Efficient Variational Inference in miniKanren with Weighted Model Counting

EVAN DONAHUE, University of Tokyo, Japan

We extend miniKanren with a collection of primitives for describing probabilistic generative models. We further describe modifications to the language's stream-based implementation that permit the efficient variational learning of such models via weighted model counting. We begin with a naive implementation that requires minimal changes to the core miniKanren implementation, and then describe two modifications to achieve practical levels of efficiency. The first alters the search to factorize conditionally independent conjuncts, avoiding unnecessary combinatorial explosion. The second modifies tabling to recover standard probabilistic dynamic programming algorithms such as Viterbi, forward-backward, and Baum-Welch. The end result is a simple extension to miniKanren that is nevertheless efficient enough to be of use in writing practical probabilistic relational programs.

ACM Reference Format:
Evan Donahue. 2022. Efficient Variational Inference in miniKanren with Weighted Model Counting. In . ACM, New York, NY, USA, Article 5, 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Probabilistic programming languages (PPLs) augment traditional programming languages with probabilistic constructs that allow for the economical expression of probabilistic models. Such languages enable the straightforward construction and use of models that are otherwise difficult to implement correctly. Probabilistic logic programming languages in particular permit the specification of models describing the kinds of complex, discrete structures that frequently admit concise representations in logic programming languages [18]. In this paper, we introduce a probabilistic extension of miniKanren, an embedded relational programming language, with the twin aims of making the implementation efficient enough for practical use yet simple enough to be easy to implement.

Even among logic programming languages, miniKanren has excelled in handling certain complex discrete structures—particularly programs—in the context of work on relational interpretation [5]. This efficacy is due in part to miniKanren's unique stream-based, interleaving search and pure relationality. As a result of miniKanren's unique position among logic programming languages, it is reasonable to conjecture that a probabilistic extension of miniKanren would likewise enable as yet unforeseen applications even among existing probabilistic logic programming languages [6, 20].

As a first step towards novel applications of probabilistic relational programming, this paper reproduces several classical algorithms pertaining to probabilistic graphical models including Gaussian mixture modeling, K-Means clustering, the Viterbi algorithm, and the Baum-Welch algorithm in order to demonstrate the utility of the proposed probabilistic extensions for solving practical and well understood probabilistic modeling tasks. To accomplish this, we describe a simple yet efficient probabilistic extension to miniKanren that employs variational inference via weighted model counting (WMC). Variational methods are fast, deterministic optimization-based methods

miniKanren 2022, September 11 2022, Ljubljana, Slovenia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

that, while offering weaker accuracy guarantees than sampling-based methods, have nevertheless proven effective in other PPLs [1, 14]. WMC describes a strategy for performing probabilistic inference that conforms well to the existing contours of miniKanren's standard stream-based implementation. Answer states returned by run correspond to models and each model can be assigned a weight using special probabilistic goals, making the implementation relatively simple.

This paper makes three main contributions: (1) a minimal extension to the miniKanren state data structure to allow for WMC, (2) an extension to miniKanren streams that efficiently factorizes conditionally independent conjuncts, allowing for the efficient expression of models such as Gaussian mixture models and K-Means, and (3) an extension to tabling that recovers analogues of the standard dynamic programming algorithms for hidden Markov models (HMMs) and probabilistic context-free grammars (PCFGs) such as the Viterbi, Baum-Welch, forward-backward, and inside-outside algorithms.

The remainder of this paper is divided into six sections. Section 2 describes related work in probabilistic logic programming and related areas. Section 3 offers an overview of the proposed extensions. Section 4 describes the implementation details of each successive extension. Section 5 describes an empirical evaluation of the proposed extensions. Section 6 presents a discussion of the results and offers suggestions for future work.

The probabilistic extensions described in this paper are part of the fully-featured SmallKanren implementation of miniKanren in Smalltalk, which is available through the author's website. All examples in this paper are written in Scheme for greatest legibility to the miniKanren community, however, these should be regarded as pseudocode transliterations of the original Pharo Smalltalk implementation.

2 RELATED WORK

Even setting aside the wide range of contemporary research on PPLs more generally, the subfield of probabilistic logic programming alone has a long history [15, 16, 22]. Several lines of research in particular bear on the results discussed in this paper. PRISM [20], developed an approach to aggregate tabling for machine learning [25] in a fashion analogous to the aggregate streams described in Section 4.3, with differences to be discussed in that section. DICE [10], while not a logic programming language, pioneered one of the primary optimization techniques used here to factorize inference in conditionally independent conjuncts. Early versions of Dyna [9] placed an emphasis on recovering many of the same dynamic programming algorithms discussed here using an extension of Datalog. Finally, the goals of making variational inference easy to access extend beyond PPLs per se to frameworks such as Pomegranate [21], although the implementation details differ correspondingly.

Within work on miniKanren specifically, the most natural comparison is with probKanren [26], which offers a comparable syntax for specifying, at least at present, similar classes of probabilistic models. probKanren differs primarily in its implementation, which is based on a sequential Monte Carlo sampling strategy that is slower but potentially more accurate. As performance is not a stated goal of probKanren, it does not prioritize integration with tabling or achieve the exponential speedups that are the focus of this paper, although it does manage a comparably simple implementation without pushing the burden of explicitly managing the model parameters outside of miniKanren onto the user, as is done in this paper to keep the implementation simple.

Beyond probKanren, Zhang et al. [24] demonstrates an alternative configuration of miniKanren search and probabilistic computation by learning from the language's internal representations a heuristic function to guide the search behavior. Although the goal of specifying probabilistic models differs, Section 6 discusses some potential confluences between these two lines of work. The miniKanren implementation described here corresponds most closely to that described in Byrd

[4], particularly including its implementation of tabling, although see Section 4.4 for a discussion of some differences.

3 OVERVIEW OF PROBABILISTIC EXTENSIONS

This section offers an overview of the probabilistic extensions to miniKanren, the implementations of which will be described in Section 4. We begin by describing the syntactic extensions needed to express probabilistic models. We then describe three common types of queries and two inference strategies for answering those queries that we have implemented in miniKanren.

3.1 Modeling

We extend the language with two new features: a collection of primitives representing exponential family probability distributions such as Bernoulli, categorical, and normal, and the observe goal for associating data with the distributions defined by those primitives. For the purposes of exposition, we assume the existence of a run macro analogous to the usual run$*$, but which returns the state data structure that contains the substitution, constraints, and associated data, rather than reified query variables. This will make it easier to define operations other than reification on the answer stream that are necessary for probabilistic analysis.

The following example, which calculates the likelihood of a single toss of a fair coin coming up heads, will clarify the relationship between these two language features:

```
(define-values (coin heads) (values (bernoulli .5) 1))
(run () (observe coin heads))
```

In this example, (**bernoulli** 0.5) defines a Bernoulli distribution with $p = 0.5$, representing the outcome of a fair coin toss. **observe** associates a single observation of a "heads"[1] with the distribution representing the coin presumed to have generated that outcome. Running this program will return a single state with an associated likelihood score of 0.5, representing the likelihood of a fair coin yielding a heads. In the remainder of this paper, we will refer to the collection of distributions defined and used in a given miniKanren program (in this case only the Bernoulli distribution labeled *coin*) as the "model parameters." Using these parameters to evaluate data and using data to optimize these parameters will be the primary objectives of inference.

Conjoined **observe** statements represent successive events, and any given state, interpreted as a conjunction of some subset of the overall program's goals, consequently represents a joint distribution over the **observe** goals it conjoins.

```
(observe coin heads)
(observe coin heads)
```

This program fragment represents the successive observation of two flips of the same fair coin defined above, with each observation coming up heads. The current implementation adopts the mean-field assumption that each observation is conditionally independent of the others. It therefore calculates the likelihood of the above conjunction as the product of the likelihoods of each individual observation, or $0.5 \times 0.5 = 0.25$. Observing a 0 probability event causes the implementation to fail early in the corresponding branch, as a 0 probability state can have no influence on subsequent calculations in other branches of the search.

