

Infinite probability computation by cyclic explanation graphs

TAISUKE SATO
Tokyo Institute of Technology, 2-12-1 Ookayama, Meguro, Tokyo, Japan
(*e-mail:* `sato@mi.cs.titech.ac.jp`)

PHILIPP MEYER
Technical University Munich, Munich, Germany
(*e-mail:* `meyerphi@in.tum.de`)

submitted 13 October 2012; revised 22 February 2013; accepted 22 August 2013

Abstract

Tabling in logic programming has been used to eliminate redundant computation and also to stop infinite loop. In this paper[1] we investigate another possibility of tabling, i.e. to compute an infinite sum of probabilities for probabilistic logic programs. Using PRISM, a logic-based probabilistic modeling language with a tabling mechanism, we generalize prefix probability computation for probabilistic context-free grammars (PCFGs) to probabilistic logic programs. Given a top-goal, we search for all proofs with tabling and obtain an explanation graph which compresses them and may be cyclic. We then convert the explanation graph to a set of linear probability equations and solve them by matrix operation. The solution gives us the probability of the top-goal, which, in nature, is an infinite sum of probabilities. Our general approach to prefix probability computation through tabling not only allows to deal with non-probabilistic context-free grammars such as probabilistic left-corner grammars but has applications such as plan recognition and probabilistic model checking and makes it possible to compute probability for probabilistic models describing cyclic relations.

KEYWORDS: tabling, probability computation, prefix, probability equation

1 Introduction

Combining logic and probability in a logic programming language provides us with a powerful modeling tool for machine learning. The resulting language allows us to build complex yet comprehensible probabilistic models in a declarative way. PRISM Sato and Kameya (1997, 2001, 2008) is one of the earliest attempts to develop such a language. It covers a large class of known models, including Bayesian

[1] This paper is based on Sato and Meyer (2012) (Sato, T. and Meyer, P. 2012. Tabling for infinite probability computation. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP12)*, Budapest, Hungary, Leibniz International Proceedings in Informatics, vol. 17. Kluwer, Boston, MA, 348–358) and extended with a theorem for prefix PCFGs, a detailed explanation for tabling, the addition of probabilistic left-corner grammars, experiments with a real corpus and two non-linguistic applications: plan recognition and probabilistic model checking.

networks (BNs), hidden Markov models (HMMs) and probabilistic context-free grammars (PCFGs), and computes probabilities with the same time complexity as their standard algorithms[2] as well as unknown models such as probabilistic context-free graph grammars (Sato 2008).

The efficiency of probability computation in PRISM is attributed to the use of tabling (Tamaki and Sato 1986; Warren 1992; Rocha *et al.* 2005; Zhou *et al.* 2008; Zhou *et al.* 2010)[3] that eliminates redundant computation. Given a top-goal G, we search for all proofs of G[4] while tabling probabilistic goals and recording their logical dependencies as a set $expl(G)$ of propositional formulas with a graphical structure, which we call an *explanation graph* for G (Sato and Kameya 2001). By applying dynamic programming to $expl(G)$ when it is acyclic and partially ordered, we can efficiently compute the probability of G in time linear in the size of the graph. The use of tabling also gives us another advantage over non-tabled computation; it stops infinite loop by detecting recurrence patterns of goals. Tabled logic programs thus can directly use left recursive rules in CFGs without the need of converting them to right recursive ones.

In this paper we investigate another possibility of tabling that has gone unnoticed in the non-probabilistic setting; we apply tabling to compute an infinite sum of probabilities that typically appears in the context of prefix probability computation for PCFGs (Jelinek and Lafferty 1991; Stolcke 1995; Nederhof and Satta 2011a). PCFGs are a probabilistic extension of CFGs in which CFG rules are assigned probabilities and the probability of a sentence is computed as a sum-product of probabilities assigned to the rules used to derive the sentence (Baker 1979; Manning and Schütze 1999). A prefix u is an initial substring of a sentence. The probability of the prefix u is a sum of probabilities of infinitely many sentences of the form uv for some string v. Prefix probability is useful in speech recognition as discussed in Jelinek and Lafferty (1991). We generalize this prefix probability computation for PCFGs to probability computation on *cyclic explanation graph*s generated by PRISM programs using tabled search. Since we can use arbitrary programs, our approach not only allows us to deal with non-PCFGs, such as probabilistic left-corner grammars (PLCGs) in addition to PCFGs, but also opens a way to practical applications such as planning and model checking, as will be demonstrated in Sections 5 and 6 respectively.

PRISM constructs an explanation graph for a top-goal G by collecting clauses used in a proof of G while checking if there is a loop, i.e. if there is a proved goal that calls itself as one of its descendent goals. Loops easily occur, for example, in programs for prefix of PCFGs and in ones for the Markov chain containing self-loops. By default whenever PRISM detects a loop during the construction of the explanation graph, it fails with an error message but by setting `error_on_cycle`

[2] They are the junction tree algorithm for BNs, the forward–backward algorithm for HMMs and the inside–outside (IO) algorithm for PCFGs.

[3] Tabling is also employed by other probabilistic logic programming languages such as ProbLog (Mantadelis and Janssens 2010) and PITA (Riguzzi and Swift 2011).

[4] In this paper, we mean by a proof of a goal G an SLD-refutation of $\Leftarrow G$.

flag to `off` using `set_prism_flag/2`, we can let PRISM skip loop checking, and as a result can obtain a cyclic explanation graph. So constructing cyclic explanation graphs requires no extra cost in PRISM.

However, while computing probability from such cyclic graphs is possible (Etessami and Yannakakis 2009), efficient computation is difficult except for the case of *linear cyclic explanation graph*s that can be turned into a set of linear probability equations straightforwardly solvable by matrix operation. So the practical issue is to guarantee the linearity of cyclic explanation graphs. We specifically examine a PRISM program for prefix probability computation for PCFGs and prove that the program always generates linear cyclic explanation graphs. We also prove that the probability equations obtained from the linear cyclic explanation graphs are solvable by matrix operation under some mild assumptions on PCFGs.

To empirically test our approach, we conduct experiments of computing prefix probability for a PCFG and also for a PLCG using a real corpus of moderate size. To our knowledge, prefix probability computation for PLCGs is new and has not been attempted so far. As applications, we apply prefix probability computation to plan recognition in which action sequences are derived from plans using a PCFG. Our task is to infer, given an action sequence, the plan underlying it. Note that we do not require the action sequence to be complete as a sentence unlike previous approaches (Bobick and Ivanov 1998; Amft *et al.* 2007; Lymberopoulos *et al.* 2007; Geib and Goldman 2011) as we are able to deal with prefix action sequences. We also apply our approach to the reachability probability problem in probabilistic model checking (Hinton *et al.* 2006; Gorlin *et al.* 2012). This class of problems needs to describe Markov chains and to compute the reachability probability between two states. The experiment suggests that our approach is reasonably fast.

In what follows, we first review probability computation in PRISM in Section 2. In Section 3 we explain how prefix probability is computed for PCFGs in PRISM together with some formal proofs. Then we tackle the problem of prefix probability computation for PLCGs in Section 4. We apply prefix probability computation to plan recognition in Section 5, and to the reachability probability problem in probabilistic model checking in Section 6. Section 7 contains related work and Section 8 is the conclusion. We assume the reader has a basic familiarity with PRISM (Sato and Kameya 2001; Sato and Kameya 2008).

2 Probability computation in PRISM

We review probability computation in PRISM for self-containedness. PRISM[5] is a probabilistic extension of Prolog with built-in predicates for machine learning tasks such as parameter learning and Bayesian inference (Sato and Kameya 2001; Sato and Kameya 2008). Theoretically a PRISM program *DB* is a union $R \cup F$ of a set of definite clauses R and and a set F of ground probabilistic atoms of the form `msw(`id`,`v`)` that represent simple probabilistic choices, where *id* and v are ground

[5] `http://sato-www.cs.titech.ac.jp/prism/`

terms.[6] Using probabilities assigned to msw atoms, DB uniquely defines a probability measure $P_{DB}(\cdot)$ over possible Herbrand interpretations from which the probability of an arbitrary closed formula is calculated. Practically however PRISM programs are just Prolog programs that use msw atoms introduced by values/2 declarations[7] as probabilistic primitives[8] as shown in Figure 1 of Section 3.1.

In PRISM, the probability $P_{DB}(G)$ of a ground atom G w.r.t. a program DB is basically computed as a sum of probabilities of all *explanations* for G, where an *explanation for* G is a conjunction $E = \mathtt{msw}_1 \wedge \cdots \wedge \mathtt{msw}_k$ of ground msw atoms such that $\mathtt{msw}, \ldots, \mathtt{msw}_k, comp(R) \vdash G$.[9] However, naively computing $P_{DB}(G)$ is computationally expensive because of exponentially many explanations. Instead we compute $P_{DB}(G)$ in three steps. In the first step, we perform tabled search for all proofs of G while recording clause instantiations used in a proof in the external memory area (through some C-interface predicates). In the second step, we construct an explanation graph $expl(G)$ for G from recorded clause instantiations. It compactly represents all possible explanations for G by sub-formula sharing. In the third step, we convert $expl(G)$ to a set of probability equations and obtain $P_{DB}(G)$ by solving it using dynamic programming. In the following we discuss each of them in detail.

2.1 Tabled search and explanation graphs

In general, there are exponentially many proofs of G and so are explanations. Fortunately, we can often compress them to an equivalent but much smaller representation by factoring out common sub-conjunctions as intermediate goals (Sato and Kameya 2001; Zhou *et al.* 2008). We can express the set of all explanations as a set of *defining formula*s that take the form $H \Leftrightarrow \alpha_1 \vee \cdots \vee \alpha_M$. Here H is the top-goal G or an intermediate goal. Hereafter the top-goal and intermediate goals are collectively called *defined goal*s. We call each $H \Leftarrow \alpha_i$ $(1 \leqslant i \leqslant M)$ a *defining clause* for H, where α_i is a conjunction $C_1 \wedge \cdots \wedge C_m \wedge \mathtt{msw}_1 \wedge \cdots \wedge \mathtt{msw}_n$ $(0 \leqslant m, n)$ of defined goals $\{C_1, \ldots, C_m\}$ and msw atoms $\{\mathtt{msw}_1, \ldots, \mathtt{msw}_n\}$.

We say that H is a *parent* of C_j $(1 \leqslant j \leqslant m)$ and call the transitive closure of this parent–child relation the *ancestor relation*. The whole set of defining formulas is denoted by $expl(G)$ and called an *explanation graph* for G as is called so far. In $expl(G)$ each defined goal has only one defining formula and possibly is referred to by other defined goals.

[6] We use lower case strings to represent ground terms, atoms etc. in this paper.

[7] A declaration $\mathtt{values}(id, [v_1, \ldots, v_N])$ introduces a set of ground probabilistic atoms $\mathtt{msw}(id, v_i) (1 \leqslant i \leqslant N)$. They represent as a group a discrete random variable on a sample space $V_{id} = \{v_1, \ldots, v_N\}$. So only one of them becomes probabilistically true and others are false. To specify their distribution we use a PRISM command $\mathtt{set\_sw}(id, [\theta_1, \ldots, \theta_N])$ that sets $P_{DB}(\mathtt{msw}(id, v_i))$, the probability of $\mathtt{msw}(id, v_i)$ being true, to θ_i $(1 \leqslant i \leqslant N)$, where $\sum_{v \in V_{id}} \theta_v = 1$.

[8] Procedurally, executing $\mathtt{msw}(id, \mathtt{X})$ as a PRISM goal returns $\mathtt{X} = v_i$ with probability θ_i. On the other hand, a ground goal $\mathtt{msw}(id, v)$ is equivalent to $\mathtt{msw}(id, \mathtt{X}), \mathtt{X}{=}v$ and fails if the value returned in X differs from v. We assume that different occurrences of msw/2 atom in a program or in a proof are independent and if they have the same id, they represent samples from independent and identically distributed random variables (Sato and Kameya 2001).

[9] $comp(R)$ is the completion of R. It is a union of the if-and-only-if form of R and the so-called Clark's equational theory.

