Lazy Factored Inference for Probabilistic Programming

Avi Pfeffer, Brian Ruttenberg, Amy Sliva, Michael Howard, Glenn Takata
Charles River Analytics

Abstract

Probabilistic programming provides the means to represent and reason about complex probabilistic models using programming language constructs. Even simple probabilistic programs can produce models with infinitely many variables. Factored inference algorithms are widely used for probabilistic graphical models, but cannot be applied to these programs because all the variables and factors have to be enumerated. In this paper, we present a new inference framework, called lazy factored inference (LFI), that enables factored algorithms to be used for models with infinitely many variables. LFI expands the model to a bounded depth and uses the structure of the program to precisely quantify the effect of the unexpanded part of the model, thereby producing lower and upper bounds to the probability of the query.

1. INTRODUCTION

Probabilistic models are growing in their richness, diversity, and widespread usage. One of the challenges to using probabilistic models, especially for users without deep machine learning expertise, is the need to create representations and reasoning algorithms for models. Probabilistic programming (PP) (Koller, McAllester, Pfeffer et al., 1997) addresses these challenges by providing expressive languages to represent models using programming language constructs and inference algorithms that apply automatically to models written in the languages.

One of the biggest challenges in PP inference is that even compact programs can result in models with very large or an infinite number of variables. Currently, the typical method for performing inference in such models is to use Metropolis-Hastings (MH) (Metropolis, Rosenbluth, Rosenbluth, 1953; Hastings, 1970), which has become a standard algorithm in languages such as BLOG (Milch, Marthi, Russell et al., 2005), Church (Goodman, Mansinghka, Roy et al., 2008), and Figaro (Pfeffer, 2012) for this reason. Unfortunately, MH is extremely hard to understand and requires significant expertise to achieve convergence at a reasonable rate in many applications.

Factored algorithms, such as variable elimination (VE) (Zhang & Poole, 1994; Dechter, 1999) and belief propagation (BP) (Pearl, 1988; McEliece, Mackay, Cheng, 1998), are alternative and widely used inference algorithms and are generally preferred to MH where they are applicable. For instance, in the 2010 UAI Approximate Inference Competition, many of the entrants used factored algorithms, while none used MH. However, current factored algorithms require enumerating all the variables in the model and creating factors for them, which is infeasible for models with a very large number of variables, and impossible if there are infinitely many variables. Indeed, Infer.NET (Winn, 2008) has achieved excellent results on real-world inference tasks (Herbrich, Minka, Graepel, 2006) using expectation propagation (Minka, 2001), a factored algorithm, at the cost of severely restricting the expressivity of the language to avoid recursion, thereby eliminating infinite models.

We believe that just as factored algorithms have been instrumental in the success of probabilistic graphical models in general, making factored inference work for PP is essential to its eventual success. In this paper we describe an inference framework—*lazy factored inference (LFI)*—that achieves this goal, making factored algorithms applicable to models with very many or infinitely many variables. LFI works by expanding a potentially infinite model up to a bounded depth and characterizing precisely the effect of the unexpanded part of the model on the probability of the query. As we show, characterizing the effect of the unexpanded part of the model can be performed using standard factored inference algorithms as a subroutine with no modification to the algorithms, with the addition of preprocessing and postprocessing steps. The result of LFI is a pair of lower and upper bounds on the probability of the query. By

iteratively expanding the model to increasing depths, we obtain an anytime algorithm that can produce progressively tighter bounds. Although LFI is a general inference framework for graphical models, it works particularly well for PP, because PP languages typically have the necessary constructs to guide the lazy expansion.

The remainder of this paper is organized as follows. In Section 2 we provide a running example that will be used to illustrate the LFI approach. Sections 3 and 4 present the basic intuition and technical details of LFI. In Section 5 we present theoretical results and analysis of the LFI approach. In Section 6 we describe an implementation of two lazy factored algorithms—VE and BP—in the open source Figaro PP language(Pfeffer, 2012) and present experimental results on reasoning with probabilistic context-free grammars, which would otherwise be intractable using standard factored algorithms. Finally, in Section 7 we discuss related work and in Section 8 we conclude.

2. RANDOM LISTS EXAMPLE

As a simple running example to motivate our approach, we use a model that generates random lists of unbounded length. Each list consists of the symbol `'a` or the symbol `'b` at each index. Lists are created by a generator function that grows the list one symbol at a time. At each step, the generator terminates with probability 0.5, adds an `'a` with probability 0.3, or adds a 'b with probability 0.2. We can query the list for certain properties, such as whether the list contains a `'b`.

This list generator and the containment queries can both be defined in Figaro, a PP language embedded in Scala and capable of representing and reasoning about a wide variety of probabilistic models. The following three lines of code define the random list in Scala using the Figaro `Element` construct, which represents random variables. Line 1 defines a general class of random lists, named `L`. A random list consists of two possible cases: either it is defined to be the Empty list (Line 2), or it is a Cons of two Elements (random variables) as defined in Lines 3 and 4, where the `head` is an `Element[Symbol]` and the tail is an `Element[L]` (i.e., a random list).

```
1 abstract class L
2 case object Empty extends L
3 case class Cons(head: Element[Symbol],
4                 tail: Element[L]) extends L
```

