Combining Static and Dynamic Optimizations Using Closed-Form Solutions

Daniel Lundén*
KTH Royal Institute of Technology
Sweden

David Broman*
KTH Royal Institute of Technology
Sweden

Lawrence M. Murray*
Uppsala University
Sweden

Abstract

It is sometimes possible to optimize probabilistic programs, either statically or dynamically. We introduce two examples demonstrating the need for both approaches. Furthermore, we identify a set of challenges related to the two approaches, and more importantly, how to combine them.

1 Introduction

In probabilistic programming [2, 3, 6, 8], certain programs have substructures that can be optimized using closed-form solutions. These optimizations can be performed either statically, before execution of a probabilistic program, or dynamically, during execution of a probabilistic program. In this extended abstract, we discuss the need for both of these approaches. We also identify more general challenges related to combining static and dynamic optimizations for probabilistic programs.

The closed-form solutions are as follows: for two random variables X and Y with distributions $p(x)$ and $p(y|x)$, we can find both the marginal distribution $p(y) = \int_X p(y|x)p(x)dx$ and the conditional distribution $p(x|y)$ (given that y is observed) in closed-form. This is, for instance, the case when $p(x)$ is a *conjugate prior* for the likelihood function $p(y|x)$. As a consequence, we can analytically condition some unknown random variables in our program on the observed random variables. We have previously developed a systematic and correct method for doing this dynamically for importance sampling methods, including Sequential Monte Carlo (SMC) methods [1], called *delayed sampling* [4, 5]. Using delayed sampling for SMC gives potentially increased effective sample sizes and reduced variance for estimators based on the generated samples.

2 The need for dynamic optimization

If a probabilistic program simply represents a probabilistic graphical model, we can identify the previously introduced closed-form solutions before executing the program. As a consequence, it is easy to encode (and possibly even further optimize) the operations performed by delayed sampling in the program statically.

One of the motivations of probabilistic programming is, however, to *add* expressiveness compared to probabilistic graphical models, which are inherently static. This increased

```
(defquery dynamic
  (let [data [0 -1.1 2.4 1.2 -0.1 -1.4 -1.9]
        x (sample (normal 0 1))
        mix (fn [anc]
              (if (sample (flip 0.5))
                (normal anc 1)
                (normal 0 (+ 1 (abs anc)))))
        foo (fn foo [root depth]
              (let [left (mix root)
                    right (mix root)]
                (if (= depth 1)
                  [left right]
                  (concat (foo (sample left)
                               (- depth 1))
                          (foo (sample right)
                               (- depth 1))))))
        leaves (foo x 3)]
    (map (fn [dist obs] (observe dist obs))
         leaves data) x))
```

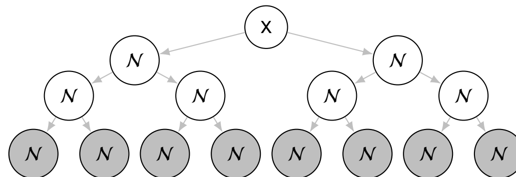


Figure 1. A program (written in Anglican [8]) in which the dependencies between normal random variables may, or may not, have a closed-form solution. This is decided during execution by flipping a coin. The function `flip` returns either true or false, with 50% probability.

expressiveness is due to two features of probabilistic programming: recursion (possibly infinite number of random variables) and stochastic branching (random variables affect control flow). These additions make it possible for a probabilistic program to give *different* probabilistic graphical models for each simulation of the program. Here, different means that there can be a different number of random variables, different distributions, different types of relationships between random variables, and even different topologies for the graphical models generated when simulating the program. Consequently, there is possibly an infinite number of graphical models for a single probabilistic program. As such, it sometimes becomes infeasible to exploit our closed-form solutions statically.

As an example, consider the program with corresponding graphical model in Figure 1. For each node in the graph (not including the root node), there is a 50% chance that

*Financially supported by the Swedish Foundation for Strategic Research (ASSEMBLE RIT15-0012).

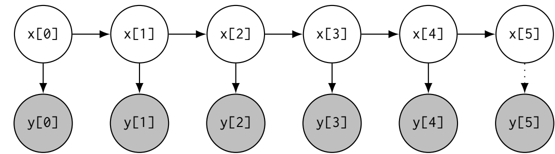
a closed-form solution[1] between it and its ancestor exists. Attempting to exploit these closed-form solutions statically entails considering 2^{14} possible graphical models, all of which must be handled differently (see [4, 5]). The combinatorial difficulties become even more apparent if the topology of the graph is also allowed to change between simulations. This demonstrates that a dynamic approach operating during execution (such as delayed sampling) is essential for this particular optimization in probabilistic programs.

