Contents

1 Introduction

Probabilistic programming languages are generalizations of programming languages, in which procedures are replaced with random procedures that induce distributions. By allowing for easy description and manipulation of distributions, probabilistic programming languages let one describe classical AI models in compact ways, and provide a language for very richer expression.

A core operation of probabilistic programming languages is inference, which is a difficult computational problem in general[2]. However, Markov chain Monte Carlo (MCMC) methods converge quite quickly to the posterior distribution for a large class of inference problems. Much effort has been devoted to studying in which cases this inference is efficient, and how to make it so.

We explore implementations of a programming language much like Church [4]. A naive implementation of inference uses a database to store all random choices [5], and re-performs all computations on each transition.

To improve this, we introduce a "traces" data structure, letting us represent the program structure in a way that allows us to do minimal re-computation. During inference, we make one small change and propagate the changes locally.

Unfortunately, the trace structure takes up significantly more space than the random database. Thus we introduce "reduced traces", which takes the same amount of space (asymptotically) as the random database. While it is asymptotically slower than traces for some programs, we demonstrate that the time complexity is similar in practice.

We also remark on another significant advantage of reduced traces. While writing interpreters in Python is typically considered slow, the PyPy translation toolchain aims to let developers compile their interpreters into lower level languages, such as C. Furthermore, one can use various hints to generate a tracing JIT. Generating a tracing JIT for inference could potentially improve efficiency greatly, and reduced traces make this far easier to implement.

Thus far, in this project, we have done the following:

1. Explore the theoretical aspects of the traces and reduced traces inference algorithms.
2. Implement the different inference engines, in RPython, so as to have C versions of the interpreter. All engines conform to a common REST API. [1]

[1]Server code here: https://github.com/WuTheFWasThat/Church-interpreter

3. Run various benchmarks on the engines to compare the engines' performance on various problems.

There are many research directions to pursue after this, all towards the goal of improving performance of the inference engine:

1. Generate a tracing JIT version of the interpreter.

2. Creating a parallelized version of reduced traces.

3. Potentially exploring some "adaptive" generalizations of Gibbs sampling.

We would ultimately like to recover the efficiency of special-cased inference algorithms for topic-modeling, mixture modeling, and Hidden Markov Models, in a much more general setting, and with far fewer lines of code.

2 Language description

2.1 Values and expressions

A **value** is a valid binding for a variable, in our language. For example, integers, floats, and booleans are valid values. We call these primitive values. Another type of value is a **procedure**, a function which gets applied to a tuple of argument values and returns another value. These procedures may, of course, be non-deterministic.

Our language contains some default procedures, from which we can then build up more complicated ones. In fact, as we all see later, the types of procedures we allow will end up being more general than that which is typically thought of as a procedure.

Expressions are syntactic building blocks. Expressions are **evaluated**, so that they take on some value. Their evaluation is relative to an **environment**, in which variable names may be bound to values. All expressions are one of the following:

- A name, e.g. x or y, which refers to a value bound in the current environment.
- A constant value, e.g. 3, 3.1415, True, or a default procedure.
- A lambda expression, i.e. $(\lambda\ (x_1 \ldots x_n)$ body_expression), which evaluates to a procedure taking n arguments.
- An application, i.e. $(f\ x_1\ x_2\ \ldots\ x_n)$, where f is a procedure taking n arguments.

We will see examples of the syntax later. For those familiar with Scheme [1] [8], it should be quite familiar.

2.2 Directives

There are three basic operations, called **directives**, which the language supports. They are:

- `PREDICT [expr]`

  Evaluates with respect to the global environment to sample a value for the expression `expr`.

- `ASSUME [name] [expr]`

  Predicts a value for the expression `expr`, and then binds `name`, a string variable name, to the resulting value in the global environment.

- `OBSERVE [expr] [val]`

  Observes the expression `expr` to have been evaluated to the value `val`. Runs of the program should now be conditioned on the expression evaluating to the value.

There are many other commands. But by far the most interesting and important one is inference.

- `INFER [rerun] [iters]`

  Runs `iters` steps of the Markov Chain, also deciding whether to first rerun the entire program. The result of running sufficiently many iterations from the prior should be that sampling gives the posterior distribution for expressions, conditioned on all the observed values. Restarting from the prior will not be typically used, but is useful for performance comparisons.

Other commands do various useful things like report various benchmarks (including time, space, and entropy use), help us gather statistics, allow seeding of the underlying PRNG, and deleting parts of the program or clearing it entirely.

3 Inference

Inference is by far the most complicated of the directives, and also the most important. Performing inference is the reason we are interested in probabilistic programming languages in the first place.

3.1 First try

Of course, the most obvious way to implement inference is to use rejection sampling. But this scales poorly with the number of observations. We'd like to instead run Metropolis-Hastings [6] on the space of all possible executions of the program.

Our proposal density, which chooses the new state (program execution history) to potentially transition to, does the following:

1. Pick, at random, one of the random choices made, at any point in the program's execution history. Make a new choice for this randomness.

2. Rerun the entire program, changing only this choice. If this causes us to evaluate new sections of the code (e.g. a different side of an if statement, or a procedure called with different arguments) with new choices of randomness, simply run these new sections freshly.

We then use the update rule of Metropolis-Hastings to decide whether we enter this new state, or whether we keep the old one. However, if the new execution history doesn't agree with our observations, we assign it a probability of 0, so that we never transition to such a state.

3.2 Adding noise to observations

Unfortunately, as it has been described, this random walk doesn't converge to the posterior! Consider the following example, in which I flip two weighted coins and observe that they came up the same:

```
ASSUME a (bernoulli 0.5)
ASSUME b (bernoulli 0.5)
ASSUME c (^ a b)
OBSERVE c True
```

It is easy to verify that the posterior distribution should have the state {`a:True, b:True`} with probability $\frac{1}{2}$ and {`a:True, b:True`} with probability $\frac{1}{2}$. However, we can see that if we start in the state {`a:True, b:True`}, we will never walk to either of the two adjacent states {`a:False, b:True`} and {`a:True, b:False`}. So the random walk stays in the initial state forever! The same argument applies if we start from {`a:False, b:False`}. Thus this Markov chain does not mix properly.

To fix this problem, we should allow some small probability of transitioning to states where observations are false. After all, a Bayesian should never trust their eyes 100%! Suppose instead of

```
OBSERVE [expr] [val]
```

we instead used:

```
OBSERVE (bernoulli (if (= [expr] [val]) 0.999 0.001)) True
```

Now, when we run our walk, if the expression `(= [expr] [val])` evaluates to `False` (so the original observation would've failed), we force the outermost `bernoulli` application to be `True`. If the noise level is small, this should not affect results much, since the execution histories in which the original observation was false are weighted against heavily. However, the space of states is always connected in the random walk and our Markov chain will converge properly.

For convenience, our language uses the function `(noisy [observation] [noise])` as syntactic sugar for `(bernoulli (if [observation] (- 1 [noise]) [noise]))`, where you have observed some expression `[observation]` to be true.

3.3 A short example

Let's start with a simple example, to get familiar with our language. We will see how easy it is to write simple Bayes' nets, in our language (and in fact, it is easy to write much larger ones as well).

```
>>> ASSUME cloudy (bernoulli 0.5)
id: 1
value: True
>>> ASSUME sprinkler (if cloudy (bernoulli 0.1) (bernoulli 0.5))
id: 2
value: False
```

Initially, we have this model, and we know nothing else. So our prior tells us that there's a 30% chance the sprinkler is on.

```
>>> INFER sprinkler 10000 10
False: 6957
True: 3043
```

We now look outside, and see that the sprinkler is in fact on. We're 99.9% sure that our eyes aren't tricking us.

```
>>> OBSERVE (noisy sprinkler .001) True
id: 3
```

Now, we haven't yet looked at the sky, but we have inferred something about the weather.

```
>>> INFER cloudy 10000 10
False: 8353
True: 1647
```

This is quite close to the correct value of $\frac{5}{6} = 0.8\overline{333}$ False, which is what we'd get if there was no noise in our observation. However, there is still a number of worlds in which the sprinkler was actually on. In those worlds, the weather was more likely to be cloudy. So we should expect error on the order of 0.001, but more error comes from our small sample size.

