TLP **10** (4–6): 449–464, 2010. © Cambridge University Press 2010
doi:10.1017/S1471068410000219

Inference with constrained hidden Markov models in PRISM

HENNING CHRISTIANSEN, CHRISTIAN THEIL HAVE,
OLE TORP LASSEN and MATTHIEU PETIT*

Research Group PLIS: Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, P.O.Box 260, DK-4000 Roskilde, Denmark
(*e-mail:* {henning,cth,otl,petit}@ruc.dk)

submitted 7 February 2010; revised 10 April 2010; accepted 21 May 2010

Abstract

A Hidden Markov Model (HMM) is a common statistical model which is widely used for analysis of biological sequence data and other sequential phenomena. In the present paper we show how HMMs can be extended with side-constraints and present constraint solving techniques for efficient inference. Defining HMMs with side-constraints in Constraint Logic Programming has advantages in terms of more compact expression and pruning opportunities during inference. We present a PRISM-based framework for extending HMMs with side-constraints and show how well-known constraints such as `cardinality` and `all_different` are integrated. We experimentally validate our approach on the biologically motivated problem of global pairwise alignment.

KEYWORDS: hidden Markov model with side-constraints, inference, programming in statistical modeling

1 Introduction

Hidden Markov Models (HMMs) are one of the most popular models for analysis of sequential processes taking place in a random way, where "randomness" may also be an abstraction covering the fact that a detailed analytical model for the internal matters is unavailable. Such a sequential process can be observed from outside by its emission sequence (letters, sounds, measures of features, all kinds of signals) produced over time, and an HMM postulates a hypothesis about the internal machinery in terms of a finite state automaton equipped with probabilities for the different state transitions and emissions. A common inference for a given observed sequence means to compute the "best" state transitions that the HMM may go through to produce the sequence, and thus this represents a best hypothesis for the internal structure or "content" of the sequence. HMMs are widely used in speech recognition and biological sequence analysis (Rabiner 1989; Durbin *et al.* 1998).

* This work is supported by the project Logic-statistic modeling and analysis of biological sequence data funded by the NABIIT program under the Danish Strategic Research Council.

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

The efficiency of computations on HMMs heavily depends on the Markov property. Decisions made during a process run depends only on a limited past. Dynamic programming algorithms, such as Viterbi and Forward-Backward, are then used to perform efficient inference. However, many problems would require more complex dependencies among elements of the process. For example, it may be interesting to constrain an HMM to visit only different states or limit the number of visits to a given state. It is possible to model the `all_different` constraint for the states visited by extending the underlying finite state automaton, but for the price of a factorial number of new states and with an obvious impact on inference. As an alternative to modifying the HMM structure, we instead extend the HMM with side-constraints (Sato and Kameya 2008; Roth and Yih 2005). However, classical algorithms, such as Viterbi, must be modified to take care about these side-constraints (Chang *et al.* 2008; Christiansen *et al.* 2009).

In this paper, we extend HMMs with side-constraints, leading to what we call Constrained HMMs (CHMMs). Side-constraints are external constraints declared in addition to those defined by the structure of an HMM. The concept of CHMMs was introduced by Sato *et al.* in (Sato and Kameya 2008), although earlier and unrelated systems have used the same or similar names (discussed in section 6). The contribution of this paper is to define CHMMs as constraint logic programs extended with probabilistic choices and to show how to employ this setting for more efficient Viterbi computation, i.e., computation of the most probable explanation of an observation. Moreover, defining HMMs with side-constraints in Constraint Logic Programming has advantages in terms of more compact expression and pruning opportunities during inference. We show how to implement CHMMs in PRISM (Sato and Kameya 1997) and how to integrate well-known constraints, such as `cardinality` and `all_different`, into this framework. We validate our approach experimentally on the biologically motivated problem of global pairwise alignment.

The paper is organized as follows: section 2 describes background on HMMs. In section 3, we formally introduce the constraint model associated with a CHMM. Section 4 describes our PRISM-based framework to define CHMMs. Section 5 presents an experimental validation. Finally, sections 6 and 7 present related work and conclusions.

2 Background

Here we define Hidden Markov Models (HMM)s and illustrate their application to the problem of pairwise global alignment.

2.1 Hidden Markov models

For simplicity of the technical definitions, we limit ourselves to a discrete Hidden Markov Model with a distinguished initial state.

Definition 2.1
A *Hidden Markov Model* (HMM) is a 4-tuple $\langle S, A, T, E \rangle$, where

- $S = \{s_0, s_1, \ldots, s_m\}$ is a set of *states* which includes an *initial* state referred to as s_0;

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

- $A = \{e_1, e_2, \ldots, e_k\}$ is a finite set of emission symbols;
- $T = \{(p(s_0; s_1), \ldots, p(s_0; s_m)), \ldots, (p(s_m; s_1), \ldots, p(s_m; s_m))\}$ is a set of *transition probability distributions* representing probabilities to transit from one state to another;
- $E = \{(p(s_1; e_1), \ldots, p(s_1; e_k)), \ldots, (p(s_m; e_1), \ldots, p(s_m; e_k))\}$ is a set of *emission probability distributions* representing probabilities to emit each symbol from each state.

