flip-hoisting: Exploiting Repeated Parameters in Discrete Probabilistic Programs

Ellie Y. Cheng
MIT

Todd Millstein
UCLA

Guy Van den Broeck
UCLA

Steven Holtzen
Northeastern University

Abstract

Many of today’s probabilistic programming languages (PPLs) have brittle inference performance: the performance of the underlying inference algorithm is very sensitive to the precise way in which the probabilistic program is written. A standard way of addressing this challenge in traditional programming languages is via program optimizations, which seek to unburden the programmer from writing low-level performant code, freeing them to work at a higher level of abstraction. The arsenal of applicable program optimizations for PPLs to choose from is scarce in comparison to traditional programs; few of today’s PPLs offer significant forms of automated program optimization. In this work we develop a new family of program optimizations specific to discrete-valued knowledge compilation based PPLs. We identify a particular form of program structure unique to these PPLs that tangibly affects exact inference performance in these programs: redundant random variables – variables with repeated parameters and inconsistent path conditions. We develop a new program analysis and associated optimization called flip-hoisting that identifies these redundancies and optimizes them into a single random variable. We show that flip-hoisting yields inference speedups of up to 60% on applications of probabilistic programs such as Bayesian networks and probabilistic verification.

1 INTRODUCTION

Probabilistic programming languages (PPLs) have immense promise as a general-purpose and widely accessible probabilistic modeling solution, but they suffer from brittle inference performance: two programs with the same semantics, written with subtly different syntax, can have drastically different inference performance. As a consequence, many of today’s PPLs are difficult to use for non-experts that cannot navigate the subtle ways in which the structure of the program impacts inference. For example, Gorinova et al. (2020) showed that subtle re-parameterizations of programs, such as centering, yields radically different inference performance for Hamiltonian Monte Carlo inference; an average end-user should ideally be blissfully unaware of this.

The standard way in which the challenge of brittle performance is addressed in traditional programming languages is via program optimizations (Aho et al., 1986). Optimizations permit the programmer to work at a high level of abstraction, and delegate to the compiler the job of synthesizing a more efficiently written program: this separation of concerns is critical to the design of modern compilers and programming languages such as LLVM (Lattner, 2008). However, in contrast to traditional programming languages, there is a much broader variety of optimizations for PPLs due to the large number of possible inference algorithms: the effect an optimization might have on a program is intimately connected with which inference algorithm is ultimately used. For instance, Gorinova et al. (2020) identified an optimization that is specific to HMC, and Huang et al. (2017) identified optimizations specific to parallel Gibbs sampling. An optimization that may work for a PPL may be ineffective in another. Hence, optimizing general-purpose PPLs necessitates developing a broad suite of optimizations specific to many kinds of inference algorithms.

One particular family of inference algorithms in need of new optimizations is those of discrete PPLs. Discrete PPLs such as Dice and ProbLog enable new forms of inference since they only apply to programs with discrete random variables (Holtzen et al., 2020; Eijers et al., 2011). The dominant form of inference for these kinds of PPLs is exact inference via knowledge compilation (Darwiche and Marquis, 2002; Sang et al., 2008; Chavira and Darwiche, 2008; Niepert and Domingos, 2015; Mateescu et al., 2008b; Chavira et al., 2006; Sanner and McAllester, 2005). The core idea of the approach is to compile the probabilistic program into a representation that affords fast exact inference.

The performance of exact inference via knowledge compilation is sensitive to the way in which programs are written in ways that can be surprising or unintuitive for novice programmers. We identify one particularly pernicious form of program structure that is easy for a novice programmer
to accidentally introduce and has a negative impact on inference performance: redundant random variables. Two syntactic random variables are redundant if they have the same parameters and have inconsistent path conditions. Redundancies do not affect the performance of sampling-based inference algorithms: a single path through the program can never encounter both redundant random variables. However, redundancies can significantly impact knowledge compilation based inference methods like Dice because it performs exact inference, and hence implicitly represents the joint configuration of all random variables. In this work, we propose a new family of probabilistic program optimizations called flip-hoisting that unifies redundant random variables into a single random variable through program analysis and transformation. This optimization is specific to the knowledge compilation approach utilized by discrete PPLs such as Dice. The key behind our approach is a sound but incomplete branch-sensitive analysis of probabilistic programs that statically determines when it is safe to re-use a random variable. This analysis is efficient in the size of the program and is effective on realistic examples.

In Section 1.1 we give a motivating example that provides intuition behind how flip-hoisting works. Section 2 formally describes the flip-hoisting procedure and argues for its correctness. Section 3 describes how to implement flip-hoisting for Dice and demonstrates empirically that flip-hoisting is an effective optimization by showing (1) that practical problems that exhibit opportunities for flip-hoisting exist in the wild on examples from probabilistic graphical models and probabilistic verification; and (2) that our automated flip-hoisting approach reduces the runtime of Dice inference on these examples, giving new state-of-the-art performance for probabilistic programming languages on these tasks. Section 4 gives an overview of related work. Section 5 concludes.

1.1 Motivating Example and Overview

Throughout this paper, examples will be written in Dice (Holtzen et al., 2020). Dice is a functional probabilistic programming language that supports discrete random variables; in Dice, the syntax flip θ denotes a Boolean random variable that is true with probability θ. We introduce subscripts to flips in the examples as auxiliary notation to uniquely refer to them in our discussion.

Consider the example probabilistic program in Figure 1a. The program encodes a simple distribution with a Boolean random variable y that is conditioned on the values of x and z, which are themselves Boolean random variables with probability 0.1 and 0.2 respectively. The program has some potentially redundant flips; flip 0.3 occurs twice in the program as flip3 and flip5. So, we ask: can we optimize this program to a version where these flip 0.3 occurs exactly once? Intuitively, instead of flipping two coins, we would like to flip a single coin instead, without changing the semantics of the program. We can indeed do so by hoisting, wherein we introduce a variable tmp in order to use a single flip in place of the original two occurrences. In Figure 1b we show the optimized version of the program after hoisting flip3 and flip5.

Removing redundant flips is an optimization that specifically improves the performance of knowledge compilation, similar to how centering is an optimization unique to improving HMC (Gorinova et al., 2020). To see this, we will inspect some details of how Dice inference works. Dice compiles probabilistic programs into binary decision diagrams (BDDs) and computes queries through weighted model counting (Holtzen et al., 2020). Each of the nodes in a BDD corresponds to a flip in the program; the solid edge leaving a node has a weight equal to the parameter θ of the flip and the dotted edge leaving a node has a weight equal to 1 − θ. Performing weighted model counting on a BDD then includes tracing through each path from the root nodes to the root. Figure 2 shows the BDD of Figure 1a.
with each node labeled with the \texttt{flip} subscripts. When a \texttt{flip} is hoisted the BDD consequently reduces in size as redundant nodes are removed. Combining \texttt{flip}_3 and \texttt{flip}_5 results in the node being shared in the corresponding BDD in Figure\textsuperscript{[24]}, resulting in a smaller BDD. The size of the BDD is the critical component in determining the ultimate runtime of \texttt{Dice} inference, and so reducing size has practical improvements on runtime (Holtzen et al., 2020).

