Sampling with Memoization

Avi Pfeffer
School of Engineering and Applied Sciences
Harvard University
avi@eecs.harvard.edu

Abstract

Memoization is a fundamental technique in computer science, providing the basis for dynamic programming. This paper explores using memoization to improve the performance of rejection sampling algorithms. It is shown that reusing values produced in previous samples and stored in a cache is beneficial. The paper goes on to explore the idea of recursive memoization, in which values are aggressively reused from the cache even in the process of computing a value to store in the cache. This leads to the values in the cache becoming dependent on each other, and therefore produces a biased sampler. However in practice this seems to be quite beneficial. Furthermore, we show that the error in the cache tends to zero in the long run. We demonstrate the usefulness of memoized sampling in a duplicate bridge simulation, and in experiments with probabilistic grammars.

Memoization is a fundamental technique in computer science. In memoization, when a function is evaluated on arguments, the value returned is cached, so that whenever the function is called again on the same arguments the value can be retrieved from the cache. This can potentially save a great deal of computation, as the function does not have to be evaluated again. Memoization provides the basis for dynamic programming.

Probabilistic models are a basic tool of artificial intelligence. The approach is to represent the domain using a probabilistic model, make observations, and infer probabilities of query variables using probabilistic inference. In many cases, the models are too complex to perform inference exactly, so approximate algorithms are needed. Sampling is one of the most important approaches to approximate inference in probabilistic models. There are various families of sampling algorithms, including rejection sampling (see e.g. (Robert & Casella 2004)), importance sampling (Srinivasan 2002), and Markov chain Monte Carlo algorithms (Gilks, Richardson, & Spiegelhalter 1996), each of which has a useful role. This paper explores the idea of exploiting memoization to improve the performance of rejection sampling algorithms.

The context is a highly expressive language for probabilistic modeling called IBAL (Pfeffer 2007a). By defining an inference algorithm for this language, we automatically obtain an algorithm for a wide variety of models. Furthermore, IBAL provides a natural basis for thinking about breaking up a complex model into the evaluation of different expressions. It also allows us to explore the issues that arise for sampling with memoization in full generality.

Straightforward memoization as applied to deterministic programs does not work for sampling. In deterministic programs, we want to get the same value every time an expression is evaluated. For sampling, we want to get independent samples from the distribution defined by the expression; we explicitly don't want to reuse the value previously computed. However, there is a natural alternative. Instead of using the values computed for the same sample, we can reuse values computed in previous samples. The different values selected can be made independent of each other, so that samples are taken from the correct distribution.

This simple idea raises many questions which are explored in this paper. One important extension is to use the cache recursively, while we are actually computing values to be stored in the cache. This allows a great deal more computation to be saved, but it makes the values in the cache dependent on each other. As a result, samples taken using the cache will be biased. We show that this bias tends to zero in the long run. We present experimental results from a duplicate bridge simulation and from probabilistic grammars that show that sampling with memoization works well.

IBAL

IBAL (Pfeffer 2007a) is a highly expressive and general language for representing probabilistic models. It can represent any generative model over discrete variables that can be expressed as a program. This includes Bayesian networks, hidden Markov models, dynamic Bayesian networks, probabilistic relational models, probabilistic context free grammars and many more. The basic idea behind IBAL is that a program describes the way in which the world is generated. In an ordinary programming language, an expression represents a computation that produces a value. In IBAL, an expression represents a computation that *stochastically* produces a value. The meaning of an expression is the probability distribution over the value that it produces.

IBAL uses the following concepts. A *value* is a symbol, integer, Boolean, tuple, or function. A *pattern* is a template that matches values. An *environment* is a mapping from variable names to values. An *expression* is any of the following:

c	// constant
a	// variable
$\langle a_1 = \epsilon_1, ..., a_n = \epsilon_n \rangle$	// tuple construction
$\epsilon.a$	// component access
`if` ϵ_1 `then` ϵ_2 `else` ϵ_3	// conditional
`case` ϵ_0 `of` $\{\pi_1 : \epsilon_1, ..., \pi_n : \epsilon_n\}$	// pattern matching
`dist` $[p_1 : \epsilon_1, ..., p_n : \epsilon_n]$	// stochastic choice
`let` $a = \epsilon_1$ `in` ϵ_2	// variable binding
`let` $a_0(a_1, ..., a_n) = \epsilon_1$ `in` ϵ_2	// function definition
$\epsilon_1 \otimes \epsilon_2$	// operator
$\epsilon_0(\epsilon_1, ..., \epsilon_n)$	// function application
$\epsilon_1 == \epsilon_2$	// equality test
`obs` π `in` ϵ	// observation

where c denotes a constant (symbol, integer or Boolean), a denotes a variable name, ϵ denotes an expression, π denotes a pattern, p denotes a probability, and $\otimes$ denotes a predefined operator. The intuitive meaning of most of the expression forms is the same as in ordinary programming languages. A `dist` expression defines a process that stochastically chooses the value of one of its subexpressions, each with a given probability. An expression `let` $a = \epsilon_1$ `in` ϵ_2 defines the process of binding a to the result of evaluating ϵ_1, and evaluating ϵ_2 in the environment extended by this binding. An expression `obs` π `in` ϵ evaluates to the result of ϵ, but conditioned on the result matching π. A *program* is simply a top-level expression that we are trying to evaluate.