We define a *run* of an HMM as a pair consisting of a sequence of states $s^{(0)}s^{(1)}\ldots s^{(n)}$, called a *path* and a corresponding sequence of emissions $e^{(1)}\ldots e^{(n)}$, called an *observation*, such that

- $s^{(0)} = s_0$;
- $\forall i, 0 \leqslant i \leqslant n-1, p(s^{(i)}; s^{(i+1)}) > 0$ (probability to transit from $s^{(i)}$ to $s^{(i+1)}$);
- $\forall i, 0 < i \leqslant n, p(s^{(i)}; e^{(i)}) > 0$ (probability to emit $e^{(i)}$ from $s^{(i)}$).

The *probability* of such a run is defined as $\prod_{i=1..n} p(s^{(i-1)}; s^{(i)}) \cdot p(s^{(i)}; e^{(i)})$.

2.2 An example HMM: Pairwise global alignment

As an example of an HMM that we later extend with constraints, we consider the problem of aligning two sequences. Sequence alignment is among the most common tasks in computational biology, where it is used to discover preserved regions from a common ancestor. Notice that we here use a so-called pair HMM (Durbin *et al.* 1998) which emits two sequences at the same time, and which is a straightforward extension of the definition above.

In the global alignment problem, two sequences x and y must be aligned optimally, based on a scoring scheme for comparison of different alignments. In probabilistic modeling, a probability is associated with each pair of symbols emitted from a state and similarly a probability for introducing gaps, δ, and extending gaps, ϵ, in the alignment of the sequences is defined. The probability of an alignment is then the product of probabilistic transitions performed to recognize the alignment. In biology, these probabilities are defined to reflect observed statistics about sequence mutation and conservation.

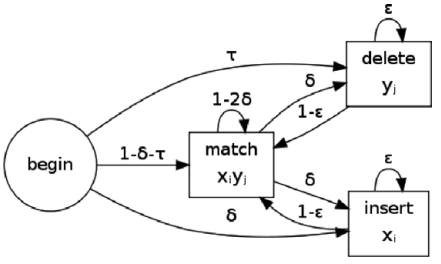


Fig. 1. A pair HMM for pairwise global alignment of sequences. States, represented by squares for emitting states and circles for silent states, are connected by arrows representing transitions labeled with probabilities.

Fig. 1 shows an HMM capable of generating a pair of aligned sequences. When given two sequences to align, then a path from the initial state, `begin`, such that

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

the model emits the two sequences, corresponds to an alignment. The initial state, `begin`, does not emit symbols. The `match` state emits a pair of symbols (x_i, y_j), one for each sequence corresponding to alignment of the symbol at position i in sequence x and the symbol at position j in sequence y. Emitted symbols can be identical or different. A difference represents a potential mutation between the two sequences. The `insert` state emits only the next symbol of sequence x, effectively aligning position x_i to a gap in y. Oppositely, the `delete` state aligns a symbol y_j to a gap in sequence x.

The following example shows an alignment of two short protein sequences, where the third line indicates the state sequence of this alignment abbreviated with the first letter of the state name:

```
Sequence x:   H G K K G A     A Q V
Sequence y:         K G P K K A Q A
alignment : b i i i m m m d d m m m
```

In this context, a common task is to find the optimal alignment. This means to find a state sequence that can recognize the two sequences and has maximal probability. A second task is to calculate the probability to observe an emission sequence. A third type of inference is parameter learning, where we are given a set of alignments and estimate the "best" parameters for the model, where best usually means that they maximize likelihood of the alignments.

3 A constraint model for HMM with side-constraints

In this section, we give a formal definition of CHMMs and propose a constraint model for CHMM runs. Then, the computation of the most probable path is adapted for CHMMs.

3.1 Constrained hidden Markov model

A CHMM extends an HMM with constraints that limit the set of valid runs and leave fewer paths to consider for any given sequence.

Definition 3.1
A *constrained HMM (CHMM)* is defined by a 5-tuple $\langle S, A, T, E, C\rangle$ where $\langle S, A, T, E\rangle$ is an HMM and C is a set of constraints, each of which is a mapping from HMM runs into $\{true, false\}$.

A *run* of a CHMM, $\langle path, observation\rangle$ is a run of the corresponding HMM for which $C(path, observation)$ is true.

Notice that we define constraints in a highly abstract way, independently of any specific constraint language. In the following, constraints over finite domains (Van Hentenryck *et al.* 1995) are used, although other constraint languages such as $CLP(Q)$ and $CLP(R)$ could have been used as well.

3.2 Runs of a CHMM as a constraint program

In this section, we propose to model runs of CHMM by a constraint program over finite domains. In this context, a run of CHMM is a solution of the constraint program.

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

Let $\langle S, A, T, E, C\rangle$ be a CHMM and n the sequence length. A constraint program for runs is given by the following predicate.

$$run([s^{(0)}, S_1, \ldots, S_n], [E_1, \ldots, E_n])$$

where each variable S_i and E_i represents the state and the emission at the step i. The domains of S_i and E_i, are given as $\mathrm{dom}(S_i) = S \setminus \{s_0\}$ and $\mathrm{dom}(E_i) = E$. The *run* predicate is specified as follows.