4 XRPs and Mem

Up until now, we have neglected to mention a crucial aspect of our language. The procedures allowed in our language are much more general than what we typically think of as procedures. In this section, we introduce the most general type of procedures we can allow inference over.

4.1 Exchangeability

Evaluating and unevaluating sections of code is very easy when all random choices are made independently of one another. But actually, it is easy to evaluate and unevaluate so long as the sequence of random choices is **exchangeable sequence**, meaning, roughly speaking, that the probability of some set of outcomes does not depend on the order of the outcomes. In particular, as long as it is safe to pretend that the section of code we're unevaluating was the last section of code evaluated, we're fine! For more background, see [3].

Thus instead of merely being able to flip independent coins as a source of randomness for procedures, the most general thing we are able to allow what is called an **exchangeable random procedure (XRP)**. An XRP is a type of random procedure, but different applications of it are not guaranteed to be independent; they are guaranteed merely to be an exchangeable sequence. That is, the sequence of `(arguments, result)` tuples is exchangeable. The probability of a sequence is independent of its order (assuming we got the arguments in that order).

For example, if f is an XRP taking one argument, we may find that $f(1) = 3$, and later that $f(2) = 4$. Unlike a typical procedure, the result $f(2) = 4$ was not necessarily independent from the earlier result $f(1) = 3$. However, the probability that this execution history happens should be the same as the probability that we first apply $f(2)$ and get 4, and then later apply $f(1)$ and get 3. Of course, $f(2)$ can be replaced with $f(1)$ everywhere in this paragraph, as well — multiple applications to the same arguments can also be different and non-independent, so long as they are exchangeable.

4.1.1 Formally specifying XRPs

XRPs used in our probabilistic programs are specified by the following four functions:

- `initialize()`, which returns an initial state.
- `sample(state, args)`

  Samples a new `value`, applying to the `args`, given the current `state`.
- `incorporate(state, value, args)`.

  Incorporates an application to `args` that resulted in the `value`, returning a new `state`.
- `remove(state, value, args)`

  Undoes an incorporate, taking out the application to `args` resulting in `value`, and returning a new `state`.
- `unsample(state, value, args)`

  Undoes a sampling. It's purpose is to "undo" any randomness or state change used in sample. By default, it does nothing, and few XRPs use this. Returns the new `state`.
- `prob(state, value, args)`

  Returns the marginal (log) probability of returning the value, if the XRP in its current state is applied to the arguments.

The state of the XRP is a sufficient state, meaning it alone lets you determine the XRP's behavior. Minimally, the state may be a history of all previous applications of the XRP, although in most cases, it is much more succinct. Whenever we apply the XRP, we use `sample` to determine the resulting value, and we may use `prob` to determine the probability that we got that value, conditioning on all previous applications. This value can then incorporated into the XRP using `incorporate`, and the state is updated. Finally, `remove` lets you remove a value, undoing an incorporate.

Notice that if applications are independent, then we do not need a state, and both incorporate and remove can do nothing. Thus this recovers the special case of normal random procedures.

XRPs are quite a general object, and replacing random procedures with them significantly increase the expressive power of our language. They also come in a variety of flavors, and we will see some examples later.

4.1.2 Re-apply or re-score

If our XRP is capable of returning any value in its range, regardless of its arguments, then we will say it is a "rescoring" XRP. That is, if we are dynamically changing the program, and the XRP is passed new arguments but wants to retain its old value, we can do so, but perhaps with a modification to the probability.

If our XRP is not capable of this, we call it a "reapplying" XRP. That is, it should reapply these new arguments to get a new value as well.

Essentially, in inference, when values change, and we are propagating the changes upwards, reapplying XRPs will "pass on" the changes, whereas rescoring XRPs will "absorb" them.

Now, we can allow for something much more general than the noisy observations above. We now simply require that all observed expressions have an outermost rescoring/absorbing XRP application. When running inference, we always force the XRP to give the observed value by using `inc` (without using `sample`). The probability of getting this observed value should always be non-zero.

The noisy expressions obtained with `noisy` are a special case of this, so long as `noise` was non-zero.

Also, notice that we could make all XRPs re-scorers, by adding noise to their output. That is, we could have all XRPs have an internal error rate, as if `noisy` were being applied.

4.2 Mem

A special XRP worth noting is `mem`, which lets us memoize procedures. Applying `mem` to a procedure returns another XRP, which is a memoized form of that procedure. That is, when using a `mem`'d procedure, we never reapply randomness, given old arguments. Thus whenever we apply the memoized procedure once to some particular arguments, applying it again to those arguments will result in the same value.

Implementation of `mem` is extremely tricky, and we will decline to work out its details in this paper. However, it is important to understand its utility. As a simple example, consider a naive implementation of Fibonacci:

```
ASSUME fib (lambda (x) (if (< x 2) 1 (+ (fib (- x 1)) (fib (- x 2)))))
```

Of course, evaluating fibonacci in this manner is an exponential time operation. Evaluating `(fib 20)` took nearly 2 seconds.

Simply wrapping a `mem` around the lambda expression will make it so we don't need to recompute `(fib 20)` the next time we call it. Not only that, but when we recurse, we will only compute `(fib [x])` once, for each $x = 0, \ldots, 20$.

```
ASSUME fib (mem (lambda (x) (if (< x 2) 1 (+ (fib (- x 1)) (fib (- x 2))))))
```

Evaluating `(fib 20)` now takes well less than a millisecond.

5 Traces

The random database implementation of inference is not very interesting. The inference engine simply reruns the program after each iteration, remembering all previous randomness chocies, and having only one choice changed. Let's skip to the "traces" implementation.

5.1 Basic overview

In the traces implementation of inference, we create an **evaluation node** for every expression that is evaluated, which keeps track of the expression, environment, values, and many other things.

The evaluation nodes are connected in a directed graph structure, via dependencies. Thus if the evaluation of one node, x, depends on the evaluation of another (e.g. a subexpression, or a variable lookup), y, we say that y is a **child** of x and that x is a **parent** of y. By interpreting these child-parent relationships as edges (from parent to child), these evaluation nodes form a directed acyclic graph, which we refer to as the **trace**.

Each node caches the relative locations and values of all its children. Furthermore, if it is an application node, we also cache the procedure and arguments. And for every node, we cache its most recent value.

The **trace** is the data structure which contains all the nodes and their dependencies, the environments in which the nodes are evaluated, and a data structure storing XRP applications.

5.2 Propagation

How does propagation actually work? This question is surprisingly interesting, and we explore two different candidate propagation schemes.

5.2.1 Recursive propagation

The easiest version of propagation to implement is simply a recursive, "depth-first" implementation. We simply start at the node which we reflipped, and recursively re-evaluate anything which depends on it, stopping only when we reach XRP application nodes. In pseudocode:

```
function propagate(node):
  node.evaluate_based_on_children()
  if node is an assume node:
    for all node' which reference node:
      propagate(node')
  if node has a parent:
    propagate(parent)
```

Although this is how I implement propagation, there is one crucial optimization to be made. Suppose we just propagated to the argument to a procedure. When we propagate up, we will re-evaluate this procedure application. However, instead of re-evaluating the entire body, it is sufficient to merely change the environment to reflect the new value of the variable corresponding to the changed argument.