An n-ary predicate `p/n` is said to be *probabilistic* if the predicate symbol `p` is `msw` or recursively, there is a clause in *DB* such that the head contains the predicate symbol `p` and a probabilistic predicate occurs in the body. Likewise, an atom `p(`$t_1,\ldots,t_n$`)` is probabilistic if `p/n` is probabilistic. Then roughly $expl(G)$ is obtained from exhaustive tabled search for all proofs of G while tabling probabilistic predicates in *DB*. What we actually use however is not *DB* but another non-probabilistic Prolog program DB' translated from *DB* that has a mechanism of recording instantiated clauses used in a proof of G. We construct $expl(G)$ by tabled search for all proofs of G w.r.t. DB' while tabling probabilistic predicates and collect instantiated clauses used in a proof as defining clauses constituting $expl(G)$ (Kameya and Sato 2000; Zhou and Sato 2003).

DB' is obtained by translating each clause in *DB* as follows (Zhou and Sato 2003).[10] Suppose, for example, `p(X,f(V)):-msw(X,V),q(g(X,V)),r(V)` is a clause in *DB* and also suppose `p/2` and `q/1` are probabilistic but `r/1` is not (generalization is easy). We replace `msw(X,V)` with `(get_values(X,Vs),member(V,Vs))`[11] and further add a special goal to store a defining clause in the external memory area. So the translated clause is

```
p(X,f(V)):- get_values(X,Vs),member(V,Vs),q(g(X,V)),r(V),
            add_to_db(path(p(X,f(V)),[q(g(X,V))],[msw(X,V)])).
```

Here `member(V,Vs)` is a backtrackable predicate and returns an element `V` in a list `Vs`. The combined goal `(get_values(X,Vs),member(V,Vs))` thus succeeds with some value `V` in the outcome space `Vs` for `msw(X,·)`.

When all goals in `(get_values(X,Vs),member(V,Vs),q(g(X,V)),r(V))` succeed, `add_to_db/1` is invoked. `add_to_db(path(`a`,`b`,`c`))` is a special goal that always succeeds and stores a defining clause a `<=` b `&` c for a in the external memory area, where b is a list (conjunction) of probabilistic atoms and c is a list (conjunction) of `msw` atoms.

The translated program DB' is a usual Prolog program and runs isomorphically to *DB* as far as tabled search is concerned. We mean by *tabled goal*s goals containing a tabled predicate, by *answer*s goals proved successfully and by *tabled answer*s tabled goals proved successfully. Then in tabled search if a call to a tabled goal H occurs, H is unfolded by a clause in the program and tabled search continues, or unified with a tabled answer stored in the table and returns with success. In the former case, if the search succeeds and $H\theta$ is proved, where θ is an answer substitution, the answer $H\theta$ is added to the table. In the latter case, the tabling strategy determines when tabled answers are consumed. More details are given in Section 3.2. In the rest of the paper, since *DB* and DB' behave identically, when the context is clear, we use *DB* and DB' interchangeably for simplicity and say, for example, "all proofs of G w.r.t. *DB*" instead of "all proofs of G w.r.t. DB'".

[10] The actual implementation is slightly different. Also, another translation is possible which stores defining clauses in the table (Kameya and Sato 2000).

[11] For `X` $= id$, `get_values(X,Vs)` returns the list of possible values `Vs` for `msw(`id`,·)`.

2.2 *From explanation graphs to probability computation*

The probability $P_{DB}(G)$ of a given goal G is precisely defined in terms of the distribution semantics of PRISM. But the problem is that the semantics is so abstractly defined that we cannot know the actual value of $P_{DB}(G)$ easily. Here we describe how to compute it from $expl(G)$ under some assumptions.

To compute $P_{DB}(G)$, we convert each defining formula $H \Leftrightarrow \alpha_1 \vee \cdots \vee \alpha_M$ in $expl(G)$ to *a set of probability equations for* H:

$$P(H) = P(\alpha_1) + \cdots + P(\alpha_M) \tag{1}$$

$$\text{where}$$

$$P(\alpha_i) = P(C_1) \cdots P(C_m) P_{DB}(\mathtt{msw}_1) \cdots P_{DB}(\mathtt{msw}_n) \quad (1 \leqslant i \leqslant M)$$

$$\text{for } \alpha_i = C_1 \wedge \cdots \wedge C_m \wedge \mathtt{msw}_1 \wedge \cdots \wedge \mathtt{msw}_n.$$

We denote by $eq(G)$ the entire set of probability equations thus obtained. Note that the conversion assumes exclusiveness among disjuncts $\{\alpha_1, \ldots, \alpha_M\}$ and independence among conjuncts $\{C_1, \ldots, C_m, \mathtt{msw}_1, \ldots, \mathtt{msw}_n\}$.[12] We consider the $P(H)$'s in $eq(G)$ as numerical variables representing unknown probabilities and refer to them as *P-variables*. Then the right-hand side of (1) is a multivariate polynomial in P-variables with non-negative coefficients which are products of $P_{DB}(\mathtt{msw})$s.

We say that $expl(G)$ *is acyclic* if the ancestor relation in $expl(G)$ is acyclic. When $expl(G)$ is acyclic as is the case with standard generative models such as BNs, HMMs and PCFGs, defined goals in $expl(G)$ are hierarchically ordered by the ancestor relation (with G as top-most element) and the P-variables in $eq(G)$ are also hierarchically ordered. As a result $eq(G)$ is uniquely and efficiently solved in a bottom-up manner by dynamic programming using the generalized inside–outside algorithm (Sato and Kameya 2001) in time linear in the size of $eq(G)$ and the unique solution gives $P(G) = P_{DB}(G)$.

There are however cases where $expl(G)$ is cyclic and so is $eq(G)$, and hence it is impossible to apply dynamic programming to $eq(G)$, or even worse $eq(G)$ may not have a unique solution when $eq(G)$ is a system of polynomial equations of second degree or higher. Nonetheless, no matter whether it is cyclic or not, we can prove at least the existence of a solution for $eq(G)$, thanks to the special form and properties of $eq(G)$ under the *generative exclusiveness condition*; at any choice point in any execution path of the top-goal, a choice of alternative path is made by the value of X sampled from $\mathtt{msw}(id,\mathtt{X})$. We quickly remark that this condition is naturally satisfied by PRISM programs for generative models in general and BNs, HMMs and PCFGs in particular, because in a generative model an outcome is generated by a sequence of probabilistic choices and the process is simulated by msw atoms.

The generative exclusiveness condition implies that every disjunction in a defining formula is exclusive and originated from a probabilistic choice made by some msw. So a defining formula $H \Leftrightarrow \alpha_1 \vee \cdots \vee \alpha_M$ iswritten as $H \Leftrightarrow (\mathtt{msw}(id_H, v_1) \wedge \beta_1) \vee \cdots \vee$

[12] In this paper we assume these conditions are always satisfied. In particular, we assume the generative exclusiveness condition stated later which implies the exclusiveness among disjuncts.

$(\texttt{msw}(id_H, v_M) \wedge \beta_M)$ for some $\texttt{msw}(id_H, \cdot)$ that has a sample space V_{id_H} such that $V_{id_H} \supseteq \{v_1, \ldots, v_M\}$. Denote the vector of P-variables in $expl(G)$ by $\mathbf{X}^G$ and write a component $P(H)$ as X_H. Then the probability equation about $P(H)$ is represented as $X_H = T_H(\mathbf{X}^G) = \sum_{i=1}^{M} P_{DB}(\texttt{msw}(id_H, v_i))\varphi_i^H(\mathbf{X}^G)$, where $\varphi_i^H(\mathbf{X}^G)$ is a product of some $P_{DB}(\texttt{msw})$'s and variables in $\mathbf{X}^G$. We represent $eq(G)$ as $\mathbf{X}^G = T(\mathbf{X}^G)$. Now define a vector sequence $\{\mathbf{X}_k^G\}_{k=0}^{\infty}$ by $\mathbf{X}_0^G = \mathbf{0}$[13] and $\mathbf{X}_{k+1}^G = T(\mathbf{X}_k^G)$ for $k \geqslant 1$. Then $\mathbf{X}_k^G = T^{(k)}(\mathbf{0})$ $(k \geqslant 1)$. First we prove two lemmas.

Lemma 1

$T(\cdot)$ is monotonic, i.e. $\mathbf{X}^G \leqslant \mathbf{Y}^G$ implies $T(\mathbf{X}^G) \leqslant T(\mathbf{Y}^G)$.[14]

Proof

It is enough to prove that $\mathbf{X}^G \leqslant \mathbf{Y}^G$ implies $T_H(\mathbf{X}^G) \leqslant T_H(\mathbf{Y}^G)$ for an arbitrary component $T_H(\mathbf{X}^G)$ of $T(\mathbf{X}^G)$. Suppose $\mathbf{X}^G \leqslant \mathbf{Y}^G$ and write $T_H(\mathbf{X}^G) = \sum_{i=1}^{M} P_{DB}(\texttt{msw}(id_H, v_i))\varphi_i^H(\mathbf{X}^G)$. Since every $\varphi_i^H(\mathbf{X}^G)$ is a product of some $P_{DB}(\texttt{msw})$s and variables in $\mathbf{X}^G$, $\mathbf{X}^G \leqslant \mathbf{Y}^G$ implies $\varphi_i^H(\mathbf{X}^G) \leqslant \varphi_i^H(\mathbf{Y}^G)$ for every i. Hence,

$$\begin{aligned} T_H(\mathbf{X}^G) &= \sum_{i=1}^{M} P_{DB}(\texttt{msw}(id_H, v_i))\varphi_i^H(\mathbf{X}^G) \\ &\leqslant \sum_{i=1}^{M} P_{DB}(\texttt{msw}(id_H, v_i))\varphi_i^H(\mathbf{Y}^G) = T_H(\mathbf{Y}^G). \end{aligned}$$

□

Lemma 2

Suppose the generative exclusiveness condition is satisfied. $\{\mathbf{X}_k^G\}_{k=0}^{\infty}$ is bounded from above; $\mathbf{X}_k^G \leqslant \mathbf{1}$ for every $k \geqslant 0$.

Proof

For $k = 0$, $\mathbf{X}_0^G = \mathbf{0} \leqslant \mathbf{1}$ holds. Suppose $k > 0$ and inductively assume $\mathbf{X}_k^G \leqslant \mathbf{1}$ holds. Let $X_{k+1}^H = T_H(\mathbf{X}_k^G)$ be a probability equation in $\mathbf{X}^G = T(\mathbf{X}^G)$. We see

$$\begin{aligned} X_{K+1}^H = T_H(\mathbf{X}_k^G) &= \sum_{i=1}^{M} P_{DB}(\texttt{msw}(id_H, v_i))\varphi_i^H(\mathbf{X}_k^G) \\ &\leqslant \sum_{i=1}^{M} P_{DB}(\texttt{msw}(id_H, v_i)) \leqslant \sum_{v \in V_{id_H}} P_{DB}(\texttt{msw}(id_H, v)) = 1. \end{aligned}$$

Here we use the fact that since $\varphi_i^H(\mathbf{X}^G)$ is a product of some $P_{DB}(\texttt{msw})$s and variables in $\mathbf{X}^G$, $\mathbf{X}_k^G \leqslant \mathbf{1}$ implies $\varphi_i^H(\mathbf{X}_k^G) \leqslant 1$. □

[13] We use $\mathbf{0}$ (resp. $\mathbf{1}$) to denote a vector of 0s (resp. a vector of 1s).

[14] For N dimensional vectors $\mathbf{X} = (x_1, \ldots, x_N)$ and $\mathbf{Y} = (y_1, \ldots, y_N)$, we write $\mathbf{X} \leqslant \mathbf{Y}$ (resp. $\mathbf{X} < \mathbf{Y}$) if $x_i \leqslant y_i$ (resp. $x_i < y_i$) for every i $(1 \leqslant i \leqslant N)$.

Theorem 1
Under the generative exclusiveness condition, $\{\mathbf{X}_k^G\}_{k=0}^{\infty}$ monotonically converges to the least fixed point $\mathbf{X}_{\infty}^G = T(\mathbf{X}_{\infty}^G)$ which gives a solution for $eq(G)$.