$run([s^{(0)}, S_1, \ldots, S_n], [E_1, \ldots, E_n])$ is *true* iff

$$\begin{aligned}
&\exists s^{(1)} \in \mathrm{dom}(S_1), \ldots, \exists s^{(n)} \in \mathrm{dom}(S_n) \text{ and} \\
&\qquad \exists e^{(1)} \in \mathrm{dom}(E_1), \ldots, \exists e^{(n)} \in \mathrm{dom}(E_n), \\
&\qquad C(s^{(0)}s^{(1)}\ldots s^{(n)}, e^{(1)}\ldots e^{(n)}) \text{ is true}, s^{(0)} = s_0 \text{ and} \\
&\qquad\qquad p(s^{(0)}; s^{(1)}) \cdot p(s^{(1)}; e^{(1)}) \ldots p(s^{(n-1)}; s^{(n)}) \cdot p(s^{(n)}; e^{(n)}) > 0. \quad (1)
\end{aligned}$$

Formula (1) states that $s^{(0)}s^{(1)}\ldots s^{(n)}$ and $e^{(1)}\ldots e^{(n)}$ is a run of the HMM that satisfies C. By the definition of *run*/2, (local) relationships between S_i and S_{i+1} and S_i and E_i can be established, since the probability of a run must be positive. Indeed, valuation of S_i to $s^{(i)}$ and S_{i+1} to $s^{(i+1)}$ can be part of a solution of the constraint program whenever $p(s^{(i)}; s^{(i+1)}) > 0$. These relationships between variables of *run*/2 are modeled by the following constraints,

$$trans(S_{i-1}, S_i) \text{ and } emit(S_i, E_i), \text{for all } i,\ 1 \leqslant i \leqslant n$$

where S_i, S_{i+1} and E_i are the variables of *run*/2. These constraints are defined as follows.

- $trans(S_i, S_{i+1})$ is *true* iff $\exists s^{(i)} \in \mathrm{dom}(S_i)$ and $s^{(i+1)} \in \mathrm{dom}(S_{i+1})$ such that $p(s^{(i)}; s^{(i+1)}) > 0$;
- $emit(S_i, E_i)$ is *true* iff $\exists s^{(i)} \in \mathrm{dom}(S_i)$ and $e^{(i)} \in \mathrm{dom}(E_i)$ such that $p(s^{(i)}; e^{(i)}) > 0$.

Section 4 below shows an implementation of this framework such that a solution of the constraint program corresponds to a valid derivation of a PRISM program.

3.3 Example: Constrained pairwise global alignment

We consider the HMM presented in section 2.2 and extend it into a CHMM by the following set of constraints,

$$\begin{aligned}
C = \{&\texttt{cardinality_atmost}(N_d, [S_1, \ldots, S_n], \text{delete}), \\
&\texttt{cardinality_atmost}(N_i, [S_1, \ldots, S_n], \text{insert})\}.
\end{aligned}$$

A constraint $\texttt{cardinality_atmost}(N, L, X)$ is satisfied whenever L is a list of elements, out of which at most N are equal to X. In a biological context, it is reasonable to consider only alignments with a limited number of insertions and deletions given the assumption that the two sequences are related.

As described above, we can consider this CHMM as a constraint program

$$run([s^{(0)}, S_1, \ldots, S_n], [E_1, \ldots, E_n])$$

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

$$\begin{array}{ll} trans_ctr: & \Sigma := \Sigma \cup \{\langle s', i{+}1, p \cdot p(s;s') \cdot p(s';e^{(i+1)}), \pi\, s', \sigma \wedge S_{i+1} = s' \rangle\} \\ & \text{whenever } \langle s, i, p, \pi, \sigma \rangle \in \Sigma, p(s;s'), p(s';e^{(i+1)}) > 0 \\ & \textit{check_constraints}(\sigma \wedge S_{i+1} = s') \text{ and } prune_ctr \text{ does not apply.} \\ \\ prune_ctr: & \Sigma := \Sigma \setminus \{\langle s, i{+}1, p', \pi', \sigma' \rangle\} \\ & \text{whenever there is another } \langle s, i{+}1, p, \pi, \sigma \rangle \in \Sigma \text{ with} \\ & p \geq p' \text{ and } \mathrm{sol}(\sigma') \subseteq \mathrm{sol}(\sigma). \end{array}$$

Fig. 2. Rewriting rules for the computation of most probable paths for CHMM.

where $\mathrm{dom}(S_i) \in \{\mathrm{match}, \mathrm{delete}, \mathrm{insert}\}$, $\mathrm{dom}(E_i) \in \{A, C, D, \ldots, W, Y\}$[1] and the constraints C are as described above.

3.4 Computation of the most probable path for a CHMM

The Viterbi algorithm (Viterbi 1967) is a dynamic programming algorithm for finding a most probable path corresponding to a given observation. The algorithm keeps track of, for each prefix of an observed emission sequence, the most probable (partial) path leading to each possible state, and extends those step by step into longer paths, eventually covering the entire emission sequence. Here, we adapt this algorithm for CHMMs.