Disjunction, by contrast, represents a probabilistic choice, the semantics of which depend on the inference strategy employed. Consider the following program representing a coin flip that determines from which of two normal distributions from which to observe a real-valued data point:

[1]Bernoulli distributions in this implementation produce 1 or 0, which can be considered "heads" and "tails" by convention.

```
(define-values (tails datapoint n0 n1)
  (values 0 0 (normal -1 1) (normal 1 1)))
(run (z)
  (observe coin z)
  (conde
    [(== z tails) (observe n0 datapoint)]
    [(== z heads) (observe n1 datapoint)]))
```

As will be discussed in Section 4, disjunction may be interpreted either as the summation of each disjoined branch or as the selection of the single branch with the maximum likelihood. In order for either interpretation to be fully well defined, however, it is necessary that, for a **conde** clause that branches on an observed variable, there must be exactly one disjunct for each element of the support of the distribution associated with that variable. In this case, because the support of a Bernoulli distribution consists of 0 and 1, there must be two branches in the **conde** clause corresponding to tails and heads, respectively. This constraint ensures that each **conde** that returns multiple answers can be viewed as observing an event generated by the distribution in question and making a probabilistic decision based on the outcome. **conde** clauses that return a single answer due to mutually exclusive branches, such as those that iterate through a fully ground list, need not have probabilistic choices associated with them.

Intuitively, a probabilistic miniKanren program that obeys this constraint on **conde** can be viewed in terms of a "generative story." In the same sense that base miniKanren can be viewed as "generating" discrete structures that satisfy the constraints described in a given program, probabilistic miniKanren can be viewed as generating a dataset and associated latent variables subject to the constraints of the actually observed data and the probabilistic model. The above program describes a Gaussian mixture model or equivalently K-Means model used for clustering real-valued data points into two clusters, each corresponding to one of the normal distributions. For each datapoint, a coin is flipped and, based on the result, one of two normal distributions is sampled, conceptually "generating" a dataset of samples from the pair of normal distributions along with the latent variable representing the decision of from which distribution each datapoint arose. If the above goal is evaluated for each datapoint, and the results are conjoined, the likelihood of the final program corresponds to the likelihood that a given coin and pair of normal distributions generated the observed dataset. By varying the query used, the same program can serve to compute a likelihood, calculate missing values, or optimize the model parameters, as discussed in more detail in the next section.

3.2 Querying

Given a model that includes the model parameters and the program relating them to data via the **observe** interface, there are many different possible queries one could ask of the model. Loosely following the framework outlined in Rabiner [17], this paper will concern itself, for expository purposes, with three basic types of queries—likelihood, missing values, and training—and two inference strategies—argmax and summation.

The likelihood query returns a single score representing the likelihood of the data given the current model parameters. If z were bound to *heads* in the above example, a likelihood query would return the likelihood that *datapoint* was generated by the normal distribution associated with an observation of heads. If z were free, a likelihood query would either return the likelihood of the most likely category to which the datapoint belongs, corresponding to the most likely assignment of z and its corresponding branch selection in the **conde**, or the sum of the likelihoods of both possibilities, depending on whether the argmax or summation strategy was used. These strategies

correspond to the "hard" and "soft" assignments that differentiate a Gaussian mixture model from the related K-means clustering algorithm. Such a query can be useful in determining whether a particular datapoint is well explained by the model.

The missing value query attempts to fill in unbound logic variables with the most probable variable assignments. In the case of the Gaussian mixture model, this corresponds to determining the cluster z to which a given datapoint most likely belongs. Under the argmax strategy, this assignment corresponds to the binding in the most likely answer. Under the summation strategy, this assignment corresponds to the overall most likely assignment for each variable as determined by summing up its total likelihood across all possible answers. In the case of this GMM, because the z variable for a single datapoint is only observed once (each datapoint in a full implementation would be associated with its own z), these strategies produce the same answer. Note that, due to miniKanren's relationality, it is possible to leave arbitrary logic variables free in the usual way, and answer a variety of different probabilistic queries. Although this paper will describe the usual directionalities of these models, any permutation of ground and free variables can be computed.

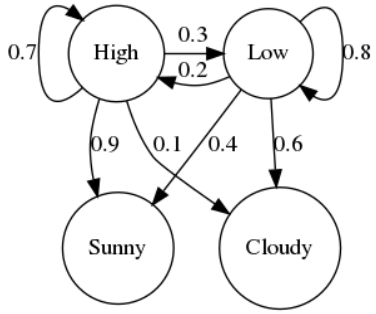
The training query is an iterative query, rerunning the miniKanren program repeatedly and attempting to find the locally optimal values for the model parameters by tuning them to maximize the likelihood of a given dataset. Under the argmax strategy, this query corresponds to tuning the model parameters to optimize the likelihood of the most likely variable assignment, corresponding to a "hard EM" strategy, while the summation strategy corresponds to the "soft EM" strategy of tuning the model parameters to maximize the average likelihood of all possible variable assignments.

4 IMPLEMENTATION

In this section, we describe the implementation of the probabilistic primitives introduced in Section 3. We begin with a simple and fully relational but inefficient implementation. We then describe two optimizations that, while not relational, taken together, recover the standard efficient dynamic programming solutions to optimizing probabilistic models in a variational framework as well as enable the efficient querying of several other types of models. We use a simple HMM for weather forecasting as the running example in most of this section to motivate and illustrate the implementation details.

4.1 Hidden Markov Model for Weather Forecasting

A hidden Markov model, or HMM, is a probabilistic model that describes a sequence of unobserved states, each of which produces an observable phenomenon. In this instance, the sequence of states is that of the daily barometric pressure, which cannot be observed directly without a barometer, and the observations are sunny or cloudy days generated in part by the barometric pressure. On any given day, the barometric pressure may be high or low and the weather may be sunny or cloudy. High pressure systems are more likely to yield sunny days, while low pressure systems are more likely to indicate cloud cover and storms. Given a barometer, it is possible to measure the barometric pressure and on that basis predict whether it will be a sunny or a cloudy day. However, lacking a barometer, it is still possible to observe the weather of the past several days and estimate the likelihood of being in the middle of a high or low pressure system, using that estimate as the basis for a prediction of the day's weather. In particular, assume the true weather system conforms to the following model:



During a high pressure system, the following day will remain within the high pressure system with a probability of 0.7, and will transition to a low pressure system with probability 0.3. Conversely, a low pressure system will remain low pressure with probability 0.8, and transition back into a high pressure system with probability 0.2. During a high pressure system, there is a 0.9 probability of emitting sun and a 0.1 probability of emitting cloud cover. During a low pressure system, there is a 0.6 probability of cloud cover and 0.4 of sun.

