`flip`-hoisting: Exploiting Repeated Parameters in Discrete Probabilistic Programs

Yu-Hsi Cheng,[1] **Todd Millstein,**[1] **Guy Van den Broeck,**[1] **Steven Holtzen**[2]

[1]University of California, Los Angeles [2]Northeastern University
ellieyhc45@g.ucla.edu, todd@cs.ucla.edu, guyvdb@cs.ucla.edu, s.holtzen@northeastern.edu

Abstract

Probabilistic programming is emerging as a popular and effective means of probabilistic modeling and an alternative to probabilistic graphical models. Probabilistic programs provide greater expressivity and flexibility in modeling probabilistic systems than graphical models, but this flexibility comes at a cost: there remains a significant disparity in performance between specialized Bayesian network solvers and probabilistic program inference algorithms. In this work we present a program analysis and associated optimization, `flip`-hoisting, that collapses repetitious parameters in discrete probabilistic programs to improve inference performance. `flip`-hoisting generalizes *parameter sharing* – a well-known important optimization from discrete graphical models – to probabilistic programs. We implement `flip`-hoisting in an existing probabilistic programming language and show empirically that it significantly improves inference performance, narrowing the gap between the performances of probabilistic programs and probabilistic graphical models.

1 Introduction

Probabilistic graphical models and probabilistic programs are two of the most widely used probabilistic modeling frameworks in artificial intelligence. Probabilistic programs are a flexible modeling paradigm that allows users to craft detailed and concise descriptions of complex probabilistic systems using the standard syntax and semantics of a programming language. Currently, the price for this modeling expressivity is inference performance: probabilistic program inference lags behind the sophisticated tools and techniques that have been developed by communities focused on more constrained modeling paradigms such as discrete probabilistic graphical modeling.

What is the key behind the impressive scalability of graphical model inference? In short, it is *identifying and exploiting structure during inference*. In the worst-case, inference is #P-hard (Roth 1996); hence, to scale, it is necessary to find problem-specific structure that avoids this worst-case. A long-running theme in the development of graphical models has been to eke out ever more performance by identifying increasingly nuanced and subtle kinds of structure and exploiting this structure through specially crafted inference algorithms. The end result of these efforts is a suite of scalable discrete graphical model systems that significantly outperform all existing probabilistic programming languages (PPLs) on the special case of discrete Bayesian networks. However, these methods are limited to graphical models: thus far they do not apply to probabilistic programs which are much more general, involving data structures, arithmetic, functions, and other operations.

The traditional approach to making programs run faster is to apply *program optimizations*, a standard suite of program transformations that maintain the program's original behavior while improving performance. For example, *common sub-expression elimination* (CSE) attempts to reduce redundant code by computing the result once, storing it in a local variable, and then re-using that local variable many times. In the context of probabilistic programs, CSE can have tremendous benefits on inference performance, since it can potentially reduce the overall number of random variables introduced by the probabilistic program: intuitively, when possible, one would like to flip a single coin and reuse it rather than to flip a new coin for every use. However, CSE is challenging to implement for PPLs since common sub-expressions might be correlated probabilistically and reusing them naively can change the semantics of the program.

Closing the gap between graphical model and PPL inference, as well as bringing to bear insights from Bayesian network optimizations to probabilistic programs, will require a rich suite of probabilistic program optimizations. In this work we propose a new family of generic probabilistic program optimization called `flip`-hoisting that generalizes common sub-expression elimination to probabilistic programs, reducing the overall number of random variables introduced by the probabilistic program and hence exponentially reducing the state space of the probabilistic program. The key behind our approach is a sound but incomplete *branch-sensitive analysis* of the probabilistic program 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.

First, in Section 1 we give a motivating example that gives more intuition behind how `flip`-hoisting works. Section 2 gives necessary background. Section 3 formally describes the `flip`-hoisting procedure and argues for its correctness. Section 4 describes how to implement `flip`-hoisting for an existing probabilistic program. Section 5 shows empirically that `flip`-hoisting is effective on realistic examples from graphical modeling and probabilistic verification, giv-

```
1  let x = flip1 0.1 in let z = flip2 0.2 in
2  let y = if x && z then flip3 0.3
3      else if x && !z then flip4 0.2
4      else flip5 0.3
5  in y
```

(a) No optimizations; subscripts are for referencing.

```
1  let x = flip 0.1 in
2  let tmp = flip 0.2 in
3  let z = tmp in
4  let y = if x && z then flip 0.3
5      else if x && !z then tmp
6      else flip 0.3
7  in y
```

(b) Invalid `flip`-hoisting.

```
1  let x = flip 0.1 in let z = flip 0.2 in
2  let tmp = flip 0.3 in
3  let y = if x && z then tmp
4      else if x && !z then flip 0.2
5      else tmp
6  in y
```

(c) Valid `flip`-hoisting.

Figure 1: Examples of `flip`-hoisting on `Dice` programs.

ing new state-of-the-art performance for probabilistic programming languages on these tasks. Section 6 gives an overview of related work. Section 7 concludes.

Motivating Example and Overview

Consider the example probabilistic program in Figure 1a. Throughout this paper examples will be written in `Dice`, though we note here that `flip`-hoisting is generic and generalizes in a straightforward manner to other PPLs. `Dice` is a functional probabilistic programming language that supports discrete random variables; in `Dice`, the syntax `flip` θ denotes a random variable that is true with probability θ (Holtzen, Van den Broeck, and Millstein 2020).