Proof
$\{\mathbf{X}_k^G\}_{k=0}^{\infty}$ is a monotonically increasing sequence $\mathbf{0} = \mathbf{X}_0^G \leqslant \mathbf{X}_1^G \leqslant \cdots$ by Lemma 1 which is bounded from above by Lemma 2. Consequently, $\{\mathbf{X}_k^G\}_{k=0}^{\infty}$ converges to a limit $\mathbf{X}_{\infty}^G$. Furthermore, because T is continuous, we have $T(\mathbf{X}_{\infty}^G) = T(\lim_{k\to\infty}\mathbf{X}_k^G) = \lim_{k\to\infty} T(\mathbf{X}_k^G) = \lim_{k\to\infty}\mathbf{X}_{k+1}^G = \mathbf{X}_{\infty}^G$. So we have $\mathbf{X}_{\infty}^G = T(\mathbf{X}_{\infty}^G)$. Let $\mathbf{X}'^G \geqslant \mathbf{0}$ be another fixed point of T. $\mathbf{X}_k^G \leqslant \mathbf{X}'^G$ for all $k \geqslant 0$ is inductively proved. Therefore, $\mathbf{X}_{\infty}^G = \lim_{k\to\infty}\mathbf{X}_k^G \leqslant \mathbf{X}'^G$. Hence, $\mathbf{X}_{\infty}^G$ is the least fixed point of T. □

3 Prefix probability computation for PCFGs in PRISM

In this section, using a concrete example, we have a close look at how cyclic explanation graphs are constructed and investigate their properties. The reader is assumed to have a basic knowledge of CFG parsing.

3.1 *A prefix parser*

Before proceeding we introduce some terminology about CFGs for later use. Let X be a nonterminal in a CFG, α, β a mixed sequence of terminals and nonterminals. A rule for X is a production rule of the form $X \to \alpha$. If there is a rule of the form $X \to Y\beta$, we say X and Y are in the direct left-corner relation. The *transitive closure* of the direct left-corner relation is called *left-corner relation* and we write $X \to_L Y$ if X and Y are in the left-corner relation. The left-corner relation is *cyclic* if $X \to_L X$ holds for some nonterminal X. We say that a rule is *useless* if it does not occur in any sentence derivation. A nonterminal is *useless* if every rule for it is useless. Otherwise it is *useful*. In this paper we assume that CFGs have "s" as a default start symbol and have no epsilon rule and no useless nonterminal.

Finally, let $X \to \alpha_1 : \theta_1, \ldots, X \to \alpha_n : \theta_n$ be the set of rules for X in a PCFG with *selection probabilities* $\theta_1, \ldots, \theta_n$ where $\sum_{i=1}^{n} \theta_i = 1$. We assume that every rule has a *positive* selection probability. If the sum of probabilities of sentences derived from the start symbol is 1, the PCFG is said to be *consistent* (Wetherell 1980). We also assume that PCFGs are consistent.

Now we look at a concrete example of prefix probability computation based on cyclic explanation graphs. Consider a CFG, $\mathbf{G}_0 = \{\ \mathtt{s} \to \mathtt{s\,s}\ ,\ \mathtt{s} \to \mathtt{a}\ ,\ \mathtt{s} \to \mathtt{b}\ \}$ and its PCFG version, $\mathbf{PG_0} = \{\ \mathtt{s} \to \mathtt{s\,s} : 0.4,\ \ \mathtt{s} \to \mathtt{a} : 0.3,\ \ \mathtt{s} \to \mathtt{b} : 0.3\ \}$. Here "s" is a start symbol in $\mathbf{G}_0$ and "a" and "b" are terminals. $\mathtt{s} \to \mathtt{s\,s} : 0.4$ says that the rule $\mathtt{s} \to \mathtt{s\,s}$ is selected with probability 0.4 when "s" is expanded in a sentence derivation.

A PRISM program DB_0 in Figure 1 is a prefix parser for $\mathbf{PG_0}$. It is a slight modification of a standard top-down CFG parser and parses prefixes accepted by $\mathbf{G}_0$ such as "a" (as list [a]). The only difference is that it can have *pseudo success*

```
values(s,[[s,s],[a],[b]]).
:- set_sw(s,[0.4,0.3,0.3]).

pre_pcfg(L):- pre_pcfg([s],L,[]).                  --(1) % L is a prefix
pre_pcfg([A|R],L0,L2):-                            --(2) % L0 is ground when called
   ( values(A,_)-> msw(A,RHS),                     --(3) % if A is a nonterminal
       pre_pcfg(RHS,L0,L1)                         --(4) % select rule A->RHS
   ;  L0=[A|L1] ),                                 --(5) % else consume A in L0
   ( L1=[] -> L2=[]                                --(6) % (pseudo) success
   ; pre_pcfg(R,L1,L2) ).                          --(7) % recursion
pre_pcfg([],L1,L1).                                --(8) % termination
```

Fig. 1. Prefix PCFG parser DB_0.

at line `(6)`, i.e. it immediately terminates with success as soon as the input prefix is consumed even when there remain some nonterminals in `R` at line `(2)`.[15]

A `values/2` declaration `values(s,[[s,s],[a],[b]])` in the program introduces three `msw` atoms: `msw(s,[s,s])`, `msw(s,[a])` and `msw(s,[b])`. The next command `:- set_sw(s,[0.4,0.3,0.3])` sets $\theta_{\mathtt{s}\to\mathtt{ss}} = P_{DB_0}(\mathtt{msw(s,[s,s])}) = 0.4$, $\theta_{\mathtt{s}\to\mathtt{a}} = P_{DB_0}(\mathtt{msw(s,[a])}) = 0.3$ and $\theta_{\mathtt{s}\to\mathtt{b}} = P_{DB_0}(\mathtt{msw(s,[b])}) = 0.3$ respectively when the program is loaded. Thus, $\mathbf{PG_0}$ is encoded. We point out that DB_0 is general, applicable to any PCFG just by replacing the `values/2` declaration and `set_sw` command with appropriate ones that encode a given PCFG.

3.2 Tracing linear-tabling

Once a program DB and a top-goal G are given for which the probability is computed, the next task is to construct an explanation graph for G by searching for all proofs while tabling answers and recording their defining clauses in the external memory area. Using a simple example, we illustrate how tabled search for all proofs is done by *linear-tabling with the lazy strategy* in B-Prolog (Zhou *et al.* 2008), which has been a standard platform for PRISM.

One of the unique features of linear-tabling is to iterate exhaustive tabled search to obtain all answers when there are looping subgoals.[16] More precisely, if a call `:-(A,...)` on a path of an SLD-tree has a sub-path containing sub-derivation `:-A` $\Rightarrow \cdots \Rightarrow$ `:-(A',...)` such that `A` and `A'` are variants, `A` and `A'` are called *interdependent looping subgoal*s. Interdependent looping subgoals constitute a cluster. The first looping subgoal `A` in the cluster that appears in the SLD-tree is said to be a *top-most looping subgoal* (Zhou *et al.* 2008).

Although a looping subgoal causes an infinite loop, it can be proved by non-looping paths in the SLD-tree. We preserve answers from such non-looping paths

15 This is justifiable because as we assume that every nonterminal is useful, we can prove that every nonterminal derives a terminal string with probability 1.

16 In this section, the terms "subgoal" and "goal" are used synonymously.

in the table and make them available as tabled answers when looping subgoals are called. Linear-tabling with the lazy strategy tries to collect all answers for looping subgoals by iterating *round*s for a top-most looping subgoal. In a round exhaustive search by backtracking is performed to generate all proofs of the top-most looping subgoal while consuming tabled answers and adding newly found answers to the table. The lazy strategy does not allow other subgoals outside the looping path to consume tabled answers of the top-most looping subgoal until no more round generates new answers for the looping subgoals (Zhou *et al.* 2008).

Figure 2 sketches tabled search for all proofs of a top-goal G_0 = `pre_pcfg([a])` w.r.t. DB_0 while tabling `pre_pcfg/1` and `pre_pcfg/3`. Here `(1)`,`(2)`,... correspond to line numbers in Figure 1.[17] Although Figure 2 is self-explanatory, we add some comments. The top-call to G_0 = `pre_pcfg([a])` leads to a call to a subgoal `TG` = `pre_pcfg([s,s],[a],L1)` via a call to `pre_pcfg([s],[a],[])` in which `values(s,_)` is tested true and `msw(s,RHS1)` is executed at line `(3)`. Since `TG` is a top-most looping subgoal, exhaustive tabled search is iterated on `TG` until no new answer is obtained.

In the first round, a proof by a branch in the SLD-tree specified by `RHS2` = `[a]` succeeds with `L1` = `[]` and gives a tabled answer `pre_pcfg([s,s],[a],[])` for which a defining clause is recorded in the external memory area. In the second round a branch specified by `RHS2` = `[s,s]` succeeds as well using the previously tabled answer, giving a new defining clause `pre_pcfg([s,s],[a],[])` `<=` `pre_pcfg([s,s],[a],[]) & msw(s,[s,s])`. The third round generates no new answer and the call to `TG` terminates successfully. `TG` now exports its tabled answer `pre_pcfg([s,s],[a],[])` which leads to the success of the top-call.

After all proof search is done, PRISM constructs an explanation graph $expl(G_0)$ by tracing tabled answers starting from G_0 while collecting defining clauses recorded in the external memory area. When PRISM encounters looping subgoals in the body of a defining clause, it looks at the PRISM-flag `error_on_cycle` and if the value is "`off`," these goals are treated as succeeded normally and as a result a cyclic explanation graph is obtained.

3.3 Computing prefix probability: an example

In this section, using the continuing example, we describe probability computation in cyclic explanation graphs.

An explanation graph for G_0 = `pre_pcfg([a])` is obtained by executing a command `?- probf(pre_pcfg([a]))`[18] w.r.t. DB_0. The command initiates exhaustive tabled search described in Section 3.2 and generates an explanation graph shown in Figure 3 consisting of defining clauses in Figure 2.

[17] Recall that as we explained in Section 2.1, the program we actually use in the tabled search is a translated program DB'_0, but as it behaves exactly the same way as the original one except that defining clauses are recorded in the external memory area, we explain the tabled search in terms of DB_0 for intuitiveness and conciseness.

[18] `probf/1` is a built-in predicate in PRISM and `probf(`G`)` displays the explanation graph of G.

```
:- pre_pcfg([a])
  :- pre_pcfg([s],[a],[])
    :- msw(s,RHS1),pre_pcfg(RHS1,[a],L1)..
      (first round)
      :- pre_pcfg([s,s],[a],L1)..    % RHS1=[s,s], top-most looping subgoal TG
        :- msw(s,RHS2),pre_pcfg(RHS2,[a],L1)..
          :- pre_pcfg([s,s],[a],L1)..
             % RHS2=[s,s], fails at (4) as no anwser available in the table
             % for :- pre_pcfg([s,s],[a],L1) yet.
          :- pre_pcfg([a],[a],L1)..
             % RHS2=[a], executes (5) and succeeds at (6) with L1=[], resulting
             % in tabled answers pre_pcfg([a],[a],[]) and pre_pcfg([s,s],[a],[])
             % with defining clauses
             %   pre_pcfg([a],[a],[]) and
             %   pre_pcfg([s,s],[a],[]) <= pre_pcfg([a],[a],[]) & msw(s,[a])
          :- pre_pcfg([b],[a],L1)..
             % RHS2=[b], fails at (5)
      (second round)
      :- pre_pcfg([s,s],[a],L1)..    % RHS1=[s,s], top-most looping subgoal TG
        :- msw(s,RHS2),pre_pcfg(RHS2,[a],L1)..
          :- pre_pcfg([s,s],[a],L1)..
             % RHS2=[s,s], this time can consume the tabled answer
             % pre_pcfg([s,s],[a],[]) in the previous round and
             % succeeds with L1=[], giving pseudo success at (6) and
             % a defining clause
             %   pre_pcfg([s,s],[a],[]) <= pre_pcfg([s,s],[a],[]) & msw(s,[s,s])
             % no further answer generated
          :- pre_pcfg([a],[a],L1)..  % RHS2=[a], succeeds with L1=[]
          :- pre_pcfg([b],[a],L1)..  % RHS2=[b], fails at (5)
      (third round)
      :- pre_pcfg([s,s],[a],L1)..
             % yields no new answer, so :- pre_pcfg([s,s],[a],L1)
             % is completely evaluated with one answer pre_pcfg([s,s],[a],[])
             % which results in the success of :- pre_pcfg([s],[a],[])
             % giving a defining clause
             %   pre_pcfg([s],[a],[]) <= pre_pcfg([s,s],[a],[]) & msw(s,[s,s])
      :- pre_pcfg([a],[a],L1)..
             % RHS1=[a], succeeds with L1=[], results in the success of
             % :- pre_pcfg([s],[a],[]) giving a defining clause
             %   pre_pcfg([s],[a],[]) <= pre_pcfg([a],[a],[]) & msw(s,[a])
      :- pre_pcfg([b],[a],L1)..      % RHS1=[b], fails at (5)
      ...
```

Fig. 2. A sketch of SLD-tree(s) for `:- pre_pcfg([a])`.