We now define the random list generator function. The body of the `generate` function, which returns an `Element[L]` (random list) is found in lines 2-5. First, it uses `Flip(0.5)` (Line 3) to generate a random Boolean that is true with probability 0.5. If the Boolean is true, it produces the `Empty` list (Line 3) to terminate the list. Otherwise, it produces a `Cons` in which the head is `'a` with probability 0.6 and `'b` with probability 0.4 (Line 4), and the tail is the result of a recursive call to `generate()` (Line 5). Sampling from this `generate()` function could generate lists of unbounded length, while full expansion of all the possibilities results in a model with infinitely many variables.

```
1 def generate(): Element[L] = {
2  Apply(Flip(0.5), (b: Boolean) =>
3   if (b) Empty
4   else Cons(Select(0.6 -> 'a, 0.4 -> 'b),
5             generate()))
6 }
```

Now, suppose we want to know whether this list contains a particular symbol. We can define a `contains` predicate, which takes two arguments: the `target` symbol and the random list `el` we are checking. The implementation of the `contains` predicate in Figaro is shown below.

```
1 def contains(target: Symbol,
2  el: Element[L]): Element[Boolean] = {
3  Chain(el, (l: L) => {
4    l match {
5     case Empty => Constant(false)
6     case Cons(head, tail) =>
7      If(head === target,
8         Constant(true),
9         contains(target, tail))
10   }})}
```

The result of the `contains` predicate is a random variable denoted by the type `Element[Boolean]` (Line 2). Even though `contains` works deterministically, the result is random because the list argument is random. The body of `contains` is found in Lines 3-11. It uses `Chain`, a Figaro construct that chains random processes together through two arguments: an `Element` (random variable) and a function that takes a value of the `Element` and produces another `Element`. A `Chain` will first sample a value from the given `Element` argument. Then it applies the given function to this value to produce a new `Element`. Finally, it samples a value from this new `Element`.

In the case of `contains`, the `Element` argument is the random list `el`. The function argument takes a particular value of `el`, which is a list `l`, and returns an `Element[Boolean]`. The body of this function is found in Lines 4-10 using pattern matching on the type of `l`. If `l` is `Empty` (Line 5), the function returns `Constant(false)`, which is the element whose value is `false` with probability 1. Otherwise (Lines 6-10), `l` must be a `Cons`. If the value of `head` is equal to the `target`, it returns `Constant(true)`, otherwise it recursively calls `contains` on the tail.

Using this model, we want to be able to observe evidence and ask queries about the contents of a random list. The Figaro code below generates a random list in Line 1 using the `generate` function. Lines 2 and 3 create random Booleans indicating whether `el` contains the symbols `'a`

or `'b`, respectively. Now suppose we observe that the random List contains `'a`. Line 4 sets this observation. Given this evidence that the list contains `'a`, we want to determine the probability that the list also contains `'b`. Although the answer can be determined analytically in this simple example, a general algorithmic solution would need to sum over infinitely many sequences of unbounded length, motivating the need for a lazy solution. The fifth line creates a lazy version of VE capable of solving this otherwise intractable query. In the next section we describe how LFI makes factored analysis of very large or infinite models possible in PP.

```
1 val el = generate()
2 val ca = contains('a, el)
3 val cb = contains('b, el)
4 ca.observe(true)
5 val alg = new LazyVariableElimination(cb)
```

3. LAZY FACTORED INFERENCE

So, how do we make this VE algorithm work without enumerating the infinitely many variables in the model? The main intuition is that variables that are far from the query and evidence have little impact on the query. However, it is not just the distance from the query and evidence that matters, it is the fact that other variables need to take on particular values to make these faraway variables relevant. In our example, the query to determine the probability of the symbol `'b` occurring in the list, given that we have evidence of symbol `'a`, is a `contains` function that recursively processes the list from the beginning. In this case, variables that correspond to symbols far along the list, or variables determining whether the list terminates at some point far along the list are considered less relevant, because they are only relevant if the list has not terminated earlier.

Because not all variables contribute equally—and in fact because many variables have only a minor impact—to the query result of any given model, it is not necessary to enumerate the entire probability space for accurate inference. LFI is a new approach that expands the model, beginning with the query and the evidence, up to a bounded depth, and characterizes quantitatively the effect of the unexplored part of the model on the query. This expansion will only explore *relevant* parts of the model to a specified depth, following the definition of (Baker & Boult, 1990a).

Definition 1 (Relevant Variables). Given a set **Q** of query variables and set **E** of evidence variables in a Bayesian network, relevant variables w.r.t. **Q** and **E** are variables X in the set $\mathbf{Q} \cup \mathbf{E} \cup \text{An}(\mathbf{Q} \cup \mathbf{E})$, such that X is not d-separated from **Q** by **E**, where $\text{An}(\mathbf{Q} \cup \mathbf{E})$ contains all ancestors of **Q** and **E**.

In LFI, we expand the model to a bounded depth, producing a Bayesian network, and only consider relevant variables in that network. We explore close relevant variables first—those that are closest to the query and evidence variables along non blocked paths—ignoring nodes that are either distant from the query or "barren" in that their distributions supply no information to the beliefs of the query variables(Shachter, 1988; Baker & Boult, 1990b).

For the variables **Y** that is distant from the query, we determine values **x** of the expanded variables **X** that render **Y** irrelevant, in the sense that the probability of the query is independent of **Y** when $\mathbf{X} = \mathbf{x}$. In other words, the query is fully determined by **x**. We can then assign the probability mass $P(\mathbf{X} = \mathbf{x})$ to different possible query outcomes. This contributes to a lower bound for the query outcomes. For any value **x'** that does not render **Y** irrelevant, the query is undetermined, and the probability mass $P(\mathbf{X} = \mathbf{x'})$ could potentially be added to any of the query outcomes. As a result, through this limited expansion of the model, we will be able to apply factored inference algorithms to a reduced, tractable number of variables to compute lower and upper bounds on the query result.