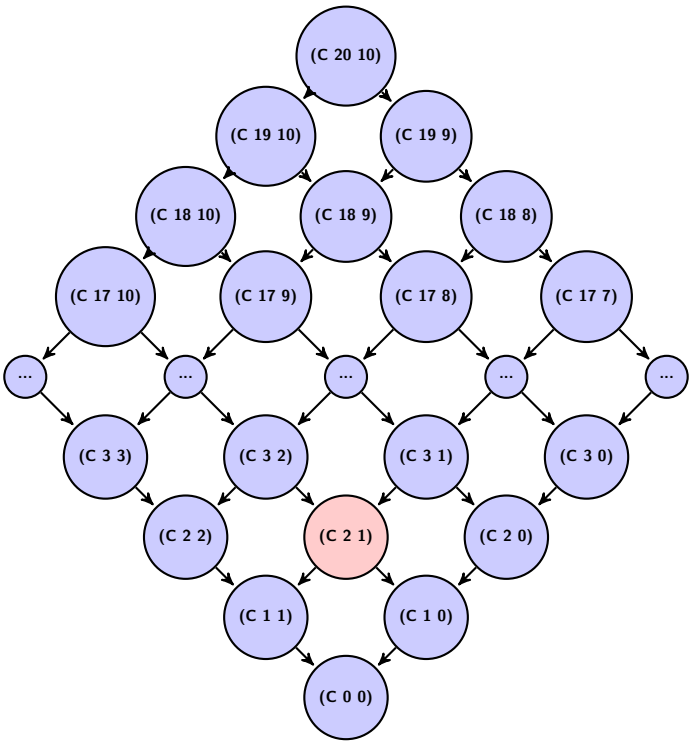
There is one big problem with recursive propagation. We do not guarantee that we only propagate each node once. In fact, this propagation has exponentially bad worst-case scenarios, and can be poor in some practical settings.

Suppose we have the following program, which computes a familiar multivariate recurrence.

```
ASSUME binom (lambda (n k) (if (or (= k 0) (= k n)) 1
...                            (+ (binom (- n 1) (- k 1)) (binom (- n 1) k))))
```

Let's say we want to randomly replace one of the entry $n = 2, k = 1$ of Pascal's triangle with a random real number between 0 and 2. And we are curious in the resulting distribution for the entry $n = 20, k = 10$.

```
ASSUME binom-wrong (lambda (n k c) (if (and (= n 2) (= k 1)) c (binom n k)))
PREDICT (binom-wrong 20 10 (uniform-continuous 0 2))
```



Propagating from the red node upwards. Here the C function is `binom-wrong`.

Now suppose we decided to reflip the `uniform-continuous` application. And suppose we have the policy of always propagating to the right first, if possible. We can then verify that the worst-case number of times we may propagate to the application `(binom-wrong a b)` is essentially `(binom a b)`.

Intuitively, we wanted to propagate to the smaller values first, in this modified Pascal's triangle. Then, we would only have needed to propagate about 100 times; once for each node. Thus we see that the order is extremely important, in general.

However, although the worst case scenario is terrible, this method of propagation is okay in practice for many problems.

5.2.2 Full location

So, we see is that the order of propagation is quite important. How can we know which parents to propagate our values to first? Intuitively, since our trace structure is a DAG, we simply propagate to those "lowest" in the DAG. Essentially, we want to maintain a priority queue, from which we will repeatedly remove the minimum and add his parents to the queue, until the queue is empty. Here, we are simply interpreting the parent-child relationships as comparisons in a partial ordering.

```
function propagate(node, q):
  q = priority_queue(node)
  while q is not empty:
    node' = q.pop_min()
    propagate_and_add(node', q)

function propagate_and_add(node, q):
  node.evaluate_based_on_children()
  for all node' which reference node:
    q.add(node')
  if node has a parent:
    q.add(parent)
```

Unfortunately, a heap does not work correctly on a set of elements with only a partial ordering (with no guarantee on how ties are broken), and so we must extend to some full ordering. So what information can we use to do comparisons?

We cannot simply use timestamps of the initial evaluation, since it is possible for unevaluated branches earlier in the program to be evaluated later.

We can use a full description of the node's location (something like a call stack), including the ID of the top directive node. Unfortunately, this can be quite expensive, as comparisons would now take $O(h)$, where h is the height of the DAG, defined to be the length of the

longest directed path starting from a directive node.

Is it possible to do something more efficient? We believe we can do the following optimization, which has not been implemented: Consider the full path from a directive node to the node in question. Remove the edges which correspond to being the computation of an application node (the application of a procedure), i.e. the link between an application node and its child which isn't the arguments or procedure. We now have a set of path segments. Simply store the ordered list of their lengths.

5.2.3 Analysis

As described above, the trace is not static across different runs of the same program. When referring to this dynamic data structure, we will say, the **dynamic trace**. We may think of the dynamic trace as a random variable, where the randomness is over the different runs of the program.

For every node x, we say the **envelope** of x, denoted V_x, is the set of nodes reached by propagating upwards in the dynamic trace until we reach "re-scoring" XRP application nodes (including the application nodes, as well as x), and then propagating down through the bodies of any procedure applications we pass through. We think of V_x a random variable as well, but it is only defined over worlds in which node x is active in the dynamic trace.

Now let p_x be the probability that node x is chosen to be reflipped in the Markov chain. In other words, p_x should represent the distribution where, first, we sample from a run of the program, and then we pick one of the XRP applications uniformly at random. Lastly, we let V be the random variable which first picks some x with probability p_x, and then picks a random V_x, conditioned on x being active.

We claim now that the expected propagation time of the "full location" propagation scheme is

$$E_V[h|V| \cdot h \log |V||] = h^2 \cdot E_V[|V| \log |V|].$$

Most terms follow directly from our propagation algorithm. For each of V things, we must add and pop it from the queue in $h \log |V|$, as well as do $O(1)$ work of evaluation. It's true that we un-evaluating as well as evaluate, but this introduces just a factor of 2, and we can "amortize" by considering the un-evaluation to always happen right away (after evaluation, instead of before).

So where did the second factor of h come from? This hidden complexity comes from variable lookups. It may take us $O(h)$ to find the environment containing the value of a variable. This hidden cost exists in our current implementation, but can perhaps be alleviated by using path-compression and sacrificing memory usage.

Another hidden complexity may come from complexity internal to XRPs. For the sake of analysis, we have assumed (erroneously, in practice) that all XRP operations are $O(1)$.

In the greedy version, we've shown that the worst case is essentially

$$E_W[h|W^W|],$$

in the worst case.

6 Reduced Traces

6.1 Motivation

The major downside of the traces implementation is that it is not memory efficient. The number of nodes stored is essentially proportional to the time it takes to compute the original program, since we keep a node for each expression evaluation.

To highlight this, consider the following program, which samples from the categorical distribution, which returns i with probability p_i. The distribution is represented by some procedure `categorical-dist`, which takes an argument i and returns p_i. It is guaranteed that for some finite n, $\sum_{i=0}^{n-1} p_i = 1$

```
ASSUME sample-categorical-loop
        (lambda (v i)
                (if (< v (categorical-dist i))
                    i
                    (sample-categorical-loop
                     (- v (categorical-dist i))
                     (+ i 1))))

ASSUME sample-categorical (lambda ()
                            (sample-categorical-loop (rand) 0)
```

Notice that using traces, we will create a node for each application of the sub-loop. And yet it is clear that this space is un-needed. We feed the function a single choice of randomness, and it creates up to n nodes. But if we were to reflip the application of `rand`, we would need to re-evaluate the entire application again. Thus if n is large, this could be a phenomenal waste of space. Supposing a loop like this were on the critical path of some program, we could be using a multiplicative factor of $O(n)$ more space.

One solution is to "collapse" this sampling into an XRP. That is, we turn `sample-categorical` into an XRP, since applications of it are independent, and thus exchangeable. This is a useful idea which we will turn back to later, but this is clearly not a satisfactory solution in general.