Consider a given observation $e^{(1)} \ldots e^{(n)}$, a CHMM $\langle S, A, T, E, C \rangle$, and its constraint program

$$run([s^{(0)}, S_1, \ldots, S_n], [e^{(1)}, \ldots, e^{(n)}]).$$

The most probable path is computed by finding the valuation $s^{(1)}, \ldots, s^{(n)}$ that maximizes the objective function: the probability of a run.

Computation of the most probable path for CHMM is expressed as a rewriting system on a set of 5-tuples Σ. Each such 5-tuple is of form $\langle s, i, p, \pi, \sigma \rangle$ where π is a partial path ending in state s and representing a path for the emission sequence prefix $e^{(1)} \cdots e^{(i)}$; p is the computed probability for the emissions and transitions applied in the construction of π, and σ is the current constraint store seen as a conjunction of constraints. Any ground and satisfied constraint will be removed from the constraint store, and *true* refers to the empty conjunction. The set of solutions of a constraint store σ is denoted by $\mathrm{sol}(\sigma)$. The two rewriting rules in Fig. 2 describe an iteration step of the computation of the most probable path.[2] The computation starts from an initial set of 5-tuples

$$\{\langle s^{(0)}, 0, 1, \epsilon, C \wedge trans(s^{(0)}, S_1) \wedge \bigwedge_{1 \leqslant i \leqslant n-1} trans(S_i, S_{i+1}) \wedge \bigwedge_{1 \leqslant i \leqslant n} emit(S_i, e_i) \rangle\}. \quad (2)$$

[1] This set of letters refers to the 21 different amino acids from which proteins are composed.

[2] When any reference to constraints and the constraint store are removed from Fig. 2, we have a compact representation of one iteration step of the Viterbi algorithm for HMMs.

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

The *trans_ctr* rule expands an existing partial path one step in directions that preserve the satisfaction of the constraint store; this satisfiability check is denoted *check_constraints* (and depends thus on the particular C). The *prune_ctr* rule removes partial solutions that are not optimal for the current observation prefix *and* shares the same set of complete solutions with the better partial solution. The second condition is necessary in case no partial path contained in $\mathrm{sol}(\sigma)$ can be extended into a full path without violating the constraints. We take the following correctness property for granted.

Proposition 3.1
Assume a CHMM H with the notation as above and an observation $Obs = e^{(1)} \cdots e^{(n)}$. When the Viterbi algorithm in Fig. 2 is executed from an initial set of 5-tuples given the formula (2), it terminates with a set of 5-tuples Σ_{final}. It holds that

- For any $\langle s, n, p, \pi, true \rangle \in \Sigma_{final}$, π is a most probable path for Obs ending in s and with probability p.
- Whenever there exists a path for Obs ending in s, Σ_{final} includes a 5-tuple of the form $\langle s, n, p, \pi, true \rangle$.

Notice that all the variables of the constraint program are valuated when a final state is reached, and thus any final constraint store is equivalent to *true* (as *trans_ctr* prevents any inconsistent store to arise).

The classical Viterbi algorithm is guaranteed to run in time linear to the length of the given sequence, whereas our algorithm may in the worst case run in exponential time; this may occur if *prune_ctr* cannot be applied at all. In other words, a representation of the constraint store that allows an efficient comparison as in "$\mathrm{sol}(\sigma') \subseteq \mathrm{sol}(\sigma)$" is essential for the practicability of our algorithm. On the other hand, for those problems that can be formulated as a CHMM with effective and efficient definitions of *check_constraints* and the comparison test, the Σ states may stay of a reasonable size. Notice that our algorithm is still correct if we use approximations of these tests, more specifically, *check_constraints* may occasionally return *true* when the correct answer is *false* and the opposite for the comparison.

4 Implementation of CHMMs in PRISM

After briefly introducing PRISM, we propose a methodology to define CHMMs in this framework.

4.1 A brief introduction to the PRISM system

PRISM (Sato and Kameya 2008) is a powerful system for working with probabilistic-logic models, based on an extension to Prolog with discrete random variables, called multi-valued switches. We illustrate this with a simple example HMM with two states `s0` and `s1`. A switch declaration,

```
values(x,0).
```

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

associates the named random variable x with a set of outcomes O. Whenever the goal msw(x,X) is called from the program, then a probabilistic choice will be made unifying X with an element of O. Switches can also be defined in a parametric form,

```
values(emit(_),[a,b]).  % symbol emission
values(trans(_),[s0,s1]). % state transition
```

where each declaration defines a family of switches, one for each possible instance of emit(_) and trans(_) and each instance is given a distinct probability distribution. This parametrization can serve to model dependencies: in our HMM example we define the parameters to be the states s0 and s1 (plus init for trans(_)), thus defining emissions and transitions for each state with the Markov property. Finally, we define a logic program to implement the probabilistic model,

```
hmm(L):- run_length(T), hmm(T,init,L).
hmm(0,_,[]).
hmm(T,State,[Emit|EmitRest]) :-
   T > 0,
   msw(trans(State),NextState),
   msw(emit(NextState),Emit),
   T1 is T-1,
   hmm(T1,NextState,EmitRest).
run_length(10).
```

Here, a derivation of the goal hmm corresponds to what we define as a *run* in section 2.1. As shown by (Sato 1995), Prolog's traditional Herbrand model semantics generalizes immediately to a probabilistic semantics when probabilities are given for each random variable (provided that a few restrictions are respected on how msw is used in the program). Thus a PRISM program defines a probabilistic model that provides a probability distribution for all goals that can be formulated in the program's logical language. PRISM assigns each possible derivation of a goal a probability defined as the product of the probabilities of the selected switch outcomes of the derivation. Under normal conditions, it will be the case that the sum of probabilities of all possible derivations of such a goal is unity, but these conditions can be violated in a constrained model. If a program attempts to unify the stochastically selected outcome of a switch with some other value distinct from that outcome, this unification will fail resulting in a failed derivation.