For example, from a partial expansion of our model to the first n elements of a list *el*, we can compute:

- p_1 = P(*el* has length ≤ n and does not contain `'b`)
- p_2 = P(*el* contains `'b` in the first n elements)
- p_3 = P(*el* has length > n and does not contain `'b` in the first n elements)

In the first case, the query for whether the list contains `'b` is definitely false, in the second case it is definitely true, and in the third case the query is not yet determined. So (p_2, $p_2 + p_3$) are lower and upper bounds on the probability that the list contains `'b`. When we have evidence that the list contains `'a`, we get more cases, but the principle is similar.

Of course, since we are only partially exploring the model along relevant paths, we cannot guarantee that all unexplored portions are irrelevant to the query. To represent the unexplored probability mass in LFI, we extended the range of values a variable can take.

Definition 2 (Extended Variable Range). *A variable with an* extended variable range *can take a regular value, or it can take the special value * (pronounced "star").*

For example, the possible extended values of a Boolean are { false, true, * }. Intuitively, * stands for "unknown result of the rest of the computation," and the probability associated with * represents the amount of probability mass resulting from the unexplored part of the computation. If we quantify this, we know how much remaining probability mass could be added to each of the regular values.

As will be discussed in Section 4, by computing sums and products involving extended values in the ordinary way, we can keep track of this probability mass.

The above concepts can be formalized into a LFI algorithm consisting of four steps:

1. Expand the model to the desired depth and compute the extended ranges of relevant elements
2. Produce factors for the relevant elements
3. Apply a factored inference algorithm to the factors
4. Finalize the result to produce bounds on the query

The LFI algorithm naturally lends itself to an iterative deepening approach, where we gradually increase the depth and improve the resulting bounds on the query (given that all evidence is known). This produces an anytime algorithm for factored inference on very large or infinitely large models using PP. In the following section, we discuss each step of the LFI algorithm in detail.

4. THE LFI ALGORITHM FOR PP

We now provide details on the four steps of the LFI algorithm for PP and its implementation using Figaro.

4.1 STEP 1: EXPAND THE MODEL

The first step of the LFI algorithm is to expand the model, beginning with the query and evidence, up to a depth *d*. This step must determine which variables are relevant when the model is expanded to this depth and the range of each relevant variable, which is a set of extended values (possibly including *). We present two approaches to the expansion, a basic algorithm (Section 4.1.1) suitable for simple queries, and a backtracking version (Section 4.1.2) that can be used to compute more complex queries and evidence. Section 4.1.3 specifically addresses lazy expansion of evidence.

We explain these algorithms using Figaro constructs, but they are all generalizable to other PP languages. Recall from Section 2 that in Figaro a random variables is represented by an `Element`. Some elements are atomic, meaning they do not depend on any arguments (e.g., `Select(0.6 -> 'a, 0.4 -> 'b)`is the probabilistic model that produces `'a` with probability 0.6 and `'b` with probability 0.4). An element can also consist of the more complex `Chain` structure for chaining random processes together (see Section 2). As we will see in Section 4.2, the `Chain` construct helps to control and limit the impact of the unexplored part of the computation on the query. Most functional probabilistic programming languages have a structure similar to `Chain` that can be used in this manner. Finally, an element can have the form `Apply`(*arguments*, *function*), in which the arguments are elements, and the `Apply` element corresponds to the random variable produced by applying the deterministic *function* to the arguments. Since Figaro constructs can in general be expressed in terms of atomic elements, `Chain`, and `Apply`, it suffices to define the algorithm for these element classes.

4.1.1 Basic expansion algorithm

The basic expansion algorithm begins with a list of relevant elements consisting of the query and evidence, represented as Figaro elements, and proceeds recursively to depth *d* as follows.

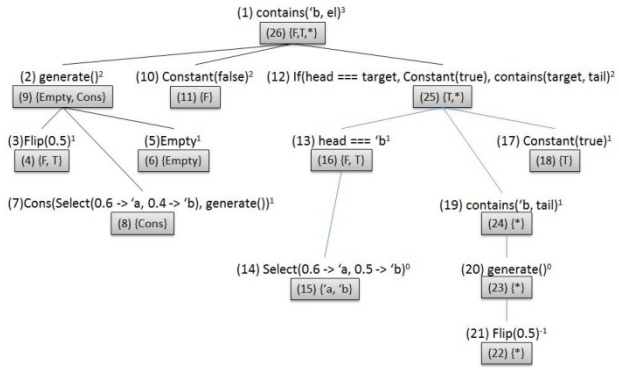
For a relevant element *E*:

1. If $d < 0$, return { * } for the range of *E*
2. If *E* is atomic, return its known range of regular values.
3. If *E* is a `Chain(`*X*`,`*F*`)`, where *X* is an element and *F* is a function that maps a value of *X* to another element:
    a. Expand *X* to depth *d* - 1.
    b. For each regular value *x* in the range of *X*:
        i. Compute $Y = F(x)$.
        ii. Expand *Y* to depth *d* - 1.
        iii. Each value, regular or *, in the range of *Y* is added to the range of *E* .
    c. If the range of *X* includes *, the range of *E* also includes *.
4. If *E* is `Apply(X,F)`, where **X** is a sequence of argument elements and *F* is a deterministic function of values of **X**:
    a. Expand each *X* in **X** to depth *d* - 1.
    b. For each combination **x** of regular values of **X**, the range of *E* contains $F(\mathbf{x})$.
    c. If any argument in **X** contains * in its range, the range of `E` also contains *.