As can be seen, the top-most looping subgoal `pre_pcfg([s,s],[a],[])` calls itself. We convert the cyclic explanation graph in Figure 3 to the corresponding set of probability equations shown in Figure 4. Here we used abbreviations: $\theta_{\mathtt{s}\to\mathtt{ss}} = P_{DB_0}(\mathtt{msw}(\mathtt{s},[\mathtt{s},\mathtt{s}]))$ and $\theta_{\mathtt{s}\to\mathtt{a}} = P_{DB_0}(\mathtt{msw}(\mathtt{s},[\mathtt{a}]))$.

Although we know that the set of probability equations in Figure 4 has a solution (see Theorem 1), we do not know their actual values. To know their actual values, we need to compute them by solving the equations. Fortunately, equations are linear

```
pre_pcfg([a]) <=> pre_pcfg([s],[a],[])
pre_pcfg([s],[a],[]) <=>
   pre_pcfg([s,s],[a],[]) & msw(s,[s,s]) v pre_pcfg([a],[a],[]) & msw(s,[a])
pre_pcfg([s,s],[a],[]) <=>
   pre_pcfg([a],[a],[]) & msw(s,[a]) v pre_pcfg([s,s],[a],[]) & msw(s,[s,s])
pre_pcfg([a],[a],[])
```

Fig. 3. Explanation graph for prefix “a”.

$$\begin{aligned}
P(\texttt{pre\_pcfg}([\texttt{a}])) &= \texttt{X} = \texttt{Y}\\
P(\texttt{pre\_pcfg}([\texttt{s}],[\texttt{a}],[])) &= \texttt{Y} = \texttt{Z}\cdot\theta_{\texttt{s}\to\texttt{ss}} + \texttt{W}\cdot\theta_{\texttt{s}\to\texttt{a}}\\
P(\texttt{pre\_pcfg}([\texttt{s},\texttt{s}],[\texttt{a}],[])) &= \texttt{Z} = \texttt{W}\cdot\theta_{\texttt{s}\to\texttt{a}} + \texttt{Z}\cdot\theta_{\texttt{s}\to\texttt{ss}}\\
P(\texttt{pre\_pcfg}([\texttt{a}],[\texttt{a}],[])) &= \texttt{W} = 1
\end{aligned}$$

Fig. 4. Probability equations for prefix “a”.

in the P-variables X, Y, Z and W and easily solvable. By substituting $\theta_{\texttt{s}\to\texttt{ss}} = 0.4$ and $\theta_{\texttt{s}\to\texttt{a}} = 0.3$ for the equations and solving them, we obtain X = Y = Z = 0.5, and W = 1[19] respectively. Hence, the prefix probability of “a”, $P(\texttt{pre\_pcfg}([\texttt{a}]))$, is 0.5. Note that this prefix probability is greater than the probability of “a” as a sentence which is 0.3. This is because the prefix probability of “a” is the sum of the probability of sentence “a” *and* the probabilities of infinitely many sentences extending “a”.

By looking at the set of probability equations in Figure 4 more closely, we can understand the way our approach computes prefix probability in PCFGs. For example, consider $\texttt{Z} = P(\texttt{pre\_pcfg}([\texttt{s},\texttt{s}],[\texttt{a}],[]))$ and the equation $\texttt{Z} = \texttt{W}\cdot\theta_{\texttt{s}\to\texttt{a}} + \texttt{Z}\cdot\theta_{\texttt{s}\to\texttt{ss}}$. We can expand the solution Z into an infinite series:

$$\texttt{Z} = \frac{1}{1-\theta_{\texttt{s}\to\texttt{ss}}}\texttt{W}\cdot\theta_{\texttt{s}\to\texttt{a}} = (1+\theta_{\texttt{s}\to\texttt{ss}}+\theta^2_{\texttt{s}\to\texttt{ss}}+\cdots)\texttt{W}\cdot\theta_{\texttt{s}\to\texttt{a}}$$

It is easy to see that this series represents the probability of infinitely many leftmost derivations of prefix “a” from nonterminals “s s” by partitioning the derivations based on the number of applications of rule s → s s to derive “a”, i.e. 1 for no application ($\texttt{s s} \Rightarrow_{\texttt{s}\to\texttt{a}} \texttt{a s}$), $\theta_{\texttt{s}\to\texttt{ss}}$ for one ($\texttt{s s} \Rightarrow_{\texttt{s}\to\texttt{ss}} \texttt{s s s} \Rightarrow_{\texttt{s}\to\texttt{a}} \texttt{a s s}$) and so on.[20]

3.4 Properties of explanation graphs generated by a prefix parser

We here examine properties of cyclic explanation graphs. Let **PG** be a PCFG and **G’** its underlying CFG, i.e. the CFG obtained by removing probabilities from **PG**. Throughout this section we use $DB_{\mathbf{PG}}$ for a prefix parser for **PG** obtained by replacing the `values/2` declaration in DB_0 in Figure 1 with an appropriate set of

[19] W = 1 because `pre_pcfg([a],[a],[])` is logically proved without involving `msw`s.

[20] Here we use $\alpha{\Rightarrow}\beta$ (resp. $\alpha \overset{*}{\Rightarrow} \beta$) to indicate β is derived from α by one step derivation (resp. zero or more steps derivation) using CFG rules. Also, recall here that it is assumed that PCFGs are consistent. So the sum of probabilities of sentences derived from “s” is 1. Consequently, for example, we may safely ignore s in “a s” when computing the probability of prefix “a” derived from “a s”.

`values/2` declarations encoding **PG**. In what follows, we first prove a necessary and sufficient condition under which a prefix parser $DB_{\mathbf{PG}}$ generates cyclic explanation graphs. We then prove that $DB_{\mathbf{PG}}$ always generates a system of linear equations for prefix probabilities. Finally we prove that the linear system is solvable by matrix operation under our assumptions on PCFGs.

Theorem 2
Let $G_\ell = \texttt{pre\_pcfg}(\ell)$ be a goal for a prefix $\ell = \texttt{[}w_1,\ldots,w_N\texttt{]}$ in **G'** and $expl(G_\ell)$ an explanation graph for G_ℓ generated by $DB_{\mathbf{PG}}$. Suppose there is no useless nonterminal in **G'**. Then there exists a cyclic explanation graph $expl(G_\ell)$ if-and-only-if the left-corner relation of **G'** is cyclic.

Proof
Suppose $expl(G_\ell)$ is cyclic. Then some defined goal $\texttt{pre\_pcfg([}a\texttt{|}\beta\texttt{]},\ell_0,\ell_2\texttt{)}$ with a nonterminal "a" must have itself as a descendant in $expl(G_\ell)$, where ℓ_0 and ℓ_2 are sublists of ℓ. So an SLD-derivation exists from $\texttt{:-pre\_pcfg([}a\texttt{|}\beta\texttt{]},\ell_0\texttt{,L2),K}$ to its descendant $\texttt{:-pre\_pcfg([}a\texttt{|}\beta\texttt{]},\ell_0\texttt{,L2'),K'}$ in which the list ℓ_0 is preserved. Consequently, there is a corresponding leftmost derivation $\texttt{s} \overset{*}{\Rightarrow} a\delta \overset{*}{\Rightarrow} a\delta'$ by **G'**, the underlying CFG of **PG**. So the left-corner relation is cyclic.

Conversely, suppose the left-corner relation of **G'** is cyclic. Then there is a nonterminal "a" such that $a \rightarrow_L a$. As there is no useless nonterminal by our assumption, there is a leftmost derivation starting from "s" such that $\texttt{s} \overset{*}{\Rightarrow} \gamma a\delta \overset{*}{\Rightarrow} \gamma a\delta' \overset{*}{\Rightarrow} w_1 \cdots w_N$ for some sentence $w_1,\ldots,w_N$. In what follows, for simplicity we assume that γ is empty (but the generalization is straightforward). Let $\ell_0 = w_1,\ldots,w_j$ ($j \leqslant N$) be a prefix derived from a whose partial parse tree[21] has a as the root and no a occurs below the root a. Then it is easy to see that the tabled search for all proofs of G_{ℓ_0} generates $expl(G_{\ell_0})$ containing a goal $\texttt{pre\_pcfg([}a\texttt{|}\beta\texttt{]},\ell_0\texttt{,[])}$ which is an ancestor of itself. So $expl(G_{\ell_0})$ is cyclic. □

Let $expl(G)$ be an explanation graph for G_ℓ. We introduce an equivalence relation $A \equiv B$ over defined goals appearing in $expl(G)$: $A \equiv B$ if-and-only-if $A = B$ or A is an ancestor of B and *vice versa*. We partition the set of defined goals into equivalence classes $[A]_\equiv$. Each $[A]_\equiv$ is called a *strongly connected component* (SCC). We say that *a defining formula* $H \Leftrightarrow \alpha_1 \vee \cdots \vee \alpha_M$ *is linear* if there is no $\alpha_i = C_1 \wedge \cdots \wedge C_m \wedge \texttt{msw}_1 \wedge \cdots \wedge \texttt{msw}_n$ ($1 \leqslant i \leqslant h$, $0 \leqslant m, n$) that has two defined goals, C_j and C_k ($j \neq k$), belonging to the same SCC. Also, we say $expl(G)$ *is linear* if every defining formula in $expl(G)$ is linear.

Lemma 3
No two defined goals in the body of a defining formula in $expl(G_\ell)$ belong to the same SCC.

Proof
Let $H \Leftrightarrow \alpha_1 \vee \cdots \vee \alpha_M$ be a defining formula in $expl(G_\ell)$. Suppose some α_i contains two defined goals belonging to the same SCC. Looking at DB_0 in Figure 1, we know

[21] A partial parse tree is an incomplete parse tree whose leaves may contain nonterminals.

that the only possibility is such that $H \Leftrightarrow \alpha_1 \vee \cdots \vee \alpha_M$ is a ground instantiation of the first (compound) clause about `pre_pcfg/3`:

$$\begin{array}{l}\texttt{pre\_pcfg([}a\texttt{|}\beta\texttt{]},\ell_0\texttt{,}\ell_2\texttt{):-}\\ \texttt{msw(}a\texttt{,}\alpha\texttt{),pre\_pcfg(}\alpha\texttt{,}\ell_0\texttt{,}\ell_1\texttt{),pre\_pcfg(}\beta\texttt{,}\ell_1\texttt{,}\ell_2\texttt{)}\end{array} \quad (2)$$

and the two defined goals, $\texttt{pre\_pcfg(}\alpha\texttt{,}\ell_0\texttt{,}\ell_1\texttt{)}$ and $\texttt{pre\_pcfg(}\beta\texttt{,}\ell_1\texttt{,}\ell_2\texttt{)}$, are in the same SCC. However, since $\texttt{pre\_pcfg(}\alpha\texttt{,}\ell_0\texttt{,}\ell_1\texttt{)}$ is a proved goal, ℓ_1 is shorter than ℓ_0. On the other hand, since $\texttt{pre\_pcfg(}\beta\texttt{,}\ell_1\texttt{,}\ell_2\texttt{)}$ is an ancestor of $\texttt{pre\_pcfg(}\alpha\texttt{,}\ell_0\texttt{,}\ell_1\texttt{)}$ in $expl(G_\ell)$ because they belong to the same SCC by assumption, ℓ_0 is identical to or a part of ℓ_1, and hence ℓ_0 is equal to or shorter than ℓ_1. Contradiction. Therefore, there is no such defining formula. Hence, $expl(G_\ell)$ is linear. □

Theorem 3

Let $expl(G_\ell)$ be an explanation graph for a prefix ℓ parsed by $DB_{\mathbf{PG}}$. $expl(G_\ell)$ is linear.