PRISM includes built-in mechanisms for efficient probabilistic inference based on tabling. During inference, once a probabilistic goal has been solved, its answers are put in a global table. Later calls to the same goal will simply lookup the answer in the table in constant time. PRISM utilizes this to provide an efficient generalized Viterbi algorithm that may be used for the computation of the most likely *successful* derivation for a large number of probabilistic models including HMMs. PRISM also includes similar utilities for calculating the probability of a derivation or set of such and machine learning algorithms which produce the most likely probabilities for switch outcomes in order to explain a set of observed goals.

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

4.2 A framework for CHMMs in PRISM

We have implemented a framework for integration of side-constraints in a PRISM program.[3] The framework has been used for adding constraints to HMM based models, but it should be possible to extend to other kinds of models. The underlying idea is that the program is augmented with a constraint store and a constraint checker goal is inserted in a few strategic places of the PRISM program. This constraint checking is the direct implementation of check_constraints of *trans_ctr*. The *prune_ctr* implementation is not discussed as we use the tabling mechanism of PRISM to prune the search space.

4.2.1 Integration of side-constraints in a PRISM program

This section describes how our framework can be integrated in a PRISM program. As an example, we consider an implementation of the HMM from the previous section. Below the central recursive predicate of the implementation is shown extended with constraint checking,

```
1   hmm(T,State,[Emit|EmitRest],StoreIn) :-
2      T > 0,
3      msw(trans(State),NextState),
4      msw(emit(NextState),Emit),
5      check_constraints([NextState,Emit],StoreIn,StoreOut),
6      T1 is T-1,
7      hmm(T1,NextState,EmitRest,StoreOut).
```

Integration of side-constraint checking is done by extending relevant predicates with an extra parameter (`StoreIn,StoreOut` in the code above) to accommodate a constraint store and a call to the `check_constraints` goal (line 5), after each distinct sequence of `msw` applications.

If `check_constraints` fails during PRISM inference, then the corresponding PRISM derivation fails, and further extensions of this derivation will not be attempted since it does not constitute a valid run. In effect, inference by PRISM will only consider runs which are guaranteed not to violate any of the constraints declared for the model.

Declaration of constraints and implementation of constraint solvers are conceptually decoupled from the PRISM model. The declaration of side-constraints on the model is done by declaring facts of the form, `constraint(ConstraintSpec)`. The `ConstraintSpec` associates the constraint with a constraint checker implementation and may contain some parameters for this particular instance of the type of constraint.

A satisfiability checker maintains its own constraint store. A satisfiability checker for a particular type of constraint consists of an `init_constraint_store/2` rule and one or more `check_sat/4` rules. The `init_constraint_store/2` rule is used to

[3] The current implementation of the framework is available via http://akira.ruc.dk/~cth/chmm

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

create a starting point for the constraint store of each declared constraint and is of the form,

```
init_constraint_store(ConstraintSpec, InitialStore).
```

It is given `ConstraintSpec` and must unify `InitialStore` with an initial constraint store matching the `ConstraintSpec`. Additionally, one or more `check_sat` rules of the form,

```
check_sat(ConstraintSpec,StateUpdate,StoreBefore,StoreAfter):- ... .
```

must be implemented to check the satisfiability of the constraint.

As an example, consider an implementation of a `cardinality_atmost` constraint, called `cardinality` in our framework,

```
init_constraint_store(cardinality(_,_), 0).
check_sat(cardinality(U,Max), U, VisitsIn, VisitsOut) :-
        VisitsOut is VisitsIn + 1,VisitsOut =< Max.
check_sat(cardinality(X,_),U,S,S) :- X \= U.
```

Each time `check_constraints` is called from the PRISM model, the relevant `check_sat` goals are called for each declared constraint. If any of these fails, so will `check_constraints`. `StateUpdate` and `StoreBefore` are given and `check_constraints` is expected to unify `StoreAfter` to an updated constraint store. In our example HMM, the `StateUpdate` will consist of the `[State,Emit]` pattern given to `check_constraints`.

The call to this rule must only succeed if the constraint given by `ConstraintSpec` is not violated by the further information given by the `StateUpdate`. Constraints are checked incrementally and should only fail if any further updates to the constraint store can only lead to failure.

The constraint stores of individually declared constraints are automatically aggregated in the constraint store exposed to the PRISM model. Individual constraint checkers are unaware of each other and cannot access the individual constraint stores of other constraint checkers. The constraints are checked in the order they are declared, so this order should be optimized to do pruning as early as possible.