All the elements that are expanded in this way, including those that are expanded to a depth of -1 and so have the range { * }, are *relevant*. At the end of this step, we create a variable for each such element whose range is the computed range of extended values of the element. These variables are later used to produce factors for the inference algorithms.

Figure 1 shows an example of the basic expansion algorithm for our random list example. Each node in the graph in Figure 1 corresponds to an element whose values are to be computed, and the shaded box beneath shows the resulting values. The numbers in parentheses to the left of the elements and the resulting values indicates the order in which the elements were expanded and their values were determined. The small superscript number to the right of the element represents the depth to which the element is expanded.

In this example, we want to determine which variables are relevant to our query—whether the list `el` contains the symbol `'b`—by looking to a target depth of $d = 3$. The first step is to expand the top-level query, `contains('b, el)` to $d = 3$. This query is a compound element, so will expand its arguments in Step (2) to depth $d-1 = 2$. The algorithm first looks at the value of `el`, which is defined by a call to `generate()`, and



expands `generate()` to d = 2. Again, we have a compound element, so the arguments of the `generate()` element are expanded to depth d = 1. Step (3) first looks at `Flip(0.5)`, which immediately produces the values `{F, T}` in Step (4). There are then two possible outcomes, depending on the value of the `Flip`: `Empty`, and `Cons(Select(0.6 -> 'a, 0.4 -> 'b), generate())`, which produce the value sets `{Empty}` and `{Cons}` respectively in Steps (5)-(8) as they are expanded and their ranges computed. Note that even though `Cons` contains two random elements, `Cons` itself is just a value. So, in Step (9), we determine that the possible values of `generate()` are `{Empty, Cons}`. If `generate()` is `Empty`, the top level query is `Constant(false)`, whose value set is `{F}` (Steps (10)-(11)), so `F` will become a possible value for the top level query.

Figure 1: Basic LFI expansion on a random list.

In this depth 3 expansion, so far we have found the case where the generated list is empty. Otherwise, the top level query is the result of `If(head === 'b, Constant(true),contains(target, tail))`.

In Step (13), we expand this compound element starting with expanding `head`, which we get out of the previously computed `Cons`. The range of values for `head` are `{'a, 'b}` (Step (14)), so the values of `head === 'b` are `{F, T}`. Since the test for `'b` in the `head` could be either `T` or `F`, we expand both consequences. In the first consequence (Step (17)), the head of the list is `'b`, so we have found a case where the top level query has value `T`. In the other consequence, we have a recursive call to `contains('b, tail)` at depth d = 1 in Step 19. This results in an expansion of `generate()` at depth 0, which in turn results in expansion of `Flip(0.5)` at d = -1 in Step (21). Since the depth is negative, we immediately get the result `{*}`for the range of `Flip(0.5)`. Since the `Flip` has no regular values, we do not expand either of the two outcomes `Empty` or `Cons`. Instead, we immediately return the value set `{*}` for `generate()`, and in turn for `contains('b, tail)`. This corresponds to a possible value of `*` for the top level query. In the end in Step (26), we get the value set `{F,T,*}` for the top level query.

4.1.2 Backtracking Expansion

The above algorithm is sufficient when we are only expanding a single query with no evidence, and when the expansion forms a tree such that no element occurs in more than one path. However, if the same element is used both by the query and some evidence, or is reachable from the query by more than one path, this basic expansion algorithm encounters a subtle problem where it may compute inconsistent ranges for the same elements.

Suppose we have a query element X and an evidence element Y, and the target depth is 1. Suppose also that Y is an argument of an argument of X. If we expand X first, we will eventually expand Y to depth -1, resulting in a range of { * }. However, because Y is an evidence element, we will eventually expand it to depth 1, resulting in a different range. The computed range of Y will be incompatible with the range of X, which can cause trouble for factored computation later on.

One possible solution is to stipulate in advance that whenever an important (query or evidence) element is encountered, it is always expanded to the maximum desired depth d. However, this does not completely solve the problem, because X and Y might both depend, at different depths, on some other element Z that is not a query or evidence element.

Our solution uses backtracking to keep track of dependencies at various depths and adjust previous computations once new dependencies are revealed by the expansion algorithm. Failure to make this optimization can lead to exponential blowup as the same elements get recursively expanded again and again. Consider a case where X and Y both depend on an element Z. Suppose Y is expanded first, resulting in Z being expanded to some depth d_1. After Z has been expanded, we record a back pointer from Z to Y. When X is later expanded, it will result in a request to expand Z to depth d_2. If $d_2 \leq d_1$, we have already computed an equal or better set of values for Z, so we do not expand Z again. If, however $d_2 > d_1$, we need to expand Z to a greater depth. After doing so and computing a new set of values for Z, we know from the back pointer that Z was previously expanded from Y to a lesser depth than d_2, so Y might use an inconsistent set of values of Z. Therefore, we backtrack and re-expand Y. We will also have back pointers from Y so we can re-expand other elements that depend on Y.

Using backtracking, we can ensure that the last time the values of an element are computed by Steps 2-4 in the basic expansion algorithm occurs after the last time values have been computed for all elements on which it depends.