We have shown that \texttt{flip} 0.3 can be optimized to occur only once instead of twice. Can we do the same for \texttt{flip} 0.2, which also occurs twice in the example program? The answer is no – sound hoisting is not a simple matter of unifying all syntactically identical random variables. Consider the \textit{invalid} program in Figure\textsuperscript{[1c]} in which we hoist \texttt{flip}_2 and \texttt{flip}_4. This hoisting of the two \texttt{flips} changes the semantics of the original program. In the original program it is possible that \( x = \text{true}, z = \text{false}, \text{and } y = \text{true} \); this assignment is not possible in the hoisted version since \( y \) is constrained to be equal to \( z \) in this case. This invalid hoisting introduced a \textit{spurious dependence} between the two \texttt{flips}, coupling them when the semantics of the original program depended on them being decoupled. The spurious dependence induces incorrect inference results as the semantics of the original program has been altered.

What is a sufficient condition for ensuring that a hoisting is valid? One is that there is no path through the probabilistic program that encounters both \texttt{flips}; we call such \texttt{flips} \textit{redundant}. This is the case for \texttt{flip} 0.3. The key observation is that the \texttt{if}-expression on Line\textsuperscript{[3]} in the original program in Figure\textsuperscript{[1a]} has mutually exclusive branches, guaranteeing that the path conditions for the two separate instances of \texttt{flip} 0.3 are inconsistent; in this case, it is safe to flip a single coin instead of two without changing the semantics of the program. We note that while our motivating example only has Bernoulli random variables, our approach naturally applies to categorical distributions as well, since they can be represented as a sequence of Bernoulli random variables (Holtzen et al., 2020; Sang et al., 2005).

In this paper we give the first analysis for identifying and merging redundant \texttt{flips} in probabilistic programs. In general, determining if it is safe to merge two \texttt{flips} is a special case of \textit{reachability analysis}, and so is undecidable for general programs. Hence, we identify a sound but incomplete strategy that is effective for existing \texttt{Dice} programs (Aho et al., 1986). For example, our approach is able to perform the hoisting shown in Figure\textsuperscript{[1b]} automatically, and never performs the invalid hoisting shown in Figure\textsuperscript{[1c]}. We will show that \texttt{flip}-hoisting is efficient in the size of the program, prove that it is sound, and show empirically that it improves the performance of existing probabilistic programs derived from graphical models and verification.

### 2 \textbf{flip-HOISTING}

In this section we formally introduce \texttt{flip}-hoisting. In order to unambiguously refer to \texttt{flips} in a program, we assume that each one has a unique identifier; we assign these identifiers as subscripts as shown in Figure\textsuperscript{[1a]}. We denote syntactic probabilistic programs as \( p \) and let \([p]\) denote the probability distribution on the values returned by \( p \).

\textit{Hoisting}, denoted \texttt{hoist}(\( p, i, j \)), is the function that transforms a program \( p \) by inserting a new \texttt{flip} – the hoisted \texttt{flip} – at an ancestor node of \texttt{flip}_i and \texttt{flip}_j in the abstract syntax tree (AST) – \texttt{ancestor}(\( p, i, j \)) – and replacing \texttt{flip}_i and \texttt{flip}_j with a reference to the hoisted \texttt{flip}. \texttt{hoist}(\( p, i, j \)) assumes \texttt{flip}_i and \texttt{flip}_j have the same parameter.\textsuperscript{[7]}

For instance, if \( p_{ex} \) is the program in Figure\textsuperscript{[1a]} then \texttt{hoist}(\( p_{ex}, 3, 5 \)) outputs the program in Figure\textsuperscript{[1b]}: Hoisting itself is efficient and implementable as a single pass over the syntax of the program; the challenge is knowing when hoisting is \textit{sound}:

\textbf{Definition 1} (Sound hoisting). For a probabilistic program \( p \), hoisting \texttt{flips} \( i \) and \( j \) is sound if \([p] = [\text{hoist}(p, i, j)]\).

This definition does not yield an algorithm because determining whether a hoisting is sound is hard: it requires reasoning about whether or not two probabilistic programs are equivalent, which is a challenging computational task. Hence we require an assumption that aids in implementation. One route is analyzing the \textit{path conditions} for each \texttt{flip}, a familiar concept from symbolic execution (King, 1976):

\textbf{Definition 2} (Path conditions). The path conditions \( \text{PC}(p, i) \) for \texttt{flip}_i in probabilistic program \( p \) is the set of necessary and sufficient conditions on \texttt{flips} that ensure execution reaches \texttt{flip}_i.

For instance, \( \text{PC}(p_{ex}, 3) = \text{flip}_1 \wedge \text{flip}_2 \), since if \texttt{flip}_3 is executed then the condition of the \texttt{if}-statement on Line\textsuperscript{[3]} of Figure\textsuperscript{[1a]} must be true, which implies that these two \texttt{flips} are true. The path conditions yield a more tractable test for \texttt{flip} redundancy:

\textbf{Theorem 1} (Path redundancy). For program \( p \), hoisting \texttt{flips} \( i \) and \( j \) is sound if the \texttt{flips} have the same parameter value and \( \text{PC}(p, i) \) is inconsistent with \( \text{PC}(p, j) \).

\textbf{Proof sketch.} A trace \( \tau \) through a program \( p \) is a total assignment to \texttt{flips} encountered during an execution of the program; for example, one trace through Figure\textsuperscript{[1a]} is \( \tau = \{ \text{flip}_1 = \text{true}, \text{flip}_2 = \text{false}, \text{flip}_4 = \text{true} \} \). This information uniquely determines the result of the execution along this trace and the probability of the trace.

\textsuperscript{[7]}In general there are multiple possible places where the new \texttt{flip} can be inserted, but that is irrelevant here.
There is a bijection between the traces of $p$ and the traces of $hoist(p, i, j)$ that ensures that the programs are equivalent. In other words, a tuple of traces is in the bijection if both the resulting values and probabilities of the traces are equivalent. Thus, the bijection implies that the two programs have the same resulting probability distributions.