Observe that this program has some potentially redundant `flip`s: `flip 0.2` and `flip 0.3` both occur twice in the program. So, we ask: can we optimize this program to a version where these two `flip`s occur exactly once? For `flip 0.2` the answer is *no*. Consider the *invalid* hoisted program in Figure 1b. This hoisting changes the semantics of the original program. In the original program it is possible that `x = true, y = false`, and `z = true`; this assignment is not possible in the hoisted version since `y` is constrained to be equal to `z` in this case. This failure mode introduces a *spurious dependence* between the two `flip`s, coupling them when the semantics of the original program depended on them being decoupled.

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 candidate `flip`s; we call such `flip`s *redundant*. This is the case for `flip 0.3`, and a valid `flip`-hoisting of this is shown in Figure 1c. The key observation is that the `if`-expression on Line 2 has mutually exclusive branches, guaranteeing that there is no program execution for which the two separate instances of `flip 0.3` are simultaneously exercised; in this case, it is safe to flip just a single coin.

(a) A simple Bayesian network with 2 variables.

A	$\Pr(A)$
0	0.2
1	0.3
2	0.5

A	B	$\Pr(B \mid A)$
0	0	0.1
0	1	0.9
1	0	0.2
1	1	0.8
2	0	0.2
2	1	0.8

(b) The CPTs for A and B.

```
1  let A = discrete(0.2, 0.3, 0.5) in
2  let B = if A==0 then flip 0.1
3     else if A==1 then flip 0.2
4     else flip 0.2 in (A, B)
```

(c) A `Dice` encoding of the Bayesian network.

Figure 2: Bayesian network encoding as `Dice` program.

In this paper we give an analysis for identifying and merging redundant `flip`s in probabilistic programs. In general, determining if it is safe to merge two `flip`s is a special case of *reachability analysis*, and so is undecidable for general programs. Hence, we identify a sound but incomplete strategy that is effective for existing programs. We analyze only `if`-expressions whose guards have a particular form of structure: conjunctions of literals. This analysis is strong enough to perform the hoisting shown in Figure 1c automatically. In Section 3 we give more details about this analysis, argue that it is efficient in the size of the program and prove it is sound. In Section 5 we show empirically that this restriction is practical by showing that it benefits existing probabilistic programs derived from graphical models and problems from probabilistic verification.

2 Bayesian Network Background

Throughout this paper we will be comparing and contrasting approaches to inference in probabilistic graphical models and probabilistic programs. To facilitate this discussion in this section we introduce (1) essential background on encoding Bayesian networks as probabilistic programs and (2) an overview of existing methods in the Bayesian networks literature for exploiting parameter repetition.

Bayesian Network Encoding

Figure 2a 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 2b.

Figure 2c represents this Bayesian network as a `Dice` program. It requires the use of a new keyword, `discrete`, which defines a discrete probability distribution over integer values. Then, to define the conditional distribution

$\Pr(B \mid A)$, we branch on each possible value of A, flipping a differently weighted coin for each possible value of A. Finally, Line 4 returns a tuple (A, B) which represents the distribution on all values the variables in the Bayesian network can jointly take.

Parameter Sharing

Finding and exploiting the intricate structure of CPTs is a long-running enterprise in the PGM community (Choi, Kisa, and Darwiche 2013; 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. The CPT for $\Pr(B \mid A)$ in Figure 2b has *repetitious parameters*: $\Pr(B = 1 \mid A = 1) = \Pr(B = 1 \mid A = 2)$. Chavira and Darwiche (2008) showed how to exploit repetitious parameters while encoding a graphical model into a logical representation.

In essence parameter sharing is a special case of `flip`-hoisting, and served as an inspiration for our approach. This can be seen in Figure 2c, where `flip 0.2` is redundant and can be hoisted. There are important differences between parameter sharing and `flip`-hoisting. First, `flip`-hoisting has *global scope*: parameter sharing is limited to exploiting repetitious parameters within a single CPT, while `flip`-hoisting is a whole-program analysis that applies to arbitrary probabilistic programs with program structure like conditionals and tuples. Second, parameter sharing works on a logical *undirected* representation of the Bayesian network, and hence is tied to a specific kind of inference algorithm; `flip`-hoisting is a program optimization that works on a directed probabilistic program regardless of the down-stream inference algorithm that is applied.

Encoding Discrete Distributions A question raised by the comparison between `flip`-hoisting and parameter sharing is: how does `flip`-hoisting – which, as its name implies, only handles Boolean random variables – handle arbitrary discrete random variables? The problem of translating discrete distributions into some combination of Bernoulli random variables is known as *categorical encoding*, and it has been studied in the graphical modeling literature in the context of weighted model counting. There are many possible encodings, and an important conclusion is that the overall inference performance is quite sensitive to this choice. For instance, ENC3 from Chavira and Darwiche (2008) can exploit repetitious parameters, but is generally regarded to require more Boolean random variables than the SBK encoding introduced by Sang, Beame, and Kautz (2005). One contribution of the present work is a study of categorical encodings in the context of probabilistic programs. In particular, we show how to exploit repetitious parameters with SBK-like encodings; this will be elaborated on in Section 4.

3 `flip`-hoisting

In this section we formally introduce `flip`-hoisting. In order to unambiguously refer to each `flip` in a program, we assume that each one has a unique numeric id; we assign these ids as subscripts as shown in Figure 1a. Denote syntactic probabilistic programs as $\mathtt{p}$, and let $[\![\mathtt{p}]\!]$ denote the probability distribution on the values returned by $\mathtt{p}$.

