Efficient Inference in Probabilistic Computing

M.Eng Thesis

Jeff Wu

Advised by Vikash Mansinghka and Josh Tenenbaum

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[1]. 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 [2]. A naive implementation of inference uses a database to store all random choices, 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,at we demonstrate thits 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 hae 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]
3. Run various benchmarks on the engines to compare the engines' performance on various problems. Demonstrate good performance on a large-scale topic model!

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.

Ultimately 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.

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

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. Our language has a very Scheme-like syntax. Please refer to [2] for more details.

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.

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.

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 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_MANY 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_MANY 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

In this section, we introduce the most general type of procedures

4.1 XRPs

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.

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.

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.

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

- `init()`, which returns an initial state.
- `get(state, args)`, which returns a value.
- `prob(state, value, args)`, which returns a (log) probability.
- `inc(state, value, args)`, which returns a new state.
- `rem(state, value, args)`, which returns a new state.

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 `get` 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 `inc`, and the state is updated. Finally, `rem` 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. However, for the remainder of the paper, we will use XRP to refer to procedures which are strictly non-deterministic. All XRPs which are deterministic can simply be viewed as ordinary procedures.

We can see why this exchangeability condition still lets us do inference. During inference, we want to reflip random choices from earlier in the program, while being able to preserve later choices. Exchangeability lets us do precisely that, at no cost. We can pretend the XRP application in question was the most recent one, and thus we can safely remove the value.

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.

Lastly, we allow for something much more general than the noisy observations above. We now simply require that all observed expressions have an outermost XRP application. When running inference, we always force the XRP to give the observed value by using `inc` (without using `get`). 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.

4.2 Mem

A special XRP worth noting is `mem`, which lets us memorize 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.

When we re-flip the value of some XRP application, we can now simply remove the old value, and then incorporate a new value directly. This is valid by the exchangeability property. We'd now like to propagate the new value throughout the trace, by repeatedly propagating new values to parents. If we reach another random XRP application, we simply reuse its old value, so that we do not need to propagate computation any further (even if its arguments have changed). If new branches need to be evaluated, it is done on demand.

5.2 Propagation

Many interesting algorithmic ideas went into the implementation of traces. For example, for inference, we need to obtain a random XRP application to re-flip. Thus, we'd like to create a hash table data structure, augmented with the "get random key" operation. However, propagation is the most important detail, and we'd like to focus on the question: How does propagation actually work?

This question is surprisingly interesting, and we explore three different candidate propagation schemes.

5.2.1 Recursive propagation

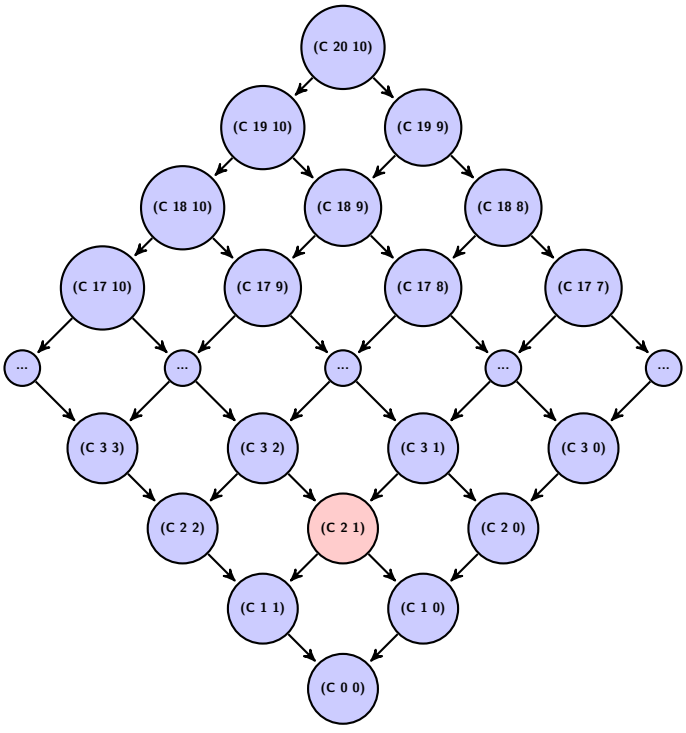
The easiest version of propagation to implement is simply a recursive, "depth-first" implementation. We simply start at the node which we reflipped, and recursively re-evaluate its parents, stopping only when we reach XRP application nodes. However, notice that in a graph where nodes have multiple children and parents, this does 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 triange 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 m l)` is essentially `(binom m l)`.