Proof

Immediate from Lemma 3. □

We next introduce a partial ordering $[A]_\equiv \succ [B]_\equiv$ over SCCs by $[A]_\equiv \succ [B]_\equiv$ if-and-only-if A is an ancestor of B but not *vice versa* in $expl(G)$. We then extend this partial ordering to a total ordering $[A]_\equiv \succ [B]_\equiv$ over SCCs. Likewise, we partition P-variables by the equivalence relation: $P(A) {\equiv} P(B)$ if-and-only-if $[A]_\equiv = [B]_\equiv$. We denote by $[P(A)]_\equiv$ the equivalence class of P-variables corresponding to $[A]_\equiv$. By construction $[P(A)]_\equiv$'s are totally ordered isomorphically to SCCs; $[P(A)]_\equiv \succ [P(B)]_\equiv$ if-and-only-if $[A]_\equiv \succ [B]_\equiv$. In the following we treat SCCs and P-variables as isomorphically stratified by this total ordering. We use $eq([P(A)]_\equiv)$ to stand for the union of sets of probability equations for defined goals in $[A]_\equiv$. Notice that in the case of PCFGs, $eq([P(A)]_\equiv)$ is a system of linear equations by Theorem 3 if we consider P-variables in the lower strata as constants. Hence, $eq(G_\ell)$ is solvable inductively from lower strata to upper strata.

Now we show that $eq([P(A)]_\equiv)$ is always solvable by matrix operation under our assumptions on PCFGs. Let "a" be a nonterminal in the underlying CFG **G'** and A be a defined goal in $expl(G_\ell)$. Write $A = \texttt{pre\_pcfg([}a\texttt{|}\beta\texttt{]},\ell_0\texttt{,}\ell_2\texttt{)}$. Since A is a proved goal, A successfully calls some ground goals $B_j = \texttt{pre\_pcfg(}\alpha_j\texttt{,}\ell_0\texttt{,}\ell_{1j}\texttt{)}$ and $C_j = \texttt{pre\_pcfg(}\beta\texttt{,}\ell_{1j}\texttt{,}\ell_2\texttt{)}$ in the clause body shown in (2) where $a \rightarrow \alpha_j$ is a CFG rule in **G'**. By repeating a similar proof for Lemma 3, we can prove that the third goal C_j does not belong to $[A]_\equiv$, the SCC containing A. Thus, $[A]_\equiv \succ [\texttt{pre\_pcfg(}\beta\texttt{,}\ell_{1j}\texttt{,}\ell_2\texttt{)}]_\equiv$. So only some B_j's can possibly belong to $[A]_\equiv$.

Let $P(A_1),\ldots,P(A_K)$ be an enumeration of P-variables in $[P(A)]_\equiv$. Introduce a column vector $\mathbf{X}_A = (P(A_1),\ldots,P(A_K))^T$. It follows from what we discussed before that we can write $eq([P(A)]_\equiv)$ as a system of linear equations $\mathbf{X}_A = M\mathbf{X}_A + \mathbf{Y}_A$, where M is a $K \times K$ non-negative matrix and $\mathbf{Y}_A$ is a non-negative vector whose component is a sum of P-variables in the lower strata multiplied by constants. M is irreducible because every goal in $[A]_\equiv$ directly or indirectly calls every goal in $[A]_\equiv$

with positive probability. $\mathbf{Y}_A$ is non-zero because some A_i must have a proof tree that only contains defined goals in the lower strata.

Theorem 4
Let **PG** be a consistent PCFG such that there is no epsilon rule and every production rule has a positive selection probability. Also, let $DB_{\mathbf{PG}}$ be a prefix parser for **PG** and $expl(G_\ell)$ be an explanation graph for a prefix ℓ. Suppose $eq([P(A)]_\equiv)$ is a system of linear equations for a defined goal A in $expl(G_\ell)$. Put $[P(A)]_\equiv = \{P(A_i) \mid 1 \leqslant i \leqslant K\}$ and write $eq([P(A)]_\equiv)$ as $\mathbf{X}_A = M\mathbf{X}_A + \mathbf{Y}_A$, where $\mathbf{X}_A = (P(A_1), \ldots, P(A_K))^T$. It has a unique solution $\mathbf{X}_A = (I - M)^{-1}\mathbf{Y}_A$.

Proof
We prove that $I - M$ has an inverse matrix. To prove it, we assume hereafter that P-variables in $[P(A)]_\equiv$ are assigned as their values probabilities from $\mathbf{X}^G_\infty$, a solution for $eq(G)$ whose existence is guaranteed by Theorem 1 and hence all equations in $eq([P(A)]_\equiv)$ are true.

By applying $\mathbf{X}_A = M\mathbf{X}_A + \mathbf{Y}_A$ k times repeatedly to itself, we have $\mathbf{X}_A = M^k\mathbf{X}_A + (M^{k-1} + \cdots + I)\mathbf{Y}_A$ for $k = 1, 2, \ldots$ Since M, $\mathbf{X}_A$ and $\mathbf{Y}_A$ are all non-negative, we have $\mathbf{X}_A \geqslant M^k\mathbf{X}_A$ and $\mathbf{X}_A \geqslant (M^{k-1} + \cdots + I)\mathbf{Y}_A$ for every k. On the other hand, since $\{(M^{k-1} + \cdots + I)\mathbf{Y}_A\}_k$ is a monotonically increasing sequence of non-negative vectors bounded by $\mathbf{X}_A$, it converges and so does $\{M^k\mathbf{X}_A\}_k$.

Let $\rho(M)$ be the spectral radius of M.[22] Suppose $\rho(M) > 1$. In general, $\rho(M) \leqslant \| M^k \|_\infty^{\frac{1}{k}}$ holds for every k, where $\| \cdot \|_\infty$ is the matrix norm induced from the ∞ vector norm. It follows from $\rho(M)^k \leqslant \| M^k \|_\infty$ that $\lim_{k\to\infty} \| M^k \|_\infty = +\infty$. Consequently, since $\mathbf{X}_A > 0$ holds because every proved goal has a positive probability from our assumption, some element of $M^k\mathbf{X}_A$ goes to $+\infty$, which contradicts the convergence of $\{M^k\mathbf{X}_A\}_k$. So $\rho(M) \leqslant 1$.

Suppose now $\rho(M) = 1$. Then in this case, we note that $\{\frac{M^{k-1}+\cdots+I}{k}\}_k$ converges to a positive matrix (Meyer 2000, Example 8.3.2) and hence $(M^{k-1} + \cdots + I)\mathbf{Y}_A = (\frac{M^{k-1}+\cdots+I}{k})\cdot k\mathbf{Y}_A$ diverges as k goes to infinity, which contradicts again the convergence of $\{(M^{k-1} + \cdots + I)\mathbf{Y}_A\}_k$. Therefore, $\rho(M) < 1$. So $(I - M)^{-1}$ exists. $\square$

Note that $\mathbf{X}_A = (I - M)^{-1}\mathbf{Y}_A = (I + M + M^2 + \cdots)\mathbf{Y}_A$. By further analyzing the matrix M, we understand that multiplying M by $\mathbf{Y}_A$, for example, corresponds to growing partial parse trees by one step application of production rules (reduce operation in bottom-up parsing). Hence, $P(A_i)$, a component of $\mathbf{X}_A$, becomes an infinite sum of probabilities and so is the probability of the top-goal $P(\texttt{pre\_pcfg}(\ell))$.

We sum up our discussion so far and state in Figure 5 a general procedure to compute probability on cyclic explanation graphs. In the case of PCFGs, *DB* is the prefix parser in Figure 1 with appropriate `values/2` declarations encoding a given PCFG and $G = \texttt{pre\_pcfg}(\ell)$ is a goal for a prefix ℓ. Under our assumptions on PCFGs, $eq(G)$ in **[Step 2]** is guaranteed to be linear by Theorem 3 and **[Step 3]** is always possible by Theorem 4.

[22] $\rho(M) \overset{\text{def}}{=} \underset{i}{\operatorname{argmax}}\, |\lambda_i|$, where the λ_i's are the eigenvalues of M.

[Step 1]: Given a program DB and a goal G, construct an explanation graph $expl(G)$.
[Step 2]: Convert $expl(G)$ to a set of probability equations $eq(G)$.
[Step 3]: Solve $eq(G)$ inductively from lower strata by matrix operation and obtain $P_{DB}(G)$.

Fig. 5. Probability computation on cyclic explanation graphs.

We emphasize that the procedure is general and applicable to arbitrary programs that generate linear explanation graphs,[23] not restricted to those generated by a prefix PCFG parser. Also, we add that even if $eq(G)$ is nonlinear, it is still solvable (Theorem 1). This fact is applied to the computation of infix probability for PCFGs (Nederhof and Satta 2011a), although it is beyond the scope of this paper and we do not discuss it.

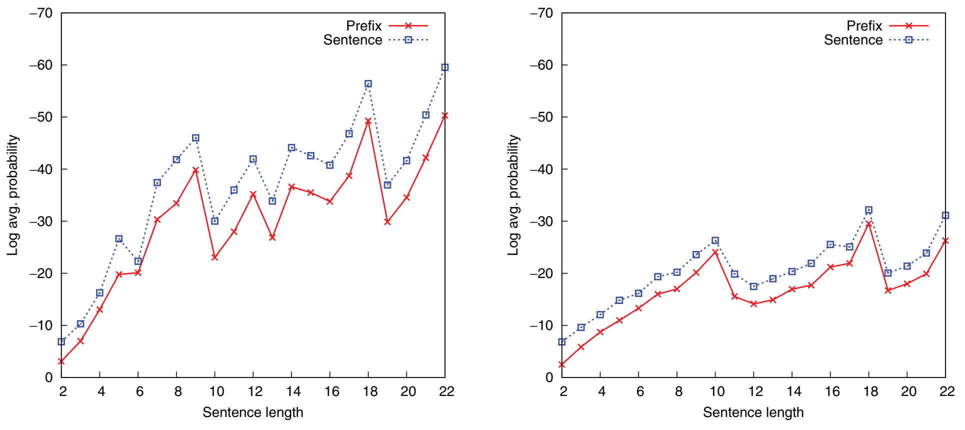
3.5 Prefix probability computation for a real PCFG

Here we apply our approach to real data to show the effectiveness of our approach. We use the ATR corpus and its PCFG (Uratani *et al.* 1994).[24] The corpus contains labeled parse trees for 10,995 Japanese sentences whose average length is about 10. The associated manually developed CFG comprises 861 CFG rules (168 nonterminals and 446 terminals[25]) and yields 958 parses/sentences on average. A PCFG is prepared by assigning probabilities (*parameter*s) to CFG rules and is encoded as a PRISM program just like the one in Figure 1 with appropriate `values/2` declarations. Using this PCFG, we computed the average probability of sentence and that of prefix in the ATR corpus for comparison. We randomly sampled 100 sentences of a given length from the ATR corpus and computed their average probability. We then deleted their last word and created 100 prefixes for which we also computed the average probability.

Figure 6 contains results of plotting the (minus) logarithm of average prefix probability and that of average sentence probability for a length varying from 2 to 22. We used two parameter sets for the PCFG. For Figure 6(a), parameters are uniform, i.e. if a nonterminal X has n rules $\{X \to \alpha_i \mid 1 \leqslant i \leqslant n\}$, each rule is selected with probability $1/n$. For Figure 6(b), parameters are learned from the entire ATR corpus by the built-in EM algorithm in PRISM.

Seeing these figures we first note that the average prefix probability is always greater than the average sentence probability at each length in both Figures 6(a) and (b) as expected, and second that the curves in Figure 6(b) are much smoother than the ones in Figure 6(a) and shifted downward considerably (the y-axis is scaled with minus logarithm) due to the effect of parameters learned by maximum

[23] Currently the PRISM system returns an error message when $eq(G)$ is not linear.
[24] All experiments in this paper are done on a single machine with Core i7 Quad 2.67 GHz ×2 CPU and 72-GB RAM running OpenSUSE 11.2.
[25] In this paper, we use part-of-speech (POS) tag sequences derived from the sentences instead of sentences themselves. So terminals in the grammar are POS tags.



(a) Uniform parameters (b) Learned parameters

Fig. 6. (Colour online) Prefix and sentence probability for the ATR corpus.

likelihood estimation. It is also observed that the difference between the two curves in Figure 6(b) is smaller than the one in Figure 6(a), which is statistically confirmed by t-test at 0.05 significance level.[26]

One potential explanation for this phenomenon is as follows. Let uw be a sentence in the ATR corpus, where w is the last word. The difference between the probability of prefix u and the probability of sentence uw is the sum of infinitely many probabilities of the sentences D extending u except uw. Since most members of D do not appear in the corpus, their total probability computed from the parameters learned from the corpus by maximum likelihood estimation considerably decreases compared with the case of using uniform parameters where any one of D receives non-negligible probability mass. Since this happens to every prefix used in the experiment, we see the narrowing difference between the average sentence probability curve in Figure 6(a) and the average prefix probability curve in Figure 6(b).