3 The need for static optimization

As already mentioned, if the probabilistic program simply represents a graphical model, we can perform our optimizations statically. If we instead perform the optimizations dynamically, there would be a certain overhead in maintaining algorithm data structures. Possibly, we would also repeat calculations in each run of the program.

Figure 2a demonstrates a probabilistic program in the form of a linear-Gaussian state-space model. In the program, no random variables affect control flow, making this a straightforward graphical model. Therefore, we can, and should, optimize the program statically. We do this by repeated application of the closed-form solutions between the normal variables (i.e., we apply a Kalman filter). The result is shown in Figure 2b. The very last observation `y[5]` does not have a closed-form solution with its ancestor, and can therefore not be optimized. Using a dynamic approach such as delayed sampling would, for this program, lead to both unnecessary overhead and repeated marginalization and conditioning calculations.

```
(defquery static
  (let [data [0.4 0.9 -0.1 -1.3 0.2 2.1]
        x (sample (normal 0 1))]
    (observe (normal x 1) (first data))
    (loop [x x, data (rest data)]
      (if (seq data)
        (let [x (sample (normal x 1))]
          (observe (if (> (count data) 1)
                     (normal x 1)
                     (normal 0 (+ 1 (abs x))))
                   (first data))
          (recur x (rest data))) x))))
```



(a)

```
(defquery static-opt
  (let [data [2.1]
        x (sample (normal -0.157 (sqrt 1.617)))]
    (observe (normal 0 (+ 1 (abs x))) (first data)) x))
```

(b)

Figure 2. Figure (a) shows a program (written in Anglican [8]) with neither stochastic branching nor recursion, and the corresponding graphical model. The dotted edge for the very last observation denotes the lack of a closed-form solution. Figure (b) shows an optimized version of the same program, where the observations `o[0]` to `o[4]` have been absorbed into the distribution for `x[5]`, and where `x[0]` to `x[4]` have been marginalized out.

4 Related work

While there is much work in both exact and approximate inference for probabilistic programs, there is not much work presented for optimization of probabilistic programs. Hakaru [6] is a probabilistic programming language that uses symbolic computation in Maple for static optimization of programs. Another language is R2 [7], which statically optimizes programs by propagating observations backwards.

5 Challenges and conclusion

We have introduced two examples showing the need for both static and dynamic optimizations. An important question remains: how can we decide when to use which approach? Remember that dynamic optimization is needed when the graphical models produced by a single program can change significantly between simulations. This introduces a future challenge in probabilistic programming—what do we mean by significant change between program runs? Can we define some measure of this property that we can subsequently use to decide between static and dynamic optimizations?

Furthermore, there can also be cases when a *combination* of both approaches is appropriate. Simple parts of a program can be statically optimized, while other, more complicated parts, must be optimized dynamically. To automatically separate these two cases within one program is a challenging task that deserves further exploration. Also, is there such a thing as an *optimal* way of combining static and dynamic optimizations? Can we define this formally?

The two approaches have separate challenges related to them as well. A dynamic approach for optimization requires a proof of correctness for an algorithm that operates during execution of the program. A static approach instead involves program transformations, and requires that the transformed program is *equivalent* to the original program. This also raises the important issue of establishing equivalence between two probabilistic programs.

In summary, we argue that there is an important need for optimizing probabilistic programs using closed-form solutions. In particular, we contend that this should be done by combining static and dynamic optimization techniques.

[1] We can find both the marginal and conditional distributions described in Section 1 if we have a normally distributed random variable as the mean of another normally distributed variable.

References

[1] Arnaud Doucet, Nando de Freitas, and Neil Gordon. 2001. *Sequential Monte Carlo Methods in Practice.* Springer New York.

[2] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A language for generative models. In *In UAI*. 220–229.

[3] Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. (2014). Accessed: 2017-10-16.

[4] Daniel Lundén. 2017. *Delayed sampling in the probabilistic programming language Anglican.* Master's thesis. KTH Royal Institute of Technology.

[5] Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. 2017. Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. (2017). arXiv:1708.07787

[6] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*. Springer, 62–79.

[7] Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI'14)*. AAAI Press, 2476–2482.

[8] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*. 1024–1032.