Hoisting, denoted $\mathtt{hoist}(\mathtt{p}, i, j)$, is a function that produces a program wherein `flip`s i and j (assumed to have the same parameter value) are replaced by a reference to a single new `flip`.[1] For instance, if $\mathtt{p}_{ex}$ is the program in Figure 1a, then $\mathtt{hoist}(\mathtt{p}_{ex}, 3, 5)$ outputs the program in Figure 1c. Hoisting itself is efficient and simple to implement as a single pass over the program; the challenge is knowing when hoisting is *sound*:

Definition 1 (Sound hoisting)**.** *For a probabilistic program* $\mathtt{p}$*, hoisting* `flip`s i *and* j *is* sound *if* $[\![\mathtt{p}]\!] = [\![\mathtt{hoist}(\mathtt{p}, i, j)]\!]$.

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

Definition 2 (Path conditions)**.** *The* path conditions $\text{PC}(\mathtt{p}, i)$ *for* `flip` i *in probabilistic program* $\mathtt{p}$ *is the set of necessary and sufficient conditions on* `flip`s *that ensure execution reaches* `flip` i*.*

For instance, $\text{PC}(\mathtt{p}_{ex}, 3) = \mathtt{flip}_1 \wedge \mathtt{flip}_2$, since if $\mathtt{flip}_3$ is executed then the guard of the `if`-statement on Line 2 must be true, which implies that these two `flip`s are true. The path conditions yield a more tractable test:

Theorem 1 (Path redundancy)**.** *For program* $\mathtt{p}$*, hoisting* `flip`s i *and* j *is sound if the* `flip`s *have the same parameter value and* $\text{PC}(\mathtt{p}, i)$ *is inconsistent with* $\text{PC}(\mathtt{p}, j)$.

Proof sketch. A *path* through a program $\mathtt{p}$ is a total assignment to `flip`s encountered during an execution of the program; for instance, one path through Figure 1a is $\{\mathtt{flip}_1 = \mathit{true}, \mathtt{flip}_2 = \mathit{false}, \mathtt{flip}_4 = \mathit{true}\}$. This information uniquely determines the result of the execution along this path as well as the probability of taking the path. Intuitively there is a correspondence between the paths of $\mathtt{p}$ and $\mathtt{hoist}(\mathtt{p}, i, j)$ that ensures that the programs are equivalent.

The most interesting paths are those that encounter one of the hoisted `flip`s — we can show that such paths are in one-to-one correspondence between $\mathtt{p}$ and $\mathtt{hoist}(\mathtt{p}, i, j)$. Without loss of generality, consider a path through $\mathtt{p}$ that includes `flip` i. Since $\text{PC}(\mathtt{p}, i)$ is inconsistent with $\text{PC}(\mathtt{p}, j)$ this path must not include `flip` j. Hence by the definition of the `hoist` function there is a path through $\mathtt{hoist}(\mathtt{p}, i, j)$ that is identical to this path but with `flip` i replaced by the newly introduced `flip`, and with the same result value. Similarly, for each path through $\mathtt{hoist}(\mathtt{p}, i, j)$ that goes through the line of code where `flip` i was in $\mathtt{p}$, there must exist an equivalent path in $\mathtt{p}$ containing `flip` i instead of the new `flip`. □

[1] In general there are multiple possible places where the new `flip` can be inserted, but that is irrelevant here.

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 and Bayesian networks; these strengthenings yield polytime soundness checks in the size of the program.

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 guards 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`.*

Determining whether or not two `flips` are locally redundant is an efficient syntactic check on the program. `flips` 3 and 5 in the program in Figure 1a are locally redundant: they occur in disjoint branches of the `if`-expression on Line 2.

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

Proof. Follows from Theorem 1 and the fact that two locally redundant `flips` by definition have inconsistent PC. □

While seemingly simple, local hoisting is already a surprisingly powerful and general optimization. Because each CPT of a BN is encoded as a multi-way `if` expression, as shown earlier, local hoisting brings to probabilistic programs the ability to exploit repeated parameters within each CPT. Further, local hoisting is more generally applicable, as we demonstrate later on existing probabilistic programs that are not derived from graphical models.

Global Hoisting

Local hoisting is limited: it can only find hoisting opportunities that are local to each `if`-expression. In this section we develop a more complete analysis, which we call *global hoisting*, that retains the tractability of local hoisting while finding hoisting opportunities that span multiple `if`s.

Consider the minimal example in Figure 3a that we label $\mathtt{p}_g$. In this program `flips` 2 and 4 are path redundant since $\text{PC}(\mathtt{p}_g, 2) = \mathtt{flip}_1$ and $\text{PC}(\mathtt{p}_g, 4) = \neg\mathtt{flip}_1$, 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. The essence behind 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 itself is efficient.

Figure 3c shows $\mathtt{p}_g$ annotated with the data-flow facts required to perform global hoisting. We track two kinds of facts: (1) *aliasing facts*, marked with blue boxes, that relate local variables to the `flips` that they must be equal to; and (2) *constraint facts*, in yellow boxes, that list assignments to `flips` that are implied by `if`-statement guards. For instance, we know that on Line 4 it must be the case that {`1 = true`} since we (1) know from the aliasing facts that `x` is assigned to be equal to $\mathtt{flip}_1$ and (2) know that the guard constrains `x` to be `true` since the branch was taken.

```
1  let x = flip1 0.1 in
2  let y = if x then flip2 0.2 else flip3 0.3 in
3  let z = if !x then
4    flip4 0.2 else flip5 0.4 in (y, z)
```

(a) Example unhoisted program $\mathtt{p}_g$.

```
1  let x = flip1 0.1 in
2  let tmp = flip2,4 0.2 in
3  let y = if x then
4    tmp else flip3 0.3 in
5  let z = if !x then
6    tmp else flip5 0.4 in (y, z)
```

(b) Global `flip`-hoisting.

```
1  let x = flip1 0.1 in
2  {x=1}
3  let y = if x then
4      {1 = true} flip2 0.2
5    else
6      {1 = false} flip3 0.3 in
7  let z = if !x then
8      {1 = false} flip4 0.2
9    else
10     {1 = true} flip5 0.4 in (y, z)
```

(c) The program $\mathtt{p}_g$ annotated with data-flow facts.