One of the usage of prefix probability computation is to predict the most likely next word of a prefix u. Let $P_{\mathrm{cfg}}(\cdot)$ be a distribution over sentences by a PCFG. Then the *conditional prefix probability* $P_{\mathrm{prefix}}(w \mid u)$ of a word w given u is computed as $P_{\mathrm{prefix}}(w \mid u) \overset{\mathrm{def}}{=} \dfrac{P_{\mathrm{prefix}}(uw)}{P_{\mathrm{prefix}}(u)}$, where $P_{\mathrm{prefix}}(u) = \sum_{uv:\mathrm{sentence}} P_{\mathrm{cfg}}(uv)$.

Since we found the prefix probability computation is computationally burdensome for long prefixes, we tested short prefixes. For example, for a prefix $u =$ [`t_interj_hesit`, `t_interj_pre`, `t_daimeisi_domo`] of length three and a word $w =$ `t_myoji_first`, we calculated $P_{\mathrm{prefix}}(w \mid u)$, assuming equiprobable rule selection, as $P_{\mathrm{prefix}}(w \mid u) = 0.00103$. Thus, by computing $P_{\mathrm{prefix}}(w \mid u)$ for all possible ws, we can predict the most likely next word of a given prefix as $\operatorname{argmax}_w P_{\mathrm{prefix}}(w \mid u)$.

[26] In Figure 6(a), the average difference between the two curves is 6.57 whereas in Figure 6(b) the average difference is 3.59.

```
values(lc(s,s),[rule(s,[s,s])]).  values(lc(s,a),[rule(s,[a])]).
values(lc(s,b),[rule(s,[b])]).    values(first(s),[a,b]).
values(att(s),[att,pro]).

pre_plcg(L):- g_call([s],L,[]).     % L is a prefix
g_call([],L,L).
g_call([G|R],[Wd|L],L2):-
   ( G = Wd -> L1 = L               % shift operation
   ; msw(first(G),Wd),lc_call(G,Wd,L,L1) ),
   ( L1 == [] -> L2 = []            % (pseudo) success
   ; g_call(R,L1,L2) ).
lc_call(G,B,L,L2):-                 % B-tree is completed
   msw(lc(G,B),rule(A,[B|RHS2])),
   ( G == A -> true ; values(lc(G,A),_) ),
   ( L == [] -> L1 = []             % (pseudo) success
   ; g_call(RHS2,L,L1) ),
   ( G == A -> att_or_pro(A,Op),    % attach or project
     ( Op == att -> L2 = L1 ; lc_call(G,A,L1,L2) )
   ; lc_call(G,A,L1,L2) ).
att_or_pro(A,Op):- ( values(lc(A,A),_) -> msw(att(A),Op) ; Op=att ).
```

Fig. 7. Prefix PLCG parser DB_1.

4 Prefix probability computation for PLCGs

In this section, we deal with PLCGs and their prefix probability computation to test the generality of our approach.

PLCGs are a probabilistic version of left-corner grammars (LCGs), which in turn are a generative version of left-corner (LC) parsing (Manning 1997; Roark and Johnson 1999; Van Uytsel *et al.* 2001) that performs bottom-up parsing using three parsing operations, i.e. shift, attach and project. Although PLCGs and PCFGs may share a common CFG, they assign probability differently. PCFGs assign probability to the expansion of nonterminals by CFG rules in top-down parsing whereas PLCGs assign probability to the three operations in bottom-up parsing. As a result they define different classes of distribution.

Since prefix probability computation for PLCGs does not seem to be attempted before, we detail how a prefix PLCG parser DB_1 in Figure 7 works. It is a serial parser and specialized for a PLCG whose underlying CFG is $\mathbf{G}_0 = \{$ s→ s s, s→ a, s→b$\}$, the same as the one for DB_0 in Section 3.1. In the program `values(lc(`g`,`b`),`r`)` introduces `msw` atoms to choose a CFG rule $g \rightarrow b\beta$ from r, where g and b are in the left-corner relation of $\mathbf{G}_0$. So `values(lc(s,s),[rule(s,[s,s])])` introduces just one `msw` atom `msw(lc(s,s),[rule(s,[s,s])])`.[27] On the other hand, `values(first(s),[a,b])` that encodes the first set of "`s`" in $\mathbf{G}_0$[28] introduces

[27] Consequently, executing `msw(lc(s,s),X)` returns `X` = `rule(s,[s,s])` with probability 1.
[28] The first set of a nonterminal A is the set of terminals in the left-corner relation with A.

```
pre_plcg([a,b]) <=> g_call([s],[a,b],[])
g_call([s],[a,b],[]) <=> lc_call(s,a,[b],[]) & msw(first(s),a)
lc_call(s,a,[b],[])
   <=> g_call([],[b],[b]) & att_or_pro(s,pro)
          & lc_call(s,s,[b],[]) & msw(lc(s,a),rule(s,[a]))
g_call([],[b],[b])
lc_call(s,s,[b],[])
   <=> g_call([s],[b],[]) & att_or_pro(s,att)                    --(1)
          & msw(lc(s,s),rule(s,[s,s]))
     v g_call([s],[b],[]) & att_or_pro(s,pro)                    --(2)
          & lc_call(s,s,[],[]) & msw(lc(s,s),rule(s,[s,s]))
g_call([s],[b],[]) <=> lc_call(s,b,[],[]) & msw(first(s),b)      --(3)
lc_call(s,b,[],[])
   <=> att_or_pro(s,att) & msw(lc(s,b),rule(s,[b]))
     v att_or_pro(s,pro) & lc_call(s,s,[],[])
          & msw(lc(s,b),rule(s,[b]))
lc_call(s,s,[],[])
   <=> att_or_pro(s,att) & msw(lc(s,s),rule(s,[s,s]))
     v att_or_pro(s,pro) & lc_call(s,s,[],[])
          & msw(lc(s,s),rule(s,[s,s]))
att_or_pro(s,att) <=> msw(att(s),att)
att_or_pro(s,pro) <=> msw(att(s),pro)
```

Fig. 8. Explanation graph for `pre_plcg([a,b])`.

{`msw(first(s),a)`,`msw(first(s),b)`}. Similarly, `values(att(s),[att,pro])` introduces {`msw(att(s),att)`,`msw(att(s),pro)`} to make a probabilistic choice between attach and project. All probabilistic choices are equiprobable by default.

Suppose `pre_plcg(`ℓ`)` is given as a top-goal, where ℓ is a prefix. To parse ℓ, the parser repeatedly performs shift by `g_call/3` and attach and project by `lc_call/4` just as in LC parsing. The role of `g_call(`α`,`ℓ`,L2)` is to construct a partial parse tree whose leaves are a substring ℓ`-L2` (as d-list) spanned by α while instantiating `L2` to a sublist of ℓ. Let `G` be the left-most symbol of α and `Wd` the left-most word of ℓ. When `G` is a terminal and coincides with `Wd`, shift is performed and `Wd` is read from ℓ as an initial partial parse tree consisting of `Wd`. Otherwise `Wd` is considered as a word randomly selected from the first set of `G` using `msw(first(G),Wd)` as an initial partial parse tree.

A call to `lc_call(G,B,L,L2)` occurs when a `B`-tree (partial parse tree whose root node is `B`) is constructed and `G` is in the left-corner relation with `B`. It grows the `B`-tree probabilistically either by attach or by project using a CFG rule of the form `A`$\rightarrow$`B`β until a `G`-tree is constructed while consuming words in `L`, leaving `L2`. When the input is a prefix, the parser returns with pseudo success as soon as the prefix is consumed as indicated by the comment "`(pseudo) success`."

When `pre_plcg([a,b])` is given as a top-goal, for example, a linear explanation graph shown in Figure 8 is constructed in which `lc_call(s,s,[],[])` calls itself as a top-most looping subgoal. Now we analyze Figure 8 to confirm that our

$$\begin{array}{rclcl}
P(\texttt{pre\_plcg}([\texttt{a},\texttt{b}])) & = & \texttt{X1} & = & \texttt{X2} \\
P(\texttt{g\_call}([\texttt{s}],[\texttt{a},\texttt{b}],[])) & = & \texttt{X2} & = & \texttt{X3}\cdot 0.5 \\
P(\texttt{lc\_call}(\texttt{s},\texttt{a},[\texttt{b}],[])) & = & \texttt{X3} & = & \texttt{X4}\cdot\texttt{X10}\cdot\texttt{X5}\cdot 1 \\
P(\texttt{g\_call}([],[\texttt{b}],[\texttt{b}])) & = & \texttt{X4} & = & 1 \\
P(\texttt{lc\_call}(\texttt{s},\texttt{s},[\texttt{b}],[])) & = & \texttt{X5} & = & \texttt{X6}\cdot\texttt{X9}\cdot 1+\texttt{X6}\cdot\texttt{X10}\cdot\texttt{X8}\cdot 1 \\
P(\texttt{g\_call}([\texttt{s}],[\texttt{b}],[])) & = & \texttt{X6} & = & \texttt{X7}\cdot 0.5 \\
P(\texttt{lc\_call}(\texttt{s},\texttt{b},[],[])) & = & \texttt{X7} & = & \texttt{X9}\cdot 1+\texttt{X10}\cdot\texttt{X8}\cdot 1 \\
P(\texttt{lc\_call}(\texttt{s},\texttt{s},[],[])) & = & \texttt{X8} & = & \texttt{X9}\cdot 1+\texttt{X10}\cdot\texttt{X8}\cdot 1 \\
P(\texttt{att\_or\_pro}(\texttt{s},\texttt{att})) & = & \texttt{X9} & = & 0.5 \\
P(\texttt{att\_or\_pro}(\texttt{s},\texttt{pro})) & = & \texttt{X10} & = & 0.5
\end{array}$$

Fig. 9. Probability equations for prefix “ab”.

PLCG program correctly recognizes all partial parse trees for prefix “ab”. Figure 8 compactly represents a form of propositional PRISM program of all computation paths (sequences of probabilistic choices made by `msw` atoms) that generate prefix “ab”. Each path corresponds to a partial parse tree for “ab”. We write partial parse trees like `s(s(s(a),s(b)),s)`. We denote by $\mathbf{T}_i$ $(i = 1, 2, 3)$ a set of partial parse trees generated by computation paths corresponding to line (i) in Figure 8.

Then observe, for example, that computation paths going through `(1)` yield partial parse trees by combining ones generated by `g_call([s],[b],[])` and ones obtained by `s`-trees grown by attach operation using rule `s` → `s s`. This observation leads to an equation $\mathbf{T}_1 = \texttt{s(s(a),}\mathbf{T}_3\texttt{)}$, where $\texttt{s(s(a),}\mathbf{T}_3\texttt{)}$ stands for the set $\{\texttt{s(s(a),}\tau\texttt{)} \mid \tau \in \mathbf{T}_3\}$. In this way we obtain three equations below.

$$\begin{array}{ll}
\texttt{Eq 1:} & \mathbf{T}_1 = \texttt{s}(\texttt{s}(\texttt{a}),\mathbf{T}_3) \\
\texttt{Eq 2:} & \mathbf{T}_2 = \texttt{s}(\mathbf{T}_1,\texttt{s}) \cup \texttt{s}(\mathbf{T}_2,\texttt{s}) \\
\texttt{Eq 3:} & \mathbf{T}_3 = \{\texttt{s}(\texttt{b})\} \cup \texttt{s}(\mathbf{T}_3,\texttt{s})
\end{array}$$

By solving them we know that $\mathbf{T}_3 = \{\overbrace{\texttt{s}(\cdots\texttt{s}(}^{m}\,\texttt{s}(\texttt{b}),\overbrace{\texttt{s})\cdots\texttt{s})}^{m} \mid m \geqslant 0\}$ and so on. Also, recall that all computation paths for “ab” have to prove `lc_call(s,s,[b],[])` and hence have to go through `(1)` or `(2)` in Figure 8. Consequently, the set of partial parse trees for prefix “ab” generated by DB_1 is represented as $\mathbf{T}_1 \cup \mathbf{T}_2$, where $\mathbf{T}_1 \cup \mathbf{T}_2 = \{\overbrace{\texttt{s}(\cdots\texttt{s}(}^{n}\,\texttt{s}(\texttt{s}(\texttt{a}),\overbrace{\texttt{s}(\cdots\texttt{s}(}^{m}\,\texttt{s}(\texttt{b}),\overbrace{\texttt{s})\cdots\texttt{s})}^{m}\,)\,\overbrace{\texttt{s})\cdots\texttt{s})}^{n} \mid m \geqslant 0, n \geqslant 0\}$, which certainly represents all partial parse trees for prefix “ab”.