6.2 Reducing the traces graph

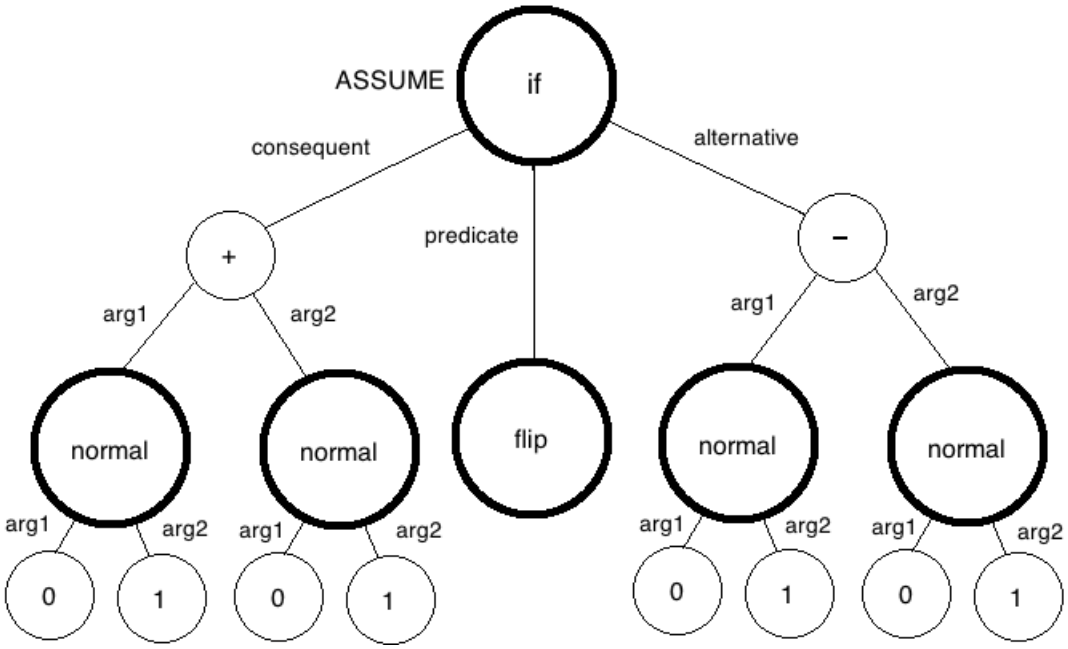
The idea of reduced traces is to essentially "contract" all edges of deterministic computation in the traces graph. Whenever we need to perform deterministic computation, we will simply recompute, instead of having nodes all along the way with cached values.

In contracting all these edges, the only nodes which remain will be nodes corresponding to XRP applications, and nodes which correspond to directives [2].

To illustrate, let's consider the following simple program:

```
ASSUME a (if (flip)
             (+ (normal 0 1) (normal 0 1))
             (- (normal 0 1) (normal 0 1)))
```

Let's now draw the trace structure corresponding to this program.



Circles correspond to nodes which would exist in a traces data structure, but only the large circles correspond exist in the reduced traces data structure. We can see that here, the number of nodes has reduced from 16 to 6. In general, the reduction can be arbitrarily large.

[2]There are also nodes corresponding to applications of `mem`, in the standard implementation of `men`. However, this is not necessarily always the case

6.2.1 Naming locations

One issue which comes up when implementing reduced traces is the naming of child node locations. Let's revisit the example above. In regular traces, the top node would have 3 children, corresponding to

1. consequent
2. predicate
3. alternative

In reduced traces, however, the top node has 5 children, corresponding to the following lists:

1. [consequent, arg1]
2. [consequent, arg2]
3. [predicate]
4. [alternative, arg1]
5. [alternative, arg2]

These locations are important for being able to recover a specific child node, in order to continue computations while propagating up, or while unevaluating. Unfortunately, these lists can be arbitrarily long, if there is a lot of deterministic computation in-between nodes, taking up space. Furthermore, the lists must be copied, in order to propagate the location values to children, taking up time. Thus this is an algorithmic disaster.

The solution is to use a rolling hash on these lists (first converting them into numbers, if necessary). We want the property that the hash can be computed for the list even if we don't know an entire prefix, so long as we have the hash of the prefix. In our implementation, we use the standard Rabin-Karp hash[7].

Now, all the top node will remember is a dictionary with 5 random-looking integer keys, mapping to 5 children nodes. We have effectively pinned down the relative location of a node within the computation, taking up minimal space, and with rather strong guarantees against collision.

This technique is also used in the random database implementation of the engine.

6.3 Propagation in reduced traces

In reduced traces, we essentially have the same options as in regular traces. In fact, the pseudocode is identical to that of traces, as outlined in Section 5.2. However, the subroutine `evaluate_based_on_children` is no longer as straightforward.

First, we should store a copy our children dictionary. Then, we must evaluate our expression as normal, but stopping the recursion whenever we reach what should be a new node. If that node is in our child dictionary, we can use its cached value. Otherwise, we should evaluate it newly. At the end, we evaluate anything which we didn't see in our old child dictionary.

One subtlety which comes up here, is how to name locations when applying functions. The easiest thing to do, and indeed what our implementation does, is to simply create a random ID for each XRP and procedure created. However, this causes us to need to repeat calculation any time we are dealing with a procedure or XRP. This is quite bad; much worse than not doing the changed-argument optimization we could do in traces.

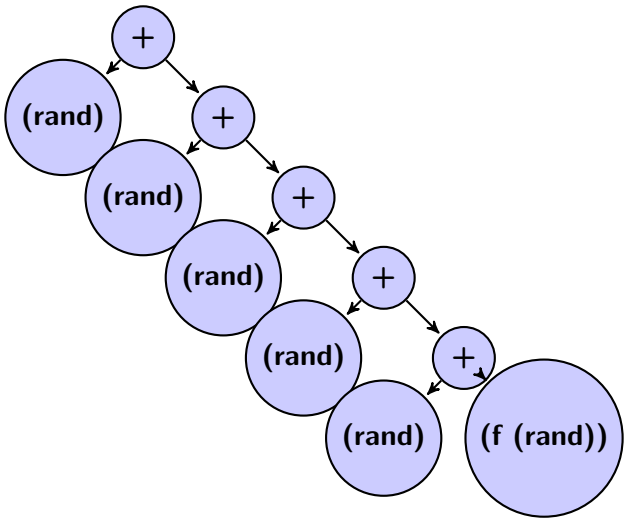
Luckily, we believe this can be fixed. Essentially, when evaluating, we should always keep track of the environment. Thus our reduced trace structure reflects all values of nodes, modulo environments of procedures. The environment is instead kept at the node separately (it still exists in memory, but isn't factored into the location). And of course, we should now hash the body code of procedures and use those for our location naming.

Though this takes care of the majority of our pains, we will pay a price for our lack of structure. Let's consider the following program, which indexes into an array using the max of two samples from the categorical distribution from the last example:

```
ASSUME categorical-sum (+ (sample-categorical) (sample-categorical))
```

Now, each time we reflip the coin at the bottom of some `sample-categorical`, we must re-evaluate both applications of sample-categorical, in order to propagate up to `x`'s value. Even in the analysis given for the static traces, evaluating this second argument was not necessary.

The structure of a reduced trace is different from that of the traces we saw in the previous section, and this can lead to drastically worse running time. For example, consider the following trace structure:



Now suppose `f` is an extremely expensive deterministic computation, so that the time it takes dominates all other runtime considerations. In regular traces, we only need to recompute `f` with probability 1 in 6, when its argument is reflipped. However, in reduced traces, we must recompute it every time! We no longer have values cached along the edges, and so we have traded in space complexity for some time complexity.

It's clear that we can generalize this example to a class of problems for which reduced traces does drastically worse. However, as we will see, the runtime of reduced traces is empirically comparable to that of traces.

6.4 Analysis

To analyze the propagation, we imagine we were still propagating in the old dynamic trace. Although our notion of nodes is now different, we are really interested in the amount of computation done (essentially, number of expressions evaluated).