Proposition 1. *For all elements Y that have been expanded by the LFI expansion algorithm with backtracking, the last expansion for Y occurs after the last expansion of all elements on which it depends.*

Please see the supplement for all proofs. This fact ensures that the value sets will be consistent.

4.1.3 Lazily Expanding Evidence

There is an additional optimization we can make to the expansion phase of LFI. Consider a large model with many evidence elements and a single query. Implementing the above expansion algorithm will require us to expand all the evidence variables regardless of their distance from the query, resulting in a large number of elements. However, as with irrelevant parts of the model that are represented by *, distant evidence may not be relevant to the query (i.e., there will be no path from the evidence variables to a query variable within depth *d* of the query). Ideally, we will only expand evidence that is close to the query and can actually contribute to the probability bounds computation.

We can accomplish this by modifying the basic expansion algorithm to lazily expand in multiple iterations, beginning with only the query elements *Q*.

1. Set `ExpandList = Q` with depth *d*
2. For each element *E* in `ExpandList`, expand *E* to specified depth *d* as described in Section 4.1.1
3. For each iteration where $d \geq 0$
    a. Identify all elements *X* that use the current element *E* and have not be expanded to *d* - 1
    b. If X is an evidence element, then add to `ExpandList` with depth *d* - 1
    c. Recursively expand *X* until $d < 0$
4. Continue until `ExpandList = Empty`

After this process has completed, we guarantee that all elements relevant to the query within a distance of *d* have been expanded.

Theorem 1. *Let* $\mathbf{Q}$ *denote a set of query variables and* $\mathbf{E}$ *a set of evidence variables with known values in a probabilistic graphical model* $\mathbf{G}$*. Lazy expansion of* $\mathbf{G}$ *to depth d will expand all variables relevant to* $\mathbf{Q}$ *and* $\mathbf{E}$ *within depth d* of $\mathbf{Q}$.

4.2 STEP 2: PRODUCE FACTORS FOR THE RELEVANT ELEMENTS

Once the model has been lazily expanded to the desired depth to identify the relevant elements and their possible values, the next step is to produce factors for these elements so they can be used in a factored inference algorithm. Figaro already contains an algorithm for producing factors for a finite set of elements whose corresponding variables have ordinary (not extended) ranges. Producing factors for elements whose variables have extended ranges extends this procedure in a straightforward way.

In general, there are two kinds of factors produced by Figaro. The first encodes the relationship between an element and its arguments resulting from the definition of the element's generative model. The second encodes conditions and constraints on a variable.

For the first kind of factor for an element *E*:

1. If *E* is atomic, it's factor is the usual factor over its regular values
2. If *E* is `Apply(`$\mathbf{X}$`,`*F*`)`, then the factor assigns a probability to each assignment $\mathbf{x}$ to the arguments and *y* to the result, as follows:
    a. If none of the arguments are *, and $y = F(\mathbf{x})$, the probability is 1.
    b. If any of the arguments is *, and $y = *$, the probability is 1.
    c. Otherwise, the probability is 0.
3. If *E* is `Chain(`*X*`,`*F*`)`, then we build off a technique used in Figaro for constructing factors for a chain without extended values. Since every value of *X* results in a different element, a naïve factor would include a variable for each such element, potentially resulting in extremely large factors if *X* has many values. Instead, many three variable factors are constructed. For each regular value *x* of *X*, we construct a factor φ_x over *X*, the specific element $Y = F(x)$ for some value *x* of *X*, and *E*. Without extended values, these factors are defined so that their product equals the single naïve factor. We extend this construction to extended values as follows.
    a. For each regular value *x* of *X*, we define a factor φ_x that specifies a probability for each value *x'* of *X*, *y* of $F(x)$, and *e* of E, as follows:
        i. If $x' \neq x$, the probability is 1. This is a "don't care" case.
        ii. If $x' = x$ and $e = y$, the probability is 1. This also applies if $e = y = *$.
        iii. Otherwise the probability is 0.
    b. We also create a binary factor φ_* that specifies a probability for each value *x* of *X* and *e* of *E*, as follows:
        i. If $x \neq *$, the probability is 1 (don't care).
        ii. If $x = *$ and $e = *$, the probability is 1.
        iii. Otherwise the probability is 0.

To see how this construction for chains helps control the effect of *, consider the following element from our random list:

```
If(head == target, Constant(true), contains(target, tail))
```

`If` is actually syntactic sugar for `Chain`, in which the first argument is the test, and the function maps the result of the test to the appropriate consequence. Here, if the test is true (i.e., the value of `head` is equal to the target symbol), only the `then` clause `Constant(true)` is relevant, so the factor $\varphi_{\texttt{true}}$ will not include the variable

for the `else` clause, while the factor $\varphi_{\texttt{false}}$ will have a don't care case. Therefore, even if the value of the `else` clause is *, the value `true` for the entire `If` expression will have probability 1 in each factor. This is the essential insight that prevents * contaminating the entire computation.

The second kind of factor corresponds to a condition or constraint. First we consider conditions, which are predicates on elements that are either satisfied or not satisfied. To produce a factor for an element `E` and condition `C`:

1. If `E` has a regular value, we can determine if `C` is satisfied and compute an entry of 0 or 1 as usual.
2. If the value of `E` is *, we do not know whether `C` would be satisfied by the eventual value * would resolve to if we expanded it fully, so we create bounds of [0, 1] on the entry.