We can represent this weather system over the course of three days, on which the observed weather was sunny, cloudy, and cloudy, with the following program:

```
(define-values (low cloudy high sunny) (values 0 0 1 1))
(define-values (high-t low-t) ;Transition distributions
  (values (bernoulli 0.7) (bernoulli 0.2)))
(define-values (high-e low-e) ;Emission distributions
  (values (bernoulli 0.9) (bernoulli 0.4)))
(define observations (list sunny cloudy cloudy)) ;Data

;prev-t - Transition distribution of previous timestep
;states - List of states
;observations - List of observations
(define (hmm prev-t states observations)   ;Given the transition,
  (conde                                   ;states, and observations,
    [(== states '())                       ;either we have reached the
     (== observations '())]                ;end of the observations,
    [(fresh (curr-s rest-s curr-o rest-o)
       (== states (cons curr-s rest-s))    ;or we take the latest state
       (== observations
           (cons curr-o rest-o))           ;and observation, compute the
       (observe prev-t curr-s)             ;transition probability,
       (conde                              ;and randomly choose
         [(== curr-s low)                  ;a low
          (observe low-e curr-o)
          (hmm low-t rest-s rest-o)]
         [(== curr-state high)             ;or high state from which to
          (observe high-e curr-o)          ;observe an emission
          (hmm high-t rest-s rest-o)])])])) ;and transition to next.

(run (states)
  (hmm high-t states observations)) ;Start in high state for simplicity
```

Listing 1. Weather forecasting HMM

At each state, the HMM observes the current state given the transition probability distribution corresponding to the previous state, observes an emission depending on the observed current state, and transitions to the subsequent state using the transition distribution associated with the current state. For simplicity, it is assumed that observations begin in the high pressure state, hence *high-t* is passed in as the first *prev-t*, representing the previous state's transition distribution.[2]

In the following sections, we describe a naive implementation of the inference procedures as well as two optimizations involving aggregating substreams and tabling.

4.2 Naive Implementation

4.2.1 Likelihood & Missing Values. The simplest aspect of the system to implement is the likelihood calculation, which supplies the information necessary to answer both the likelihood and missing values queries with light post-processing of the normal miniKanren answer stream. The implementation will require extending the state, the central data structure containing the substitution and constraint store, with a new term representing the likelihood score, which is set to 1 by default in a new state:

```
(substitution constraints likelihood)
```

Consider the likelihood of observing the sequence sunny, cloudy, cloudy, on three consecutive days during which the barometric readings were high, low, and low, respectively. The likelihood of this event would be the product of each transition and corresponding emission, or $0.7 \times 0.9 \times 0.3 \times 0.6 \times 0.8 \times 0.6 = 0.054432$. In order to compute this, each **observe** simply multiplies[3] the state's current likelihood by the likelihood of the observation given the corresponding distribution:

```
(define observe (distribution observation)
  (lambda (s)
    (set-likelihood s (* (likelihood distribution observation)
                         (get-likelihood s)))))
```

Each state returned by **run** will therefore possess a likelihood that is the product of the likelihoods of every **observe** statement encountered in the path that state traveled through the search tree.

Observations that are ground or are already bound in the substitution can be handled directly by computing the likelihood of the value given the model parameters associated with the relevant distribution. In order to make the **observe** statement relational in the case where the observed variable is free, a simple constraint can be added to the store that waits for the variable to be unified and multiplies the likelihood appropriately at that time.[4] The behavior of a state that returns from **run** with unbound but observed variables still in the constraint store is undefined for the purposes of this paper, although one could imagine a number of ways of interpreting such a constraint, including dropping it or computing its expected value, depending on the nature of the query.

Using only this extension, it is possible to handle the first four queries in the naive implementation: likelihood and missing values under both argmax and summation strategies. It is first worth noting, however, that if all states and observations are ground, then argmax and summation yield the same result, as only one path through the search tree has a nonzero likelihood. If, however, only the

[2] A more general approach might be to define a separate distribution over start states and set it equal to the stationary distribution of the current model, which describes the likelihood of being in a given state at any given time based on the current transition probabilities.

[3] Code examples in this paper multiply probabilities to most closely correspond to the mathematical identities described, although in practice a real implementation should work in logarithmic space using logarithmic identities and the log-sum-exp trick, as is common in probabilistic applications.

[4] For distributions over complex terms that may contain additional free variables, it will be necessary to suspend until all variables are bound.

observations sunny, cloudy, cloudy, are observed, and not the barometric states, it is necessary to take into account all possible assignments to the state variables in computing the likelihood.

Under the argmax strategy, the likelihood of the observation sequence is defined as the likelihood of the most probable single assignment of state variables. To compute the likelihood of the most probable variable assignment, simply fold the stream of answers returned by **run**, retaining the single answer with the maximum likelihood:

```
(reduce max (map get-likelihood (run ...)))
```

The likelihood of this answer is the likelihood of the data as a whole under this strategy.

Under the summation strategy, the likelihood of the observation sequence is the sum of the likelihoods of that sequence under all possible assignments to the state variables. To compute this likelihood, simply sum the likelihoods of all states returned by **run**:

```
(reduce + (map get-likelihood (run ...)))
```

For missing values, the argmax strategy defines the expected value of the state variables to be their assignment in the state with the highest likelihood. To compute this, simply fold the stream of answers and return the highest likelihood state:

```
(reduce (lambda (a b) (if (< (get-likelihood b) (get-likelihood a)) a b))
        (run ...))
```

The summation strategy has an unusual interpretation from the perspective of the standard semantics of miniKanren, corresponding in the HMM to the posterior decoding. This strategy defines the value of each state variable as the value with the greatest total likelihood across all possible states. To compute this, for each state variable and for each potential value it attains in any state, sum the likelihoods of the states in which that variable took that value and assign the variable to the value with the highest resulting likelihood. Free logic variables make the interpretation of this strategy particularly tricky. Note too that because this assignment is defined individually per variable, the resulting assignments may in the general case not match the assignments in any single returned state, and therefore may not in fact be a valid solution to the constraints of the miniKanren program. In this simple example, however, the assignments will be valid since no set of assignments can fail, they just may not match the argmax solution of the most likely joint assignment.

4.2.2 Training. The only queries remaining for the naive implementation are those for optimizing the model parameters given a set of training data. For these queries, in addition to the likelihood computed above, it will also be necessary to keep track of the count of the number of times each distribution was observed and the values observed from it. We extend the state with another term representing an association list relating each distribution to two objects, a vector of running counts of sufficient statistics associated with observations of the distribution and the current estimates for the optimized model parameters. Note that for streaming large amounts of data through a miniKanren implementation, an extension that avoids needing to bind every cell of a list in the substitution in order to destructure it, such as Donahue [8] or Ballantyne [2], will be useful to avoid an unbounded memory leak when iterating over data points.

```
(substitution constraints likelihood
              ((distribution . (counts . parameter-estimates)) ...))
```

distribution is an identifier corresponding to a specific instantiation of a distribution using the **bernoulli**, **normal**, or other constructors. This can be a globally unique integer, or perhaps the unique object identity of a new object in systems that support this.

counts represent a summary of the values observed in association with the given distribution that will be used in generating new estimates for the model parameters. Each exponential family

distribution has the property that all of the information necessary to estimate its parameters can be summarized by a weighted sum of statistics derived from the observed values associated with it. In the case of the Bernoulli distribution, it is enough to know how many heads and tails were observed in order to estimate the probability of the coin. In this case, those outcomes correspond to how many times each state transitioned into each successive state and how many times each type of weather was observed in each state. The form of the counts will be specific to each type of distribution.

parameter–estimates is a record containing the current best estimates of the parameter values corresponding to a particular distribution, such as the weight of the coin corresponding to a Bernoulli distribution. The expectation maximization algorithm used to estimate model parameters is iterative, and will incrementally update the model parameters associated with each distribution on every iteration. For example, beginning with a fair coin distribution of (**bernoulli** 0.5) for *parameter–estimates*, if 6 heads and 4 tails are observed, *parameter–estimates* will become (**bernoulli** *0.6*), corresponding to the newly maximized probability estimate. In order to find the final estimate for each state transition and emission distribution, the programmer can look up the distribution identifier in this association list and read out the model parameters of its optimized estimate.