The probability equations derived from Figure 8 are shown in Figure 9. We have $P($`g_call([],[b],[])`$) = 1$ as `g_call([],[b],[b])` is logically proved. Suppose the probabilities of `msw(att(s),att)`, `msw(att(s),pro)`, `msw(first(s),a)` and `msw(first(s),b)` are all set to 0.5. Then the solution becomes `X1` $=$ `X2` $= 0.125$, `X3` $= 0.25$, `X4` $= 1$, `X5` $=$ `X6` $= 0.5$, `X7` $=$ `X8` $= 1$ and `X9` $=$ `X10` $= 0.5$. So the probability of `pre_plcg([a,b])` is computed as `X1` $= 0.125$.

Finally, we test prefix probability computation for PLCGs with real data. We prepared a prefix PLCG parser like the one in Figure 7 adapted for the ATR corpus and conducted prefix probability computation. Since the prefix PLCG parser

S	→	Pl	: 0.1 |	St	: 0.4 |	Cl	: 0.3 |	Mo	: 0.2
Pl	→	play	: 0.5 |	play Pl	: 0.3 |	Cl	: 0.1 |	Mo	: 0.1
St	→	study	: 0.1 |	study St	: 0.3 |	Pl St	: 0.2 |	Cl St	: 0.4
Cl	→	clean	: 0.4 |	clean Cl	: 0.5 |	Pl Cl	: 0.1		
Mo	→	mow	: 0.3 |	mow Mo	: 0.1 |	Pl Mo	: 0.4 |	Cl Mo	: 0.2

Fig. 10. PCFG for plan recognition.

is much larger than the corresponding prefix PCFG parser, containing over 20,000 `values/2` declarations, learning time and computation time are expected to be much longer than the PCFG case. Indeed, we measured CPU time for the PCFG and the PLCG respectively used to compute the probabilities of 100 prefixes created from 100 sentences in the ATR corpus by deleting their last word. The PCFG case took 12.3 ms/prefix whereas the PLCG case took 5.9 s/prefix, 48 times slower than the PCFG case. We also computed conditional probability $P_{\text{prefix}}(w \mid u)$ for PLCG prefixes. For a pair of the prefix $u = [$`t_interj_hesit`, `t_interj_pre`, `t_daimeisi_domo`$]$ and the word $w =$ `t_myoji_first` used in Section 3, for example, $P_{\text{prefix}}(w \mid u)$ is computed as 0.00032, which is considerably smaller compared with 0.00103 computed for the PCFG case.

5 Plan recognition

Prefix probability computation has practical applications. In this section we apply it to plan recognition using artificial data. Plan recognition is a task of inferring a plan (intension) from a sequence of observed actions and has been pursued, for example, in robotics to interpret video scene data and sensor data. One way to perform plan recognition is to use a formal grammar to describe the relation between plans and action sequences by equating sentences with action sequences and nonterminals with plans. However, to cope with noisy observations, it is natural to use probabilistic grammars such as PCFGs (Bobick and Ivanov 1998; Amft *et al.* 2007; Lymberopoulos *et al.* 2007; Geib and Goldman 2011; Pomponio *et al.* 2011).

Consider a simple PCFG in Figure 10 where S is a start symbol. It describes how four plans, i.e. { Pl(playing), St(studying), Cl(cleaning), Mo(mowing) } generate sequences of observable actions, i.e. { play, study, clean, mow }.

This PCFG generates action sequences such as "play clean", "play study study" and so on. Note that although "play clean" is a sentence derivable from Pl and Cl, it is also derivable from Cl and Mo as a prefix. Our task is to predict, given such a sequence x of actions which may be a prefix, the most likely plan $y^* = \operatorname*{argmax}_y P_{\text{prefix}}(\mathrm{S} \rightarrow y, y \overset{*}{\Rightarrow} x)$, where y ranges over { Pl, St, Cl, Mo } as a recognized plan for x. For example, for $x =$ "play clean", $y^* =$ St is the recognized plan giving the highest probability 0.0272 for $P_{\text{prefix}}(\mathrm{S} \rightarrow y, y \overset{*}{\Rightarrow} x)$.

To evaluate the accuracy of our prediction method, we take a random sample of 100 prefixes (action sequences) together with their plans and evaluate the accuracy of prediction by predicting the plan for each sampled action sequence. Prefixes are

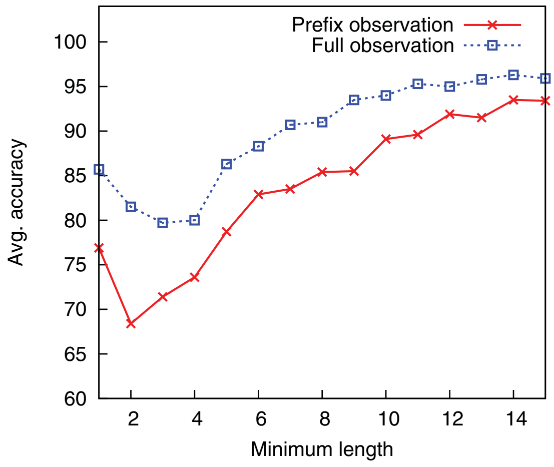


Fig. 11. (Colour online) Plan recognition.

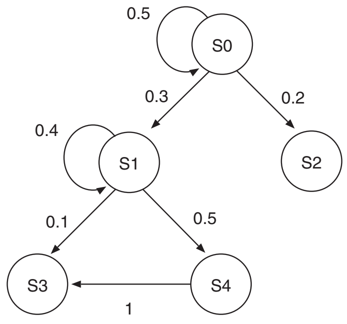
sampled so that they are not shorter than a threshold, referred to hereafter as "minlen" (minimum length). Finally, we compute the average accuracy of prediction over 10 runs. We tested two cases. One is full observation, i.e. prefixes are restricted to sentences. The other case is no restriction. Figure 11 shows the average accuracy w.r.t. minlen varying from 1 to 15. The blue curve corresponds to full observation (sentence) whereas the red one corresponds to prefix observation.[29]

We first notice that full observation always gives a better accuracy than prefix observation. This may be attributed to the fact that ambiguity measured by the average number of possible plans for an action sequence, termed "amb" here, in the case of full observation is less than the amb in the case of prefix observation at all minlen values (1 to 15). We also observe that the average accuracy (almost) monotonically increases as minlen increases in both cases. This is intuitively obvious because longer action sequences should give more clue to prediction and reduce the ambiguity about possible plans. Actually amb monotonically decreases w.r.t. minlen. On the other hand, however, this explanation conflicts with the initial drop in both curves w.r.t minlen, so we still need a coherent explanation.

6 Reachability probability

Computing probability through cyclic explanation graphs has applications beyond prefix probability computation. In this section, inspired by Gorlin *et al.* (2012), we take up the problem of computing reachability probability in discrete Markov chains. Figure 12 illustrates an example of Markov chain (the left-hand side (a))

[29] We measured the CPU time for plan recognition with randomly generated 100 action sequences whose average length is 4.18 (with std. 3.02). We obtained 1.79 ms/action sequence as the average time for plan recognition.



```
values(t(s0), [s0,s1,s2]).
:- set_sw(t(s0),[0.5,0.3,0.2]).
values(t(s1), [s1, s3, s4]).
:- set_sw(t(s1),[0.4, 0.1, 0.5]).
values(t(s4), [s3]).

trans(S,T):-
  ( S=s0;S=s1;S=s4 ), msw(t(S),T).

reach(S,T):-
  trans(S,U), reach(U,T).
reach(S,S).
```

(a) (b)

Fig. 12. (a) Markov chain, and (b) a program.

and its PRISM program (the right-hand side (b)), both borrowed from Gorlin *et al.* (2012) with a slight modification of the program.

A state transition in a Markov chain is made by a probabilistic choice of next state. Since the choice is exclusive and independent at each state, PRISM can simulate Markov chains, except when there is a self-loop, or more generally there is a set of state transitions forming a loop. In this case probability computation requires an infinite sum of probabilities which PRISM has been unable to deal with. However, by applying the general procedure described in Figure 5, we are now able to compute an infinite sum of probabilities, in particular for the reachability probability problem. For example, the reachable probability from `s0` to `s3` is represented as P(`reach(s0,s3)`) and is computed by the program as 0.6.

In the following we tackle a more complicated problem and verify the Synchronous Leader Election Protocol as described in a web page[30] for the PRISM model checker (Kwiatkowska *et al.* 2011) as one of the case studies. The protocol probabilistically elects a leader among processors distributed over a ring network communicating by synchronous message passing. It has two parameters, N, the number of processors, and K, the number of candidate ids used for election. Our task is to show that a leader will be elected with probability one. We use a PRISM program faithfully translated from the one shown in the web page with one exception. That is, we separate probabilistic transition from deterministic transition and only the predicate representing the former is tabled using PRISM's `p_table` declaration.[31]

Figure 13 shows CPU time taken for verification, varying N and K. As we see, the plotted curves for N = 5 and N = 6 look alike and the CPU time is almost

[30] `http://www.prismmodelchecker.org/casestudies/synchronous_leader.php`

[31] `:- p_table q/n` implies that the probabilistic predicate `q/n` is tabled *and* all other probabilistic predicates not declared by `p_table` will not be tabled. By default, all probabilistic predicates are tabled in PRISM, which sometimes makes explanation graphs unnecessarily large in view of probability computation due to the introduction of defining clauses without `msws` in the body. Selective tabling by `p_table` declarations prevents this.

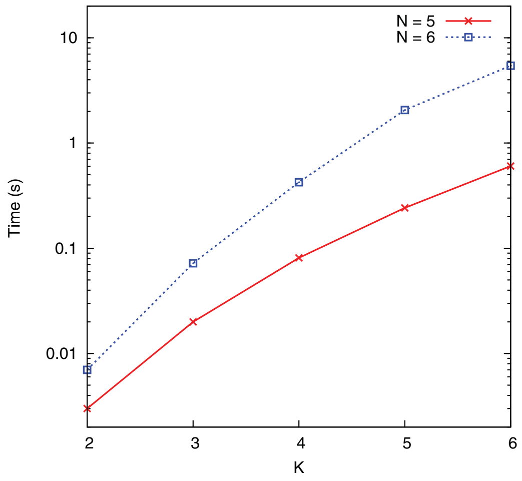


Fig. 13. (Colour online) CPU time for checking the Synchronous Leader Election Protocol.

exponential in K. We note that they are similar in shape to the ones (PIP-full) obtained by Gorlin *et al.* (2012) who conducted the same experiment to compare their approach with the PRISM model checker. However, an exact comparison with our approach would be difficult because of the difference in CPU processors and more seriously because of the difference in coding.[32]

7 Related works and future work

Tabling in logic programming has traditionally been used to eliminate redundant computation and to avoid infinite loop, but the use of loop detected by tabling for infinite probability computation seems new, although tabling for (finite) probability computation is well known and implemented in some probabilistic logic programming languages such as PRISM (Sato and Kameya 2001), ProbLog (Mantadelis and Janssens 2010) and PITA (Riguzzi and Swift 2011). This is probably because looping goals have long been considered useless despite the fact that they make sense if probabilities are involved and the loop computes converging probabilities like prefix probability computation.

Technically, our approach is closely related to Gorlin *et al.* (2012) in which the authors proposed probabilistic inference plus (PIP) that computes the probability of infinitely many explanations, and applied PIP to model checking. In PIP, to compute the probability of a query Q w.r.t. a probabilistic logic program P, a residual program is first constructed using XSB Prolog (Swift and Warren 2012) from P and Q. Then it is converted to a DCG called equation generator that generates possible explanations for Q as strings, from which a factored explanation diagram (FED) is derived. It is a compressed representation of the set of (possibly infinitely many) explanations for Q

[32] In Gorlin *et al.* (2012) the authors used a 2.5-GHz processor and encoded the Synchronous Leader Election Protocol problem via a PCTL model checker whereas we used a 2.67-GHz processor and directly encoded the problem as a PRISM program.

w.r.t. P and further converted to a system of polynomial equations. The probability of Q is obtained by solving the equations.