Figure 3: Example of global `flip`-hoisting.

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 j are redundant, check if their corresponding constraint facts are inconsistent by checking if they disagree on any assignments to literals, which is efficient. By definition, inconsistency of constraint facts would imply 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 `if` guards that are *conjunctions of literals*, which can be analyzed in a simple linear pass; for other guards we conservatively derive no 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 ensures soundness.

Global hoisting is a strict generalization of local hoisting. When applied to BNs encoded as probabilistic programs, it enables forms of *cross-CPT* parameter sharing. More generally, global hoisting is able to identify `flip` redundancies in the presence of complex control flow.

4 Implementation in `Dice`

The previous section described `flip`-hoisting as a generic optimization that can be applied to any probabilistic programming language (albeit with minor modifications to lan-

guage syntax). To validate the effectiveness of this optimization in practice we implemented it in the `Dice` probabilistic programming system (Holtzen, Van den Broeck, and Millstein 2020). This implementation requires solving two additional technical problems that we detail in this section. The first, categorical encodings, was foreshadowed previously: it is the mechanism by which we translate a discrete probability distribution into a collection of `flip`s. It is an important decision for performance reasons and affects how many opportunities for `flip`-hoisting will be available.

`Dice`'s performance is very sensitive to the order in which `flip`s are introduced in the program. Hence, the second implementation detail, *order preservation*, refines our description of hoisting to be respectful of the variable order specified by the programmer. Ultimately order preservation allows us to prove that `flip`-hoisting can never hurt `Dice`'s inference performance (Theorem 3).

Categorical Encodings

It is well-known from the graphical modeling literature that inference performance is sensitive to the way in which categorical (or discrete) random variables are encoded into Boolean random variables (Chavira and Darwiche 2008). This encoding is often necessary because of details of the underlying inference algorithm: for instance, in `Dice` it is assumed that all random variables are binary because inference relies on a logical encoding. In this section we give a new categorical encoding strategy for probabilistic programs that is *hoisting aware*: it seeks to surface hoisting opportunities during the encoding.

First we need a way to represent integer *values* using a collection of Booleans. We do this naturally using a binary encoding and tuples. For instance the integer 2 can be encoded as a little-endian binary tuple `(true,false)`, where the first value is the 2's place and the second value is the 1's place; this encoding requires knowing a-priori a fixed bit-width, which is a requirement in `Dice` programs.

Now we use this binary encoding on integers to define the categorical encoding. Consider the simple discrete distribution `discrete(0.1, 0.4, 0.5)`. One way to encode this distribution using `flip`s is to flip a sequence of coins for each value; this is analogous to the encoding introduced by Sang, Beame, and Kautz (2005) in the context of graphical models. The *sequential encoding* is:

```
1 if flip 0.1 then (false, false)
2 else if flip 0.4/0.9 then (false, true)
3 else (true, false)
```

The above listing defines a distribution on pairs of Booleans that matches the original `discrete`: the tuple `(false, false)` – which corresponds with integer value 0 – is given a probability of 0.1. On each subsequent `flip` it is necessary to *re-normalize* by dividing by the remaining probability mass, which is why the second `flip` is true with probability $0.4/0.9$. As a consequence, changing the order in which values are output may result in very different parameters being present in the program. For instance, we could change the above encoding to output `(false, true)` before `(false, false)`, in which case the first `flip` would have parameter 0.4 and the second $0.1/0.6$.

Now consider the case when there are multiple `discrete`s that share one or more parameters, a fairly common circumstance. We would like to encode these `discrete`s in such a way to maximize hoisting opportunities. As a heuristic to accomplish this, we choose an order that places parameters in the output order in descending order according to the total number of times each parameter appears in the program: this way, during re-normalization, these common parameters are less likely to be lost.

Maintaining Variable Order

To discuss the effects of `flip`-hoisting on the `Dice` inference algorithm, we must first briefly take a look at the compilation process. A `Dice` program is compiled into a binary decision diagram (BDD) along with a weight function such that probabilistic inference of the original program is reduced to weighted model counting (WMC) on the BDD (Holtzen, Van den Broeck, and Millstein 2020). The BDD represents all possible paths through the program and has a variable for each `flip` made along these paths. Since the size of the BDD is worst-case exponential in the number of variables, and since the cost of WMC on the BDD is linear in its size, eliminating variables from the BDD can have a substantial performance benefit, exactly what our `flip`-hoisting optimization targets by sharing `flip`s across different execution paths.

The size of a BDD is heavily dependent on the variable order used to create it. For each function, the `Dice` compiler generates a variable order for its compiled BDDs that is consistent with program order. For example, in Figure 1a, the BDD variable corresponding to `flip 0.1` will appear in the variable order before that corresponding to the `flip 0.2` in `z`, as they are always executed in that order.

As described above, `flip`-hoisting has the effect of "moving" flips earlier in the program, which changes their order relative to other `flip`s and hence changes the BDD variable order. Changing the variable order means that the resulting BDD can be drastically different than the original one, and possibly much larger in size. To avoid this problem, our `flip`-hoisting optimization performs additional hoisting of `flip`s in order maintain the original variable order. This guarantees the compiled size will not increase (and, as we show later, it the size sometimes substantially decreases):

Theorem 3. *Let* $|\mathtt{p}|$ *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* $|\mathtt{hoist}(\mathtt{p}, i, j)| \leq |\mathtt{p}|$.