4.2.2 Efficient inference with a separate constraint store stack

The tabling mechanism in PRISM makes Viterbi computation and EM learning efficient, but when extra parameters such as the constraint store are introduced in the probabilistic goals, PRISM considers these as goals with distinct derivations and stores a tabled entry for each version of the goal. This behavior is undesired when the extra parameters are used only for internal bookkeeping. The effect of this excessive tabling is that the dynamic programming advantages are lost with exponential time inference as consequence.

In (Christiansen and Gallagher 2009) a related problem concerning tabling of annotations produced by running Viterbi on PRISM programs is approached using a program transformation that removes non-discriminating arguments, which do

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

not affect the control flow. The annotation can then be recovered from the program derivation of the transformed program.

This approach is not applicable for the constraint store argument because the constraint store implicitly affects control flow by limiting possible future derivation extensions. The constraint store has to be considered in the inference process; otherwise it would be possible to produce invalid derivation paths.

B-Prolog, on which PRISM is based, supports table modes, but this is not directly usable with probabilistic goals in PRISM. It is possible with these modes to declare an argument of a tabled goal as an *output argument*, which means that it will not be used as key in the table lookup, but will be unified with the value of the argument stored in a tabled goal. For our purpose, declaring the constraint store arguments as output arguments would not be feasible since different derivations of the same goal may have differing constraint stores and these determine possible derivation extensions.

To deal with the tabling problem we have introduced a separate constraint store stack, which avoids storing data locally in parameters of probabilistic goals by maintaining the constraint store with assert and retract. This stack is maintained in parallel to the derivation stack of Prolog. PRISM utilizes Prolog's backtracking to explore possible solutions, so the constraint store stack implementation is required to be able to restore a previous constraint store when PRISM encounters failures during inference and performs backtracking to find alternative solutions.

To utilize this functionality, the user should use the goal `check_constraints/1`, which omits the store arguments, rather than `check_constraints/3` as stated above. We then define `check_constraints/1` as

```
check_constraints(StateUpdate) :-
    get_store(StoreBefore),
    check_constraints(StateUpdate,StoreBefore,StoreAfter),
    forward_store(StoreAfter).
```

The new `check_constraints/1` make use of the goal `get_store/1` to retrieve the current version of the constraint store and `forward_store/1` is used to assert the updated store,

```
get_store(S) :- !, store(S).
forward_store(S) :- (asserta(store(S)) ; retract(store(S)),fail).
```

If a derivation fails, PRISM backtracks to the choice point in the `forward_store` rule and retract the most recently asserted store. Then, when exploring alternative derivation extensions, the previously asserted constraint store will be used as expected.

4.2.3 Complexity analysis of our implementation

Due to tabling, PRISM guarantees familiar best known complexity bounds of common inference tasks on a variety of the models that can be expressed in PRISM, which includes HMMs (Sato 2000). This implicitly limits the number of calls of

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

`check_constraints` to the same bound. The added complexity of doing constraint checking depends on incremental constraint checking cost of individual constraints checkers and the number of constraints expressed on the model.

Space complexity is influenced by table space usage and maximal length of a derivation at any given point. Since the asserted constraint store stack contains a constraint store fixpoint for each step of the current derivation, it is bounded by $O(n\max(|c|))$ where n is the length of the sequence and $\max(|c|)$ is the maximal size of the constraint store in any derivation step. Note that the space complexity of the separate constraint store stack is unaffected by time complexity and the number of states in the model. With more complex models like the pair HMM, the table space required for dynamic programming becomes the dominating concern.

5 Experimental validation

In this section, we validate our CHMM implementation with the pair HMM presented in section 2.2. The experiments were run on a computer with 16 2.4 GHz, 64 bit Intel Xeon(R) E7340 CPUs and 64 GB of memory. All of the experiments utilized only a single processor at a time.

Our experiments utilize implementations of some common constraints adapted for the CHMM framework: `cardinality(UpdatePatterns,Max)` ensures that entries from the list `UpdatePatterns` occurs at most `Max` times in the derivation sequence. `alldiff` ensures that all updates in a derivation are different; `lock_to_sequence(Seq)` ensures that the sequence of derivation updates is identical to the sequence represented by the list `Seq`; `lock_to_set(Set)` ensures that all updates belong to members of the list `Set`. The operator `forall_subseq(L,C)` applies the constraint C to every subsequence of length L in the derivation sequence and `for_range(From,To,C)` applies C only the range, `To-From`, both inclusive; `state_specific(C)` applies C only to the `State` part of the update.

5.1 Running time of constrained alignment

The addition of side-constraints to an HMM involves some computational overhead in order to check the satisfiability of the constraints, but may also reduce the number of possible solutions and therefore the amount of work required to find the optimal path. As a practical experiment to demonstrate this, we consider global alignment with the pair HMM discussed in section 2.2.

The overhead of integrating the constraint checking machinery in the model is demonstrated in the left part of Fig. 3, where sequences of increasing length are aligned. It can be observed that the running time penalty is a constant factor and that the polynomial time complexity of the pair HMM is preserved in our framework. Obviously, polynomial time inference presupposes incremental constraint checking to be a constant time operation, which may not be the case for certain types of constraints.