Recall that we had defined the envelope to be the set of nodes reached by propagating upwards in a dynamic trace until we reached "re-scoring" XRP application nodes, and then propagating down through the bodies of any procedure applications we passed through. The set of nodes we evaluate is very similar, except that we additionally propagate downward from all other nodes with deterministic computation that we pass through, until we hit another application node (which has its value cached).

Viewing the picture with only reduced traces, the set of nodes we propagate to is extremely analogous to the notion of Markov blankets, and also captures explicitly a notion of conditional independence.

Let us call this set of nodes the **full envelope**, W_x. From the reduced trace point of view, we simply propagate to all parents, and then propagate down to their children which needed to be evaluated. However, it is the set of nodes from the traces point of view that we are interested in.

Again, we can consider the reduced trace to be dynamic, in this analysis, using the same sort of amortized analysis we had done earlier. Thus the runtime is

$$E_W[h^2|W| \cdot \log|W|],$$

where W is the random variable which first picks some x with probability p_x, and then picks a random full envelope W_x, conditioned on x being active.

And in the greedy version, the worst case is

$$E_W[h|W^W|],$$

analogously to before.

6.5 Engine Complexity Trade-offs Summary

To summarize, we have the following table of asymptotic complexities (we abuse notation and drop all expectations, which are implicitly over the random variables of the expression):

Table 1: Engine Complexity Trade-offs

	Time / Iteration	**Space**	**Time**
Random DB	T	H + P	T
Traces 1 (recursive)	$h \cdot V^V$	T $(+ H + P)$	T
Traces 2 (full location)	$h^2 \cdot V \log V$	T $(+ H + P)$	$h \cdot T$
Reduced Traces (recursive)	$h \cdot W^W$	H + P	T
Reduced Traces (full location)	$h^2 \cdot W \log W$	H + P	T

Here, the variables mean the following:

- P = Size of the program description (the directives and their expressions)
- T = Expected runtime of the original program
- H = Expected entropy (number of XRPs)
- V = Envelope size, in dynamic trace
- W = Full envelope size, in dynamic reduced trace
- h = The "height" of the program. That is, the largest number of evaluations any node is away from a directive.

Recall that $T > H$ and $T > P$, and $V < W$.

Though none of the traces implementation is strictly dominated by another, we believe the 1st and especially 3rd are more likely to be efficient in practice. And we believe that reduced traces should have much better space usage and only marginally worse run-time. See the appendix for some preliminary empirical results.

7 Extended XRPs

We now discuss some extensions to the XRP interface, to give the user more power over how inference works.

7.1 XRP weights

By default, all XRP applications are equally likely to be reflipped. But what if we would like to reflip some XRP applications more than others? This won't affect the resulting distribution, so long as we correctly calculate transition probabilities, but it could be helpful for making our chain mix much faster.

To achieve this, we will assign each XRP a weight, which tells us how often we should reflip its applications, relative to other ones. This is extremely useful for certain programs. However, in order to use this, we'd like a data structure supporting the following operations:

- Insert(key, weight) : Insert a key, along with a weight
- Remove(key) : Remove a key
- Sample() : Sample keys, with probability proportional to their weight.

Previously, we needed a data structure to support this in the special case where all weights were 1. In this case, a special augmented hash table can achieve these operations in $O(k)$, where k is the key size. The idea is to keep the keys in an array, as well as a hash table, and to keep a hash table from key to index in that array. When we delete an element, we swap it with the last item of the array and update everything appropriately.

Unfortunately, the weighted case is much more difficult. It is slightly reminiscent of the problem of implementing malloc. A simple solution is to do the following: Maintain the same augmented hash table as before, but also keeping track of weights. Also, keep track of the maximum weight currently held, using a heap. When we sample, use rejection sampling, by dividing by the maximum weight currently held.

Unfortunately, this sampling algorithm has complexity proportional to the ratio of the maximum weight to the average weight, which can get arbitrarily large. However, insertion and removal happen in $O(k \log n)$. Insertion and removal are much more important, as they can happen many times (during propagation), whereas sampling only happens once per round of iteration.

In fact, one can imagine having weights more finely controllable. For example, they could be different for particular applications on the same XRP, changing based on the XRP's internal state. They could even perhaps be given as an argument to the XRP application, and be themselves random variables in the program. This would let us do a certain form of inference over inference. These extensions have not yet been implemented or fully thought out, but would be one research direction to head in.

7.2 Application proposals

Suppose we have a program containing many applications of `(flip 0.9)`, in which, because of our observations, we expect common program executions to contain half true and half false.

In our standard proposal, it might take a while for this program to mix, simply because each iteration, we propose to have coins be tails far more often than we propose they be heads, even if they are already tails. Wouldn't it be better if we simply always proposed to turn over the coin, and rejected more often? Indeed, this seems better most of the time, but for many XRPs it makes less sense to do something like this. Furthermore, in some situations, it may actually be harmful to include this "optimization".

The solution is to let the user customize! Of course, this sort of customization is entirely optional, and would be potentially confusing if front-facing for a amateur probabilistic programmer. Essentially, we offer the following simple function which can be optionally implemented for any XRP:

- `application_mh_propose(state, args, val)`

  Returns a newly proposed value, a forwards probability (a contribution to the forwards term in the MH proposal Q ratio), and a backwards probability. Furthermore, if XRP applications are weighted, it returns any application weight removed, and any application weight added. Lastly, it returns a new state.

Notice the default implementation of this, in terms of the most primitive XRP applications, would be the following:

1. First remove the old value.
2. Then, score the old value to determine the backwards probability contribution.
3. Then, sample a new value, independently.
4. Then, score the new value to determine the forwards probability contribution.
5. Lastly, incorporate the new value.

After the MH proposal is evaluated, using the probabilities and weights returned to help calculate the acceptance ratio, we must be able to then restore or keep the old state. It is thus often convenient to explicitly implement a `restore` and `keep` function for XRPs, since the state is often kept internally. In this case, guarantee that after any `application_mh_propose` is called, exactly one of `restore` and `keep` is called (corresponding to whether we rejected or accepted).

7.3 State weights and state proposals

Memoization is typically not implemented through the usual XRP interface, despite its being an XRP. And in fact, implementing it as a true XRP would normally lead to some undesired behavior. For example, we would not be able to reflip individual bits of randomness within a memoized procedure application. Furthermore, we would only choose to re-flipping mem as often as any other XRP, despite its potentially owning much of the program's computation and randomness.

To fix this, we will add the following optional additions to the XRP interface:

- `internal_mh_propose(state, args, values)`

  Re-proposes part of any XRP's internal randomness, i.e. parts of its state. Returns a new list of values, a forward probability contribution, and a backwards probability contribution. Also returns any state weight added and state weight removed. And finally, it returns a new state.

- `restore(state)`, `keep(state)`

  These two functions are trivial functions. However, we include them to emphasize that, if the state is implemented internally in some object, it should allow for restoration and keeping after an internal MH propose, analogously to application proposal.

The above description of `internal_mh_propose` needs a bit more clarification. Firstly, we mentioned "state weight". This is fairly self explanatory. We are allowed to assign weights to the state of an XRP, in addition to weights to individual applications. This is particularly useful for a true-XRP implementation of mem, in which applications are all deterministic and thus have no weight, but the state should be internally repurposed often. We should intuitively set the state weight to be equal to the sum of the application and state weights which happen internally in all the mem's procedure application computation.

Another important thing to know is that both the input and output lists of values are ordered, and correspond to each other. Thus, when we feed the internal proposal all the old values, we know exactly which new value to give to each evaluation node that applied that XRP. We then propagate upwards from each one of these new nodes [3]. At the end of propagation, we determine the acceptance probability, and then restore or keep the XRP's internal state (as well as the corresponding set of values).