We will sketch a proof here to avoid going into the details of `Dice` compilation. Since the order of variables in the program are unchanged after hoisting, so too is the order of variables in the compiled BDD. Then, hoisting can be thought of as relabeling `flip`s 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.

However, one thing to note is that it is not always possible to perform 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 between the redundant flips. So, to satisfy the conditions of Theorem 3, it is occasionally necessary to forego hoisting opportunities for the sake of preserving order.

5 Experiments

The previous sections described the `flip`-hoisting optimization and how to implement it specifically in `Dice`. In this section we seek to validate the efficacy of `flip`-hoisting on realistic probabilistic programs. To accomplish this, we implemented `flip`-hoisting as an optimization in the `Dice` compiler, and tested the performance on a variety of example probabilistic programs coming from Bayesian networks and probabilistic verification.[2] The experiments were run on an Intel Xeon E5-2640 with 512 GB of memory and CentOS Linux 7. We consider two primary performance metrics before and after optimization: (1) number of `flips`, which measures how effectively `flip`-hoisting reduces the state space of the program; and (2) BDD size, which measures how much `flip`-hoisting improves the performance of `Dice`'s default compilation. We evaluated both global and local `flip`-hoisting on examples from Bayesian networks and probabilistic verification, and found that on many examples `flip`-hoisting found numerous optimization opportunities and reduced the BDD size on some examples by an order of magnitude.

Local `flip`-hoisting

First we evaluate local `flip`-hoisting on a collection of well-known examples from the graphical models community encoded as `Dice` programs.[3] Prior work observed the presence of redundant parameters in these models (Chavira and Darwiche 2008), so our question is *can `Dice` with `flip`-hoisting exploit these redundant parameters?*

`flip`-hoisting cannot be beneficial for inference if there are no hoisting opportunities, so first we measure the number of `flips` for each benchmark before and after optimization in Table 1. The first column, "No Opt", is a baseline with no optimizations; the second column, "Base Opt", includes some baseline optimizations that ensures that our improvements are due to `flip`-hoisting;"Hoist" applies `flip`-hoisting to `Dice` programs with a default encoding for discrete random variables; "Seq+Hoist" applies `flip`-hoisting to sequentially encoded `Dice` programs. In almost every example `flip`-hoisting results in a significant decrease in the overall number of `flip`. In general, the sequential encoding provided further reductions on several examples; particularly notable is the PIGS benchmark, where sequential hoisting provided a 50% reduction.

Now we report how local `flip`-hoisting helps `Dice`'s inference performance, shown in Table 2. We observe that hoisting provides BDD size benefits on all examples, experimentally validating Theorem 3. The benefits of the se-

[2] All our code will be released as open-source.

[3] See https://www.bnlearn.com/bnrepository/.

Table 1: Number of `flips` in the program when local `flip`-hoisting is applied for various Bayesian networks.

Benchmarks	No Opt	Base Opt	Hoist	Seq+Hoist
ALARM-A	585	504	201	**137**
ALARM-S	585	504	247	**222**
ALARM	585	504	201	**133**
ANDES	1,157	1,084	**328**	**328**
BARLEY	135,059	114,005	59,254	**41,998**
BN 78	829	829	**328**	**328**
BN 79	829	829	**328**	**328**
CHILD	330	227	171	**140**
CPCS54	829	829	**328**	**328**
DIABETES	619,292	77,185	15,194	**11,967**
DIAG A	5,005	3,910	**377**	415
DIAG AM	5,005	3,910	**377**	415
DIAG B	22,600	19,136	**672**	695
DIAG BM	22,600	19,136	**672**	695
EMDEC6G	826	807	**712**	**712**
HAILFIND	3,683	2,155	1,149	**871**
HEPAR2	1,582	1,453	1,283	**1,272**
INSURANCE	1,217	650	320	**315**
LINK	14,503	496	**458**	**458**
MILDEW	690,940	30,916	26,318	**21,498**
MOISSAC3	57,813	13,419	2,472	**1,766**
MUNIN	119,529	25,625	19,148	**16,487**
MUNIN1	22,956	4,712	3,428	**2,918**
MUNIN2	103,439	22,825	17,963	**15,525**
MUNIN3	106,660	23,478	18,856	**16,281**
MUNIN4	119,289	25,385	19,145	**16,474**
PATHFIND	91,358	28,703	1,897	**1,652**
PIGS	8,427	2,066	1,178	**586**
TCC4E	1,604	1,604	**545**	**545**
WATER	10,203	3,113	1,813	**1,783**
WIN95PTS	574	350	**125**	**125**

Table 2: BDD size for compiled programs for local `flip`-hoisting. A "–" denotes a 20-minute time out.