The most interesting traces are those that encounter one of the hoisted flips — we will give a bijection between the traces of $p$ and $hoist(p, i, j)$ for this case. Consider a trace $\tau$ through $p$ that includes flip$_i$. Since PC$(p, i)$ is inconsistent with PC$(p, j)$, it must be the case that $\tau$ does not include flip$_j$. Now let $p_h = hoist(p, i, j)$, and assume that flip$_i$ is hoisted to and relabeled to a new variable flip$'$ in the hoisted program. By the definition of the hoist function there exists a trace $\tau'$ through hoist$(p, i, j)$ that is identical to $\tau$ but with flip$_j$ replaced by flip$'$ with the same value in the trace. Similarly, for each trace through $hoist(p, i, j)$ that goes through the line of code where flip$_i$ was in $p$, there exists an equivalent trace in $p$ containing flip$_i$ in place of flip$'$.

Path redundancy reduces the problem of checking hoisting soundness to satisfiability, which is still too computationally hard to be implemented as a practical optimization. Next, we consider two strengthenings of path redundancy that capture common cases occurring in probabilistic programs; these strengthenings yield efficient soundness checks.

2.1 Local Hoisting

One of the strictest strengthenings of path redundancy is local redundancy, which avoids the need to reason about the intricacies of if-statement conditions altogether.

**Definition 3** (Locally redundant flips). Two flips $i$ and $j$ are locally redundant if they have the same parameter value and appear in disjoint branches of the same if-statement.

Determining whether or not two flips are locally redundant is an efficient syntactic check on the program. Figure 1a gives an example where flip$_3$ and flip$_5$ are locally redundant: they occur in disjoint branches of the if-expression on Line 3.

**Proposition 2.** It is sound to hoist locally redundant flips.

The proof follows from Theorem 1 and the fact that two locally redundant flips by definition have inconsistent path conditions.

While seemingly simple, local hoisting is already a surprisingly powerful and general optimization. Further, local hoisting applies to a surprisingly broad set of programs including those encoding Bayesian networks as well as existing programs from probabilistic verification.

2.2 Global Hoisting

Local hoisting identifies hoisting opportunities within a single if-expression. In this section we develop a separate analysis, which we call global hoisting, that retains the tractability of local hoisting while finding hoisting opportunities that span multiple ifs.

Consider the minimal example in Figure 3a that we label $p_g$. In this program flip$_2$ and flip$_4$ are path redundant since PC$(p_g, 2) = flip_1$ and PC$(p_g, 4) = \neg flip_1$, which are clearly inconsistent logical sentences. However, these two flips are not locally redundant. We would like to efficiently certify that it is safe to hoist these flips and others similar to them. To accomplish this we will perform a form of data-flow analysis on the program (Aho et al., 1986). The essence behind a data-flow analysis is to traverse the program and collect a set of facts — stated as logical propositions — that hold at each point in the program. By suitably constraining the structure of these facts we ensure that the analysis is efficient.

Figure 3c shows $p_g$ annotated with the data-flow facts required to perform global hoisting. We track two kinds of facts: (1) aliasing facts, marked with orange boxes, that relate local variables to the flips that they must be equal to; and (2) constraint facts, in cyan boxes, that list assignments to flips that are implied by if-statement conditions. For instance, at the point where flip$_2$ occurs it must be the case that $\{1 = true\}$ because we know from the aliasing facts that $x$ is assigned to be equal to flip$_1$, and we know that the condition constrains $x$ to be true since the branch was taken. The data-flow analysis records these aliasing facts and constraint facts in a linear pass through the AST.

Once the data-flow analysis is complete, it is straightforward to use the set of constraint facts that hold at each flip to determine when it is safe to hoist. To check if flip$_i$ and
flip_i are redundant, we check if their corresponding constraint facts are inconsistent, determined by if they disagree on any assignments to literals, which is efficient. This is done with a simple iteration through the constraint facts of flips with equal parameters to determine if there is a variable assigned to true in one and to false in the other. By definition, inconsistency of constraint facts implies path redundancy of the two flips, and hence hoisting these two will be sound.

There are a few more details about the data-flow analysis that are necessary to make it work. First, in order to efficiently construct the constraint facts, we only derive such facts from the conditions of if-statements that are conjunctions of variables for the then branch and disjunctions of variables for the else branch, which can be analyzed in a simple linear pass; for other more complex conditions, we take a best-effort approach, recording partial facts. Next, at join points—the points in the program at which two branches merge back into a single flow of execution—we take the intersection of all data-flow facts to conservatively ensure soundness.

3 IMPLEMENTATION & EVALUATION

In order for an optimization to be useful it must (1) be fast to implement in practice; (2) have a potential to significantly improve performance; and (3) benefit existing practical programs. In this section we show these three criteria hold for flip-hoisting. First, Section 3.1 shows how to implement flip-hoisting in an efficient way for the Dice system [Holtzen et al., 2020]—this implementation requires a minor generalization to flip-hoisting that preserves an additional ordering invariant. With this invariant we can prove that flip-hoisting can help and never hurt Dice’s inference performance, making it useful as a “run-by-default” optimization. Next Section 3.2 gives a best-case scenario on a synthetic set of programs and shows that flip-hoisting has the potential to speed up inference significantly. Finally, Section 3.3 evaluates flip-hoisting on a broad family of existing probabilistic programs and shows that flip-hoisting never hurts and sometimes significantly helps performance. It also shows that there exist programs in the wild—such as programs from the literature on probabilistic verification [Holtzen et al., 2021]—that naturally exhibit opportunities to hoist flips and benefit significantly from the optimization.

3.1 Implementation in Dice

To validate the effectiveness of this optimization in practice we implemented it in the Dice probabilistic programming system [Holtzen et al., 2020]. Dice supports exact inference and supports procedures, control-flow, categorical distributions on discrete values, and bounded recursion. Dice works by compiling a probabilistic program into a binary decision diagram (BDD), a tractable probabilistic model that supports linear-time probabilistic inference [Darwiche and Marquis, 2002; Choi et al., 2020].

Unlike many other PPLs, the performance of Dice is extremely sensitive to the order in which variables are introduced in the program [Holtzen et al., 2020]. This is because the variable ordering of the BDD is dictated by this order, and the size of the compiled BDD—which is one of the primary factors that determines the runtime performance of inference for the program—is heavily influenced by the variable order. For example, in Figure 14, the BDD variable corresponding to flip_1 will appear in the variable order before the BDD variable corresponding to flip_2, as they are always executed in that order.

Ultimately our goal is to design an optimization that can never hurt—and often help—inference performance. Hence, it is critical in the context of Dice that flip-hoisting maintains the variable ordering of the program. We therefore extend the flip-hoisting transformation so that whenever it hoists a flip it also hoists earlier flips as necessary to maintain the original order.

Definition 4 (Ordering-maintaining). If the ordering of the flip definitions are the same between p and hoist(p, i, j) —where branches of if-expressions are considered interchangeable in ordering—then hoist(p, i, j) is ordering-maintaining.