7.4 Links

Consider the following setup:

[3] This only works in the priority queue method of propagation (with full locations). In the greedy propagation method, there may be some unexplored subtleties in the order of propagation here, although in normal use cases it may also work...

```
ASSUME weight (uniform-continuous 0 1)
ASSUME coin (lambda () (flip weight))
...
```

Here, the remainder of the program uses coin extensively. In this scenario, if `weight` changes, `coin` becomes a different function, and we must propagate this change to all instances of `coin`. Not only that, but we will reflip each instance of `coin`, and propagate all those changes, even if the weight has barely changed.

We would like to provide a shortcut, by which we can tell all the `coin` applications that it has not changed very much, and that they can in fact retain their original flipped values. But in general, `coin` could've been something else, which changed completely in structure once `weight` changed, so that all applications indeed needed to be reflipped.

In another scenario, suppose we have an XRP which takes many arguments. When one argument changes, it can quickly adjust its state so that all its applications do not change. But how can we recognize this possibility and support this behavior?

Or what if the XRP outputs something somewhat different, but in a way such that the change can be passed on in a more intelligent way? In fact, this is exactly what happens when we implement mem as an XRP.

In the previous section, we dealt with changes to a particular mem'd procedure application. But what about changes to the mem'd procedure itself? In particular, consider the following example:

```
ASSUME cluster1-center (normal 0 1)
ASSUME cluster2-center (normal 0 1)
ASSUME cluster-point (mem (lambda (i) (if (= (mod i 2) 1)
                                          (normal cluster1-center 1)
                                          (normal cluster2-center 1))))
OBSERVE (cluster-point 1) 1.4
OBSERVE (cluster-point 2) -1.2
OBSERVE (cluster-point 3) 1.7
...
```

Suppose we are running inference, and we reflip `cluster1-center` (after all, this XRP should probably be weighted heavily). Now, the argument to the memoization XRP has changed. This means, we should remove and reapply the memorization application, resulting in a new version of cluster-point, all of whose applications must be propagated to. In particular, even the even-numbered cluster points will change in value!

However, typically, in traces, we can do an optimization here. Since only the environment of the procedure changed, and not the body itself, we can simply propagate that new value

within the body, neglecting to redo any computation. So here, for example, we would only need to propagate within the odd-numbered memoization application nodes.

To solve this problem (and potentially many other problems), we make explicit the notion of a "link". A link is a connection maintained from a "broadcaster" node to a "receiver" node, along which the broadcaster may send messages of the form "I changed in this manner". The receivers must create the links to the broadcaster, who will then broadcast his message to all receivers once he receives it. Typically, the message is simply "I changed my value to x".

Essentially, all propagation happens in this way. When parent nodes create children, they by default, contact them to become their receiver, and implement a protocol where they simply listen for any change in their value and then continue broadcasting these value changes.

We suggest the following: **Whenever there is a variable reference, we perform a lookup in the current environment, and a link is created to the node responsible for that variable setting. If it is the global environment, it will be an assume node. Otherwise, it should be the argument to a procedure application.**

Now, we will add the following stipulations to mem's implementation:

- When the memoizer is applied, the created XRP internally creates a new RIPL. The internal RIPL's global environment is simply the environment of the procedure that the memorizer is being applied to.
- When the memoized procedure is applied to new arguments, the internal RIPL creates a new node. This application node then creates a link to the internal node!
- When the internal nodes do lookups which reach the internal RIPL's global environment, they contact any node responsible for that assignment (just like normal variable references contact assume nodes, or the node of the argument producing the assignment). In this case, they traverse through the environment and find a node in the outer RIPL.

Considering the cluster example again, let's see what can happen when `cluster1-center` changes. First, it broadcasts to the applications of the procedure `cluster-point` where `cluster1-center` was indeed used. This gets passed on, up to the top of these applications, in the internal RIPL. This is then passed to the applications of `cluster-point`, which are passed the new memorized values. These application nodes then can broadcast their new values!

In general, nodes which take on XRPs as values could, could be linked to by all appearances of that XRP's application. To accomplish this, we simply allow the node which evaluated to the XRP to tell it about itself, and then add a hook in the XRP's `apply`, which lets other nodes identify that node and contact it. This could be useful for a variety

of reasons, since we could pass arbitrary messages to allow different protocols for propagation.

Lastly, this abstraction is highly useful for a seemingly unrelated problem. Suppose we are running a distributed version of our engine. What happens if something needs to access a variable in an environment stored on a remote computer? And what happens when that variable's value changes and needs to propagate? Making links explicit helps us think about the intercommunication required for this to happen.

7.5 And more...

There are many more extensions to the XRP interface which have yet to be explored much. As the user base for probabilistic programs grow, and the usage becomes more advance, demand will increase for more specialized control. Some potential other extensions to consider, which we haven't explored:

- We might let XRPs decide when to be a resampler and when to be a rescorer, depending on the value.
- One might hope we could allow XRP transitions that are forced to leave the resulting output values fixed. In this case, there would be no accept or reject step, we would simply sample from the desired distribution. A side effect of this is that weights for an XRP implementing such a feature cannot depend on internal state.
- Oftentimes, we score a sequence of XRP applications by taking the marginal probabilities and incorporating them in one by one. One might imagine a function which helps efficiently computes scores entire sequence of applications.
- We could break out of the Metropolis-Hastings paradigm and try to do an "enumerative Gibbs".
- And much more...

And there are also open problems to address, such as

- There are some slight downsides to the internal RIPL version of mem. Although the design is both more natural, and usually more flexible (as different parts of the XRP API can be easily changed), it can't support recursive or mutually recursively memoized procedures.
- In a model where things independently broadcast to each of their receivers, it seems very difficult to use a priority queue method of propagation.

8 Conclusion and future work

The fact that reduced traces makes it so easy to identify the envelope-like sets is potentially extremely useful.

Firstly, it should be easier to work with for parallelization than other implementations. For parallelization, we'd like to do inference on different parts of the program in parallel. Conditional independence is precisely the property we'd want to be true of the different parts.

Secondly, it should be much easier to generating a tracing JIT for, at the right level of granularity. Essentially, we would like to consider inference loops which have similar structural behavior. Since the randomness choices are sufficient to determine what the computation will be like, all the deterministic structure in-between nodes which is made explicit in the traces data structure is irrelevant. This makes reduced traces much more convenient for implementing a tracing JIT.

These two factors lead us to believe that reduced traces will be crucial in developing even faster versions of this inference engine.

Ultimately we would like to be able to recover the efficiency of special-cased inference algorithms for topic-modeling, mixture modeling, and Hidden Markov Models, in a much more general setting, and with far fewer lines of code.

We're already making large strides towards this goal. In addition to the work mentioned, the Probabilistic Computing group at MIT has developed much shared infrastructure and benchmarking suites, and has made some demos showcasing our engines' ability to perform.

However, with parallelism and JITing still in baby stages, there is undoubtedly much more to be done. Reduced traces were not conceived of or implemented until recently, but will hopefully contribute to the goal of fast, general-purpose inference.

9 Acknowledgements

Besides my advisors Vikash Mansinghika and Josh Tenenbaum, I should acknowledge Tejas Kulkarni, Ardavan Saeedi, Dan Lovell, and especially Yura Perov for their work on the MIT Probabilistic Computing Project in conjunction with me, providing valuable discussions and developing useful infrastructure.

A Appendix A: Empirical results

In this section, we examine the time and space complexity of our various engines, on various classic inference problems. We simultaneously showcase the flexibility and generality of the language, as well as the ease of defining models.

You will notice that we have comparisons between Python and C. This was possible because the engine was written in RPython, a subset of the Python language which can be

translated (by the PyPy toolchain) into C.