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.

Unfortunately, a heap does not work correctly on a set of elements with only a partial ordering (even with ties arbitrarily broken), and so we must extend to some full ordering.

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 could use a full description of the node's location (something like a call stack), but it would be quite expensive, leading to asymptotic increases in both time and space. Comparisons would 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. And each node would require $O(h)$ space, instead of $O(1)$ space.

Unfortunately, the story is somewhat worse than that. When evaluating the program itself, we also take a hit of $O(h)$.

5.2.3 Partial location

A final solution is to use a tuple, `(directive-id, distance from directive node)`. To decide whether to propagate to x or to y, we first compare the directive they are immediately underneath. We prefer to propagate to nodes with smaller id. In case of a tie, we then look at distance from the node corresponding to the directive itself. We prefer to propagate to nodes further from the directive.

By propagating this way, we ensure that we only do constant work for each node. We do some work evaluating while propagating up, and we potentially do work unevaluating, if the branch becomes irrelevant (e.g. the predicate of an if statement changes).

We might hope that our solution results in the minimal amount of propagation needed, in general. However, this is false for at least some families of problems. For example, consider the following code, where `f` is some arbitrary deterministic function:

```
ASSUME roll (uniform 1 6)
ASSUME result (if (> roll 1)
                  (if (> roll 2)
                      (if (> roll 3)
                          (if (> roll 4)
                              (if (> roll 5)
                                  (f 6) (f 5)) (f 4)) (f 3)) (f 2)) (f 1))
```

Suppose `roll` is currently `6`, so that the branch at `(f 6)` is currently active. If we reflip the application of `uniform` so that `roll` becomes `1`, we might hope that the correct value is propagated to the outermost occurrence of `roll`, so that we merely unevaluate the branch to `(f 6)` and evaluate the branch to `(f 1)`.

However, what happens in reality is different. We first propagate to the lowest occurrence of `roll`, which leads us to unevaluate `(f 6)` and evaluate `(f 5)`. We then propagate to the second lowest occurrence, which leads us to unevaluate `(f 5)` and evaluate `(f 4)`. This continues, so that by the end of our reflipping, we have evaluated `(f x)` for all `x = 1, ..., 6`.

What we'd like to do is to make sure that the predicate of an if branch precedes the consequents. We run into a similar problem when evaluating procedure applications, where we'd like to evaluate the arguments before the body. However, to do these things appears to require using longer keys for comparison, and we recover the "full location" method.

5.3 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 XRP application nodes (including the application nodes, as well as x), and then propagating down through the consequents of any conditionals we pass through and the bodies of any procedure applications we pass through. [2] 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 then have that the expected propagation time of the "full location" propagation scheme is

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

Although we may evaluate new parts, outside of V_x, when choosing x, we can use amortized analysis to show this is okay; essentially, we evaluate this work as part of the next reflipping instead.

Let us now turn towards a different picture. Though the trace may change structures, depending on the values at various nodes, let us imagine a static version of the graph, in which all branches of computation are active. We call this the **static trace**.

Let's pay special attention to application nodes, which correspond to the application of a procedure to some arguments. We would like to consider the static trace to have a child for each argument, a child for the procedure, and *only a single child* for the body, even if the function is called with many different tuples of arguments, across different computation histories.

In a dynamic trace, there is indeed only one child corresponding to the body, but in a static trace, there is a subtlety. If the structure of the body is known in advance, then we may essentially have only one child. But occasionally, we may not even know, statically, the structure of the procedure which is being applied. For example, the child for the procedure could have a branching if statement. Thus we imagine the static trace to have one node for each possible structure of the procedure body.

Notice that this static trace graph is not necessarily finite; after all, the computation itself may not be guaranteed to terminate. For example, the following simple program has an infinite static trace.