In the right part of Fig. 3, two sequences of equal length (32) are aligned, but with varying amounts of constraints being enforced. The global cardinality constraint

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

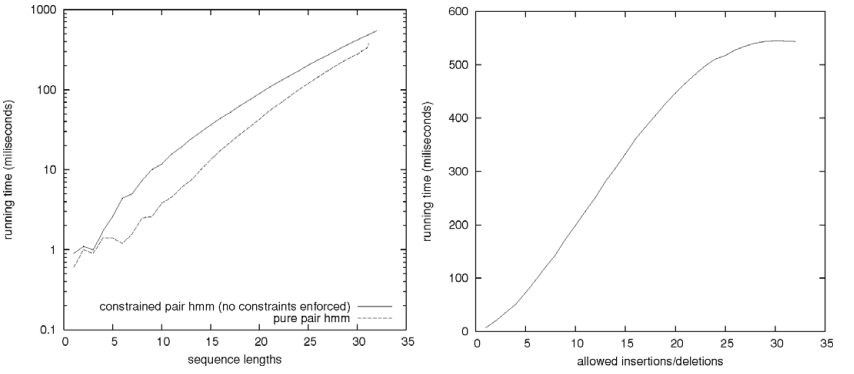


Fig. 3. Left: Running time of alignment with a pure pair HMM compared to alignment with a CHMM with no constraints enforced. Right: Running time of alignment of two sequences of length 32 with varying amounts of allowed insertions and deletions.

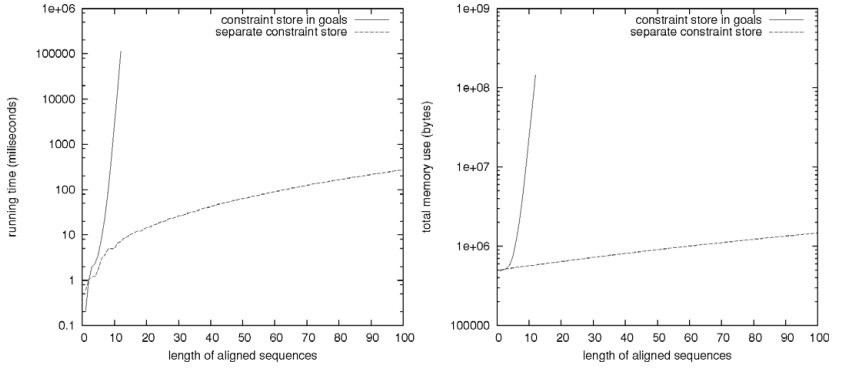


Fig. 4. A comparison of the running time (left) and memory usage (right) of constrained alignment of two sequences with tabled constraints versus a separate constraint store stack.

is used to enforce an upper limit, `L`, on the amount of inserts or deletes in the alignment,

```
constraint(state_specific(cardinality([insert,delete],L))).
```

By constraining the alignment (allowing fewer gaps), the space of viable solutions is reduced. The more constrained the alignment is, the more pruning opportunities arise. With a large amount of pruning opportunities, the running time is reduced quite significantly. Note that, since the imposed constraint is `state_specific`, the number of possible alignments, and hence running time, is unaffected by input sequence structure.

5.2 Efficiency of the separate constraint store stack

To verify the efficiency of our constraint store implementation, alignment with a local cardinality constraint was measured for different sizes of input sequences. From the measurements, which are reported in Fig. 4, it is apparent that our implementation does not incur the same exponential overhead as the naive implementation where the constraint store is maintained in the goals and hence tabled.

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

Table 1. *Running time and memory consumption for alignment with different kinds of constraints.*

Constraint	Sequence lengths	Running time (in ms)		Memory consumption (in kb)	
		in goals	separate	in goals	separate
cardinality([insert],20)	50	15460	3176	42296	5723
cardinality([insert],40)	50	29557	3968	93845	6703
for_range(1,50, lock_to_set([match]))	100	24649	4544	105498	7137
for_range(1,90, lock_to_set([match]))	100	20	48	1641	1198
for_range(1,50, lock_to_sequence ([match,..,match]))	100	24829	4544	1641	1198
for_range(1,90, lock_to_sequence ([match,..,match]))	100	20	48	105498	7137
alldiff	20	100442	28	85654	256
forall_subseqs(5,alldiff)	10	1664	12	60098	137

Running times and memory usage for a range of different constraints are reported in Table 1. For the sake for completeness, the table also includes running times for the version where the constraint store is tabled.

In most cases the separate constraint store performs better in terms of both running time and memory consumption. In the cases where performance is worse, it can be attributed to a very small number of possible derivations or constraints which rarely change the store.

6 Related work

The term "Constrained HMM" is used in (Roweis 1999; Landwehr *et al.* 2007) and refers to restrictions on the finite automaton associated with an HMM but not as constraints on HMM runs. In (Sato and Kameya 2008), CHMMs were introduced to exemplify an EM algorithm, suited for PRISM programs which allow the possibility of derivation failures. Our approach differs, as we augment PRISM programs with side-constraints and use constraint solving techniques to achieve efficient inference.

In (Riezler 1998), Riezler proposes techniques for inference in probabilistic constraint logic programming. In (Costa *et al.* 2008) relationships between elements of a Bayesian Network are expressed as a constraint logic program, which is similar to the way we define HMMs. However, our paper focus differs as we study the interest of checking satisfiability of side-constraints during inference.