As with the previous calculations, if the states are observed directly in the data using a barometer, then the argmax and summation strategies converge. This is the fully supervised learning case, and in the example above a single state with the following counts is returned:

```
('high−t . (<1 1> . ('bernoulli .7)))
('low−t . (<1 0> . ('bernoulli .2)))
('high−e . (<0 1> . ('bernoulli .9)))
('low−e . (<2 1> . ('bernoulli .4)))
```

The vectors expressed in angle brackets correspond to the number of tails and then heads observed in association with each distribution. Applying a maximum likelihood updating strategy, each count is multiplied[5] by the state's likelihood of 0.054432 and then the updating equation specific to the distribution type—$\frac{heads}{heads+tails}$ in the case of the Bernoulli distribution—is used to compute the new probability estimates for *parameter–estimates*,. Counts are zeroed before each new iteration, although this only matters for unsupervised cases that require more than a single iteration to converge and that incrementally weight the counts with weights that change on each iteration, as will be described in Section 4.3:

```
('high−t . (<0 0> . ('bernoulli .5)))
('low−t . (<0 0> . ('bernoulli 1)))
('high−e . (<0 0> . ('bernoulli 1)))
('low−e . (<0 0> . ('bernoulli 1)))
```

The more complex case is that of unsupervised learning, in which only the weather is observed but not the barometric readings. Given a model that still assumes two barometric states, the task then is to learn the transition probabilities and the conditional emission probabilities for each state. This task can be approached with the expectation maximization (EM) algorithm [7]. EM alternates between two steps, the expectation step (E-step), in which it fills in the most likely values for the missing barometric states given the current model parameters, and the maximization step (M-step), in which it maximizes those parameters as in the supervised case. This scheme is guaranteed to converge to a local optimum in the parameter space, although it may not find the globally best parameter values to explain the data.

[5]In the fully supervised case, multiplication by the state's likelihood is unnecessary as it simply cancels out. This step will be relevant, however, when combining multiple states with different likelihoods, as in the unsupervised case.

Each E-step performs the counting procedure described in the supervised case over all possible combinations of values for the unbound state variables using weighted model counting. Each state returned by **run** constitutes a model possessing a weight in the form of its likelihood and counts in the form of the sufficient statistics associated with each distribution. The counts of each state are scaled within their states via multiplication by the state's likelihood to yield *weighted counts* and are then combined across states using either the argmax or summation strategy to yield the final expected values, which are then used to compute the new maximum likelihood estimates:

```
(reduce max (map (lambda (s) (scale-counts (get-counts s)
                                           (get-likelihood s))) (run ...)))
(reduce + (map (lambda (s) (scale-counts (get-counts s)
                                         (get-likelihood s))) (run ...)))
```

Definition 4.1 (Weighted Count). The *weighted count C* of a sufficient statistic associated with a given distribution is equal to the sum of the individual counts c_i accumulated across all observations (c_i is 0 when the observation is not associated with that distribution, otherwise a value defined by the specific distribution) multiplied by the product of likelihoods ℓ_i accumulated across all observations of any distribution within conjuncts associated with a given final answer state:

$$C = \prod_{i=0}^{N} \ell_i \sum_{j=0}^{N} c_j$$

Where N is the total number of conjoined observations encountered by a given state on its path through the search tree.

The argmax aggregation strategy, in which only the counts corresponding to the most likely state are used to maximize parameter values, represents so called "hard" EM. The summation strategy, in which the scaled counts are summed and the total final count is used for maximization, corresponds to "soft" EM. The maximization step, given the predicted expected values computed under either strategy, is exactly as described in the supervised case. Because EM is an iterative algorithm, it is useful to define a *run−train* interface that executes **run** repeatedly, each time with new parameter values, until a specified number of iterations have been performed or until the total likelihood of the stream ceases to improve more than a small epsilon value.

4.3 Aggregating Conditionally Independent Streams

The naive method described above is complete, correct, and fully relational. Its only drawback is that it is too inefficient for practical use. Consider the case of performing unsupervised EM on a large dataset of weather sequences. For each observed sequence, EM must search over all values for every state in the sequence. In the case of the 3-length observation sequence in the above example, each expectation step produces an answer stream of $2^3 = 8$ possible assignments to the state variables. Because each datapoint in the dataset when learning over an entire dataset is conjoined, each of these 8 states would have to be conjoined with every other state produced by every other training example, leading to a combinatorial explosion. Conjunction in miniKanren operates like a multiplicative operation on answer stream lengths, assuming no failures. A dataset with three 3-length Markov chains and two states therefore yields $8^3 = 512$ states over which to search.

It is possible to avoid this combinatorial explosion using the strategy outlined in Holtzen et al. [10], which observes that conditionally independent events can be handled separately, as they have no bearing on one another's optimal assignments. Because the training examples are presumed to be independent, it is therefore possible to calculate counts and likelihoods individually for each

datapoint, resulting in only a single state for each and reducing the exponential problem to a linear one. In the above example, this reduces to searching effectively $3 * 8 = 24$ states. In the work cited, conditional independence is automatically deduced from the structure of the program, in particular, values passed to a function must necessarily be conditionally independent of values outside their present scope, and so it is possible to aggregate the results of a probabilistic function independent of the behavior of the rest of the program around its call site.

The miniKanren context is slightly more complicated given that, if a fresh logic variable is passed to a relation, it is still possible for subsequent unifications later in the program to affect the results returned by the relation. Consequently, the conditional independence of the values passed to a relation from their calling contexts must be guaranteed by programmer discipline—points at which the program is to assume conditional independence must be annotated by the programmer, and the programmer must subsequently refrain from constraining values passed to conditionally independent goals after those goals have run. Two additional goals make this annotation fairly intuitive.

The present implementation supplies the programmer with two forms, **argmax** and **marginalize**, that effectively annotate a given relation as conditionally independent of its conjuncts while also allowing the programmer to specify the inference strategy to be used and offering programmer conveniences that make the use of such forms more concise than their unannotated equivalents. Each form wraps a collection of goals and produces a stream that aggregates answers produced by its subgoals, returning only a single state representing the maximum likelihood state or summation of states respectively. Both forms accept a discrete distribution, a variable to which to bind its expected observation, and a function that will be called with every element in that distribution's support.[6] Consider lines 19-26 of the above HMM rewritten using the **argmax** form. Sections which have been changed are highlighted:

```
(argmax prev-t curr-s
        (lambda (state)
          (conde
            [(== state low) (observe low-e curr-o)
             (hmm low-t rest-s rest-o)]
            [(== state high) (observe high-e curr-o)
             (hmm high-t rest-s rest-o)])))
```

Listing 2. Argmax implementation of weather forecasting HMM, lines 19-26

Conceptually, this form is equivalent to a **conde** over all possible values of *prev–t*, in this case 0 and 1, in which each branch consists of an **observe**, a unification of the value with *curr–s*, and an invocation of the subgoal returned by the **lambda**, which will be whichever branch of the inner **conde** does not fail when supplied with the given value passed in as *state*. To make the implementation efficient and allow it to exploit conditional independence, this form produces one of two streams, depending on whether **argmax** or **marginalize** is used:

4.3.1 Argmax. The argmax stream consists of a state, initially null, and a stream.

```
(state stream)
```

[6] If the expected value of the observation is not of interest, such as in the case of a nuisance variable, the logic variable argument can be made optional with the benefit of a slight performance increase from not having to unify a variable in the substitution for each value in the support.

stream corresponds to the disjunction of streams returned by the **lambda** form called with each value of the distribution's support. Each time the **argmax** stream is advanced, much like streams produced by *bind*, its internal *stream* is advanced by one step. If *stream* produces an answer state, the likelihood of that state is compared against the likelihood of *state* (or 0 in the case of null). If the likelihood is greater, *state* is replaced with the new state. Once the sub-stream is exhausted, the **argmax** stream returns the maximum likelihood state produced by the stream.[7] By returning only a single aggregate state, this stream avoids the combinatorial explosion that arises from the multiplicative properties of stream conjunction. Moreover, because the returned state is the highest likelihood state and the **argmax** form is presumed to be conditionally independent of subsequent goals, the logic variable will implicitly be bound to the highest likelihood element of the distribution's support.