It is easy to define a rejection sampling algorithm for IBAL. A `sample` function is defined for each type of expression, that samples a value for the expression in the current environment. For example, `sample(if` e_1 `then` e_2 `else` e_3, ν), where ν is the environment, first calls `sample(`e_1, ν). If the result is `true`, `sample(`e_2, ν) is called and the result is returned, otherwise `sample(`e_3, ν) is called. `sample(obs` π `in` e_1, ν) first calls `sample(`e_1, ν) to obtain a value x. If π matches x, x is returned, otherwise a `Reject` exception is thrown. This exception is caught by the top-level sampling algorithm, and results in the sample being rejected.

In most discussions of sampling, a sampling run consists of a number of samples being taken from a distribution. However, for us the process of taking a sample from the distribution defined by a program may involve sampling many subexpressions. To avoid overloading the word "sample", we will use the following language in this paper. An *evaluation* is the process of (stochastically) *producing* a value for a particular (expression, environment) pair. A *sample* consists of one evaluation of the top-level program in the empty environment. A *sampling run* consists of a sequence of samples. In a sampling run, the algorithm keeps track of the number of successful samples, the values produced, and the number of rejections. The estimated probability of the observations is then the number of successful samples divided by the total number of samples. The estimated probability of a value given that the observations are satisfied is the fraction of successful samples that produced that value.

The Basic Idea

We now explore the possibility of improving this sampling algorithm through memoization. As a motivating example we will use the following problem: What is the probability that if two numbers are drawn from the same geometric distribution with parameter λ, the second will be greater than the first by the amount d. In this case we can express the probability in closed form, as $\frac{\lambda^2(1-\lambda)^d}{1-(1-\lambda)^2}$. But suppose we wish to compute this probability by sampling. We can express it as the following IBAL program:

```
let g() = dist [ λ :   0,
                 1-λ :  1 + g() ] in
let x = g() in
let y = g() in
y - x == d
```

where λ, $1-\lambda$ and d are filled in with appropriate values. If this was an ordinary program, we could observe that `g()` is evaluated twice, so we could store the result of the first evaluation in the cache, and reuse it for the second evaluation. A naive idea is to do the same thing for sampling. But this is a bad idea. It results in an extremely biased sampler. In our case, the value of `y` would become the same as the value of `x`, so `y - x == d` will always be `false`, so the probability of `y - x == d` will be underestimated.

A much better idea is to maintain a cache of all values previously produced for `g()`. When a new value of `x` needs to be produced, we simply draw an item uniformly from the cache. Since all the elements of the cache are drawn from the correct geometric distribution, and are independent of the value currently produced for `x`, we expect the value produced for the program in the current sample to be drawn from the correct distribution. (To make this work, we must delay storing the value produced for `x` until after the sample has been completed.) This is an *ex ante* argument. Before we know what is in the cache, we expect that the value of the program will be drawn from the correct distribution. After we know what is in the cache, however, we will believe the distribution to be biased. For example, before we start sampling, we will expect the value x_1 of `x` in the first run to be drawn from the geometric distribution, and the value x_2 of `x` in the second run to be independent of it. Since y will be assigned to be x_1, and compared to x_2, the result of the test will be `true` with the correct probability. However, once we know that $x_1 = 100$, then the test will be `true` with the probability that $x_2 = 98$, which is different from the correct probability. The hope is that this will be made up for by the fact that *ex ante* the sampler is unbiased, and we will be able to take many more samples.

In a general program it may be hard to determine automatically when to store items in the cache and when to reuse them. For this reason we introduce explicit `store` and `reuse` directives. The program above becomes

```
let g() = dist [ λ :   0,
                 1-λ :  1 + g() ] in
let x = store g() in
let y = reuse g() in
y - x == $d$
```

The key to the cache is an (expression, environment) pair, because that is what uniquely determines the evaluation. However, it does not use the complete environment in which the expression is evaluated. Rather, the environment is reduced so that it only contains variables used by the expression, since those are the only variables that affect the evaluation of this expression. Reducing the environment in this way can greatly increase the number of cache hits.

This simple memoization works quite well. The table below compares the performance of the program with memoization to the program without. The first two rows show performance when $\lambda = 0.05$. Each program was given 0.05 seconds of sampling time. All results in this paper are averaged over 10000 tests, except where otherwise stated, and all errors are relative error. We see that with memoization, the method was able to take more samples, and thus achieve significantly lower error. The number of samples was not quite double, because there is some overhead to using the cache. The next two rows shows performance when $\lambda = 0.01$, which is a much more difficult task for two reasons. It takes much longer, on average, to take a sample, and the correct probability of true is much smaller. We see similar performance benefits for memoization. The last two rows show performance when the samplers were given 10 times as much time. We see that the benefits of memoization are just as strong in this case.

Method	λ	Time	Samples	Error
No memo	0.05	0.05	782	0.1876
Memo	0.05	0.05	1386	0.1425
No memo	0.01	0.05	158	0.9100
Memo	0.01	0.05	307	0.6561
No memo	0.01	0.5	1520	0.2930
Memo	0.01	0.5	2943	0.2096

Multiple reuses

We said earlier that in order for the reused value for y to be independent of the value produced for x, we should delay storing the value of x in the cache until after the sample has completed. Unfortunately this is not a general solution. In a program with multiple reuses, the different reused values may be dependent on each other, even when we do not store values produced in the current sample. For example, consider the following program.

```
let g() = dist [ $\lambda$ :  0,
                 $1-\lambda$ :  1 + g() ] in
let x = store g() in
let y = reuse g() in
let z = reuse g() in
z - y == $d$
```

There is a possibility that y and z will reuse the value from the same evaluation. When that happens, the test will definitely be false. Therefore the memoized sampler will underestimate the true probability. This effect is small but noticeable. When the sampler is given 0.05 seconds and an average of 1312 samples are taken, the algorithm underestimates the true probability by a factor of 0.008 on average.