Benchmarks	No Opt	Base Opt	Hoist	Seq+Hoist	PL
ALARM-A	73,883	51,679	**18,859**	26,001	✓
ALARM-S	256,271	214,376	**143,068**	200,029	✓
ALARM	298,867	254,766	**87,168**	139,506	✓
ANDES	87,119,041	8,520,656	**3,301,994**	**3,301,994**	–
BARLEY	–	–	–	–	–
BN 78	3,499,826	3,305,000	**3,235,616**	**3,235,616**	–
BN 79	21,497,148	21,200,743	**20,453,340**	**20,453,340**	–
CHILD	2,542	2,112	**1,680**	2,660	✓
CPCS54	2,450,688	2,438,881	**2,310,466**	**2,310,466**	–
DIABETES	–	–	–	–	–
DIAG A	5,020,830	41,257	6,241	**5,172**	–
DIAG AM	5,020,830	26,197	6,241	**5,172**	–
DIAG B	–	97,573	10,628	**9,615**	–
DIAG BM	–	147,375	10,628	**9,615**	–
EMDEC6G	169,096	75,999	**12,214**	**12,214**	–
HAILFINDER	2,635,901	145,917	**140,963**	174,793	–
HEPAR2	36,375	36,244	**36,037**	39,218	✓
INSURANCE	190,242	90,095	**71,396**	97,231	✓
LINK	–	15,117,830	**14,964,749**	**14,964,749**	–
MILDEW	–	–	–	–	✓
MOISSAC3	–	498,240	267,648	**246,846**	–
MUNIN	–	4,998,135	**3,350,854**	4,719,494	–
MUNIN1	–	4,328,101	**2,162,624**	2,735,348	–
MUNIN2	–	10,324,147	**6,208,220**	8,760,584	–
MUNIN3	–	14,235,596	**10,708,445**	14,393,174	–
MUNIN4	–	5,067,973	**3,315,918**	4,555,212	–
PATHFIND	571,839	61,157	**16,153**	18,777	–
PIGS	409,226	180,124	140,617	**89,537**	–
TCC4E	3,112	3,112	**1,321**	**1,321**	–
WATER	39,494,260	44,703	**26,714**	28,189	–
WIN95PTS	75,182	1,545	**982**	**982**	✓

quential encoding on size are less clear-cut; on some examples it is out-performed by the default `Dice` encoding (both with hoisting). This is likely due to the fact that these two encodings produce different logical formulae, and hence non-equivalent BDDs. As a consequence, even though the sequential encoding enables more hoisting opportunities, it is not enough to overcome the implicit size advantage of the default `Dice` encoding on some examples. Three of the benchmarks – BARLEY, MILDEW, and DIABETES – are particularly challenging, and `Dice` does not terminate on these within 20 minutes on any settings, even though `flip`-hoisting found many hoisting opportunities in each. We pause here to note that we have aimed this work to perform hoisting at the program level, without touching the inference and compilation approach within `Dice`. For `Dice` to consistently benefit from the reduced number of flips in Seq+Hoist, one likely needs a tighter interplay between hoisting and compilation, which we leave for future work.

For broader context on these performance results, the column "PL" reports whether or not ProbLog (Fierens et al. 2011) was able to solve the Bayesian network within 20 minutes (a "✓" denotes successful termination). ProbLog is a well-engineered probabilistic logic programming language, and we employ its default inference pipeline that is specialized for discrete distributions. `Dice` with hoisting outperforms ProbLog on 21 benchmarks, and loses on only 1. This gives perspective on the difficulty of these benchmarks and shows the utility of our optimizations to `Dice`.

Hoisting in probabilistic verification The success of parameter sharing is well-documented in graphical models, and our experiments here show that these successes translate to probabilistic program encodings of well-known Bayesian networks. The question remains, however: *do there exist natural hoisting opportunities in other kinds of probabilistic programs?* Holtzen et al. (2021) introduced a new class of probabilistic programs coming from the probabilistic verification community; these models have very different structure from graphical models. We also ran local `flip`-hoisting on these models, and found that in 2 out of 7 programs – "Weather Factory" and "nand" – there were hoisting opportunities: for "Weather Factory" hoisting reduced the BDD size from 2,228,213 nodes to 1,687,555 nodes. Hence, local `flip`-hoisting is a general optimization that helps probabilistic models other than Bayesian networks.

Global Hoisting

We evaluated the `flip` count and the BDD sizes of the benchmarks when we applied global `flip`-hoisting with and without the Sequential encoding. Global hoisting, while more general than local hoisting, is a more nuanced phenomenon that is not as wide-spread in Bayesian networks. Nonetheless, it still helped some examples and their results are presented in Tables 3 and 4.

The largest improvement from global hoisting was EMDEC6G, where 34 `flip`s were eliminated and the BDD size was decreased by 34 nodes. TCC4E and WIN95PTS had some decreases in BDD size following their slight decrease in `flip` count as well. 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: Number of flips for global `flip`-hoisting.

Benchmarks	Local	Seq+Local	Global	Seq+Global
EMDEC6G	712	712	**678**	**678**
TCC4E	545	545	**541**	**541**
WIN95PTS	125	125	**124**	**124**

Table 4: Size of compiled BDDs for global `flip`-hoisting.

Benchmarks	Local	Seq+Local	Global	Seq+Global
EMDEC6G	12,214	12,214	**12,180**	**12,180**
TCC4E	1,321	1,321	**1,317**	**1,317**
WIN95PTS	982	982	**975**	**975**

6 Related Work

Symbolic Optimizations. Probabilistic programming systems like Psi and Hakaru internally represent probability distributions symbolically using computer algebra systems (Gehr, Misailovic, and Vechev 2016; Gehr, Steffen, and Vechev 2020; Narayanan et al. 2016). These methods differ from ours in that they operate on an internal symbolic representation, rather than the original program, and also to our knowledge they do not directly exploit state-space reduction enabled by `flip`-hoisting. A related idea is to use knowledge of conjugacy between distributions to simplify programs; this is employed by systems such as BUGS and JAGS, but exploits a distinct structure from `flip`-hoisting where no conjugacy is necessary (Plummer et al. 2003; Thomas, Spiegelhalter, and Gilks 1992).

Probabilistic Graphical Models. There is a rich literature in optimizing the representation of probabilistic graphical models for faster inference (Chavira and Darwiche 2008; Darwiche 2009; Choi, Kisa, and Darwiche 2013; Dudek, Phan, and Vardi 2020; Dilkas and Belle 2021). Chavira and Darwiche (2008) optimize the encoding the discrete graphical models that exploits an analogous notion of duplicate `flip`s. Our setting is more general and applies to broader classes of models than those that can be represented as probabilistic graphical models, and can be thought of strictly as a generalization of this approach.