4.3.2 Marginalize. The summation stream is slightly more complex by virtue of the need to combine counts and likelihoods from separate states. This will require a stream structure that separately tracks the substitution, constraint, and individual state likelihood information apart from the aggregated counts and likelihood:

```
(substitution constraints likelihood
             total-counts total-likelihood stream)
```

The basic outline of the implementation is that, as each new state is produced by *stream*, its likelihood is compared with *likelihood* and, if greater, *substitution* , *constraints* , and *likelihood* are replaced by those from the new state. The state's likelihood and counts are added (not multiplied) to *total–counts* and *total–likelihood* . Once *stream* is exhausted, *substitution* , *constraints* , and the aggregate values are packaged into a new state and returned.

Importantly, however, in order to correctly compute the summed counts, we must make a modification to the way in which we accumulate the sufficient statistics. The final weighted sum of the entire program is defined in terms of the counts of each final answer state scaled by its respective likelihood and summed with the weighted counts of other states as outlined in Definition 4.1. If we follow the above implementation naively, we will lose the association between an aggregated state's counts and its likelihood, which must be used at the end of **run** to scale those counts. Moreover, we cannot attempt the scaling at this point because doing so would overcount the likelihoods accumulated up to this point every time the counts were scaled by future likelihoods, which are a product of current state likelihoods and subsequent **observe** likelihoods.

We solve this problem by modifying **observe** to incrementally reweight our counts so that our counts are always already weighted. **marginalize** and **run** may then simply sum them directly, as in the implementation described above, without any additional weighting computations.

Definition 4.2 (Incremental Weighted Count). We define the *incremental weighted count* C_M after M observations, where $M \leq N$, the total number of observations, as:

$$C_M = \prod_{i=0}^{M} \ell_i \sum_{j=0}^{M} c_j$$

Theorem 4.3. *For incremental weighted count C_M, cumulative product of likelihoods $\mathscr{L}_M$, and count and likelihood c_{M+1} and ℓ_{M+1} corresponding to observation $M+1$, we compute $\mathscr{L}_{M+1}$ and C_{M+1} incrementally as follows:*

[7] This assumes a finite stream. Infinite streams could in theory be handled in a variety of ways, including truncation or periodic production of the currently maximal state, but in general infinite streams will require a more involved approach to fully account for their probabilistic semantics that is beyond the scope of the current paper.

$$\mathscr{L}_{M+1} = \mathscr{L}_M \ell_{M+1}$$
$$C_{M+1} = c_{M+1}\mathscr{L}_{M+1} + \ell_{M+1} C_M$$

The likelihood $\mathscr{L}_{M+1}$ corresponds to a product of the likelihoods of each **`observe`** statement encountered so far. The incremental weighted count C_{M+1} corresponds to all past counts C_M multiplied by the current likelihood ℓ_{M+1} plus the product of all past likelihoods $\mathscr{L}_M$ and the current count c_{M+1}. This arrangement guarantees that each count is multiplied by each likelihood only once, avoiding the double counting issue of the naive approach. See Appendix A.1 for a complete proof.

Corollary 4.4. *The incremental weighted count (Definition 4.2) incrementally computes the weighted count C (Definition 4.1) using the strategy outlined in Theorem 4.3 when $N = M$.*

Theorem 4.3 asserts that the incremental weighted counts are equivalent to the original weighted count calculation, all else being equal. All that remains is to prove that these incremental counts can be summed prematurely by the **`marginalize`** form without changing the final calculation:

Theorem 4.5. *Given a stream of N states i with likelihoods $\mathscr{L}_i$ and incrementally weighted counts C_i, combining these states into a single state inside* **`marginalize`** *with likelihood $\mathscr{L}$ and weighted count C according to the following update equations is equivalent to combining the states at the end of* **`run`***:*

$$\mathscr{L} = \sum_{i=1}^{N} \mathscr{L}_i$$
$$C = \sum_{i=1}^{N} C_i$$

See Appendix A.2 for a proof.

While this incremental approach resolves the issue with counts and likelihoods, there is still the question of the substitution to return, given that **`marginalize`** must merge a number of incompatible substitutions. The approach described above, which returns the substitution associated with the highest likelihood state, is one of several options. Its primary virtue is that it is easy to compute and has a coherent probabilistic interpretation given certain assumptions. Namely, if each branch of the **`marginalize`** returns only one state which itself was produced by a recursive call to **`marginalize`**, as is the case in the HMM implementation above and other similar recursive models, then selecting the substitution associated with the highest likelihood branch can be interpreted as selecting the value of the distribution passed to **`marginalize`** that maximizes the probability of all possible future assignments to all free variables. This corresponds to the assignment that maximizes the backwards probability in the context of the forward-backward algorithm.

We do not have access, at this point, to the relevant forward probabilities with which to compute the posterior decoding of the HMM, which would be a more typical variable assignment corresponding to the summation strategy, and if the branches return more than one state or a state that cannot be recursively interpreted as a summation, then the interpretation of this selection becomes unclear. All we can say is that this selection gives the programmer the opportunity to construct a program with a meaningful interpretation of its substitution. In any case, for likelihood and training queries, this decision is irrelevant as only the likelihoods and counts are required.

4.4 Dynamic Programming with Tabling

This section introduces a modification to tabling that allows tabled relations to maintain correct weights and counts, implicitly recovering the standard dynamic programming algorithms for common generative models, such as the Viterbi, forward-backward, and Baum-Welch algorithms in the case of HMMs. The aggregate streams described in Section 4.3 solve the problem of unnecessary combinatorial explosion due to a failure to exploit conditional independence. However, that is not the only source of combinatorial complexity.

In the naive implementation of HMM decoding, each position in the sequence is evaluated as high or low in relation to all possible combinations of all other positions, yielding an algorithm that is exponential in the number of observations. However, because each state is independent of all past states given the immediately previous state, it is possible to cache only the optimal value of each state for each possible transition from values of the previous states, leading to a family of dynamic programming solutions that avoids the expected combinatorial explosion.

Tabling, which generalizes memoization for logic programs, allows the straightforward implementation of these dynamic programming algorithms simply by tabling the *hmm* relation when implemented with **`argmax`** or **`marginalize`**. Two modifications to tabling as implemented in Byrd [4] are required to properly account for the weights and counts of tabled answers and to ensure termination.[8]

4.4.1 Accounting for Weights and Counts. To begin with, in order to make tabling mathematically compatible with WMC, the table must be extended to store not only answers but also counts and likelihoods as well, so that these can be returned from tabled relations along with the answer terms. Because a given tabled relation will be called from potentially many parts of the search tree with the same arguments but with different counts and likelihoods, the table must preserve the counts and aggregate likelihood of only the **`observe`** statements encountered within the tabled relation. Each consumer of the tabled relation must then combine its counts and likelihood with the cached values of the producer.

To achieve this tabling strategy, calls to tabled relations pass only the substitution and constraint store to the relation, resetting the state's counts and likelihood to 0 and 1 respectively. When the relation returns an answer, its counts and likelihood will therefore only reflect those encountered within the tabled goal. The producer and consumer streams generated by the *table* form preserve the counts and likelihoods of their calling states, combining them with their counterparts returned from the tabled relation with a weighted sum similar to the incremental counting of observations.