Factors representing soft constraints, which are functions from the value of a variable to a real number, are similar. In this case, bounds must be specified on the value of the constraint. Bounds of [0, 1] are the default, but different or more precise bounds can be provided as necessary.

Using these modifications to Figaro's factor generation algorithm to account for unexpanded parts of the computation represented by *, Step 2 will produce a set of factors over variables with extended ranges. Only factors for relevant variables within the desired depth will be produced.

4.3 STEP 3: APPLYING A FACTORED ALGORITHM

Using the factors produced by Step 2, we can now determine an answer to the query, which is defined as a sum-of-products expression over these factors. The goal is to reduce this sum-of-products expression to a single factor over the query variables. Factored algorithms such as VE and BP produce solutions or approximations to this factor.

For LFI, standard factored inference algorithms can be applied with no modification; however, now they are only being computed over factors representative of the relevant parts of the computation for answering the query to the desired depth, rather than the entire model. The standard algorithm is called once using the lower bounds and once using the upper bounds specified in the factors.

4.4 STEP 4: FINALIZING THE RESULT

By applying a factored inference algorithm in the previous step, we acquire two factors over the query, one for the lower bounds, and one for the upper bounds. These factors will, in general, be unnormalized, and * might have positive probability mass. In this finalization step of the LFI algorithm, we need to normalize the results and absorb the probability mass of * into the regular values.

Let the unnormalized lower bound of value i (regular or *) of the query be l_i and let the unnormalized upper bound be u_i. Standard normalization takes a set of unnormalized probabilities q_i, computes their sum $Z = \Sigma q_i$, and then computes $p_i = q_i / Z$ to obtain the normalized probabilities. In our case, $U = \Sigma u_i$ is an upper bound on the normalizing factor. Therefore $L_i = l_i / U$ is a lower bound on the normalized probability of value i. Meanwhile, for a regular value j of the query, any probability assigned to the regular value $i \neq j$ cannot be assigned to j, so $1 - \Sigma_{\text{regular } i \neq j} L_i$ is an upper bound on the probability of j. Since any probability mass associated with * will not be subtracted in this upper bound, that probability mass is absorbed into the upper bounds of each of the regular values.

5. ANALYSIS

Our main result is that the process of lazily expanding the program to increasing depths results in increasingly better bounds on the probability distribution over the query. Our analysis assumes there is a single variable, and in fact multiple query variables can break the result if query variables only become connected after some depth has been expanded. If multiple query variables are desired, that can easily be achieved by defining a single variable to be a tuple of the query variables, and making that the query variable instead.

In addition, our result assumes that all evidence variables have already been included before the bounds start to converge. If new evidence variables are introduced after a certain depth, they might change the query distribution. In many applications, such as the probabilistic context free grammar example we present later, this is not a problem as the evidence is reached at a shallow depth.

Also, our result assumes that the factored algorithm used to compute the bounds is exact. For an approximate algorithm like BP, we cannot provide the same guarantees.

Our main result is as follows:

Theorem 2: Let Q be a query variable, $\boldsymbol{E}$ a set of evidence variables, and q a regular value of Q. Assume that expanding to depth $d + 1$ does not produce any new evidence variables. Let $l_d(q)$ and $u_d(q)$ denote the lower and upper bounds produced by LFI expanded to depth d. Then $l_{d+1}(q) \geq l_d(q)$ and $u_{d+1}(q) \leq u_d(q)$.

For finite models, at some depth d all variables will be expanded, and the bounds will be equal to the true probability. Therefore, the true probability lies between the bounds at every depth for finite models. For infinite models, the bounds do not necessarily converge. For example, consider the probabilistic program:

```
def f() = Apply(f(), (x: Boolean) => x)
val query = f()
```

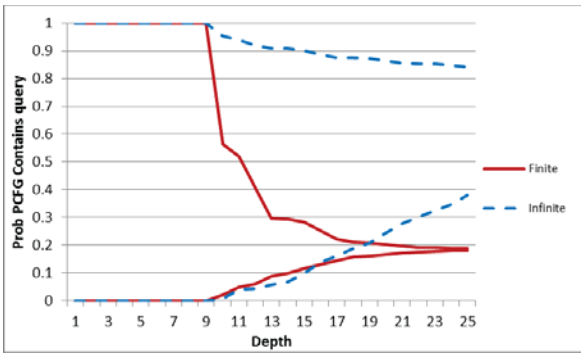


Figure 2: Probability bounds on the 'contains' query for the finite and infinite grammars.

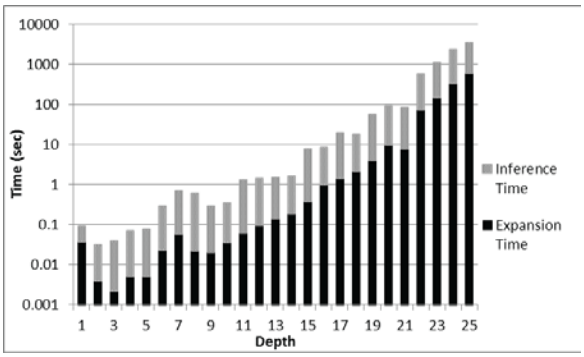


Figure 3: Running times of querying the infinite grammar for different depth expansions.