The basic idea of PIP is similar to our approach: probability computation by solving a set of equations derived from a symbolic diagram constructed from a program and a query. Nonetheless, there are substantial differences between PIP and our approach. First, PIP uses `msw/3` that has three arguments in which the second argument (trial-id) (Sato and Kameya 2001) is a term (clock) indicating when the `msw` is executed in the computing process. To ensure statistically correct treatment of the second argument for probability computation, PIP requires programs to be "temporally well-formed" and places three syntactic conditions on the occurrences of "instance arguments", i.e. arguments that work as a clock. These conditions look restrictive but how they affect the class of definable probabilistic models or how they are related to PRISM programs is unclear and not discussed in Gorlin *et al.* (2012).

PRISM, on the other hand, uses `msw/2` that omits the second argment from `msw/3` for computational efficiency and allows arbitrary programs but instead assumes that every occurrence of `msw/2` in a proof for the query is independent (*independence condition*) which guarantees the correctness of probability computation in PRISM.

Also, PIP constructs an FED, BDD-like graphical structure representing a set of explanations via a DCG (equation generator) whereas PRISM constructs an explanation graph without using a DCG. FEDs are powerful; they enable PIP to deal with programs that violate the exclusiveness condition required by PRISM while capturing common patterns in the set of explanations. However, when programs satisfy the exclusiveness condition (and the independence condition as well) as is often the case in probabilistic modeling by generative models such as BNs, HMMs, PCFGs and PLCGs, the construction of FEDs is unnecessary. A simpler structure, explanation graphs, is enough. As we have demonstrated, the sum of probabilities of infinitely many explanations can be efficiently computed by cyclic explanation graphs in such cases.

In addition, although it is not clearly stated in Gorlin *et al.* (2012), the authors seem to solve the set of equations by an iterative method described in Etessami and Yannakakis (2009) that is applicable to nonlinear cases. PRISM contrastingly assumes the linearity of equations and efficiently solves hierarchally ordered sets of system of linear equations, corresponding to SCCs, by matrix operation in cubic time in the number of variables. Considering the fact that nonlinearity occurs even in the case of PCFGs when we compute infix probability (Nederhof and Satta 2011a), however, it is an important future work to enhance PRISM's equation solving ability for nonlinear cases.

Current tabling in PRISM employs linear tabling in B-Prolog and it is straightforward to construct cyclic explanation graphs from defining clauses for tabled answers stored in the memory. Constructing cyclic explanation graphs in other Prolog systems such as XSB (Swift and Warren 2012) that employ a suspend–resume mechanism for tabling also seems possible.

Approximate computation of prefix probability seems possible, for example, by the iterative deepening algorithm used in ProbLog (De Raedt *et al.* 2007). To develop such an approximation algorithm remains a future work.

Prefix probability computation is mostly studied about PCFGs (Jelinek and Lafferty 1991; Stolcke 1995; Nederhof and Satta 2011a). Jelinek and Lafferty (1991) proposed a CKY-like algorithm for prefix probability computation in PCFGs in the Chomsky normal form. Their algorithm does not perform parsing but instead uses a single monolithic matrix whose dimension is the number of nonterminals which is constructed from a given PCFG. It runs in $O(N^3)$, where N is the length of an input prefix. Stolcke (1995) applied the Earley-style parsing to compute prefix probabilities. His algorithm uses a matrix of "probabilistic reflexive, transitive left-corner relation" computed from a given PCFG, independent of input sentences, similar to Jelinek and Lafferty (1991). Our approach differs from their approach, first in that it is general and works for arbitrary PRISM programs, and second in that it constructs an explanation graph for each input prefix and probabilities are computed on the basis of the SCCs derived from the explanation graph.

Nederhof and Satta (2011a) generalized prefix probability computation for PCFGs to infix probability computation for PCFGs. They also studied prefix probability computation for a variant of PCFGs (Nederhof and Satta 2011b). They proposed prefix probability computation for stochastic tree adjoining grammars (Nederhof *et al.* 1998). However, prefix probability computation for PLCGs has been unknown and our example in Section 4 is the first one to our knowledge.

Applying prefix probability computation to plan recognition in Section 5 is not new but our approach generalizes previous grammar-based approaches (Bobick and Ivanov 1998; Amft *et al.* 2007; Lymberopoulos *et al.* 2007; Geib and Goldman 2011; Pomponio *et al.* 2011) in that it allows for incomplete action sequences (prefixes) as observations. In relation to plan recognition, it is possible to apply prefix probability computation to predict the most likely action (word) that follows an observed action sequences (prefix) (Jelinek and Lafferty 1991), though we do not discuss it here.

We eliminated in this paper one of the restrictive assumptions on PRISM that the number of explanations for a goal is finite. However, there still remain restrictive assumptions, the exclusiveness assumption and the independence assumption (Sato and Kameya 2001). Their elimination by introducing BDDs (De Raedt *et al.* 2007; Riguzzi and Swift 2011) or FEDs (Gorlin *et al.* 2012) remains a future work.

8 Conclusions

We have proposed an innovative use of tabling: infinite probability computation based on cyclic explanation graphs generated by tabled search in PRISM. It generalizes prefix probability computation for PCFGs and is applicable to probabilistic models described by PRISM programs in general and to non-PCFG probabilistic grammars such as PLCGs in particular as we demonstrated. We applied our approach to plan recognition and to the reachability probability problem in probabilistic model checking. We expect that our approach provides a declarative way of logic-based probabilistic modeling of cyclic relations.

References

Amft, O., Kusserow, M. and Troster, G. 2007. Probabilistic parsing of dietary activity events. In *Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks*, Aachen, Germany, 2007, Springer IFMBE Proceedings, vol. 13, 242–247.

Baker, J. K. 1979. Trainable grammars for speech recognition. In *Proceedings of Spring Conference of the Acoustical Society of America*, 547–550.

Bobick, A. and Ivanov, Y. 1998. Action recognition using probabilistic parsing. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'98)*, 196–202.

De Raedt, L., Kimmig, A. and Toivonen, H. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, 2468–2473.

Etessami, K. and Yannakakis, M. 2009. Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. *Journal of ACM 56,* 1.

Geib, C. and Goldman, R. 2011. Reorgnizing plans with loops represented in a lexicalized grammar. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI'11)*, 958–963.

Gorlin, A., Ramakrishnan, C. and Smolka, S. 2012. Model checking with probabilistic tabled logic programming. *Theory and Practice of Logic Programming (TPLP) 12,* 4–5, 681–700.

Hinton, A., Kwiatkowska, M., Norman, G. and Parker, D. 2006. PRISM: A tool for automatic verification of probabilistic systems. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, LNCS, vol. 3920. Springer, New York, 441–444.

Jelinek, F. and Lafferty, J. 1991. Computation of the probability of initial substring generation by stochastic context-free grammars. *Computational Linguistics 17,* 3, 315–323.

Kameya, Y. and Sato, T. 2000. Efficient EM learning for parameterized logic programs. In *Proceedings of the 1st Conference on Computational Logic (CL'00)*, Lecture Notes in Artificial Intelligence, vol. 1861. Springer, New York, 269–294.

Kwiatkowska, M., Norman, G. and Parker, D. 2011. PRISM 4.0: Verification of probabilistic real-time systems. In *Proceeding of the 23rd International Conference on Computer Aided Verification (CAV'11)*, G. Gopalakrishnan and S. Qadeer, Eds., LNCS, vol. 6806. Springer, New York, 585–591.

Lymberopoulos, D., Teixeira, T. and Savvides, A. 2007. Detecting patterns for assisted living using sensor networks: A case study. In *Proceedings of the 2007 International Conference on Sensor Technologies and Applications (SENSORCOMM '07)*, 590–596.

Manning, C. 1997. Probabilistic parsing using left corner language models. In *Proceedings of the 5th International Conference on Parsing Technologies (IWPT-97)*. MIT Press, Cambridge, MA, 147–158.

Manning, C. D. and Schütze, H. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA.

Mantadelis, T. and Janssens, G. 2010. Dedicated tabling for a probabilistic setting. In *Proceedings of the 26th International Conference on Logic Programming (ICLP'10) (Technical Communications)*, 124–133.

Meyer, C., Ed. 2000. *Matrix Analysis and Applied Linear Algebra.* Society for Industrial and Applied Mathematics, Philadelphia, PA.

Nederhof, M., Anoop Sarkar, A. and Satta, G. 1998. Prefix probabilities from stochastic tree adjoining grammars. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics (ACL'98)*, 953–959.

Nederhof, M. and Satta, G. 2011a. Computation of infix probabilities for probabilistic context-free grammars. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing (EMNLP'11)*, 1213–1221.

Nederhof, M. and Satta, G. 2011b. Prefix probability for probabilistic synchronous context-free grammars. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL'11)*, 460–469.

Pomponio, L., Le Goc, M., Eric, P. and Alain, A. 2011. Combining timed data and expert's knowledge to model human behavior. In *Proceedings of the Health Ambient Information Systems Workshop (HamIS'11)*, http://ceur-ws.org/Vol-729/, vol. 729.

Riguzzi, F. and Swift, T. 2011. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming (TPLP) 11,* 4–5, 433–449.

Roark, B. and Johnson, M. 1999. Efficient probabilistic top-down and left-corner parsing. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, 421–428.

Rocha, R., Silva, F. and Costa, V. 2005. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming (TPLP) 5,* 1–2, 161–205.

Sato, T. 2008. A glimpse of symbolic-statistical modeling by PRISM. *Journal of Intelligent Information Systems 31,* 2, 161–176.

Sato, T. and Kameya, Y. 1997. PRISM: A language for symbolic-statistical modeling. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97)*, 1330–1335.

Sato, T. and Kameya, Y. 2001. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research 15*, 391–454.

Sato, T. and Kameya, Y. 2008. New advances in logic-based probabilistic modeling by PRISM. In *Probabilistic Inductive Logic Programming*, L. De Raedt, P. Frasconi, K. Kersting and S. Muggleton, Eds., LNAI, vol. 4911. Springer, New York, 118–155.

Sato, T. and Meyer, P. 2012. Tabling for infinite probability computation. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*, Budapest, Hungary, Leibniz International Proceedings in Informatics, vol. 17. Kluwer, Boston, MA, 348–358.

Stolcke, A. 1995. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics 21,* 2, 165–201.

Swift, T. and Warren, D. 2012. XSB: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming (TPLP) 12,* 1–2, 157–187.

Tamaki, H. and Sato, T. 1986. OLD resolution with tabulation. In *Proceedings of the 3rd International Conference on Logic Programming (ICLP'86)*, Lecture Notes in Computer Science, vol. 225. Springer, New York, 84–98.

Uratani, N., Takezawa, T., Matsuo, H. and Morita, C. 1994. ATR integrated speech and language database. Tech. Rep., TR-IT-0056, ATR Interpreting Telecommunications Research Laboratories, Kyoto, Japan.

Van Uytsel, D., Van Compernolle, D. and Wambacq, P. 2001. Maximum-likelihood training of the PLCG-based language model. In *Proceedings of the IEEE Automatic Speech Recognition and Understanding Workshop 2001 (ASRU'01).*

Warren, D. S. 1992. Memoing for logic programs. *Communications of the ACM 35,* 3, 93–111.

Wetherell, C. S. 1980. Probabilistic languages: A review and some open questions. *Computing Surveys 12,* 4, 361–379.

Zhou, N.-F., Kameya, Y. and Sato, T. 2010. Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *Proceedings of the 22nd International Conference on Tools with Artificial Intelligence (ICTAI-2010).*

Zhou, N.-F. and Sato, T. 2003. Toward a high-performance system for symbolic and statistical modeling. In *Proceedings of IJCAI-03 Workshop on Learning Statistical Models from Relational Data (SRL'03)*, http://scholarworks.umass.edu/cgi/viewcontent.cgi?article=1163&context=cs_faculty_pubs, 133–140.

Zhou, N.-F., Sato, T. and Shen, Y.-D. 2008. Linear tabling strategies and optimization. *Theory and Practice of Logic Programming (TPLP) 8,* 1, 81–109.