Theorem 4.6. *Let $C_{1..m}$ be the incremental weighted count from observation 1 to m and let $C_{m+1..n}$ be the incremental count from observation $m+1$ to n, where observations i such that $i \leq m$ precede a given tabled relation and observations such that $m < i \leq n$ are within the tabled relation. Likewise, let $\mathscr{L}_{1..m}$ be the likelihood associated with the state prior to the tabled relation and let $\mathscr{L}_{m+1..n}$ be the likelihood generated within the tabled relation and cached in the table. Then we compute the new likelihood and weighted count $\mathscr{L}_n$ and C_n at the end of the call to the tabled relation as follows:*

$$\mathscr{L}_n = \mathscr{L}_{1..m}\mathscr{L}_{m+1..n}$$
$$C_n = \mathscr{L}_{m+1..n}C_{1..m} + \mathscr{L}_{1..m}C_{m+1..n}$$

[8] A third modification, which does not directly affect the semantics of the procedures described here, but which considerably eases their implementation as well as the implementation of tabling as a whole, is to exchange the mutable implementation for an immutable version that threads the table data through the goals in the obvious way.

For a complete proof, see Appendix A.3.

The argmax strategy in conjunction with tabling leads to Viterbi training or Viterbi decoding, depending on the query. The Viterbi algorithm maintains a trellis of states and positions, populating each cell in the trellis with the maximum likelihood of observing that state at that position, and the index of the previous state that makes the present state most likely. Proceeding linearly through the sequence, for each position, each state only need consider all possible immediately previous states, and can ignore all combinations of states preceding that. It is clear that tabling *hmm* when implemented with the **`argmax`** form achieves this algorithm, albeit in the equivalent reverse direction, since the **`argmax`** form in the last position of the HMM will produce only the state that maximizes the state corresponding to that position for each possible transition. Subsequent attempts to recompute this value from different paths through the previous state will simply reuse the cached value, which in turn will build the backwards trellis within the table up to the initial position.

The summation strategy in turn recovers the forward-backward algorithm to compute the weighted counts and consequently, when used iteratively for training, yields the Baum-Welch training procedure. This equivalence is more difficult to see, especially since the conventional implementation of forward-backward uses two passes—forward and backward—through the HMM. We can gain some intuition by recognizing that the table already contains the backward probabilities. The forwards probabilities are supplied piecewise by each call to a given tabled relation from each possible prefix sequence of the HMM, which are combined according to Theorem 4.6 to produce the products of forwards and backwards probabilities that characterize Baum-Welch. Baum-Welch can, in fact, be viewed directly as a weighted sum over all possible paths through an HMM [13], which is intuitively what this implementation computes.[9]

4.4.2 Ensuring Termination. One difficulty that arises when combining aggregate streams and tabling is that it is no longer possible to detect the fixed point of the computation in the usual way. Normally, once the tabling implementation reaches a point at which all leaves of the search tree are suspended streams, it must perform a sweep through those leaves to determine whether any are capable of producing answers. If not, the computation as a whole has reached a fixed point, and the search can be terminated. Once aggregate streams are introduced, however, they may intercept answers before they can reach producer streams and be entered into the table, making the table seem artificially empty and leaving consumer streams of the table unable to produce additional answers, yielding a false positive fixed point result.

In order to deal with this, it is first necessary to observe that the particular graphical models we target in this paper are acyclic and the associated dynamic programming algorithms assume their data is finite. Consequently, there will always be a stream in the table that does not depend cyclically on other streams. For instance, the tail end of the state sequence in an HMM does not depend on further tabled calls but simply succeeds by unifying with the empty list. This means that the full fixed point computation is not required, because a fixed point need never be reached provided the tabling implementation possesses the means to clean up such streams individually.

This capability can be added by extending the tabling implementation with a boolean flag for each cache (defined as the list of memoized answers keyed to each set of unique arguments supplied to the tabled relation) that records whether or not the cache is still open. A cache starts in the open state and remains open until its associated producer stream fails, at which point that stream can close the cache by flipping the boolean flag. The next time a consumer stream attempts to examine

[9]Parenthetically, although this has not been implemented, it should in principle be possible to accumulate the forwards probabilities at the call site of each tabled relation in the table itself using another modification of the tabling implementation, which would result in the final table containing the forward and backward probabilities required to compute the full posterior decoding, with some additional work to extract these probabilities from the table at the end of run.

the cache for additional answers, it can observe that the cache is closed and, if it has consumed all the available answers, fail directly without waiting for the fixed point computation to fail as a whole.

Using this strategy, the final tabled calls will fail rather than suspend indefinitely, causing the aggregate streams above them to fail as normal and return their answers, which will pass their aggregated answers up to the tabled producers above them, which in turn will close their caches and cause their consumer streams to fail once they have consumed all their answers, and so on up the tree until the entire computation completes without ever reaching a fixed point. This strategy guarantees the termination of the dynamic programming algorithms described in this paper, and moreover has applicability to other problems beyond probabilistic computations that similarly depend on an acyclic use of tabling.

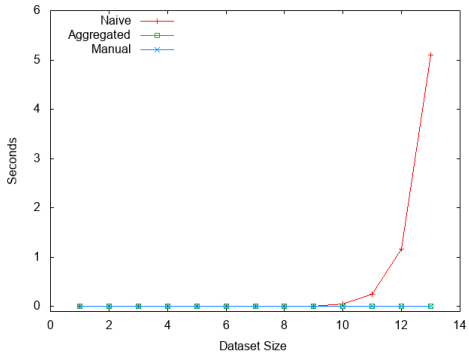
5 EVALUATION

This section compares the performance of the naive, stream aggregating, and tabled approaches to implementing HMMs and GMMs in order to confirm that these optimizations improve upon the naive implementations. We compare performance on calculating the data likelihood of a single HMM of varying observation sequence lengths, as well as a GMM of varying dataset sizes. Numbers are recorded in seconds as reported by the language's standard benchmarking utility, which measures average runtime across at least one second's worth of executions, less time spent in garbage collection. Only a single benchmark was conducted at each size, as measurement precision is less important here than the general trend. These experiments were run on a Lenovo Thinkpad T520 with an Intel i7-2760QM processor running Ubuntu 18.04.

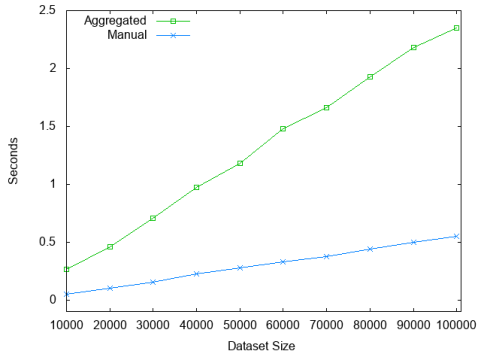
Overall, the results are as expected from the relative computational complexities of each method. Figure 1(a) displays a three-way comparison between a naive Gaussian mixture model, a mixture model implemented using aggregate streams, and a non-MiniKanren, manually implemented model. This is a test of whether the implementation can exploit the conditional independence of the independent and identically distributed data points in the dataset. The naive implementation quickly explodes while the other two remain imperceptibly close to 0.