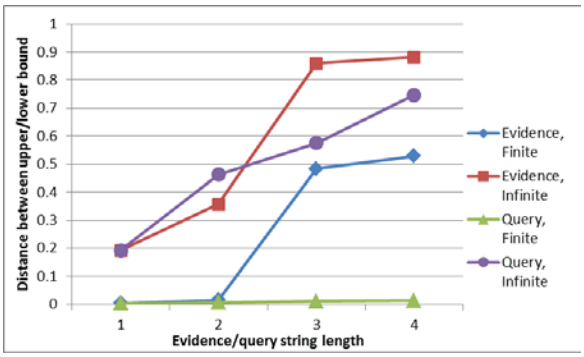


Figure 4: Probability bounds on the infinite grammar as a function ion of the query and evidence lengths. The points represent the mid-point of the bounds and the error bars show the upper and lower bounds.

This program defines an infinite chain such that each Boolean variable in the chain is equal to its predecessor. The bounds at any depth will be (0,1). This example illustrates the limits of our approach.

6. IMPLEMENTATION AND EXPERIMENTATION

We have produced two initial implementations of the LFI algorithm in Figaro, using VE and BP as the factored inference algorithms. Since BP does not provide guarantees, we have evaluated the VE implementation.

Our experiments were conducted using a probabilistic context-free grammar (PCFG). Encoding PCFGs in a PP language is straightforward, yet can present significant computational challenges when attempting to apply evidence to the PCFG and make inferences based on recognizing strings. First, all non-trivial PCFGs are unbounded, enabling generation of arbitrarily long strings that are difficult to parse and may not provide more query-relevant information than shorter strings. Second, some PCFGs are infinite, producing infinitely longs string with non-zero probability. Standard natural language parsing algorithms assume that a finite string is given as evidence, which is use to control the computation and limit the number of non-terminals that can be created. However, this technique does not work when the query is whether a string contains a particular substring, as in principle, arbitrarily long and even infinitely long strings may need to be examined to determine if the substring is present. Clearly, non-lazy factored algorithms cannot answer these queries, and sampling algorithms, such as importance sampling, can infinitely expand on certain samples. As such, we tested our LFI algorithm on both unbounded and infinite PCFGs to evaluate the algorithm's ability to reason on these otherwise intractable models.

For this experiment, we constructed a simple unbounded PCFG with three non-terminals, where the only difference between the finite and infinite grammars is the production probabilities. The grammar is encoded in Greibach Normal Form (GNF) (Greibach, 1965), which only has right-hand recursion. In other words, every production contains a terminal at the beginning, which serves to drive the generation of the string forward. This works well with LFI. Since this expansion is recursive, expanding to a fixed depth in the LFI algorithm will bound the length of the possible strings that can be generated, and thus produce bounds on the probability of the query string. Evidence can also be applied in the same manner by simply observing that the element returned by `contains` is either true or false. We refer the reader to the supplement for more details on the grammar.

Figure 2 shows the results of a query for the probability that a string produced by the PCFG contains the sub-string "de", given the observation that the string contains the sub-string "a". We show the results for both the finite and infinite versions of the grammar, expanding using LFI to a target depth ranging from $d = 1$ to $d = 25$, and using VE as the factored algorithm. As can be seen, the probability bounds on the queries in both grammars start to converge, and in the case of the finite grammar, do so quickly (at a depth of ~21). Observe that the bounds of the infinite grammar tend to stabilize for several depths (e.g., between 17 and 19), and then tighten at other depths (e.g., 19). This is an artifact of the grammar generation; increasing the depth of the lazy expansion does not always result in an increase in the possible string lengths.

For instance, expanding a non-terminal one additional depth may produce two non-terminals, which does not increase the possible size of the string until the two non-terminals are later expanded.

In Figure 3 we also show the running times of the model expansion and inference for the infinite grammar. The running times are dominated by the factored inference and not the lazy expansion of the model. Given the known limitations of VE, this result is expected but still reasonable considering that factored inference on an infinite grammar is otherwise impossible. As a comparison, we also queried the infinite PCFG using the MH and importance sampling algorithms. After 100 seconds of sampling via MH, the estimated probability of the query string is 0.001, well outside the probability bounds computed using the same amount of time for the lazy VE (~depth of 21). MH had also created more than 5 million elements, and we expect it would eventually run out of memory. The importance sampler did not even produce an estimate, as once the sampler generates a world with infinite expansion, it will never terminate.

Finally, we show how the length of the evidence and the query can affect the performance of LFI. In Figure 4, we varied the length of the query and evidence strings from one to four while keeping the expanded depth of the model fixed at 25 (the length of the query was fixed at one when the evidence was varied and vice--versa). As the query length increases, the bounds increase for the infinite grammar, with no noticable change in the finite grammar bounds. On the contrary, when the length of the evidence increases, the bounds on the query significantly increase for both the finite and infinite grammar. When the depth of the expansion is fixed, it becomes increasingly unlikely that the evidence is found in the possible generated strings at that depth, and therefore, more probability mass is assigned to unexpanded possibilities resulting in loose bounds.

7. RELATED WORK

Though lazy evaluation and execution has a long history in computation, particularly as a feature of functional programming, there is little prior work that applies this method to inference in probabilistic graphical models. For example, lazy evaluation of game trees using the alpha-beta algorithm allows computation of a potentially infinite search space (Hughes, 1989). In (Kiselyov & Shan, 2009), a domain specific language for probabilistic programming is embedded in OCaml, using continuations to represent a stochastic computation as a lazy search tree. The tree is traversed depth first and the probabilities of query values accumulated in a table used by the inference algorithms. IBAL (Pfeffer, 2001) provides an algorithm for solving infinite probabilistic models in PP with finite observations and also makes use of laziness to evaluate queries on infinitely large models. However, IBAL's approach only works if the evidence guarantees that only a finite part of the model can be constructed, working in a manner similar to natural language algorithms on finite strings. (Pfeffer & Koller, 2000) propose a scheme for inference with recursive probabilistic models, but it is not computationally expressed. None of these approaches use the structure of the model to determine the relevance of unexpanded variables and provide bounds on queries.