This problem can be solved by ensuring that the same value is not reused multiple times in one sample. Each value in the cache has a reused flag. When a value is reused, its flag is set, and the value is added to a list of values that have been reused. If a subsequent call to the cache results in a value that has already been reused, the value is discarded and a new evaluation is performed instead. At the end of the sample the flag is cleared for all values in the list.

Driving forward

To this point, we have ensured that at least one evaluation of an (expression, environment) pair is performed and its result stored in the cache before the cache is reused. In fact this is not necessary. All that is needed is that there should be some momentum driving forward to produce new evaluations. These do not necessarily need to be of the same pair. For example, consider the program

```
let g1() = dist [ $\lambda_1$ :  0,
                  $1-\lambda_1$ :  1 + g1() ] in
let g2() = dist [ $\lambda_2$ :  0,
                  $1-\lambda_2$ :  1 + g1() ] in
let x = dist [ $p$ :  reuse g1(),
               $1-p$ :  store g1() ] in
let y = dist [ $p$ :  reuse g2(),
               $1-p$ :  store g2() ] in
y - x == $d$
```

where p is a probability. In this program, there is a probability that one or two stores will occur on every iteration, so there is always forward driving momentum. In fact, memoization works in this example. Without memoization, 690 samples are taken and a error of 0.2054 is achieved (given 0.05 seconds of computation time). With memoization, with $p = 0.5$, 1192 samples are taken for a error of 0.1629. Looking more closely, we can see what happens as p is varied. The results are shown in Figure 1. Somewhat surprisingly, performance improves as p is increased even up to 0.9. This shows that the key operation in this program is comparing values from the two distributions, even when new values are not produced. Nevertheless, some degree of forward momentum is needed for this to be a correct sampler.

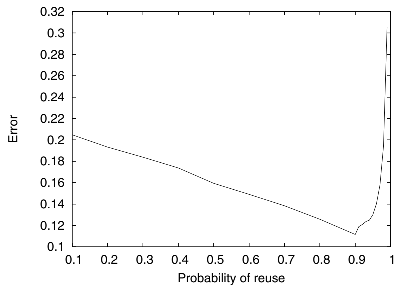

Figure 1: Forward driving performance with different probabilities of reuse

Recursive Memoization

If memoization is a good idea, maybe we can be more aggressive about it. An idea is to memoize recursive calls of an expression. This goes beyond memoization in deterministic programs. There, the first computation of an expression must finish before its value can be used. But here, we are reusing the items in the cache even to compute the result that gets placed in the cache. Our example program becomes

```
let g() = dist [ λ :  0,
              1−λ :  1 + reuse g() ] in
let x = store g() in
let y = reuse g() in
y - x == d
```

A quick test shows that this is a very bad idea. Even though many more samples are taken (6168), the average error is 1.2322, which is much worse. It is easy to see why: the entries in the cache are not independent of each other. When we store a value in the cache, it is based on a reused value already in the cache. Since we expect the cache to be biased, the new values put in it will not be drawn from the correct distribution, but will instead be drawn from a distribution related to the distribution in the cache.

Perhaps, however, if we prime the cache with a reasonable number of values, its distribution will be a good enough approximation to the true distribution that the advantage of being able to take more samples dominates. This turns out to work. For example, with 400 priming values the error is 0.0930 ($\lambda = 0.05$), which is much better than the 0.1425 achieved with non-recursive memoization. Figure 2 shows the performance of recursive memoization, as a function of the proportion of the running time used to prime the cache, under different circumstances. The dashed line shows performance when $\lambda = 0.05$. There is a trade-off to using more priming values. More primers allows the cache distribution to be closer to the true distribution, but allows fewer total samples to be taken. We see that for much of the curve, performance is significantly better than for simple memoization. The solid line shows results for the case where $\lambda = 0.01$. The shape of the curve is similar, but more time should be devoted to priming for optimal performance. A reasonable explanation for this is that since it takes longer to produce a sample, more time is needed to ensure that the cache is reasonable. This is confirmed by the dotted line, which shows performance for $\lambda = 0.01$ when the algorithm was given 10 times as long. Here the cache becomes a reasonable size in a smaller fraction of the total time.

Why does this work? After all, if the new item being stored in the cache is dependent on the cache distribution, wouldn't we expect the cache to get worse and worse? In fact we can show that, *ex ante*, we can expect the cache to get better on each evaluation. Let P be the true geometric distribution, let Q be the distribution over items in the cache, and let R be the distribution over the new item stored in the cache. We claim that $\|R, P\|_1 = (1-\lambda)\|Q, P\|_1$. This is an instance of a general theorem we will prove later, but the argument here is particularly simple. To see this, let N_i be the number of times i is in the cache, and let N be the total