We implement ordering-maintaining flip-hoisting by also hoisting all the flips defined in between ancestor(p, i, j) and flip_i and between ancestor(p, i, j) and flip_j to be defined right before the hoisted flip. To minimize this extra work, we implement ancestor(p, i, j) to always return the lowest common ancestor (LCA). With order-maintaining flip-hoisting, we can state the following:

Theorem 3. Let |p| be the number of nodes in the BDD compiled from p and let i and j be flip indexes in p. If hoisting preserves the program variable order, then |hoist(p, i, j)| ≤ |p|.

We will sketch a proof here to avoid going into the details of Dice compilation. Since the order of variables in the program is unchanged after hoisting, so too is the order of variables in the compiled BDD. Then, hoisting can be thought of as relabeling flips i and j to have a common label. This does not change the size of the BDD. The resulting BDD will then be reduced to canonical form, which may decrease the size; this is where we may profit from flip-hoisting, as there may be more compression opportunities.

Hence, order-preserving hoisting can never hurt—but can often help, as we will see—Dice compilation performance. One thing to note is that it is not always possible to perform

All our code will be released as open-source. Our experiments were carried out on a Xeon E5-2640 CPU with 512GB RAM.
an order-preserving hoisting. In particular, for global hoisting, there can be the situation that redundant flips cannot be hoisted without breaking some ordering for the flips in between the redundant flips. For example, suppose there is a flip_i and a flip_j with the same parameters and inconsistent path conditions. Let flip_i be defined strictly before flip_j. If there is a flip_k defined in between them, flip_i and flip_j cannot be hoisted. flip_i would demand flip_k be inserted after the newly hoisted variable, but flip_j needs flip_k to be inserted before the newly hoisted variable, producing conflicting orderings. So, to satisfy the conditions of Theorem 3, it is occasionally necessary to forego hoisting opportunities to preserve order.

### 3.2 Synthetic benchmarks

Now we seek to answer the question: what is the best-case scenario for how much flip-hoisting can increase performance in Dice? We evaluated local flip-hoisting on a synthetic benchmark that is designed to have many hoisting opportunities and scale in a parameter n:

```plaintext
let x_1 = flip p_1 in ... let x_n = flip p_n in
let x_{n+1} = if x_0 then if x_1 then ... if x_n then flip p_{n+1} else flip p_{n+1} else flip p_{n+1} in ...

let x_{2n} = if x_0 then if x_1 then ... if x_n then flip p_{2n} else flip p_{2n} else flip p_{2n} in ...
```

where p_i and p'_i are randomly generated probabilities and are randomly assigned to flips in x_i.

Figure 4 shows a consistent improvement in performance in both end-to-end inference time and compilation size for the hoisted version on this simple example. flip-hoisting also dramatically reduces the size of the compiled BDD, to less than a quarter of the original size, experimentally validating Theorem 3. These results show that flip-hoisting can halve the time taken to perform inference. Hence, in the best-case scenario, flip-hoisting can significantly improve inference performance for a practical PPL.

### 3.3 Evaluation on practical programs

We have shown that flip-hoisting can improve performance when there are redundant flips, and now we answer whether flip-hoisting is useful in practice. There are two questions: (1) whether opportunities for hoisting exist in realistic programs, and (2) whether exploiting those hoisting opportunities actually speeds up inference. We will show affirmative answers to both of these questions for both local and global flip-hoisting on a collection of probabilistic programs drawn from probabilistic graphical models and probabilistic verification (Holtzen et al., 2021).

**Local flip-hoisting.** First we evaluate local flip-hoisting on a collection of well-known Bayesian network examples from the graphical models community, encoded as Dice programs. We especially consider programs that are challenging for Dice, as defined by having a runtime of greater than 1 second, since these are the programs sufficiently complex enough to warrant optimizations. Table I shows how local flip-hoisting helps Dice’s inference. Note that for these programs, we measure the compilation time; that is, the time to build the BDD representation of the program from the Dice program. The end-to-end runtime of a particular query is dominated by the compilation due to inference being in linear time to the compiled BDD and polytime proportional to the compile time.

| Benchmark | Time | BDD Size |
|-----------|------|----------|
| ANDES     | -61.51% | -61.25% |
| BN_78     | -6.22%   | -2.10%   |
| BN_79     | -1.96%   | -3.53%   |
| CPCSS4    | -6.09%   | -5.27%   |
| LINK      | +0.78%   | 0.00%    |
| MOISSAC3  | -32.80%  | -27.73%  |
| MUNIN     | -17.08%  | -24.00%  |
| MUNIN1    | -44.49%  | -47.07%  |
| MUNIN2    | -31.30%  | -32.27%  |
| MUNIN3    | -18.57%  | -20.43%  |
| MUNIN4    | -22.72%  | -25.30%  |
| PATHFINDER| -58.98%  | -72.26%  |

Table 1: Change in performance for Bayesian network programs after applying local flip-hoisting. Negative value means an improvement in performance.

---

3. See https://www.bnlearn.com/bnrepository/.
Table 2: Change in performance for probabilistic verification programs after applying local flip-hoisting. Negative value means an improvement in performance. BDD Size of NAND and BRP is measured by the iterative step function used in the program.

| Benchmark | Time | BDD Size |
|-----------|------|----------|
| NAND      | -34.93% | -2.72%   |
| WEATHER   | -24.06% | -24.35% |

Thus, we report on the compilation time for a more holistic evaluation. The results show that hoisting can greatly help inference performance, barring the cases where there are no redundant flips to hoist. Even then, the incurred cost from running the optimizations on programs that do not reduce in size is still minor compared to the whole of the runtime. We can see that hoisting also provides BDD size benefits on all examples, further validating Theorem 3. In particular, on some examples, such as PATHFINDER, the BDD size is decreased by as much as 72%.

**Hoisting in probabilistic verification.** We further evaluate local flip-hoisting on a completely different set of realistic probabilistic programs. Holtzen et al. (2021) introduced a new class of programs from the probabilistic verification community; these models have very different structures from graphical models. We ran local flip-hoisting on these models, and found that in 2 out of 7 programs – WEATHER FACTORY and NAND – there were hoisting opportunities. Table 2 shows the reduction in compilation runtime and in BDD size when local flip-hoisting is applied.

One program structure present in probabilistic verification and not Bayesian networks is the use of functions. As flip-hoisting is an intra-procedural program analysis, we apply the optimization to each function, and we observe that the function step in NAND benefits from the optimization. Though the reduction in BDD size of the function is only 2.72% of the original, the improvement from flip-hoisting is compounded substantially because the function is called multiple times. These results show that local flip-hoisting is a general optimization that helps probabilistic models other than Bayesian networks.

**Hoisting with loopy probabilistic programs.** As seen in the probabilistic verification programs, loopy programs especially benefit from the effects of flip-hoisting. Dice supports statically bounded recursion and thus also bounded loops. Since flip-hoisting is an intra-procedural program analysis, the optimization is applied once to the body of the function (the body of the loop), and the benefits are reaped each time the function is called (each iteration of the loop). Figure 5 shows the end-to-end inference time for the WEATHER probabilistic verification program with increasing number of loop iterations ($n$).