Figure 1(b) reproduces the same graph without the naive implementation to better compare the aggregated and the manual versions. Both exhibit the expected linear growth due to factorizing the independent data points, with the manual implementation outperforming the miniKanren version by a factor of approximately 4. The difference is likely due to the overhead of the miniKanren search. While this overhead could perhaps be streamlined, it will necessarily encounter a fundamental limit to the extent that it is reproducing precisely the same calculation as the manual version but with the addition of extra computations for the miniKanren search. Moreover, the manual implementation used was not particularly optimized for the problem, and optimizations that exploit the structure of a given learning problem, such as vectorizing the additions and multiplications in the GMM calculation, could likely achieve significantly greater performance than is possible in the more general miniKanren-based strategy. It should not, therefore, be assumed that a miniKanren program will differ from its corresponding manual implementation by a factor of 4 in the general case. Rather, this evaluation means only to suggest that, at least for some tasks, this implementation is sufficiently practical to be of more than purely academic interest. That said, see section 6 for a discussion of possible strategies for further narrowing the performance gap.

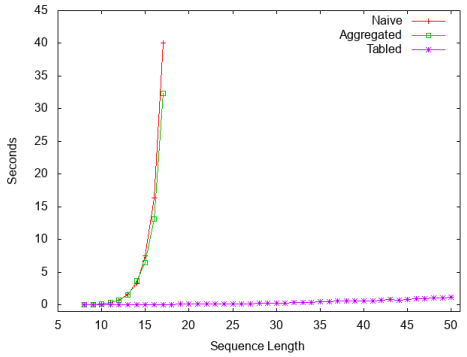
Figure 1(c) displays runtimes for decoding a single HMM based on the weather forecasting example from Section 4.1. As expected, the naive and aggregate implementations show similar exponential performance, as there is no conditional independence of sibling conjuncts to exploit. That the aggregated implementation appears slightly faster has no theoretical basis, and is likely



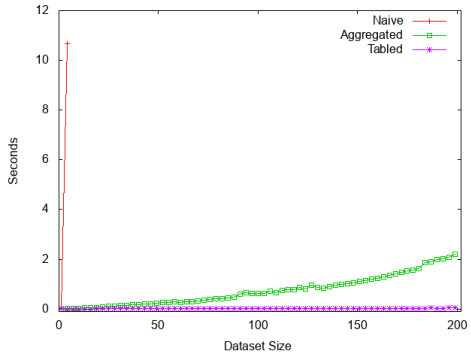
(a) 2-cluster GMM



(b) 2-cluster GMM without naive



(c) Single HMM Decoding



(d) Multiple HMM Decoding (Length 4)

Fig. 1. Performance Tests

due to idiosyncrasies of the implementation. The tabled version, by contrast, exhibits the expected linear performance of the Viterbi algorithm in the length of the sequence.

Figure 1(d) displays runtimes for the same HMM over a dataset of multiple HMMs of length 4 (chosen for best visual separation in the graph). The naive implementation quickly explodes due to failure to exploit either the independence of the samples from the dataset or of the Markov chain given its previous state. The aggregate stream exploits the independence of the dataset but not of the chain itself. The tabled implementation, finally, exploits both types of independence and avoids the combinatorial complexity. Moreover, given the small number of states in this particular model, the tabled implementation is able to amortize its runtime across multiple data points as well by simply looking up the observation sequence whole in the table without needing to run Viterbi more than once for each unique chain in the dataset.

6 DISCUSSION & FUTURE WORK

This paper has primarily focused on using miniKanren to replicate standard generative probabilistic modeling algorithms. However, there are several obvious avenues for future work building on the basic WMC framework described here. In addition to increasing the expressiveness of the language to include additional types of modeling tasks and borrowing techniques from other PPLs to further

optimize inference, there are also a number of potential avenues to use these models to enhance the existing capabilities of the miniKanren search.

While the implementation is reasonably performant, per Section 5, it is nevertheless reasonable to ask whether further gains could be made to further narrow the gap between it and manually implemented models. Despite the unavoidable overhead of the miniKanren search, it may nevertheless be possible in instances to incorporate more advanced inference strategies that improve upon the straightforward implementations of various inference algorithms. Sato et al. [19], for instance, uses an extension to tabling to improve upon even the standard dynamic programming algorithms by amortizing across EM iterations. Moreover, general approaches to accelerating EM itself, such as through better sample efficiency, may offer additional benefits during training that can mitigate some of the intra-iteration overhead and help close the distance or even surpass manual implementations that do not make full use of all possible sources of efficiency [12, 23].

In addition to efficiency concerns, there are several natural extensions to the expressivity of the language constructs described. Most obviously, the maximum likelihood updating strategy can trivially be extended to a variational Bayesian strategy simply by changing the final calculation in the maximization step of the EM algorithm to the usual Bayesian conjugate updating. Given basic Bayesian updating strategies for the standard exponential family distributions, it would be particularly useful to extend the language to handle non-parametric models. Currently, only a finite number of distributions can be defined and used in the model. Various strategies for non-parametric Bayesian modeling, which involves using the search behavior to instantiate new distributions, would fit particularly well with miniKanren's ability to generate infinite structures, and a number of strategies exist that might be effectively adapted to the framework introduced here [3, 11].

Finally, one interesting possibility for broadening the utility of these models is that of integrating them more closely with the miniKanren search. From prioritizing search branches [24] to repurposing the search to act as a sampler, there are a number of possibilities for how these models might be used beyond standard questions of statistical modeling. Given miniKanren's strengths in program synthesis, it may be interesting to explore the synthesis of probabilistic programs models themselves, although this would require further work to adapt the elements presented here to this new purpose.

7 CONCLUSION

This paper has introduced a simple extension to miniKanren allowing for compact model specification and efficient variational inference of probabilistic models via weighted model counting. The aim of this extension has been to enable the writing of probabilistic logic programs that accomplish practical machine learning tasks. Moreover, in future work, we intend to further extend this modeling framework and inference engine to capitalize on miniKanren's existing strengths in program synthesis and general constraint logic programming to explore applications at the intersection of logic and probability.

8 ACKNOWLEDGMENTS

We thank Rob Zinkov, Will Byrd, Arunava Gantait, and Julie Steele for their comments on early drafts and for extensive discussions about the fundamental ideas on which this paper was based. We also thank the anonymous reviewers for their detailed suggestions.

REFERENCES

[1] Samer Abdallah, Nicolas Gold, and Alan Marsden. 2016. Analysing Symbolic Music with Probabilistic Grammars. *Computational Music Analysis* (2016), 157–189.

[2] Michael Ballantyne. 2020. Faster miniKanren [Source Code]. (2020). https://github.com/michaelballantyne/faster-miniKanren

[3] David Blei and Michael Jordan. 2006. Variational Inference for Dirichlet Process Mixtures. *Bayesian Analysis* 1, 1 (2006), 121–143.

[4] William Byrd. 2010. *Relational Programming in Minikanren: Techniques, Applications, and Implementations*. Ph.D. Dissertation. Indiana University.

[5] William E Byrd, Eric Holk, and Daniel P Friedman. 2012. MiniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*. ACM, 8–29.

[6] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery.. In *IJCAI*, Vol. 7. Hyderabad, 2462–2467.

[7] Arthur Dempster, Nan Laird, and Donald Rubin. 1977. Maximum Likelihood from Incomplete Data Via the EM Algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)* 39, 1 (1977), 1–22.

[8] Evan Donahue. 2021. Guarded Fresh Goals: Dependency-Directed Introduction of Fresh Logic Variables. *miniKanren and Relational Programming Workshop* (2021).

[9] Jason Eisner, Eric Goldlust, and Noah A Smith. 2005. Compiling Comp Ling: Weighted Dynamic Programming and the Dyna Language. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*. 281–290.

[10] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–31.

[11] Viet Huynh, Dinh Phung, and Svetha Venkatesh. 2016. Streaming Variational Inference for Dirichlet Process Mixtures. In *Asian Conference on Machine Learning*. PMLR, 237–252.

[12] Xiao-Li Meng and David Van Dyk. 1997. The EM Algorithm—An Old Folk-Song Sung to a Fast New Tune. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 59, 3 (1997), 511–567.