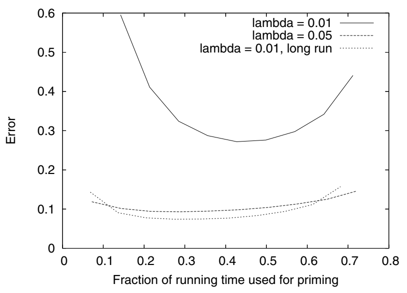

Figure 2: Performance of recursive memoization

number of items. Then $Q(i) = \frac{N_i}{N}$. Meanwhile $R(0) = \lambda$, and $R(i) = (1-\lambda)\frac{N_{i-1}}{N}$, while $P(i) = \lambda(1-\lambda)^i$. Then

$$\begin{aligned}\|R, P\|_1 &= \textstyle\sum_{i=0}^{\infty} |R(i) - P(i)| \\ &= 0 + \textstyle\sum_{i=1}^{\infty} |(1-\lambda)\frac{N_{i-1}}{N} - \lambda(1-\lambda)^i| \\ &= (1-\lambda)\textstyle\sum_{i=1}^{\infty} |\frac{N_{i-1}}{N} - \lambda(1-\lambda)^{i-1}| \\ &= (1-\lambda)\|Q, P\|_1\end{aligned}$$

In this case, a similar result can be shown for the KL-distance. As we will show later, this implies that the cache gets better and better, and eventually the bias in the cache will tend to zero at a rate of $O(M^{-\lambda})$, where M is the number of values produced after the cache has been primed.

Variations

Recursive stores

Recursive memoization means aggressively reusing values for subexpressions when storing a higher-level expression. Since it seems to be a good idea, maybe we can be similarly aggressive about recursively storing values. That is, while we are evaluating an expression to be stored, we evaluate a recursively called expression and store it. For example, the program

```
let g() = dist [λ :  0,
            1−λ:  1 + dist [
                 p :  store g(),
                 1−p :  reuse g() ] ] in
let x = store g() in
let y = reuse g() in
y - x == d
```

Recursive stores have the advantage that the cache is filled up more quickly, and if a given number of primers are needed they are produced more quickly. However, they are dangerous. With recursive stores, not only are values stored in the cache dependent on previously stored values, the different values stored within a single sample are dependent on each other. The previous result that the new sample is guaranteed to be drawn from a better distribution than the cache distribution cannot be shown in this case. When we test this idea, we find that it performs poorly. The best error is 0.0982, which is somewhat worse than the 0.0930 for recursive memoization without recursive stores.

A related idea is as follows. When we try to reuse an expression, if we do not find the expression in the cache or cannot use the sample in the cache for some reason (perhaps because we don't have enough primers), we do the work of evaluating the expression from scratch. Intuitively it makes sense that we should store the resulting sample — after all, we have gone to the work of generating it. This will result in the cache being filled up more quickly. This can be achieved without any explicit `store` directives; a `reuse` directive is taken to be an implicit `store` if the reuse does not actually happen. This in our example we could use the standard recursive memoization program from earlier. Unfortunately, this idea runs into the same problem as recursive stores. This happens because in recursive memoization we may have calls to a `reuse` expression nested recursively within other calls to the `reuse` expression. If we store samples whenever a `reuse` expression is seen and we fail to actually reuse a sample from the cache, we will end up storing values recursively, and the stored values will be dependent on each other. A quick check shows that this method also works poorly, as its best performance is 0.0997.

Handling observations

A different idea comes into play when values within a program are constrained to match observations. When different subexpressions have the same constraint, we can reuse values from the cache in the same way we have been doing so far. For example, consider the program

```
let f() = dist [ .05 : 0,
                 .95 : 1 + f() ] in
let g() = dist [ .01 : 0,
                 .99 : 1 + g() ] in
let x = f() in
let y = store (obs > x in g()) in
let z = reuse (obs > x in g()) in
z - y == 2
```

In this program, a value `x` is first generated from a geometric with parameter 0.95. The expression `obs > x in g()` means the result of evaluating `g()`, conditioned on the fact that it is greater than `x`. The program stores and reuses the result of `obs > x in g()`. However, recall that when sampling this expression, the algorithm first samples `g()` and then throws a `Reject` exception if the result is not greater than `x`. When the sampling algorithm samples a `store` expression, it handles `Reject` exceptions. If such an exception is caught, the sampler stores the special result `R` in the cache, and then it re-throws the exception. If, later, when reusing a sample from the cache, a `R` result is found, a `Reject` exception is immediately raised and sampling fails.

This is correct, but potentially wasteful. If most of the samples of `g()` are rejected, most values stored in the cache will be `Reject`, and most reuses will result in immediate rejection. It would be better to focus attention on the successful items in the cache. This leads to an importance sampling idea. In importance sampling, the sampler produces weighted samples. The probability of the evidence is estimated to be the average weight of the samples produced, and the probability distribution of a result value is the weighted fraction of successful samples that have that value. A sample that is rejected is interpreted as having weight 0. The idea behind the importance sampling algorithm is this. Instead of storing `Reject` results in the cache, we simply keep track of the number of rejections R and successes S for each (expression, environment) pair, and store only the successful samples together with their weights. When reusing, we choose one of the successful samples x with weight w, and return x with weight $\frac{Sw}{R+S}$. Combining weights for higher-level expressions in terms of subexpressions is straightforward. For example, `sample(if` e_1 `then` e_2 `else` e_3, ν), works as follows. First `sample(`e_1, ν) is called to produce (x_1, w_1). If the x_1 is `true`, `sample(`e_2, ν) is called to produce (x_2, w_2), and $(x_2, w_1 * w_2)$ is returned. Otherwise `sample(`e_3, ν) is called to produce (x_3, w_3), and $(x_3, w_1 * w_3)$ is returned.