**Global Hoisting.** We evaluate the BDD sizes of the programs when we applied global flip-hoisting in addition to local flip-hoisting. While it is possible to apply local and global flip-hoisting in either order – the difference being which of the flips are hoisted together in the program and the corresponding LCA that they are hoisted to – we perform local before global hoisting to minimize tracking data-flow facts.

While more general than local hoisting, global hoisting is a more nuanced phenomenon that is not as widespread in existing programs. It still helped some examples in terms of size and their results are presented in Table 3. The largest improvement from global hoisting was EMDEC6G, where the BDD size was decreased by 34 nodes. The compilation time suffered from the overhead of running the data-flow analysis. Though marginal, small reductions in BDD size are still important improvements with respect to the Dice system. Since Dice compilation is modular, the same BDD can be reused in further compilation. Thus, the improvements can compound to have a greater effect, as seen in the loopy programs. In the future, we aim to broaden the scope of global hoisting to richer classes of if-expressions, and further tighten the relationship between global hoisting and the underlying inference algorithm.

**Table 3: Change in performance for Bayesian network programs after applying global flip-hoisting in addition to local flip-hoisting. Negative value means an improvement in performance.**

| Benchmarks | Time | BDD Size |
|------------|------|----------|
| EMDEC6G    | +61.55% | -0.27%   |
| TCC4E      | +22.16% | -0.3%    |
| WIN95PTS   | +51.98% | -0.71%   |

4 DISCUSSION & RELATED WORK

Thus far we have demonstrated that flip-hoisting is a practical and effective program optimization for speeding up Dice inference. Now we will give broader context on
this contribution, and discuss the extent to which flip-hoisting can possibly be generalized to other kinds of PPLs as well as where flip-hoisting sits in the broader landscape of optimizations for PPLs and other kinds of probabilistic models.

**Flip-hoisting Beyond Dice.** Most of today’s PPL optimizations are tied to a single inference algorithm. Here we consider the question: is flip-hoisting a general phenomenon that can speed up inference in settings beyond Dice’s specific inference algorithm? One natural family of languages to consider are other knowledge-compilation based languages that consider other compilation targets for exact inference, such as AND/OR search trees, cutset networks, sum-product networks, etc. (Marinescu and Dechter 2009; Rahaman et al. 2014; Poon and Domingos 2011; Mateescu et al. 2008a; Saad et al. 2021). ProbLog is a knowledge compilation based probabilistic logic programming language (Fierens et al. 2011). Similar to Dice it performs weighted model counting on the compiled representation to compute queries. However, ProbLog is a logic-based programming language and has a very different semantics from Dice. For instance, ProbLog does not have if-statements, and so a strategy of identifying locally redundant variables is not possible. Nonetheless, we performed a small-scale preliminary analysis where we manually identified and hoisted redundant flips in ProbLog programs to see the extent to which an automated flip-hoisting approach might be profitable here. The example we considered is a ProbLog variant of the program in Figure 1a. We observed that, averaged over 5 runs, manual hoisting yielded a 39% speedup in ProbLog’s inference: a promising preliminary evidence that a variant of flip-hoisting would be profitable for ProbLog.

**Probabilistic Graphical Models.** Now we discuss the broader context of flip-hoisting and closely related works. Inference via knowledge compilation has been studied extensively in the areas of Bayesian networks and graphical models, and a number of significant optimizations were developed in that context (Darwiche and Marquis 2002; Sang et al. 2005; Chavira and Darwiche 2008; Darwiche 2009; Choi et al. 2013; Dudek et al. 2020; Dilkas and Belle 2021; Sanner and McAllester 2005; Boutilier et al. 1996). In particular, Chavira and Darwiche (2008) identified parameter sharing as an important optimization for speeding up exact probabilistic inference in graphical models by exploiting repeated parameters within a conditional probability table (CPT) while encoding a graphical model into a logical representation.

There are important differences between parameter sharing and flip-hoisting. First, flip-hoisting has global scope. Parameter sharing is limited to exploiting repeated parameters within a single CPT, while global flip-hoisting is a whole-program analysis that can hoist repeated parameters across CPTs. More generally, global hoisting is able to identify flip redundancies in the presence of complex control flow. Second, flip-hoisting applies to arbitrary probabilistic programs. When applied to Bayesian networks encoded as Dice programs, local hoisting optimizes the repeated parameters that parameter sharing would have targeted. However, flip-hoisting can be applied to program structures like conditionals, functions, branching, and tuples, and thus the optimization works for a much broader set of probabilistic models than just the Bayesian networks that parameter sharing applies to, such as probabilistic veriﬁcation programs. Our setting is more general as a program analysis and applies to broader classes of models. Thus, flip-hoisting can be thought of strictly as a generalization of this approach.

**Slicing and Static Analysis.** Slicing is a classic and widely-applied technique in traditional optimizing compilers for reducing code size by trimming code that is not required for the program to exhibit its intended behavior. These techniques have been generalized to probabilistic programs (Hurst et al. 2014; Amtoft and Banerjee 2020). These methods are orthogonal to ours: programs may contain redundant flips without any possibility for slicing and vice versa. Common sub-expression elimination attempts to reduce redundant code by computing the result once, storing it in a local variable, and re-using the local variable many times (Aho et al. 1986). This technique relies on sub-expressions being used more than once under the same path conditions to improve performance, while flip-hoisting targets flips that have inconsistent path conditions.

**Symbolic Optimizations.** Probabilistic programming systems like Psi and Hakaru internally represent probability distributions symbolically using computer algebra systems (Gehr et al. 2016, 2020; Narayanan et al. 2016). Techniques such as delayed sampling (Murray et al. 2018) and semi-symbolic inference (Atkinson et al. 2022) also represent probability distributions symbolically, using conjugate priors to solve the system analytically as much as possible, before resorting to sampling. These methods differ from ours as they operate on and optimize an internal symbolic representation instead of the original program. A related idea is to use knowledge of conjugacy between distributions to simplify programs; this is employed by systems like BUGS, JAGS, and Autoconj, but it exploits a distinct structure from flip-hoisting where no conjugacy is necessary (Plummer et al. 2003; Thomas et al. 1992; Hoffman et al. 2018).

**Exploiting Structure.** Scaling inference by exploiting structure is a well-known and widely studied approach. Lifted inference exploits global symmetries of the underlying probability distribution in order to scale (Poole 2003; Braz et al. 2005; Kersting 2012; Ahmadi et al. 2013;
flip-hoisting exploits local redundant parameters in the
distribution, which may or may not correspond with the
kinds of symmetries exploited by lifted inference. Sample-
Search exploits determinism in importance sampling algo-
rithms, whereas our work exploits a broader set of local
structure (Gogate and Dechter, 2011).