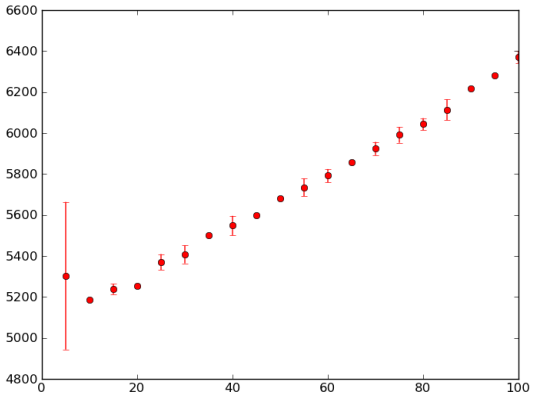
A.1 Categorical

Consider the following simple program, in which n is some integer.

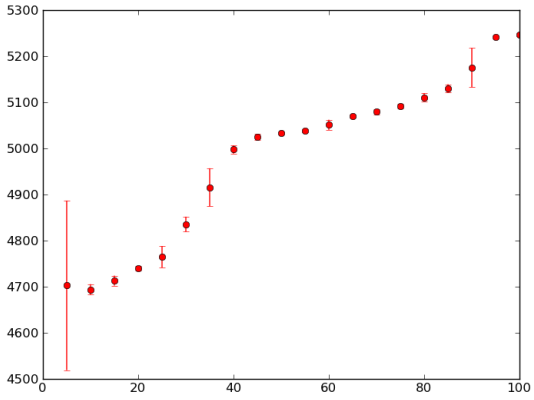
```
ASSUME categorical-dist (lambda (x) (/ 1 n))
ASSUME sample-categorical-loop
        (lambda (v i)
                (if (< v (categorical-dist i))
                    i
                    (sample-categorical-loop
                     (- v (categorical-dist i))
                     (+ i 1))))

ASSUME sample-categorical (lambda () (sample-categorical-loop (rand) 0))
ASSUME sample (sample-categorical)
```

We ran this model for varying n, measuring the time and space usage of various engines. Here are the results:

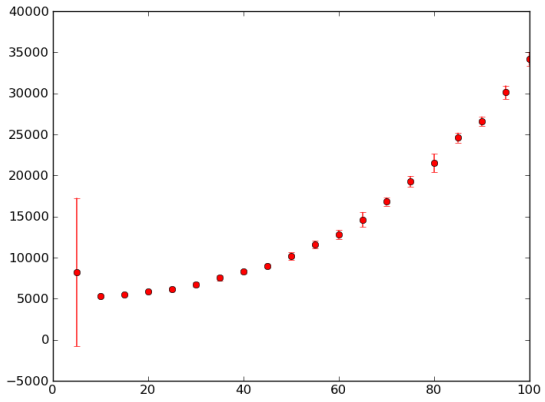
SPACE



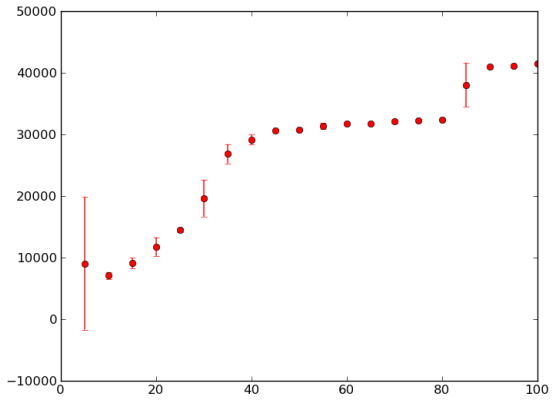
Reduced traces C



Reduced traces Python

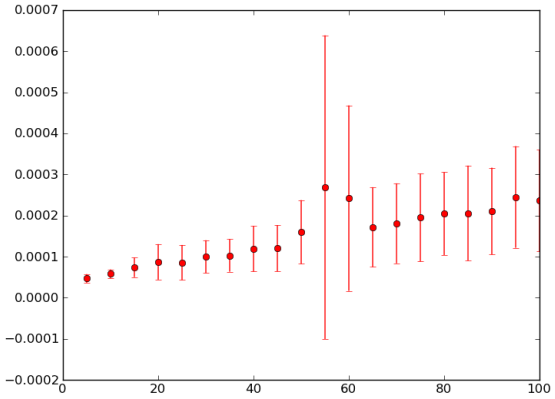


Traces C

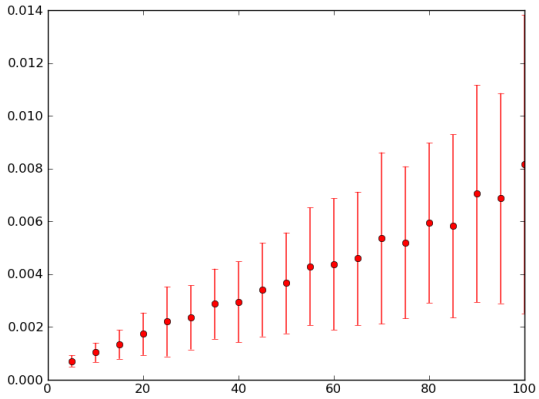


Traces Python

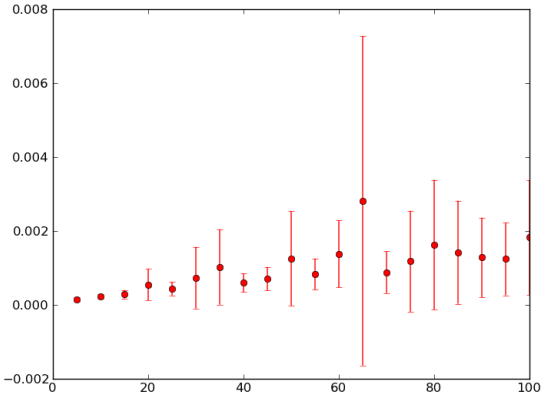
TIME



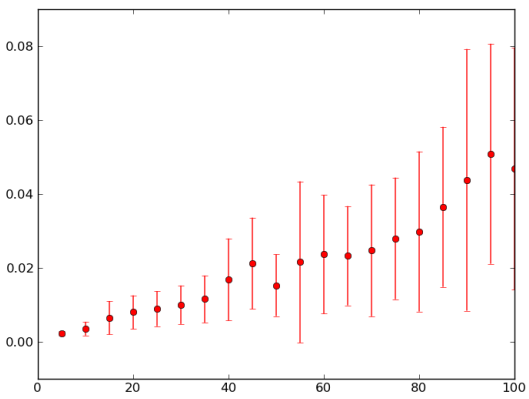
Reduced traces C



Reduced traces Python



Traces C



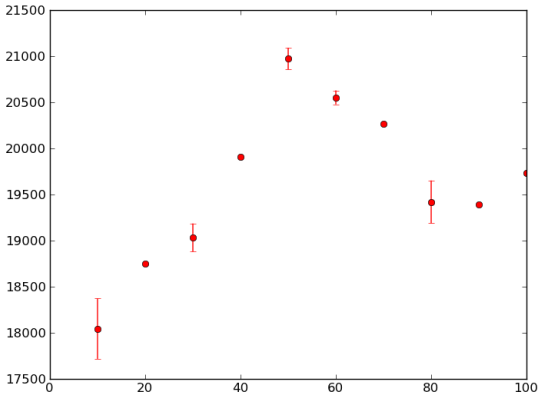
Traces Python

A.2 A Simple HMM

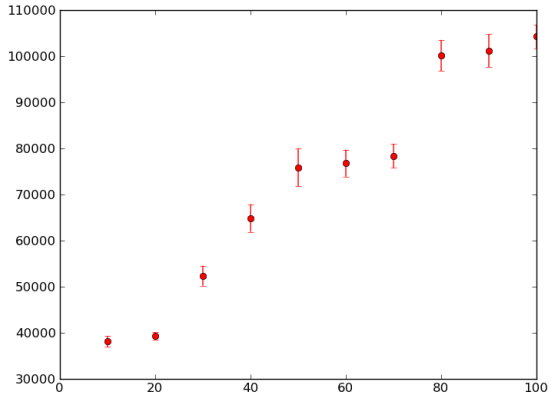
Suppose we have some integers `nstates`, and `chainlength`.

```
ASSUME get-column (mem (lambda (i) (symmetric-dirichlet 1 nstates)))

ASSUME get-next-state
       (lambda (i) \
               (let ((loop (lambda (v j) \
                                   (let ((w ((get-column i) j))) \
                                 (if (< v w) j (loop (- v w) (inc j))))))) \
               (loop (rand) 0)))

ASSUME state (mem (lambda (i) (if (= i 0) 0 (get-next-state (state (dec i))))))
ASSUME laststate (state chainlength)
```