When tried on the above program, this idea is mildly successful. Without importance sampling 0.1896 error is achieved, while the importance sampler achieves 0.1831. It remains to be seen whether the gains of the method might be larger in other examples. This is a rudimentary form of importance sampling. The only thing that generates interesting weights is the store/reuse mechanism. A better importance sampling algorithm would actually try to use the observations themselves to guide the production of values. For example, in the above program, we could use the observation that the result of `g()` is greater than `x` to guide the sampler to force that property, weighting the sample appropriately. This is quite difficult to do in a general way. We have developed a general-purpose importance sampling algorithm that uses this idea (Pfeffer 2007b).

General Programs

Thus far we have considered some simple special cases of geometric distributions. The ideas above also apply to general IBAL programs. In general, with recursive memoization the distribution over instances stored in the cache gets better over time, and as a result the error in the cache tends towards zero. We focus initially on the case where there is only one (ϵ, ν) pair that is being stored and reused.

Theorem 1: *Let (ϵ, ν) be the only (expression, environment) pair being stored and reused. Let θ be the expected total number of evaluations (the topmost evaluation plus all the recursive evaluations) of (ϵ, ν) when (ϵ, ν) is evaluated, if caching is not used. Assume that θ is finite. Let $\lambda = 1 - \frac{\theta - 1}{\theta}$. Let $P(x)$ be the correct probability that evaluating (ϵ, ν) will produce x without caching. Let $Q(x)$ be the probability that x will be produced from the cache for (ϵ, ν). Let $R(x)$ be the probability that evaluating (ϵ, ν) will produce x with caching. Then $\|R, P\|_1 \leq (1 - \lambda)\|Q, P\|_1$.*

Proof sketch: Let n^σ be the number of reuses in an evaluation σ of (ϵ, ν). We first show that $E[n^\sigma] = 1 - \lambda$. Each reuse of (ϵ, ν) when caching is used would lead to a recursive evaluation (ϵ, ν) when caching is not used. The recursive evaluation involves in expectation θ evaluations of (ϵ, ν). Thus θ, the expected number of evaluations, is equal to 1 (for the topmost evaluation) $+E[n^\sigma]\theta$. It follows that $E[n^\sigma] = \frac{\theta - 1}{\theta} = 1 - \lambda$.

We construct a Bayesian network to represent the possible evaluations of (ϵ, ν). A node in the network represents an (expression, environment) pair that may possibly be evaluated in the course of evaluating (ϵ, ν).[1] The parents of a node are the subexpressions appearing in the expression with their possible environments. For example, the parents of the node $(\texttt{if}\ \epsilon_1\ \texttt{then}\ \epsilon_2\ \texttt{else}\ \epsilon_3, \nu')$ are the nodes $(\epsilon_1, \nu'), (\epsilon_2, \nu')$ and (ϵ_3, ν'). Note that a node $(\texttt{let}\ X = \epsilon_1\ \texttt{in}\ \epsilon_2, \nu')$ may have countably many parents, corresponding to the different possible environments in which ϵ_2 may be evaluated. The conditional probability distribution of a node implements the semantics of the expression. This Bayesian network is an inverted tree, with the sole leaf being (ϵ, ν). The roots of the network are constants, variables and reuses of (ϵ, ν). Although the network may be infinite, we can define its semantics using the techniques of (Pfeffer & Koller 2000). Because θ is finite, the program is guaranteed to terminate with probability 1, and the network defines a unique distribution.

Next we assign a weight to each node in the network, representing the probability that the node will be evaluated during the course of evaluating (ϵ, ν). These weights are assigned from the leaf upward. For example, if the node $(\texttt{if}\ \epsilon_1\ \texttt{then}\ \epsilon_2\ \texttt{else}\ \epsilon_3, \nu')$ has weight w, then (ϵ_1, ν') has weight w, (ϵ_2, ν') has weight pw where p is the probability (ϵ_1, ν') evaluates to $\texttt{true}$, and (ϵ_3, ν') has weight $(1-p)w$. For any node X, let R^X denote the probability distribution over the node in the network, and let P^X denote its true distribution without caching. Let X be a node in the network with weight w, and let its parents be $U_1, U_2, \ldots$ with weights $w_1, w_2, \ldots$. It can be shown that $\|R^X, P^X\|_1 \leq \frac{1}{w}\sum_i w_i \|R^{U_i}, P^{U_i}\|_1$. This can be verified by considering each expression form in turn. For example, for a node $X = (\texttt{if}\ \epsilon_1\ \texttt{then}\ \epsilon_2\ \texttt{else}\ \epsilon_3, \nu')$ with parents U_1, U_2 and U_3, we have