**Probabilistic Program Analyses and Optimizations.**
There has been numerous recent work in techniques for
improving inference performance in PPLs in the form of
program analyses and optimizations, such as automatic
reparameterization, program decomposition, and amorti-
zation (Gorinova et al., 2020; Zhou et al., 2020; Ritchie
et al., 2016). Similar to how flip-hoisting analyzes path
conditions to find redundant variables, SYMPAIS analyzes
path conditions to estimate the satisfaction probability of
numerical constraints (Luo et al., 2020). Another approach
to reducing redundant computation using static analysis is
proposed in Nori et al., 2014, where R2 uses static analysis
to reject samples early if it is possible to determine that they
will violate subsequent observations. This method does not
focus on exact inference and so is not applicable to the same
type of programs that flip-hoisting targets.

5 CONCLUSION & FUTURE WORK

We present flip-hoisting, a program analysis and associ-
ated optimization for knowledge compilation based discrete
probabilistic programs. flip-hoisting empirically makes
inference more efficient on a range of programs from the
literature on probabilistic graphical models and probabilistic
verification. In the future, we anticipate extending flip-
hoisting to hoist continuous random variables or to optimize
approximate inference using a similar approach and pursu-
ing forms of flip-hoisting across procedures. Long term,
we expect flip-hoisting to become part of a standard suite
of probabilistic program optimizations, providing efficiency
even for programs written by non-expert users in the under-
lying probabilistic inference algorithm.

**References**

B. Ahmadi, K. Kersting, M. Mladenov, and S. Natarajan. Ex-
ploting symmetries for scaling loopy belief propagation
and relational training. 2013.

A. V. Aho, R. Sethi, and J. D. Ullman. Compilers, principles,
techniques. 1986.

T. Amtoft and A. Banerjee. A theory of slicing for impera-
tive probabilistic programs. 2020.

E. Atkinson, C. Yuan, G. Baudart, L. Mandel, and M. Carbin.
Semi-symbolic inference for efficient streaming proba-
bilistic programming. Proc. ACM Program. Lang., 6
(OOPSLA2), oct 2022. doi: 10.1145/3563347. URL
https://doi.org/10.1145/3563347

C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller.
Context-specific independence in bayesian networks. In
Proceedings of the Twelfth International Conference on
Uncertainty in Artificial Intelligence, 1996.

R. Braz, E. Amir, and D. Roth. Lifted first-order proba-
bilistic inference. In Proceedings of the Nineteenth In-
ternational Joint Conference on Artificial Intelligence.
Citeseer, 2005.

M. Chavira and A. Darwiche. On probabilistic inference by
weighted model counting. 2008.

M. Chavira, A. Darwiche, and M. Jaeger. Compiling re-
lational bayesian networks for exact inference. Int. J.
Approx. Reason., 42(1-2):4–20, 2006.

A. Choi, D. Kisa, and A. Darwiche. Compiling probabilistic
graphical models using sentential decision diagrams. In
European Conference on Symbolic and Quantitative
Approaches to Reasoning and Uncertainty. Springer, 2013.

Y. Choi, A. Vergari, and G. Van den Broeck. Probabilistic
circuits: A unifying framework for tractable probabilistic
models. 2020.

A. Darwiche. Modeling and reasoning with Bayesian net-
works. 2009.

A. Darwiche and P. Marquis. A knowledge compilation
map. 2002.

P. Dilkas and V. Belle. Weighted model counting without
parameter variables. In International Conference on The-
ory and Applications of Satisfiability Testing. Springer,
2021.

J. Dudek, V. Phan, and M. Vardi. Addmc: weighted model
counting with algebraic decision diagrams. In Proceed-
ings of the AAAI Conference on Artificial Intelligence,
2020.

D. Fierens, G. Van den Broeck, I. Thon, B. Gutmann, and
L. D. Raedt. Inference in probabilistic logic programs
using weighted cnfs. In Proceedings of the Twenty-
Seventh Conference on Uncertainty in Artificial Intel-
ligence, 2011.

T. Gehr, S. Misailovic, and M. Vechev. Psi: Exact symbolic
inference for probabilistic programs. In International
Conference on Computer Aided Verification. Springer,
2016.

T. Gehr, S. Steffen, and M. Vechev. λpsi: exact inference
for higher-order probabilistic programs. In Proceedings
of the 41st ACM SIGPLAN Conference on Programming
Language Design and Implementation, 2020.

V. Gogate and R. Dechter. Samplesearch: Importance sam-
ping in presence of determinism. 2011.

M. Gorinova, D. Moore, and M. Hoffman. Automatic repa-
rameterisation of probabilistic programs. In International
Conference on Machine Learning. Pmlr, 2020.
flip-hoisting: Exploiting Repeated Parameters in Discrete Probabilistic Programs

M. D. Hoffman, M. J. Johnson, and D. Tran. Autoconj: recognizing and exploiting conjugacy without a domain-specific language. Advances in Neural Information Processing Systems, 31, 2018.

S. Holtzen, G. Van den Broeck, and T. Millstein. Scaling exact inference for discrete probabilistic programs. 2020.

S. Holtzen, S. Junges, M. Vazquez-Chanlatte, T. Millstein, S. A. Seshia, and G. Van den Broeck. Model checking finite-horizon markov chains with probabilistic inference. In Proceedings of the 33rd International Conference on Computer-Aided Verification (CAV), 2021.

D. Huang, J.-B. Tristan, and G. Morrisett. Compiling markov chain monte carlo algorithms for probabilistic modeling. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 111–125, 2017.

C.-K. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel. Slicing probabilistic programs. 2014.

K. Kersting. Lifted probabilistic inference. 2012.

J. C. King. Symbolic execution and program testing. 1976.

C. Lattner. Llvm and clang: Next generation compiler technology. In The BSD conference, volume 5, pages 1–20, 2008.

Y. Luo, A. Filieri, and Y. Zhou. Sympais: Symbolic parallel adaptive importance sampling for probabilistic program analysis. 2020.

R. Mateescu and R. Dechter. And/or branch-and-bound search for combinatorial optimization in graphical models. 2009.

R. Mateescu, R. Dechter, and R. Marinescu. And/or multivalued decision diagrams (aomdds) for graphical models. 2008a.

R. Mateescu, R. Dechter, and R. Marinescu. AND/OR multivalued decision diagrams (aomdds) for graphical models. J. Artif. Intell. Res., 33:465–519, 2008b.

L. Murray, D. Lundén, J. Kudlicka, D. Broman, and T. Schön. Delayed sampling and automatic raoblackwellization of probabilistic programs. In International Conference on Artificial Intelligence and Statistics. Pmlr, 2018.