[2] The envelope should bear some resemblance to the notion of a Markov blanket. Indeed, it captures some aspect of conditional independence.

```
ASSUME decay-rate (rand)
ASSUME geometric (lambda (x) (if (bernoulli decay-rate) x (geometric (+ x 1))))
ASSUME decay-time (geometric 0)
```

For every node x, we say the **static envelope** of x, denoted V_x^+, is the set of nodes reached by propagating upwards in the static trace until we reach XRP application nodes (including the application nodes, as well as x), and then propagating down through the consequents of any conditionals we pass through and the bodies of any procedure applications we pass through. Similarly, 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.

So we then have that the expected propagation time of the "partial location" propagation scheme is

$$E_{V^+}[|V^+| \cdot \log |V^+|].$$

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 [3].

One minor issue which comes up is the locating of child nodes. We need to be able to recover a specific child node, if we perform the computation leading to what should be its evaluation. We also need to be able to figure out the location of children to unevaluate. The relative locations may be long, and we don't want to store them in memory.

In order to do this, we using a rolling hash on the "stack", a list which pins down the relative location of a node within the computation. This technique is also used in the random database implementation of the engine.

6.3 Propagation in reduced traces

We'd like to use the propagate nearly the exact same way as the "partial location" method, in reduced traces. However, the granularity at which we propagate up is much greater now. Whenever we re-evaluate a node, we re-perform all the deterministic computation along the way, downwards in the DAG of computation, before propagating up again.

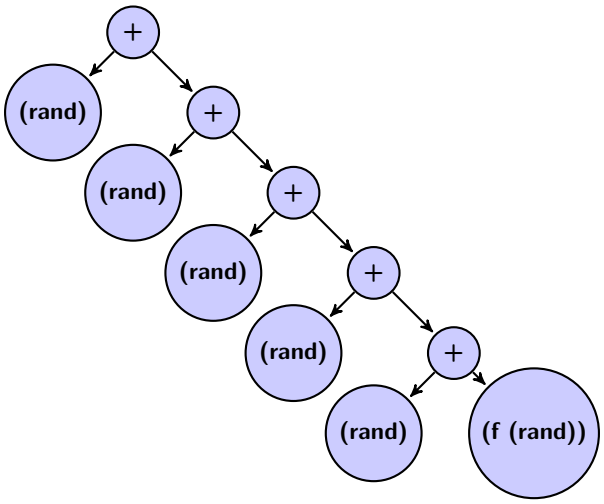
Interestingly, we can see that we will avoid the problem which we saw earlier with the dice roll. Because we evaluate expressions from top-to-bottom, we have taken care of that issue for free!

However, we will have paid a fairly steep price. Suppose we write a 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:



[3]There are also nodes corresponding to applications of `mem`.

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 empriically comparable to that of traces.

6.4 Analysis

To analyze the propagation, we imagine we were still propagating in the old dynamic trace. Recall that we had defined the envelope to be the set of nodes reached by propagating upwards in a dynamic trace until we reached XRP application nodes, and then propagating down through the consequents of any conditionals we passed through and 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, but only until we hit an XRP application node. Let us call this set of nodes the **sagging envelope**, S_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

$$E_S[|S| \cdot \log |S|],$$

where S is the random variable which first picks some x with probability p_x, and then picks a random sagging envelope S_x, conditioned on x being active.

6.5 Engine Complexity Trade-offs Summary

To summarize, we have the following table of asymptotic complexities:

Table 1: Engine Complexity Trade-offs

	Time / Iteration	**Space**	**Time**
Random DB	T	H + P	T
Traces 1 (recursive)	2^T	T $(+ H + P)$	T
Traces 2 (full location)	$h \cdot E_V[V \log V]$	T $(+ H + P)$	$h \cdot T$
Traces 3 (partial location)	$E_{V^+}[V^+ \log V^+]$	T $(+ H + P)$	T
Reduced Traces	$E_S[S \log S]$	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
- V^+ = Envelope size, in static trace
- S = Sagging envelope size, in dynamic 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 $V > V^+$ and $V > S$.

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 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 graularity. 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.

8 Acknowledgements

Besides my advisor Vikash Mansinghika, 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.