7 Conclusion and Future Work

We present `flip`-hoisting, a family of optimizations that apply common sub-expression elimination to discrete probabilistic programs. `flip`-hoisting empirically makes inference more efficient. In the future, we anticipate extending `flip`-hoisting to programs with structure like continuous random variables and procedures. Long term, we expect `flip`-hoisting to become a part of a standard suite of optimizations that are applied to all probabilistic programs to speed up inference.

References

Boutilier, C.; Friedman, N.; Goldszmidt, M.; and Koller, D. 1996. Context-Specific Independence in Bayesian Networks. In *Proceedings of the Twelfth International Conference on Uncertainty in Artificial Intelligence*, UAI'96, 115–123. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 155860412X.

Chavira, M.; and Darwiche, A. 2008. On probabilistic inference by weighted model counting. *Artificial Intelligence*, 172(6-7): 772–799.

Choi, A.; Kisa, D.; and Darwiche, A. 2013. Compiling probabilistic graphical models using sentential decision diagrams. In *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, 121–132. Springer.

Darwiche, A. 2009. *Modeling and reasoning with Bayesian networks*. Cambridge university press.

Dilkas, P.; and Belle, V. 2021. Weighted model counting without parameter variables. In *International Conference on Theory and Applications of Satisfiability Testing*, 134–151. Springer.

Dudek, J.; Phan, V.; and Vardi, M. 2020. ADDMC: weighted model counting with algebraic decision diagrams. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 1468–1476.

Fierens, D.; Van den Broeck, G.; Thon, I.; Gutmann, B.; and Raedt, L. D. 2011. Inference in Probabilistic Logic Programs Using Weighted CNF's. In *Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence*, UAI'11, 211–220. Arlington, Virginia, USA: AUAI Press. ISBN 9780974903972.

Gehr, T.; Misailovic, S.; and Vechev, M. 2016. Psi: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*, 62–83. Springer.

Gehr, T.; Steffen, S.; and Vechev, M. 2020. λPSI: exact inference for higher-order probabilistic programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 883–897.

Holtzen, S.; Junges, S.; Vazquez-Chanlatte, M.; Millstein, T.; Seshia, S. A.; and Van den Broeck, G. 2021. Model Checking Finite-Horizon Markov Chains with Probabilistic Inference. In *Proceedings of the 33rd International Conference on Computer-Aided Verification (CAV)*.

Holtzen, S.; Van den Broeck, G.; and Millstein, T. 2020. Scaling exact inference for discrete probabilistic programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA): 1–31.

King, J. C. 1976. Symbolic execution and program testing. *Communications of the ACM*, 19(7): 385–394.

Narayanan, P.; Carette, J.; Romano, W.; Shan, C.-c.; and Zinkov, R. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *International Symposium on Functional and Logic Programming*, 62–79. Springer.

Plummer, M.; et al. 2003. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of the 3rd international workshop on distributed statistical computing*, volume 124, 1–10. Vienna, Austria.

Roth, D. 1996. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2): 273–302.

Sang, T.; Beame, P.; and Kautz, H. A. 2005. Performing Bayesian inference by weighted model counting. In *AAAI*, volume 5, 475–481.

Sanner, S.; and McAllester, D. 2005. Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In *IJCAI*, volume 2005, 1384–1390. Citeseer.

Thomas, A.; Spiegelhalter, D. J.; and Gilks, W. 1992. BUGS: A program to perform Bayesian inference using Gibbs sampling. *Bayesian statistics*, 4(9): 837–842.

A Supplemental Experiments

We present here the full tables of all experiment results for every benchmark. Table 5 shows the total number and the number of distinct raw parameters in each the benchmarks. This tally is made with respect to other parameters within the scope of the `e1` sub-expression of a `let-` statement. This number of distinct parameters is useful for comparing with Bayesian network encodings like those discussed in (Chavira and Darwiche 2008). The encodings described by Chavira and Darwiche (2008) have as a *lower bound* the number of distinct parameters reported in this table; our experiments show that we can reduce the number of overall `flips` below this bound, showing a potential theoretical advantage over Chavira and Darwiche (2008) in terms of number of required Boolean random variables.

Table 6 presents the `flip` counts of each benchmark with and without the optimizations applied. Table 7 shows all the sizes of the compiled programs with the combinations of optimizations.

Table 5: Number of parameters in a Bayesian network.

Benchmarks	Total	Distinct
ALARM-ALT	680	167
ALARM-SAFER	680	258
ALARM	752	182
ANDES	1,157	357
BARLEY	130,179	35,712
BN 78	829	328
BN 79	829	328
CHILD	306	152
CPCS54	829	328
DIABETES	461,069	16,612
DIAGNOSE A	4,764	434
DIAGNOSE A MULTI	4,764	434
DIAGNOSE B	22,634	729
DIAGNOSE B MULTI	22,634	729
EMDEC6G	826	717
HAILFINDER	3,741	835
HEPAR2	2,139	1,914
INSURANCE	1,419	427
LINK	18,317	1,264
MILDEW	547,158	6,631
MOISSAC3	49,554	1,596
MUNIN	98,423	23,114
MUNIN1	18,856	3,989
MUNIN2	82,670	21,396
MUNIN3	84,485	22,575
MUNIN4	96,327	22,731
PATHFINDER	91,486	2,138
PIGS	8,427	1,474
TCC4E	1,604	545
WATER	13,484	3,578
WIN95PTS	574	168

Table 6: Number of flips in the program. A "-" denotes a 20-minute time out.