P. Narayanan, J. Carette, W. Romano, C.-c. Shan, and R. Zinkov. Probabilistic inference by program transformation in hakaru (system description). In International Symposium on Functional and Logic Programming. Springer, 2016.

M. Niepert and P. M. Domingos. Learning and inference in tractable probabilistic knowledge bases. In UAI, pages 632–641, 2015.

A. Nori, C.-K. Hur, S. Rajamani, and S. Samuel. R2: An efficient mcmc sampler for probabilistic programs. In Proceedings of the AAAI Conference on Artificial Intelligence, 2014.

M. Plummer et al. Jags: A program for analysis of bayesian graphical models using gibbs sampling. In Proceedings of the 3rd international workshop on distributed statistical computing. Vienna, Austria., 2003.

D. Poole. First-order probabilistic inference. In Ijcai, 2003.

H. Poon and P. Domingos. Sum-product networks: A new deep architecture. In 2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops). Ieee, 2011.

T. Rahman, P. Kothalkar, and V. Gogate. Cutset networks: A simple, tractable, and scalable approach for improving the accuracy of chow-liu trees. In Joint European conference on machine learning and knowledge discovery in databases. Springer, 2014.

D. Ritchie, P. Horsfall, and N. D. Goodman. Deep amortized inference for probabilistic programs. 2016.

F. A. Saad, M. C. Rinard, and V. K. Mansinghka. SPPL: probabilistic programming with fast exact symbolic inference. In PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Design and Implementation, 2021.

T. Sang, P. Beame, and H. A. Kautz. Performing bayesian inference by weighted model counting. In Aaai, 2005.

S. Sanner and D. McAllester. Affine algebraic decision diagrams (aadds) and their application to structured probabilistic inference. In Ijcai. Citeseer, 2005.

A. Thomas, D. J. Spiegelhalter, and W. Gilks. Bugs: A program to perform bayesian inference using gibbs sampling. 1992.

Y. Zhou, H. Yang, Y. W. Teh, and T. Rainforth. Divide, conquer, and combine: a new inference strategy for probabilistic programs with stochastic support. In International Conference on Machine Learning. Pmlr, 2020.
A ADDITIONAL RESULTS

We present here the full tables of all experiment results for every benchmark. We imposed a timeout threshold of 20 minutes.

A.1 Synthetic Programs

Table 4 shows the end-to-end inference runtime for the synthetic programs, and Table 5 shows the BDD size for these programs.

Table 4: Inference runtime for synthetic programs.

| Original (s) | Local Flip-Hoisting (s) |
|--------------|-------------------------|
| 0.04         | 0.04                    |
| 0.03         | 0.03                    |
| 0.03         | 0.03                    |
| 0.04         | 0.03                    |
| 0.05         | 0.03                    |
| 0.09         | 0.04                    |
| 0.17         | 0.04                    |
| 0.37         | 0.07                    |
| 0.81         | 0.17                    |
| 1.65         | 0.34                    |
| 3.56         | 0.82                    |
| 7.42         | 2.37                    |
| 13.36        | 6.39                    |

Table 5: Size of compiled BDDs for synthetic programs.

| Original | Local Flip-Hoisting |
|----------|---------------------|
| 3        | 3                   |
| 7        | 7                   |
| 31       | 16                  |
| 72       | 40                  |
| 173      | 82                  |
| 410      | 158                 |
| 941      | 329                 |
| 2,137    | 642                 |
| 4,816    | 1,306               |
| 10,592   | 2,629               |
| 23,241   | 5,247               |
| 50,599   | 10,490              |
| 109,337  | 20,980              |
| 235,104  | 42,005              |

A.2 Bayesian Network Programs

Table 6 shows the compilation runtime for the Bayesian network programs, and Table 7 shows the BDD size for these programs. Table 8 shows the compilation runtime for running global flip-hoisting on the Bayesian network programs that show improvement in BDD size, and Table 9 shows the BDD size for these programs.

Table 6: Compilation runtime for Bayesian network programs.

| Benchmarks | Original (s) | Local Flip-Hoisting (s) |
|------------|--------------|-------------------------|
| ANDES      | 7.52         | 2.90                    |
| BN_78      | 4.97         | 4.66                    |
| BN_79      | 26.78        | 26.25                   |
| CPCS54     | 1.55         | 1.46                    |
| LINK       | 168.88       | 170.20                  |
| MOISSAC3   | 2.13         | 1.43                    |
| MUNIN      | 20.72        | 17.18                   |
| MUNIN1     | 19.53        | 10.84                   |
| MUNIN2     | 34.85        | 23.94                   |
| MUNIN3     | 50.73        | 41.31                   |
| MUNIN4     | 20.55        | 15.88                   |
| PATHFINDER | 4.53         | 1.86                    |

Table 7: Size of compiled BDDs for Bayesian network programs.

| Benchmarks | Original    | Local Flip-Hoisting |
|------------|-------------|---------------------|
| ANDES      | 8,520,656   | 3,301,994           |
| BN_78      | 3,305,000   | 3,235,616           |
| BN_79      | 21,200,743  | 20,453,340          |
| CPCS54     | 2,438,881   | 2,310,466           |
| LINK       | 14,964,749  | 14,964,749          |
| MOISSAC3   | 498,240     | 360,070             |
| MUNIN      | 4,528,204   | 3,441,252           |
| MUNIN1     | 4,328,101   | 2,290,665           |
| MUNIN2     | 9,285,498   | 6,289,405           |
| MUNIN3     | 13,528,675  | 10,764,714          |
| MUNIN4     | 4,643,895   | 3,468,929           |
| PATHFINDER | 61,157      | 16,968              |

A.3 Probabilistic Verification Programs

Table 10 shows the compilation runtime for the probabilistic verification programs, and Table 11 shows the BDD size for these programs. Some programs show improvements in the sizes of the STEP function rather than in the state BDD.
Table 8: Compilation runtime for global flip-hoisting in addition to local flip-hoisting.

| Benchmarks | Local | Global |
|------------|-------|--------|
| EMDEC6G    | 0.0814| 0.1315 |
| TCC4E      | 0.088 | 0.1075 |
| WIN95PTS   | 0.0329| 0.05   |

Table 9: Size of compiled BDDs for global flip-hoisting in addition to local flip-hoisting.

| Benchmarks | Local | Global |
|------------|-------|--------|
| EMDEC6G    | 12,214| 12,180 |
| TCC4E      | 1,321 | 1,317  |
| WIN95PTS   | 982   | 975    |

Table 10: Compilation runtime for probabilistic verification programs.

| Benchmarks | Original (s) | Local Flip-Hoisting (s) |
|------------|--------------|-------------------------|
| NAND       | 72.98        | 47.49                   |
| WEATHER    | 458.60       | 348.25                  |
| ADV-GRID   | 58.12        | 62.88                   |
| BRP        | 0.74         | 0.69                    |
| GUYMC      | 6.40         | 6.22                    |
| QUEUES     | 242.18       | 326.68                  |
| MOTIV      | 0.07         | 0.03                    |

Table 11: Size of compiled BDDs for probabilistic verification programs.

| Benchmarks | Original | Local Flip-Hoisting |
|------------|----------|---------------------|
| NAND (STEP)| 6,761    | 6,577               |
| WEATHER    | 350,155  | 264,909             |
| ADV-GRID   | 302      | 302                 |
| BRP (STEP) | 4,255    | 4,253               |
| GUYMC      | 21,139   | 21,139              |
| QUEUES     | 15,812,199| 15,812,199          |
| MOTIV      | 44       | 44                  |

A.4 Loopy Programs

Table [12] shows the inference runtime for the WEATHER probabilistic program as $n$, the number of loop iterations run, increases.