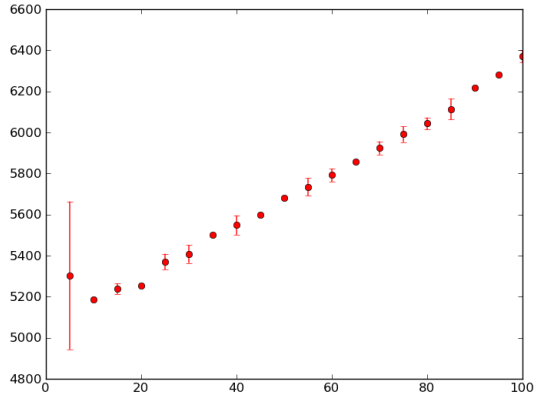
A.1 Categorical

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

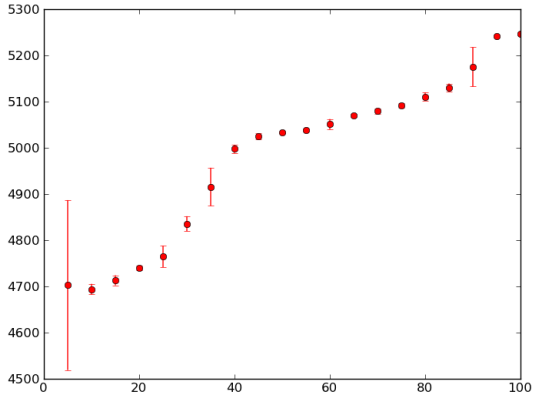
```
ASSUME categorical-dist (lambda (x) (/ 1 n))
ASSUME sample-categorical-loop
        (lambda (v i)
                (if (< v (categorical-dist i))
                    i
                    (sample-categorical-loop
                     (- v (categorical-dist i))
                     (+ i 1))))

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:

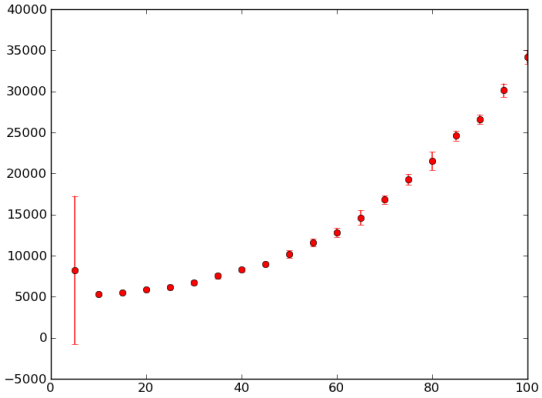
SPACE



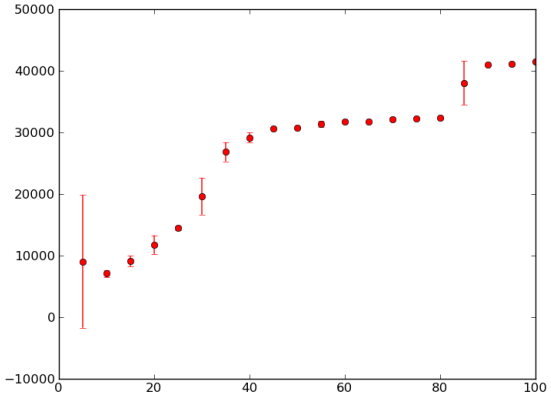
Reduced traces C



Reduced traces Python

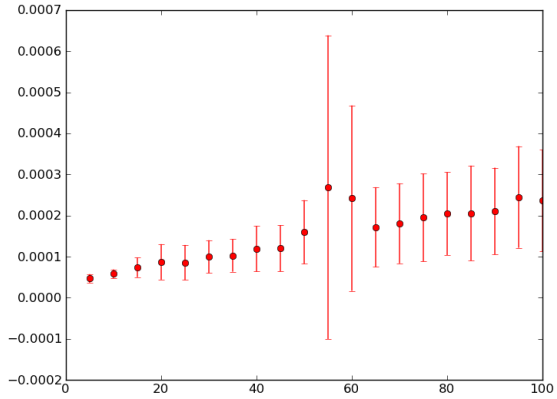