$$
\begin{array}{cl}
 & \|R^X, P^X\|_1 \\
= & \sum_z |R^X(z) - P^X(z)| \\
= & \sum_z \left| \begin{array}{l} R^{U_1}(\texttt{true})R^{U_2}(z) \\ +R^{U_1}(\texttt{false})R^{U_3}(z) \\ -P^{U_1}(\texttt{true})P^{U_2}(z) \\ -P^{U_1}(\texttt{false})P^{U_3}(z) \end{array} \right| \\
= & \sum_z \left| \begin{array}{l} (R^{U_1}(\texttt{true}) - P^{U_1}(\texttt{true}))R^{U_2}(z) \\ +(R^{U_2}(z) - P^{U_2}(z))P^{U_1}(\texttt{true}) \\ -(R^{U_1}(\texttt{false}) - P^{U_1}(\texttt{false}))R^{U_3}(z) \\ -(R^{U_3}(z) - P^{U_3}(z))P^{U_1}(\texttt{false}) \end{array} \right| \\
\leq & \sum_z \begin{array}{l} |(R^{U_1}(\texttt{true}) - P^{U_1}(\texttt{true}))R^{U_2}(z)| \\ +|(R^{U_2}(z) - P^{U_2}(z))P^{U_1}(\texttt{true})| \\ +|(R^{U_1}(\texttt{false}) - P^{U_1}(\texttt{false}))R^{U_3}(z)| \\ +|(R^{U_3}(z) - P^{U_3}(z))P^{U_1}(\texttt{false})| \end{array} \\
\leq & \begin{array}{l} |R^{U_1}(\texttt{true}) - P^{U_1}(\texttt{true})| \\ +|R^{U_1}(\texttt{false}) - P^{U_1}(\texttt{false})| \\ +\sum_z \left(\begin{array}{l} \frac{w_2}{w}|R^{U_2}(z) - P^{U_2}(z)| \\ +\frac{w_3}{w}|R^{U_3}(z) - P^{U_3}(z)| \end{array} \right) \end{array} \\
= & \frac{w_1}{w}\|R_1^U, P_1^U\|_1 + \frac{w_2}{w}\|R_2^U, P_2^U\|_1 + \frac{w_3}{w}\|R_3^U, P_3^U\|_1
\end{array}
$$

[1]The same (expression, environment) pair may appear at different points during the evaluation. Each appearance is a separate node in the network.

It follows that the error at the leaf, $\|R, P\|$ is at most the weighted sum of the errors at the roots. Roots representing constants and variables have zero error. Roots representing reuses have error $\|Q, P\|$. Now, the weight of a reuse root is the expected number of times that particular reuse will occur. Thus the total weight of the reuse roots is the expected number of reuses, i.e. $E[n^\sigma]$, which we have shown to be equal to $1 - \lambda$. Therefore $\|R, P\| \leq (1-\lambda)\|Q, P\|$. ∎

It follows from this result that the new cache after a sample will be better than the old cache. In particular, let Q^1 be the new cache distribution after one new sample. Then, by convexity of the 1-norm,

$$
\begin{array}{rcl}
\|Q^1, P\|_1 & = & \|\frac{N}{N+1}Q + \frac{1}{N+1}R, P\|_1 \\
 & \leq & \frac{N}{N+1}\|Q, P\|_1 + \frac{1}{N+1}\|R, P\|_1 \\
 & = & \frac{N+1-\lambda}{N+1}\|Q, P\|_1
\end{array}
$$

Inductively, if Q^i is the cache distribution after i new samples, we get

$$
\begin{array}{rcl}
\|Q^M, P\|_1 & \leq & \frac{(N+1-\lambda)\cdots(N+M-\lambda)}{(N+1)\cdots(N+M)}\|Q, P\|_1 \\
 & = & \frac{\Gamma(N+1)\Gamma(M-\lambda)}{\Gamma(N+1-\lambda)\Gamma(M)}\|Q, P\|_1
\end{array}
$$

where Γ is the Gamma function. According to Stirling's formula (Artin 1964),

$$\Gamma(x) \approx \sqrt{2\pi}x^{x-\frac{1}{2}}e^{-x+\frac{\alpha}{12x}}$$

where α is a constant between 0 and 1. This implies that

$$\frac{\Gamma(M-\lambda)}{\Gamma(M)} \approx \left(\frac{M-\lambda}{M}\right)^{M-\frac{1}{2}} \left(\frac{1}{(M-\lambda)^\lambda}\right) e^{-\lambda}e^{\frac{\alpha\lambda}{12M(M-\lambda)}}$$

The first term is $O(1)$, the second is $O(M^{-\lambda})$ and the last is $O(1)$. Therefore the error in the cache after M samples is $O(M^{-\lambda})$.

We would expect from this that the performance of recursive memoization would be worse when λ is small. On the other hand, when λ is small more computation is saved, and therefore relatively more samples can be taken. For the geometric distribution example, we have seen that the savings are just as great when $\lambda = 0.01$ as when $\lambda = 0.05$.[2]

Similar ideas carry over to the general case in which there may be many (ϵ, ν) pairs being stored and reused. However the situation is more complex. It may be possible for R to be worse than Q for a particular (ϵ, ν) pair in a particular evaluation. This can happen when one function calls another recursive function many times. However, in this situation, the second recursive function cannot call the first one many times, or else the computation would diverge. Therefore, intuitively, things get better overall. In fact, one can prove the following theorem:

Theorem 2: *Let $P_{\epsilon,\nu}$ denote the true distribution for (ϵ, ν), $Q_{\epsilon,\nu}$ denote the original cache distribution, and $R^k_{\epsilon,\nu}$ denote the distribution over the value produced in the k-th iteration starting with the original cache. Define the metric $\|Q, P\| = \max_{\epsilon,\nu}\|Q_{\epsilon,\nu}, P_{\epsilon,\nu}\|_1$. Then, if the number of (ϵ, ν) pairs that gets stored and reused is finite, there exists a finite K such that $\|R^K, P\| < \|Q, P\|$.*

[2]For the geometric distribution, λ as defined in this section and λ as defined earlier are equal.

Proof sketch: Let i and j denote (expression,environment) pairs. Let n_{ij}^k denote the number of recursive calls to j at depth k when i is evaluated. Note that

$$E[n_{ij'}^k] = \sum_j E[n_{ij}^1] E[n_{jj'}^{k-1}]$$

Let Q^k denote the cache distribution after k iterations. Q^k is a mixture of $\frac{N}{N+k}$ parts of Q, and $\frac{1}{N+k}$ parts of each of R^ℓ for $\ell = 1, ..., k-1$.