Table 12: Inference runtime for WEATHER programs.

| $n$ | Original (s) | Local Flip-Hoisting(s) |
|-----|--------------|------------------------|
| 5   | 30.73        | 22.38                  |
| 6   | 77.48        | 62.12                  |
| 7   | 148.26       | 105.60                 |
| 8   | 247.28       | 187.97                 |
| 9   | 365.87       | 258.61                 |
| 10  | 526.65       | 369.88                 |
| 15  | 1620.38      | 1176.86                |
| 20  | 3645.16      | 2592.16                |
A.5 ProbLog

We generated an extended version of the motivating example program with and without hoisted parameters in ProbLog. The program is a chain of variables conditioned on the previous two variables,

\[
\begin{align*}
0.1::x_0. & \quad 0.2::x_1. \\
0.3::x_2. & \quad 0.4::x_0. \\
0.4::x_0. & \quad 0.4::x_2. \\
0.3::x_2. & \quad 0.3::x_0. \\
x_2 & : = x_0, x_1, x_2p1. \\
x_2 & : = \neg x_0, x_1, x_2p2. \\
x_2 & : = x_0, \neg x_1, x_2p3. \\
x_2 & : = \neg x_0, \neg x_1, x_2p4. \\
\vdots \\
0.3::x_np1. & \\
0.4::x_np2. & \\
0.4::x_np3. & \\
0.3::x_np4. & \\
x_n & : = x_{n-2}, x_{n-1}, x_np1. \\
x_n & : = \neg x_{n-2}, x_{n-1}, x_np2. \\
x_n & : = x_{n-2}, \neg x_{n-1}, x_np3. \\
x_n & : = \neg x_{n-2}, \neg x_{n-1}, x_np4. \\
\end{align*}
\]

and the hoisted version,

\[
\begin{align*}
0.1::x_0. & \quad 0.2::x_1. \\
0.3::x_2. & \quad 0.4::x_0. \\
0.4::x_0. & \quad 0.4::x_2. \\
0.3::x_2. & \quad 0.3::x_0. \\
x_2 & : = x_0, x_1, x_2p1. \\
x_2 & : = \neg x_0, x_1, x_2p2. \\
x_2 & : = x_0, \neg x_1, x_2p3. \\
x_2 & : = \neg x_0, \neg x_1, x_2p4. \\
\vdots \\
0.3::x_np1. & \\
0.4::x_np2. & \\
0.4::x_np3. & \\
0.3::x_np4. & \\
x_n & : = x_{n-2}, x_{n-1}, x_np1. \\
x_n & : = \neg x_{n-2}, x_{n-1}, x_np2. \\
x_n & : = x_{n-2}, \neg x_{n-1}, x_np3. \\
x_n & : = \neg x_{n-2}, \neg x_{n-1}, x_np4. \\
\end{align*}
\]

We present in Table 13 the runtime of each version of the program.

| Original (s) | Hoisted (s) |
|-------------|-------------|
| 13.87       | 8.61        |
| 14.13       | 8.56        |
| 14.11       | 8.68        |
| 14.38       | 8.61        |
| 14.13       | 8.49        |

B flip-HOISTING BAYESIAN NETWORKS

flip-hoisting generalizes parameter sharing (Chavira and Darwiche 2008) to probabilistic programs by way of reducing redundant random variables. To illustrate how flip-hoisting optimizes repeated parameters in the context of Bayesian networks, we present here an example model.

Figure 6a shows an example discrete Bayesian network on two variables, A and B. The variable A takes on values in the domain \{0, 1, 2\}, and B on the domain \{0, 1\}. The conditional probability tables (CPTs) are given in Figure 6c. We can see that the CPT for \Pr(B | A)\ has repeated parameters: \Pr(B = 0 | A = 1) = \Pr(B = 0 | A = 2). Chavira and Darwiche (2008) showed how to exploit repetitious parameters while encoding a graphical model into a logical representation.

Figure 6b represents this Bayesian network as a Dice program. It requires the use of the keyword discrete, which defines a discrete probability distribution over integer values. discrete is syntactic sugar that is turned into a series of flips before compilation. To define the conditional distribution \Pr(B | A),\ we branch on each possible value of A, flipping
flip-hoisting: Exploiting Repeated Parameters in Discrete Probabilistic Programs

(a) A simple Bayesian network with 2 variables.

\[
\begin{align*}
\text{let } A &= \text{discrete}(0.2, 0.3, 0.5) \\
in & \text{let } B = \\
& \text{if } A = 0 \text{ then flip } 0.1 \\
& \text{else if } A = 1 \text{ then flip } 0.2 \\
& \text{else flip } 0.2 \text{ in } (A, B)
\end{align*}
\]

(b) A Dice encoding of the Bayesian network.

| A | Pr(A) | A | Pr(B | A) |
|---|---|---|---|
| 0 | 0.2 | 0 | 0.1 |
| 0 | 0.3 | 1 | 0.9 |
| 1 | 0.3 | 0 | 0.2 |
| 1 | 1 | 1 | 0.8 |
| 2 | 0.5 | 2 | 0.2 |
| 2 | 1 | 1 | 0.8 |

(c) The CPTs for \( A \) and \( B \).

Figure 6: Bayesian network encoding as Dice program.

a differently weighted coin for each. Finally, we return a tuple \( (A, B) \) which represents the distribution on all values the variables in the Bayesian network can jointly take. \text{flip 0.2} is redundant here, since the two copies occur on two different branches of the if-expression; they would be hoisted when flip-hoisting is applied. We can see that these hoisted flips correspond to the repeated parameters that would be targeted by parameter sharing. However, flip-hoisting is a program analysis and optimization that can be applied to any Dice program, not only Bayesian networks. Thus, flip-hoisting can benefit a broader set of probabilistic models and can be thought of strictly as a generalization of parameter sharing.