SPACE

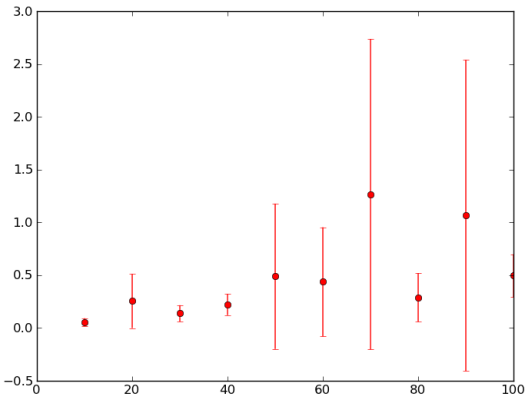


Reduced traces Python

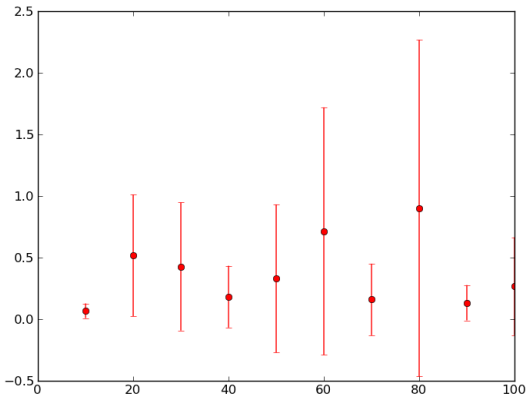


Traces Python

TIME



Reduced traces Python



Traces Python

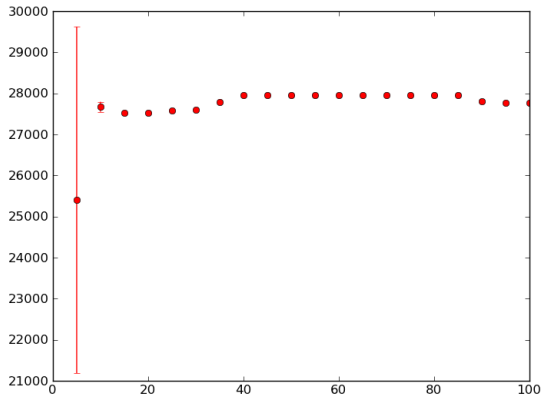
A.3 Topic Modeling

Consider the following program:

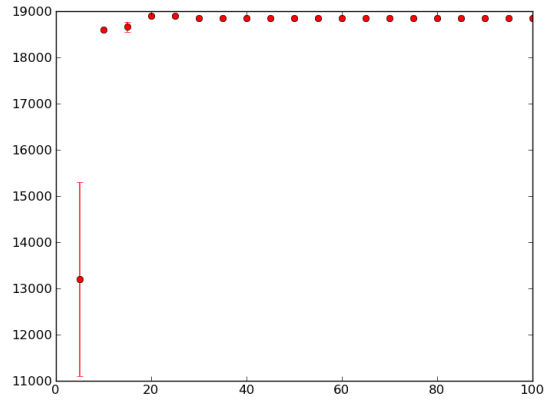
```
ASSUME get-topic-word-hyper (mem (lambda (topic) (gamma 1 1)))
ASSUME get-document-topic-hyper (mem (lambda (doc) (gamma 1 1)))

ASSUME get-document-topic-sampler
       (mem (lambda (doc)
                    (symmetric-dirichlet-multinomial/make
                     (/ (get-document-topic-hyper doc) ntopics) ntopics)))

ASSUME get-topic-word-sampler
       (mem (lambda (topic)
                    (symmetric-dirichlet-multinomial/make
                     (/ (get-topic-word-hyper topic) nwords) nwords)))

ASSUME get-word (mem (lambda (doc pos)
                ((get-topic-word-sampler ((get-document-topic-sampler doc))))))
```

When running this program (with increasingly large documents), we see that the space usage is about 50% better for reduced traces, and the time is about 50% worse.

SPACE

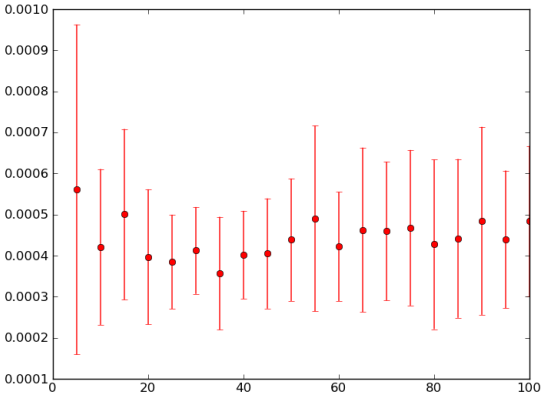


Traces Python

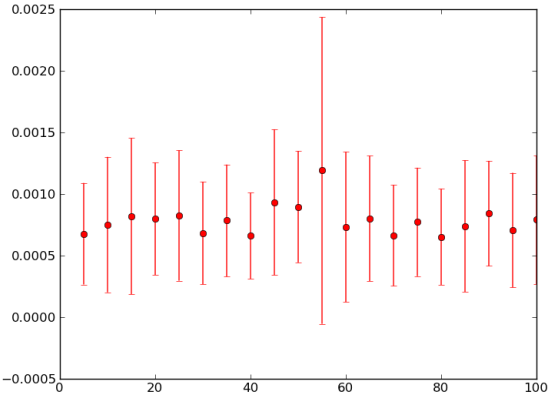


Reduced traces Python

TIME



Traces Python



Reduced traces Python

References

[1] Abelson, Harold ; Sussman, Gerald Jay; Sussman, Julie. Structure and interpretation of computer programs. MIT Press, 1996.

[2] Dagum, P. and Luby, M. Approximating probabilistic inference in Bayesian belief networks is NP-hard (Research Note), Artificial Intelligence 60 (1993) 141-153.

[3] Freer, Cameron and Roy, Daniel. (2009) "Computable exchangeable sequences have computable deÊFinetti measures", Proceedings of the 5th Conference on Computability in Europe: Mathematical Theory and Computational Practice, Lecture Notes In Computer Science, Vol. 5635, pp. 218–231

[4] Goodman, Noah D.; Mansinghka, Vikash K.; Roy, Daniel M.; Bonawitz, Keith; and Tenenbaum, Joshua B. Church. A language for generative models. Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence (2008).

[5] Goodman, Noah; Stuhlmueller, Andreas; Wingate, David. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. 15:770-778, AISTATS 2011.

[6] Hastings, W.K. (1970). "Monte Carlo Sampling Methods Using Markov Chains and Their Applications". Biometrika 57 (1): 97–109. doi:10.1093/biomet/57.1.97. JSTOR 2334940. Zbl 0219.65008.

[7] Karp, Richard M.; Rabin, Michael O. (March 1987). Efficient randomized pattern-matching algorithms. 31. Retrieved 2008-10-14.

[8] McCarthy, J. (April 1960). "Recursive functions of symbolic expressions and their computation by machine, Part I". Communications of the ACM 3 (4): 184–195.