We will show that we can write

$$\|R^\ell, P\| \leq \sum_{m=1}^{\ell} a_m^\ell \sum_j E[n_{ij}^m] \|Q, P\| \qquad (1)$$

where

$$\begin{array}{lcl} a_1^\ell & = & \frac{N}{N+\ell-1} \\ a_m^\ell & = & \frac{1}{N+\ell-1} \sum_{\ell'=m-1}^{\ell-1} a_{m-1}^{\ell'} \quad \text{for } m > 1 \end{array}$$

This is proved by induction. The base case follows from the Bayesian network construction in the proof of Theorem 1. For the induction step, it follows from the same Bayesian network construction that

$$\begin{array}{ll} & \|R_i^{k+1}, P_i\|_1 \\ \leq & \sum_j E[n_{ij}^1] \|Q_j^k, P_j\|_1 \\ \leq & \sum_j E[n_{ij}^1] \| \frac{1}{N+k} \sum_{\ell=1}^k R_j^\ell + \frac{N}{N+k} Q_j, P_j\|_1 \\ \leq & \sum_j E[n_{ij}^1] (\frac{1}{N+k} \sum_{\ell=1}^k \|R_j^\ell, P_j\|_1 + \frac{N}{N+k} \|Q_j, P_j\|_1) \\ \leq & \sum_j E[n_{ij}^1] \left(\begin{array}{l} \frac{1}{N+k} \sum_{\ell=1}^k \sum_{m=1}^\ell \sum_{j'} \\ \qquad E[n_{jj'}^m] \|Q_{j'}, P_{j'}\|_1 \\ + \frac{N}{N+k} \|Q_j, P_j\|_1 \end{array} \right) \\ = & \begin{array}{l} \frac{1}{N+k} \sum_{m=1}^k \sum_{\ell=m}^k a_m^\ell \sum_j E[n_{ij}^1] \sum_{j'} \\ \qquad E[n_{jj'}^m] \|Q_{j'}, P_{j'}\|_1 \\ + \frac{N}{N+k} \sum_j E[n_{ij}^1] \|Q_j, P_j\|_1 \end{array} \\ = & \begin{array}{l} \frac{1}{N+k} \sum_{m=1}^k \sum_{\ell=m}^k a_m^\ell \sum_j E[n_{ij}^{m+1}] \|Q_j, P_j\|_1 \\ + \frac{N}{N+k} \sum_j E[n_{ij}^1] \|Q_j, P_j\|_1 \end{array} \end{array}$$

The following two claims can be proved by induction:

1. $\forall \ell, \sum_{m=1}^\ell a_m^\ell = 1$
2. $\forall \epsilon \forall m, \exists K_m^\epsilon \text{s.t.} a_m^\ell < \epsilon, \forall \ell \geq K_m^\epsilon$

Now, let θ_i denote the total expected number of function calls (including the topmost call) when i is evaluated. Because the total expected number of calls is the sum of the expected number of calls at all depths, it is clear that $\sum_{k=1}^\infty \sum_j E[n_{ij}^k] = \theta_i - 1$. It follows that there is a H such that $\sum_j E[n_{ij}^m] < \frac{1}{2}$, for all $m \geq H$.

Let $\epsilon = \frac{1}{2H(\theta_i - 1)}$, and let $K = \max_{m=1}^{H-1} K_m^\epsilon$ from claim 2. Then $\sum_{m=1}^{H-1} a_m^K \sum_j E[n_{ij}^m] \leq (H-1)\epsilon(\theta_i - 1) < \frac{1}{2}$. By claim 1, $\sum_{m=H}^K a_m^K \sum_j E[n_{ij}^m] < \frac{1}{2}$. Therefore, from Equation 1, $\|R^k, P\| < \|Q, P\|$, as desired.

∎

Larger Experiments

To test memoization on larger models, we used the game of duplicate bridge. Bridge is a card game for four players in two pairs, one of which sits North-South and the other East-West. The cards are dealt to the players, and the game then consists of two phases. In the bidding phase, players hold an auction. The pair with the winning bid undertakes a contract to take a certain number of "tricks" in the second phase. The second phase, known as the play, consists of a sequence of thirteen tricks. In a trick each player plays a card, and the pair with the winning card takes the trick. At the end of the play, the two pairs receive a score depending primarily on whether the contract was achieved. There is a bonus for achieving contracts of specifi c levels.

In duplicate bridge, there are a certain number of tables, each involving a N-S pair playing an E-W pair. All of the tables involve the same deal of cards. At each table, the two phases of the game are played, and a score is received. The scores of all the N-S pairs are then compared, and each pair obtains a score depending on how many other N-S pairs it beat. A similar procedure is applied to the E-W pairs. Thus the true opponents of a N-S pair are not the E-W pair at the same table, but the other N-S pairs. Playing well involves not only choosing actions that lead to a good score at one's own table, but also envisioning what pairs at other tables might be doing.

Excellent bridge programs have been developed in GIB (Ginsberg 1999) and Bridge Baron (S.J. Smith & Throop 1998). We have implemented a bridge simulator in IBAL. A simulator along these lines might be used in a bridge playing program to try to predict what other players might do. The simulator works from the point of view of a particular player at a particular table, (whom we shall call the owner), who is considering what bid to make after a certain bidding sequence. The owner's cards and previous bids are given, and do not vary from sample to sample. First, the simulator randomly deals the remaining cards to the other players. Then, it proceeds to process the bidding at the table of the owner. For previous bids of other players at the table, it observes that the bid generated by the simulator matches the bid they actually made. If not, the sample is rejected. Once all previous bids have been checked, the owner makes a decision, and the rest of the bidding and play at that table are simulated. Then the bidding and play for all the other tables are simulated, using the same cards. Finally a score is obtained, and recorded along with the decision of the owner of the simulation.