There is also a body of work related to achieving more efficient inference in Bayesian networks by exploiting the structure of the graphical models to prune irrelevant nodes and manipulate the possible factorizations (Pearl, 1988; Shachter, 1988; Zhang & Poole, 1994; Baker & Boult, 1990b). As we have discussed, our LFI approach builds on and extends these concepts in a lazy way.

Our work is also related to methods for providing bounds to BP algorithms (Mooij & Kappen, 2008; Ihler, 2012). However, the methods are very different, as they use (0,1) bounds on messages at the leaves whereas we use the structure of the program to determine when the unexplored computation is relevant. Finally, computation of probability bounds in probabilistic graphical models has also been explored through other approaches. For example, in (Wexler & Meek, 2008), the multiplicative approximate inference scheme (MAS) is presented as a bounding algorithm for probability of evidence.

8. CONCLUSIONS

In this paper, we have presented an algorithm for LFI in PP, making factored inference a viable framework for full-fledged PP. LFI leverages the fact that not all variables in a probabilistic model are relevant to a particular query and provides bounds on the query probability by only exploring the most relevant portions of the model. We have provided a basic algorithm, and several optimizations to improve efficiency and accuracy. Experimental results using an implementation of LFI in Figaro demonstrate the potential of this approach for providing tractable, factored algorithms for PP.

One optimization is to reuse work between iterations of expansione, since most of the factors are the same in successive iterations. Also, we would like to explore the BP implementation more fully. First, can we provide any bounds, e.g., of convergence to the Bethe free energy? Second, how does the algorithm behave in practice? Our work is a starting point, and we hope many optimizations and refinements will come in the years ahead.

References:

Baker, M.& Boult, T. E. (1990a). Pruning bayesian networks for efficient computation. In *Uncertainty in Artificial Intelligence (UAI).*

Baker, M.& Boult, T. E. (1990b). Pruning bayesian networks for efficient computation. In *UAI*, 225-232.

Dechter, R. (1999). Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113, 41-85.

Goodman, N. D., Mansinghka, V. K., Roy, D., Bonawitz, K., and Tenenbaum, J. B. (2008). Church: A language for generative models. In *Uncertainty in Artificial Intelligence*.

Greibach, S. A. (1965). A new normal-form theorem for context-free phrase structure grammars. *Journal of the ACM (JACM)*, 12, 42-52.

Hastings, W. K. (1970). Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57, 97-109.

Herbrich, R., Minka, T., and Graepel, T. (2006). Trueskill# 8482;: A bayesian skill rating system. *Advances in Neural Information Processing Systems (NIPS)*, 569-576.

Hughes, J. (1989). Why functional programming matters. *The computer journal*, 32, 98-107.

Ihler, A. T. (2012). Accuracy bounds for belief propagation. *arXiv preprint arXiv:1206.5277*.

Kiselyov, O.& Shan, C.-c. (2009). Embedded probabilistic programming. In *Domain-Specific Languages*, 360-384.

Koller, D., McAllester, D., and Pfeffer, A. (1997). Effective bayesian inference for stochastic programs. In *National Conference on Artificial Intelligence (AAAI)*.

McEliece, R. J., Mackay, D. J., and Cheng, J. F. (1998). Turbo decoding as an instance of pearl's belief propagation algorithm. *IEEE Journal on Selected Areas in Communication*, 16, 140-152.

Metropolis, N., Rosenbluth, A. W., and Rosenbluth, M. N. (1953). Equations of state calculations by fast computing machines. *Chemistry and Physics*, 21.

Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D. L., and Kolobov, A. (2005). Blog: Probabilistic models with unknown objects. In *Proc.~19th International Joint Conference on Artificial Intelligence*, 1352-1359.

Minka, T. P. (2001). Expectation propagation for approximate bayesian inference. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, 362-369.

Mooij, J. M.& Kappen, H. J. (2008). Bounds on marginal probability distributions. In *NIPS*, 4, 3.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: Networks of plausible inference*. San Mateo, CA: Morgan Kaufmann.

Pfeffer, A. (2001). Ibal: A probabilistic rational programming language. In *International Joint Conference on Artificial Intelligence*.

Pfeffer, A. (2012). Creating and manipulating probabilistic programs with figaro. In *Workshop on Statistical Relational Artificial Intelligence (StarAI)*.

Pfeffer, A.& Koller, D. (2000). Semantics and inference for recursive probability models. In *AAAI/IAAI*, 538-544.

Shachter, R. D. (1988). Probabilistic inference and influence diagrams. *Operations Research*, 36, 589-604.

Wexler, Y.& Meek, C. (2008). Mas: A multiplicative approximation scheme for probabilistic inference. In *NIPS*, 1761-1768.

Winn, J. (2008). Infer.net and csoft. In *NIPS 2008 Workshop on Probabilistic Programming*.

Zhang, N. L.& Poole, D. (1994). A simple approach to bayesian network computations. In *The 10th Canadian Conference on Artificial Intelligence*, 171-178.