In the natural language processing community, recent work on Constrained Conditional Models feature an approach similar to ours. Indeed, Constrained Conditional Models is a general framework that augments inference and learning of conditional models with declarative constraints (Chang *et al.* 2008). However, inference is expressed as an Integer Linear Programming problem (Roth and Yih 2005). In this context, more expressive constraints, such as `cardinality` or `all_different`, can not be added on an HMM run. Moreover, our PRISM-based

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

implementation allows us to define the HMM structure separately from the side-constraints and use advanced constraint solving techniques.

7 Conclusions

In this paper, we propose a framework to define HMMs with side-constraints as a Constraint Logic program extended by probabilistic choices. Constraint Logic Programming have advantages in terms of more compact expression of CHMMs. Inference computations are adapted for CHMMs and conditions for an efficient computation are described. An implementation based on PRISM is proposed and well-known constraints and operators have been demonstrated for defining CHMMs. Finally, we experimentally validate our approach with a constrained pair HMM used for biological sequence alignment.

As current work, we study how sampling and EM-learning can be adapted for our CHMM framework. Indeed, sampling turns out to be problematic in probabilistic models with a large probability of derivation failure. In (Sato *et al.* 2005), Sato *et al.* address the problem of EM-learning with PRISM programs that can fail and their methods are also applicable for our framework.

As further work, we plan to incorporate more advanced constraint solving techniques such as those used in Weighted CSP (Larrosa and Schiex 2004) in the framework. This approach would allow us to combine soft constraints solving and inference and express this as an optimization problem. We also plan to deal with the restriction that individual constraint checkers do not share information in our framework, so that we can benefit from some of the optimization techniques used by other constraint solvers. We are working on extending the library of constraints that can be defined as side-constraints.

Acknowledgement

We thank the anonymous reviewers for their interesting comments.

References

Chang, M.-W., Ratinov, L.-A., and Rizzolo, N. Roth, D. 2008. Learning and inference with constraints. In *Proc. of AAAI Conference on Artificial Intelligence*. Chicago, USA, 1513–1518.

Christiansen, H. and Gallagher, J. 2009. Non-discriminating arguments and their uses. In *Proc. of International Conference in Logic Programming*. Pasadena, USA, 55–69.

Christiansen, H., Have, C., Lassen, O., and Petit, M. 2009. A constraint model for constrained hidden markov model: A first biological application. In *Proc. of the International Workshop on Constraint Based Methods for Bioinformatics*. Lisbon, Portugal, 19–26.

Costa, V., Page, D., and Cussens, J. 2008. CLP(BN): Constraint logic programming for probabilistic knowledge. *Probabilistic Inductive Logic Programming LNAI 4911*, 156–188.

Durbin, R., Eddy, S., Krogh, A., and Mitchison, G. 1998. *Biological Sequence Analysis*. Cambridge University Press.

Landwehr, N., Mielikinen, T., Eronen, L., Toivonen, H., and Mannila, H. 2007. Constrained hidden markov models for population-based haplotyping. *BMC Bioinformatics 8,* S-2.

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press

Larrosa, J. and Schiex, T. 2004. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence 159,* 1–2, 1–26.

Rabiner, L. 1989. A tutorial on hidden markov models and selected applications in speech recognitation. *IEEE 77,* 2 (February), 257–286.

Riezler, S. 1998. *Probabilistic Constraint Logic Programming*. PhD thesis, University of Tübingen.

Roth, D. and Yih, W. 2005. Integer linear programming inference for conditional random fields. In *Proc. of the International Conference on Machine Learning*. Bonn, Germany, 737–744.

Roweis, S. 1999. Constraint hidden markov models. In *Proc. of the International Conference of Advances in Neural Information Processing System*. Denver, USA, 782–788.

Sato, T. 1995. A statistical learning method for logic programs with distribution semantics. In *Proc. of International Conference in Logic Programming*. Tokyo, Japan, 715–729.

Sato, T. 2000. A viterbi-like algorithm and em learning for statistical abduction. In *Proc. of the Workshop on Fusion of Domain Knowledge with Data for Decision Support*. Tokyo, Japan.

Sato, T., Kameya, T., and Zhou, N. 2005. Generative modeling with failure in PRISM. In *Proc. of International Joint Conference on Aritificial Intelligence*. Edinburgh, Scotland, 847–852.

Sato, T. and Kameya, Y. 1997. PRISM: A language for symbolic-statistical modeling. In *Proc. of the International Joint Conference of on Artificial Intellingence*. Nagoya, Japan, 1330–1335.

Sato, T. and Kameya, Y. 2008. New advances in logic-based probabilistic by PRISM. In *Probabilistic Inductive Logic Programming*. LNCS. Springer, 118–155.

Van Hentenryck, P., Saraswat, V., and Deville, Y. 1995. Design, implementation, and evaluation of the constraint language cc(fd). *Constraint Programming 910*, 293–316.

Viterbi, A. J. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory 13*, 260–269.

https://doi.org/10.1017/S1471068410000219 Published online by Cambridge University Press