Traces C

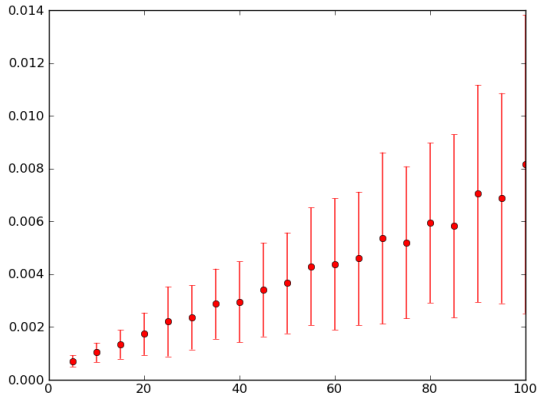


Traces Python

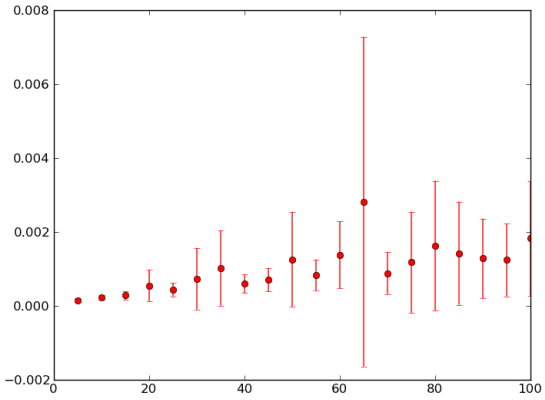
TIME



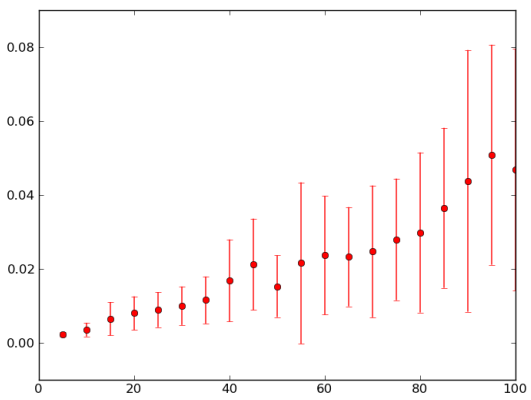
Reduced traces C



Reduced traces Python



Traces C



Traces Python

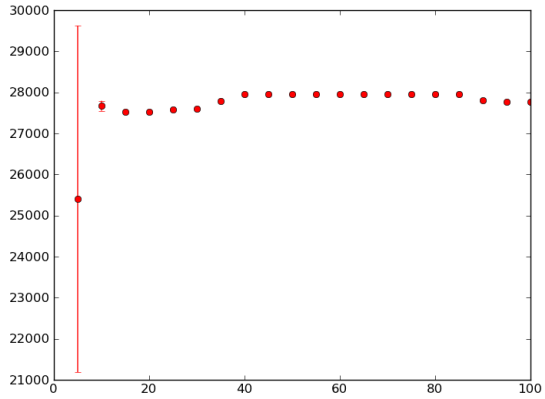
A.2 Topic Modeling

Consider the following program:

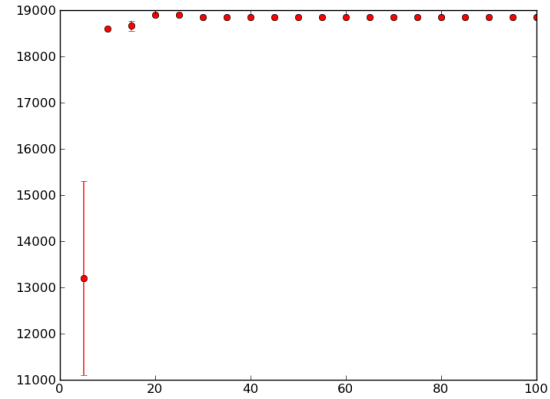
```
ASSUME get-topic-word-hyper (mem (lambda (topic) (gamma 1 1)))
ASSUME get-document-topic-hyper (mem (lambda (doc) (gamma 1 1)))
ASSUME get-document-topic-sampler (mem (lambda (doc)
          (symmetric-dirichlet-multinomial/make (/ (get-document-topic-hyper doc) ntopics) ntopics)))