Benchmarks	No Opt	Base Opt	Local Hoist	Global Hoist	Seq+Local Hoist	Seq+Global Hoist
ALARM-ALT	585	504	201	201	137	**136**
ALARM-SAFER	585	504	247	247	**222**	**222**
ALARM	585	504	201	201	**133**	**133**
ANDES	1,157	1,084	**328**	**328**	**328**	**328**
BARLEY	135,059	114,005	59,254	-	**41,998**	-
BN 78	829	829	**328**	**328**	**328**	**328**
BN 79	829	829	**328**	**328**	**328**	**328**
CHILD	330	227	171	171	**140**	**140**
CPCS54	829	829	**328**	**328**	**328**	**328**
DIABETES	619,292	77,185	15,194	15,194	**11,967**	**11,967**
DIAGNOSE A	5,005	3,910	**377**	**377**	415	415
DIAGNOSE A MULTI	5,005	3,910	**377**	**377**	415	415
DIAGNOSE B	22,600	19,136	**672**	**672**	695	695
DIAGNOSE B MULTI	22,600	19,136	**672**	**672**	695	695
EMDEC6G	826	807	712	**678**	712	**678**
HAILFINDER	3,683	2,155	1,149	1,149	**871**	**871**
HEPAR2	1,582	1,453	1,283	1,283	**1,272**	**1,272**
INSURANCE	1,217	650	320	320	**315**	**315**
LINK	14,503	496	**458**	**458**	**458**	**458**
MILDEW	690,940	30,916	26,318	-	**21,498**	-
MOISSAC3	57,813	13,419	2,472	2,472	**1,766**	**1,766**
MUNIN	119,529	25,625	19,148	19,148	**16,487**	**16,487**
MUNIN1	22,956	4,712	3,428	3,428	**2,918**	**2,918**
MUNIN2	103,439	22,825	17,963	17,963	**15,525**	**15,525**
MUNIN3	106,660	23,478	18,856	18,856	**16,281**	**16,281**
MUNIN4	119,289	25,385	19,145	19,145	**16,474**	**16,474**
PATHFINDER	91,358	28,703	1,897	1,897	**1,652**	**1,652**
PIGS	8,427	2,066	1,178	1,178	**586**	**586**
TCC4E	1,604	1,604	545	**541**	545	**541**
WATER	10,203	3,113	1,813	1,813	**1,783**	**1,783**
WIN95PTS	574	350	125	**124**	125	**124**

Table 7: BDD Size of compiled programs. A "-" denotes a 20-minute time out.

Benchmars	No Opt	Base Opt	Local Hoist	Global Hoist	Seq+Local Hoist	Seq+Global Hoist
ALARM-ALT	73,883	51,679	**18,859**	**18,859**	26,001	26,001
ALARM-SAFER	256,271	214,376	**143,068**	**143,068**	200,029	200,029
ALARM	298,867	254,766	**87,168**	**87,168**	139,506	139,506
ANDES	87,119,041	8,520,656	**3,301,994**	**3,301,994**	**3,301,994**	**3,301,994**
BARLEY	-	-	-	-	-	-
BN 78	3,499,826	3,305,000	**3,235,616**	**3,235,616**	**3,235,616**	**3,235,616**
BN 79	21,497,148	21,200,743	**20,453,340**	**20,453,340**	**20,453,340**	**20,453,340**
CHILD	2,542	2,112	**1,680**	**1,680**	2,660	2,660
CPCS54	2,450,688	2,438,881	**2,310,466**	**2,310,466**	**2,310,466**	**2,310,466**
DIABETES	-	-	-	-	-	-
DIAGNOSE A	5,020,830	41,257	6,241	6,241	**5,172**	**5,172**
DIAGNOSE A MULTI	5,020,830	26,197	6,241	6,241	**5,172**	**5,172**
DIAGNOSE B	-	97,573	10,628	10,628	**9,615**	**9,615**
DIAGNOSE B MULTI	-	147,375	10,628	10,628	**9,615**	**9,615**
EMDEC6G	169,096	75,999	12,214	**12,180**	12,214	**12,180**
HAILFINDER	2,635,901	145,917	**140,963**	**140,963**	174,793	174,793
HEPAR2	36,375	36,244	**36,037**	**36,037**	39,218	39,218
INSURANCE	190,242	90,095	**71,396**	**71,396**	97,231	97,231
LINK	-	15,117,830	**14,964,749**	**14,964,749**	**14,964,749**	**14,964,749**
MILDEW	-	-	-	-	-	-
MOISSAC3	-	498,240	267,648	267,648	**246,846**	**246,846**
MUNIN	-	4,998,135	**3,350,854**	**3,350,854**	4,719,494	4,719,494
MUNIN1	-	4,328,101	**2,162,624**	**2,162,624**	2,735,348	2,735,348
MUNIN2	-	10,324,147	**6,208,220**	**6,208,220**	8,760,584	8,760,584
MUNIN3	-	14,235,596	**10,708,445**	**10,708,445**	14,393,174	14,393,174
MUNIN4	-	5,067,973	**3,315,918**	**3,315,918**	4,555,212	4,555,212
PATHFINDER	571,839	61,157	**16,153**	**16,153**	18,777	18,777
PIGS	409,226	180,124	140,617	140,617	**89,537**	**89,537**
TCC4E	3,112	3,112	1,321	**1,317**	1,321	**1,317**
WATER	39,494,260	44,703	**26,714**	**26,714**	28,189	28,189
WIN95PTS	75,182	1,545	982	**975**	982	**975**