We tried our simulator on a deal with a moderately likely bidding sequence, using both an unmemoized sampler, and a sampler that memoized the play phase. We measured the accuracy of the estimate, its variance, and the percentage of time that the algorithm made the correct bidding decision. We also show the number of samples agreeing with the observations obtained. 10 seconds were allocated to the sampling time, and 1000 tests were taken. The results, shown in the following table, show that memoized sampling does signifi cantly better according to all three measures.

Strategy	Samples	Error	Variance	% correct
No memo	22.4	1.28	2.59	93.1
Memo	86.7	0.761	0.926	99.0

This is a good example for illustrating the role of different sampling algorithms. The problem can be decomposed into two parts: sampling likely hands for the other players given their bids, and sampling possible plays of the cards. Rejection sampling works well for the former problem when bidding sequences are reasonably likely, but runs into trouble with unlikely bidding sequences. In a real bridge application, we would probably use MCMC for the first problem. Much progress has been made on MCMC algorithms for general probabilistic models (Milch & Russell 2006). However, rejection sampling (which is just forward sampling in this case) is an excellent algorithm for the second problem, which requires simply generating values from the simulator.

We also performed experiments on a small probabilistic context free grammar (PCFG), using the example in Chapter 5 of (Charniak 1993). The task was to predict a missing word given the presence of other words. A PCFG can be easily encoded in IBAL. Each non-terminal symbol is represented by a recursive function. A production such as $S \xrightarrow{p} AB$ involves calling the function for A, calling the function for B, and concatenating the results. This example involves recursive memoization, with several mutually recursive functions calling each other. Two types of memoization strategies were used. In the first, some of the calls to non-terminal functions were always stores, and some were always reuses. We see from the table below that the gains from this strategy were modest, even though twice as many examples were taken. The second strategy uses driving forward momentum. The idea is that every time a production is encountered, it is guaranteed that one of the non-terminals on the right hand side will be evaluated, so new values for the cache are always produced. However, the choice of which non-terminal to evaluate is stochastic. So for example the production $S \xrightarrow{0.6} AB$ is turned into the expression

```
dist [0.3:append(store A(), reuse B())
      0.3:append(reuse A(), store B())]
```

As the table shows, the gains are quite significant.

Strategy	Samples	Error	Variance
No memo	6037	0.1157	0.005108
Memo 1	11425	0.1004	0.003852
Memo 2	11627	0.0663	0.001641

Conclusion

This paper has explored the idea of memoization in sampling. It has shown that the idea of reusing past samples is beneficial, and leads to significantly lower error. Furthermore, being aggressive about using memoization recursively produces additional significant improvements. While recursive memoization leads to elements of the cache being dependent on each other, we have shown that over time the error of the cache decreases to zero, so we can expect the sampler to be unbiased in the long run. We have demonstrated the efficacy of memoized sampling with a duplicate bridge simulation and with stochastic grammars.

Further work is needed to determine when exactly the method is likely to be effective. Clearly, if it is very cheap to draw from a distribution, there is no point in caching it and reusing samples from the cache. Using the cache needs to be significantly cheaper than drawing from the distribution to be worthwhile, but how much cheaper does it have to be? Another issue is the space cost of the algorithm. As more and more samples are taken, the cache could become very large. One idea is to cycle values out of the cache as it becomes full, but whether or not the cache will get better over time needs to be explored.

Rejection sampling is only one family of algorithms. While it has a useful role to play, other algorithms may be more appropriate in many circumstances. As mentioned earlier, we have developed a general importance sampling algorithm for IBAL, and we hope that many of the ideas in this paper will carry over there. Extending the ideas to MCMC is an important direction for future work.

References

Artin, E. 1964. *The Gamma Function*. Holt, Rinehart and Wilson, New York. Translated by Michael Butler.

Charniak, E. 1993. *Statistical Language Learning*. MIT Press.

Gilks, W.; Richardson, S.; and Spiegelhalter, D. 1996. *Markov Chain Monte Carlo in Practice*. Chapman & Hall/CRC.

Ginsberg, M. 1999. GIB: Steps toward an expert-level bridge-playing program. In *International Joint Conference on Artificial Intelligence*.

Milch, B., and Russell, S. 2006. General-purpose MCMC inference over relational structures. In *Uncertainty in Artificial Intelligence (UAI)*.

Pfeffer, A., and Koller, D. 2000. Semantics and inference for recursive probability models. In *National Conference on Artificial Intelligence (AAAI)*.

Pfeffer, A. 2007a. The design and implementation of IBAL: A general purpose probabilistic language. In Getoor, L., and Taskar, B., eds., *Statistical Relational Learning*. MIT Press. In press.

Pfeffer, A. 2007b. A general importance sampling algorithm for probabilistic programs. In submission.

Robert, C., and Casella, G. 2004. *Monte Carlo Statistical Methods*. Springer-Verlag, New York, 2nd edition.

S.J. Smith, D. N., and Throop, T. 1998. Success in spades: Using AI planning techniques to win the world championship of computer bridge. *AI Magazine* June.

Srinivasan, R. 2002. *Importance Sampling*. Springer-Verlag, New York.