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

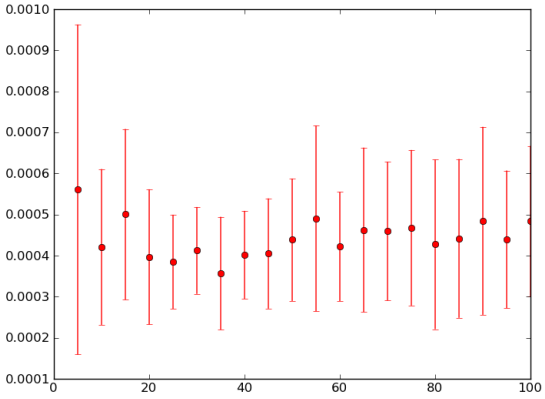


Traces Python

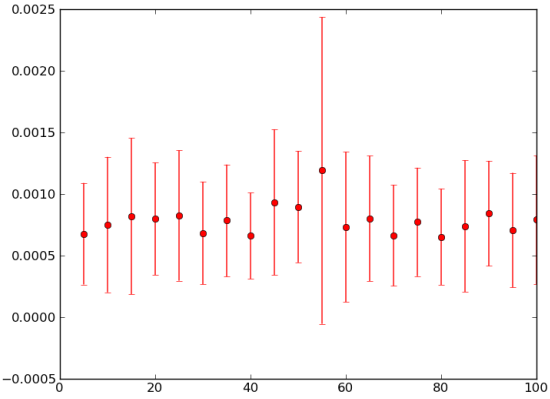


Reduced traces Python

TIME

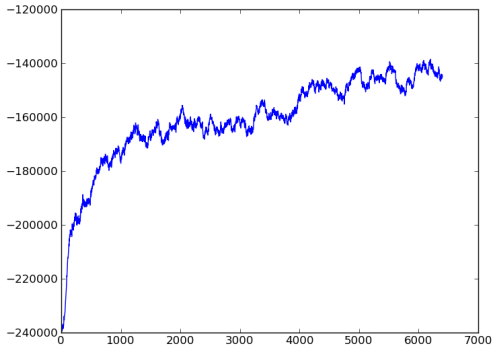


Traces Python



Reduced traces Python

In fact, this topic modeling code was used to infer topics from a real corpus of 1500 documents with over a million total words. The results were quite promising. The C engine was able to perform on the order of 100,000 inference steps per second, using around 15 gigabytes of RAM. Here is a graph of the logscore over time:



And here is a list of the final topics discovered:

```
topic 0: model(0.0337) structure(0.0278) algorithm(0.0272) network(0.0259) system(0.0226) sonn(0.0223) r
topic 1: case(0.0099) fraction(0.0083) training(0.0082) probability(0.0073) weight(0.0069) appreciated(
topic 2: tension(0.0006) lhe(0.0006) extremely(0.0005) inductive(0.0005) modularity(0.0005) ensuring(0.
topic 3: florida(0.0006) frames(0.0005) prototypes(0.0005) genes(0.0005) atr(0.0005) sym(0.0005) benign
topic 4: annealing(0.0262) equilibrium(0.0178) average(0.0175) spin(0.0157) simulated(0.0143) network(0
topic 5: count(0.0088) modify(0.0088) error(0.0082) output(0.008) total(0.0078) numeral(0.0072) set(0.0
topic 6: network(0.0412) solution(0.0402) human(0.0379) pattern(0.0339) generalization(0.0317) subject(
topic 7: solution(0.0221) generalization(0.0174) human(0.0119) cursor(0.0115) network(0.0105) external(
topic 8: learning(0.0264) point(0.0231) algorithm(0.0195) network(0.0175) function(0.0166) layer(0.0153
topic 9: classifier(0.0933) network(0.0557) system(0.0425) message(0.0407) set(0.0403) match(0.0365) noo
topic 10: dimensional(0.0191) ability(0.0172) sum(0.0108) space(0.0107) input(0.0095) inventory(0.0093)
topic 11: number(0.0088) conceptual(0.0006) overcomes(0.0006) perceptron(0.0005) development(0.0005) dia
topic 12: remarkably(0.0006) doubly(0.0006) herbster(0.0006) net(0.0005) conclusion(0.0005) empirical(0
topic 13: phonetic(0.0007) stationary(0.0006) xij(0.0006) hungry(0.0006) session(0.0005) entered(0.0005)
topic 14: examples(0.0351) function(0.0333) net(0.0268) threshold(0.0201) nodes(0.0196) random(0.0195) w
topic 15: lipschitz(0.0081) jaakkola(0.0006) sual(0.0006) synaptic(0.0005) energy(0.0005) combination(0
topic 16: effort(0.0085) britten(0.0085) hopfield(0.0072) rissanen(0.0071) drives(0.0006) accepted(0.000
topic 17: occlusion(0.0008) wechsler(0.0007) quadratic(0.0006) calcium(0.0006) getting(0.0006) brownlow
topic 18: leading(0.0087) sort(0.0079) gij(0.0006) closure(0.0006) vision(0.0005) inhibition(0.0005) nij
topic 19: resolution(0.028) data(0.025) level(0.0227) hierarchy(0.0198) input(0.0192) function(0.0183) s
topic 20: output(0.0093) bound(0.0087) criterion(0.0066) exhaustively(0.0007) table(0.0006) eoo(0.0006)
topic 21: network(0.0437) output(0.0303) input(0.0259) unit(0.0246) error(0.0223) learning(0.02) algorit
topic 22: cem(0.0006) lung(0.0006) principles(0.0005) concatenated(0.0005) alexander(0.0005) joseph(0.00
topic 23: symmetric(0.0005) wind(0.0005) token(0.0005) shepard(0.0005) molecules(0.0005) server(0.0005)
topic 24: activates(0.0006) obeying(0.0006) refer(0.0005) generic(0.0005) generality(0.0005) minus(0.000
topic 25: sequentially(0.0008) tive(0.0006) timothy(0.0006) kung(0.0006) interna(0.0006) amir(0.0006) fl
topic 26: net(0.0308) examples(0.0279) network(0.0251) function(0.0179) training(0.0177) feedforward(0.0
topic 27: order(0.0078) search(0.0006) extract(0.0006) ject(0.0006) unix(0.0005) cocktail(0.0005) hin(0.
topic 28: initializes(0.0006) tuning(0.0005) structural(0.0005) ideas(0.0005) geiger(0.0005) opponent(0.
topic 29: restrictive(0.0006) broadcast(0.0006) implicated(0.0006) patch(0.0005) replica(0.0005) slice(0
topic 30: output(0.0256) wij(0.0157) hidden(0.0152) number(0.0116) current(0.0114) weight(0.0107) unit(0
topic 31: occipital(0.0006) set(0.0005) neuron(0.0005) curve(0.0005) crossing(0.0005) incoming(0.0005) w
topic 32: extensively(0.0006) gpp(0.0006) curve(0.0005) mlp(0.0005) nucleus(0.0005) autonomous(0.0005) p
topic 33: mfa(0.0241) mean(0.0238) field(0.019) annealing(0.0143) averages(0.0127) problem(0.0103) opera
topic 34: sphering(0.0059) marginally(0.0007) ebl(0.0006) aspect(0.0005) published(0.0005) file(0.0005)
topic 35: hidden(0.0085) perform(0.0084) internal(0.0084) algorithm(0.0081) exact(0.0078) solve(0.0078)
topic 36: learning(0.0257) function(0.0161) resolution(0.0154) input(0.0152) lattice(0.0148) system(0.01
topic 37: network(0.0644) subject(0.0351) solution(0.0289) human(0.028) pattern(0.0174) minimal(0.0174)
topic 38: learning(0.0326) internal(0.0302) input(0.0271) representation(0.0267) layer(0.0237) algorithm
topic 39: learning(0.0419) problem(0.0393) algorithm(0.0374) vector(0.0221) propagation(0.0202) function
topic 40: pasadena(0.0087) unit(0.008) data(0.0079) truncating(0.0079) model(0.0078) output(0.007) latti
topic 41: appealing(0.014) function(0.0139) problem(0.0131) generate(0.0083) paper(0.008) set(0.0078) ec
topic 42: mean(0.0363) field(0.0331) mfa(0.0327) spin(0.0317) temperature(0.0256) graph(0.0234) problem
topic 43: levy(0.0006) rosci(0.0006) excised(0.0006) identical(0.0005) carried(0.0005) composition(0.000
topic 44: increased(0.0005) multilayer(0.0005) beginning(0.0005) saturation(0.0005) unlikely(0.0005) fun
topic 45: unit(0.0087) choice(0.0087) hidden(0.0081) problem(0.0074) row(0.0072) parity(0.0058) discard
topic 46: slot(0.0007) traffic(0.0006) critical(0.0005) bandwidth(0.0005) trace(0.0005) jacobian(0.0005)
topic 47: distributed(0.0005) clearly(0.0005) hardware(0.0005) subspace(0.0005) begin(0.0005) improving
topic 48: psth(0.0006) implies(0.0005) spectrum(0.0005) formation(0.0005) splines(0.0005) poorly(0.0005)
topic 49: network(0.008) uyar(0.0007) think(0.0006) srn(0.0006) coggin(0.0006) oracle(0.0006) garzon(0.0
```