[13] István Miklós and Irmtraud Meyer. 2005. A Linear Memory Algorithm for Baum-Welch Training. *BMC Bioinformatics* 6, 1 (2005), 1–8.

[14] Søren Mørk and Ian Holmes. 2012. Evaluating Bacterial Gene-Finding HMM Structures as Probabilistic Logic Programs. *Bioinformatics* 28, 5 (2012), 636–642.

[15] Stephen Muggleton et al. 1996. Stochastic Logic Programs. *Advances in Inductive Logic Programming* 32 (1996), 254–264.

[16] David Poole. 1993. Logic Programming, Abduction and Probability. *New Generation Computing* 11, 3 (1993), 377–400.

[17] Lawrence Rabiner. 1989. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proc. IEEE* 77, 2 (1989), 257–286.

[18] Taisuke Sato. 2005. A Generic Approach to Em Learning for Symbolic-Statistical Models. In *Proc. of the 4th Learning Language in Logic Workshop (LLL-05)*.

[19] Taisuke Sato, Shigeru Abe, Yoshitaka Kameya, Kiyoaki Shirai, Sato Taisuke, et al. 2001. Fast EM Learning of a Family of PCFGs. (2001).

[20] Taisuke Sato and Yoshitaka Kameya. 1997. PRISM: A Language for Symbolic-Statistical Modeling. In *IJCAI*, Vol. 97. Citeseer, 1330–1339.

[21] Jacob Schreiber. 2017. Pomegranate: Fast and Flexible Probabilistic Modeling in Python. *The Journal of Machine Learning Research* 18, 1 (2017), 5992–5997.

[22] Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. 2004. Logic Programs with Annotated Disjunctions. In *International Conference on Logic Programming*. Springer, 431–445.

[23] Jiangtao Yin, Yanfeng Zhang, and Lixin Gao. 2012. Accelerating Expectation-Maximization Algorithms with Frequent Updates. In *2012 IEEE International Conference on Cluster Computing*. IEEE, 275–283.

[24] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. 2018. Neural Guided Constraint Logic Programming for Program Synthesis. *Advances in Neural Information Processing Systems* 31 (2018).

[25] Neng-Fa Zhou, Yoshitaka Kameya, and Taisuke Sato. 2010. Mode-Directed Tabling for Dynamic Programming, Machine Learning, and Constraint Solving. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, Vol. 2. IEEE, 213–218.

[26] Robert Zinkov and Willliam Byrd. 2021. probKanren: A Simple Probabilistic Extension for microKanren.. In *Probabilistic Logic Programming Workshop*.

A PROOFS

A.1 Proof of Theorem 4.3: Equivalence of Weighted and Incrementally Weighted Counts

We want to show that Theorem 4.3 computes C_M, and in particular, C when $M = N$.

Proof.

$$
\begin{aligned}
C &= C_N && \text{(Definition 4.1)} \\
&= \prod_{i=1}^{N} \ell_i \sum_{j=1}^{N} c_j && \text{(Definition 4.2)} \\
&= \prod_{i=1}^{N} \ell_i \left(c_N + \sum_{j=1}^{N-1} c_j \right) \\
&= \left(c_N \prod_{i=1}^{N} \ell_i \right) + \left(\ell_N \prod_{i=1}^{N-1} \ell_i \sum_{j=1}^{N-1} c_j \right) \\
&= c_N \mathscr{L}_N + \ell_N C_{N-1} && \text{(Theorem 4.3)}
\end{aligned}
$$

□

A.2 Proof of Theorem 4.5: Correctness of Marginalized Counts

To show that marginalized streams yield identical calculations to the original streams, we must prove that the summed likelihood $\mathscr{L}$ and the summed weighted counts C yield the same results whether first summed and then subjected to future **observe** goals or first subjected to future **observe** goals and then summed either at the end of **run**, as per the naive implementation, or within another **marginalize**. First, consider the likelihood $\mathscr{L}' = \ell\mathscr{L}$ after an **observe** with a likelihood of ℓ. We use the prime notation so that subscripts can represent an index into the substreams of **marginalize** rather than the number of observations, as in previous proofs. Hence, prime should be taken to mean $N + 1$ where non-prime values should be taken to be N in the formalism of Theorem 4.3. We want to show that multiplication of $\mathscr{L}$ by ℓ is equivalent to the sum of the likelihoods of each L_i scaled by the same amount and later summed.

Proof. Let $\mathscr{L} = \sum_{i=0}^{N} \mathscr{L}_i$ where $\mathscr{L}_i$ is the likelihood of state i to be aggregated by **marginalize** and there are N such states in total. Then we have:

$$
\begin{aligned}
\mathscr{L} &= \sum_{i=0}^{N} \mathscr{L}_i \\
\ell\mathscr{L} &= \ell \sum_{i=0}^{N} \mathscr{L}_i \\
\mathscr{L}' &= \sum_{i=0}^{N} \ell\mathscr{L}_i
\end{aligned}
$$

This is the same as if each L_i had individually encountered the **observe** and been summed at a later point. □

Next, consider weighted count $C = \sum_{i=0}^{N}$ representing the summation of the weighted counts of N answers aggregated by **marginalize** . We want to show again that **observe** distributes over states' weighted counts.

Proof. Let incremental weighted count $C' = C_{N+1}$ where $C = C_N$ for some N representing the number of observations summarized by the final return value of **marginalize**. Theorem 4.3 gives the relationship between C' and C, and so it is necessary only to demonstrate that this relationship distributes over each C_i where $C = \sum_{i=0}^{N} C_i$.

$$
\begin{aligned}
C' &= c'\mathscr{L}' + \ell' C && \text{(Theorem 4.3)} \\
&= c'\ell' \sum_{i=0}^{N} \mathscr{L}_i + \ell' \sum_{j=0}^{N} C_j \\
&= \sum_{i=0}^{N} c'\ell'\mathscr{L}_i + \ell' C_i \\
&= \sum_{i=0}^{N} C'_i
\end{aligned}
$$

□

A.3 Proof of Theorem 4.6: Correctness of Tabled Counts

We want to show that the final likelihood $\mathscr{L}_n = \mathscr{L}_{1..m}\mathscr{L}_{m+1..n}$ and that the final incremental weighted count $C_n = \mathscr{L}_{m+1..n}C_{1..m} + \mathscr{L}_{1..m}C_{m+1..n}$ are equivalent to the corresponding values produced by the untabled conjunction of *observation* goals from 1 to n.

Proof. Beginning with $\mathscr{L}_n$:

$$
\begin{aligned}
\mathscr{L}_n &= \prod_{i=1}^{n} \ell_i \\
&= \prod_{i=1}^{m} \ell_i \prod_{j=m+1}^{n} \ell_j \\
&= \mathscr{L}_{1..m}\mathscr{L}_{m+1..n}
\end{aligned}
$$

Next, for C_n:

$$
\begin{aligned}
C_n &= \prod_{i=1}^{n} \ell_i \sum_{j=1}^{n} c_j \\
&= \prod_{i=1}^{m} \ell_i \prod_{j=m+1}^{n} \ell_j \left(\sum_{k=1}^{m} c_k + \sum_{l=m+1}^{n} c_l \right) \\
&= \prod_{j=m+1}^{n} \ell_j \left(\prod_{i=1}^{m} \ell_i \sum_{k=1}^{m} c_k \right) + \prod_{i=1}^{m} \ell_i \left(\prod_{j=m+1}^{n} \ell_j \sum_{l=m+1}^{n} c_l \right) \\
&= \mathscr{L}_{m+1..n} C_{1..m} + \mathscr{L}_{1..m} C_{m+1..n}
\end{aligned}
$$

□