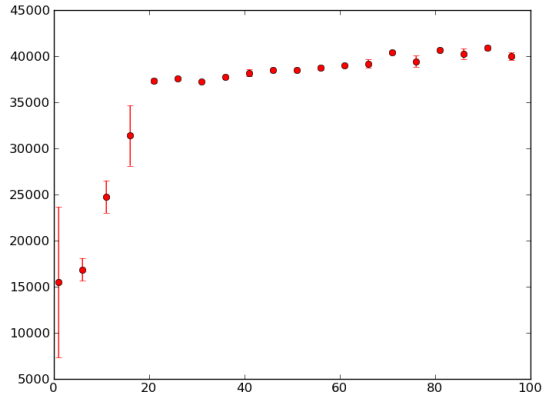
A.3 CRP Mixture Model

Lastly, we compared the two engines on a CRP mixture model.

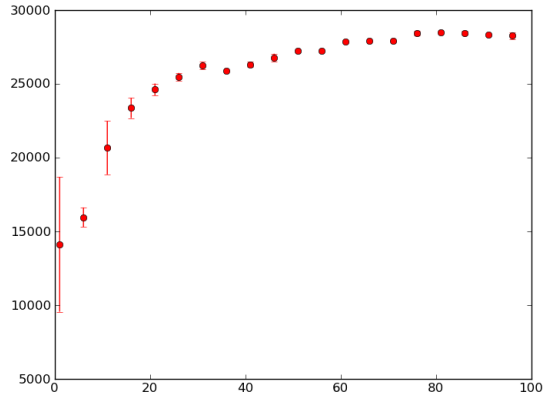
```
ASSUME alpha (gamma 0.1 20)
ASSUME cluster-crp (CRP/make alpha)

ASSUME get-cluster-mean (mem (lambda (cluster) (gaussian 0 10)))
ASSUME get-cluster-variance (mem (lambda (cluster) (gamma 0.1 100)))
ASSUME get-cluster (mem (lambda (id) (cluster-crp)))
ASSUME get-cluster-model (mem (lambda (cluster) (lambda ( ) (gaussian (get-cluster-mean cluster) (get-cluster-va
ASSUME get-datapoint (mem (lambda (id) (gaussian ((get-cluster-model (get-cluster id))) 0.1)))
```

Again, we see an improvement in space usage, and a hit in runtime.

SPACE

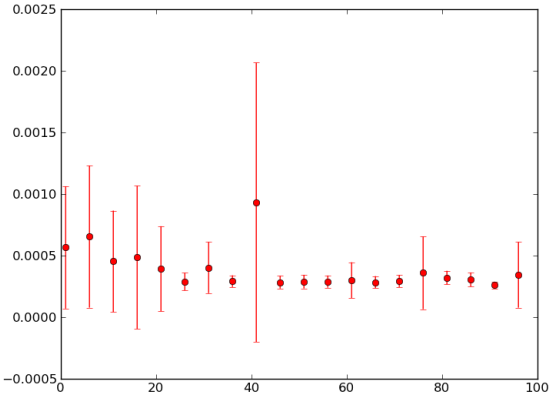


Traces Python

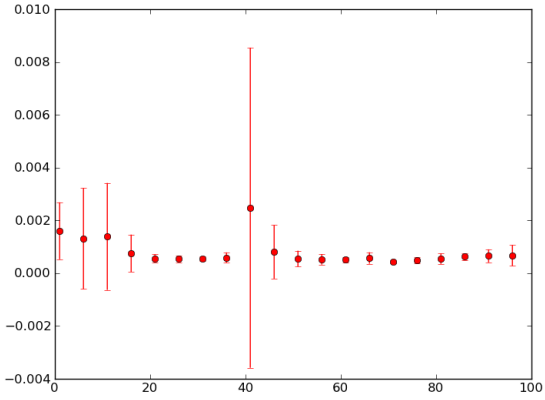


Reduced traces Python

TIME



Traces Python



Reduced traces Python

References

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

[2] 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).