Structured Factored Inference for Probabilistic Programming Supplementary Material

1 STRUCTURED FACTORED INFERENCE

1.1 Model Decomposition

The following discussion of SFI is cast in the context of a SimPPL program. However, the method is applicable to graphical models in general.

The key operation of SFI is model decomposition. This operation decomposes a model into semantically meaningful sub–models (i.e., programs) that can be reduced to a joint distribution over relevant variables. First, we define two key concepts: *uses* and *external*. An RV r *uses* an RV x if:

$$x \in \mathcal{A}_r \wedge P(r|\mathcal{A}_r) \neq P(r|\mathcal{A}_r \setminus x)$$

We denote the set of variables $\mathcal{U}_r$ for a variable r as the set of all variables r uses, either directly or recursively, plus r. This definition of uses can be difficult to verify in a program based on the semantics of SimPPL. However, in our implementation of SimPPL, we have a syntactic (but stronger) condition for r using x based on x appearing in the expression for r, or in any expression for a variable used by r. Such a condition is necessary for uses and thus still guarantees SFI's soundness.

We denote that a variable x is *external* to r if:

$$x \in \mathcal{U}_r \wedge \exists\; y \in \mathcal{RV}_Q \setminus \mathcal{U}_r \,|\, x \in \mathcal{U}_y$$

That is, an external variable to r is used in the generative process of a variable that r does not use. We denote the set of variables external to r as $\mathcal{E}^r$.

A decomposition of the model with respect to an RV $d \in \mathcal{RV}_Q$ is an operation that partitions $\mathcal{RV}_Q$ into two disjoint sets of variables, $\mathcal{RV}_Q^d$ and $\mathcal{RV}_Q^{\overline{d}}$. The RV d is called a decomposition point. We define $\mathcal{RV}_Q^d$ and $\mathcal{RV}_Q^{\overline{d}}$ as:

$$\begin{aligned} \mathcal{RV}_Q^d &= \mathcal{U}_d - \mathcal{E}^d \\ \mathcal{RV}_Q^{\overline{d}} &= \mathcal{RV}_Q - \mathcal{RV}_Q^d \end{aligned} \tag{1}$$

In other words, $\mathcal{RV}_Q^d$ is the set of variables exclusively used in the generation of d (i.e., no external uses), and $\mathcal{RV}_Q^{\overline{d}}$ is all remaining variables in Q.

As an example, consider the program in Fig. 1a from the paper. Since d can be any variable, let us choose $outcome_T^b$ as the decomposition point. In this example, $\mathcal{RV}_Q^d$ is the set of variables that $outcome_T^b$ exclusively uses, so it would be $\{outcome_T^b, x1_T^b, y1_T^b, z1_T^b, x2_T^b, y2_T^b, z2_T^b\}$ (all the variables in the left–most box of Fig. 1b from the paper). $\mathcal{RV}_Q^{\overline{d}}$ would include all other variables in the model.

1.1.1 Factored Representation

Once the variables in Q have been split into two sets via a decomposition point, we convert the decomposition to a factored representation. Each variable $r \in RV_Q$ can be converted to a set of factors Ψ_r that describe the generative semantics of the variable. For brevity, we do not provide a detailed explanation of factor creation for SimPPL, but provide a short summary in the supplement.

We denote the set of factors created from a program Q as Ψ, and each factor $\psi \in \Psi$ is defined over a set of variables x_ψ. Given Ψ, the probability distribution of an RV r in the program, $P(r)$, is formulated as:

$$P(r) = \frac{1}{Z} \sum_{x \in \mathcal{RV}_Q \setminus r} \prod_{\psi \in \Psi} \psi(x_\psi) \tag{2}$$

where Z is the normalizing constant.

As d divides $\mathcal{RV}_Q$ into two sets, it naturally divides Ψ into two sets, Ψ_d and $\Psi_{\overline{d}}$. As such, we can rewrite Eqn. 2 as:

$$P(r) = \frac{1}{Z} \sum_{x \in \mathcal{RV}_Q^{\overline{d}} \setminus r} \; \sum_{y \in \mathcal{RV}_Q^{d} \setminus r} \; \prod_{\psi \in \Psi_{\overline{d}}} \psi(x_\psi) \prod_{\psi \in \Psi_d} \psi(y_\psi) \tag{3}$$

Note that even though the variables in $\mathcal{RV}_Q^d$ and $\mathcal{RV}_Q^{\overline{d}}$ are disjoint, the variables used in the sets of factors Ψ_d and $\Psi_{\overline{d}}$ are *not* disjoint. From the definition of the sets in Eqn. 1, the only variables that are shared between Ψ_d and $\Psi_{\overline{d}}$ can be $\mathcal{E}^d$, the set of variables external to d, and d itself. As such, we can move the summation over $\mathcal{RV}_Q^d$ in Eqn. 3 inwards and the summation over

d to the outer summation, so that we get:

$$\begin{aligned}P(r) &= \frac{1}{Z} \sum_{x \in \{\mathcal{RV}_Q^{\overline{d}} \cup d\} \backslash r} \prod_{\psi \in \Psi_{\overline{d}}} \psi(x_\psi) \sum_{y \in \{\mathcal{RV}_Q^{d} \backslash d\} \backslash r} \prod_{\psi \in \Psi_d} \psi(y_\psi) \\ &= \frac{1}{Z} \sum_{x \in \{\mathcal{RV}_Q^{\overline{d}} \cup d\} \backslash r} \prod_{\psi \in \Psi_{\overline{d}}} \psi(x_\psi)\, \psi^{\mathcal{E}^d}\end{aligned}$$

where $\psi^{\mathcal{E}^d}$ is a joint factor over d and the external variables defined with respect to d. Again looking at Fig. 1a from the paper as an example, with $outcome_T^b$ as the decomposition point, we perform the summation over $\{x1_T^b, y1_T^b, z1_T^b, x2_T^b, y2_T^b, z2_T^b\}$ and are left with a factor over only $outcome_T^b$. This factor can them be multiplied with the remaining factors in the program and $outcome_T^b$ can be summed out.

In this formulation, a decomposition point d implies a structured process to compute $P(r)$ from a set of factors defined on a model: First, compute a joint distribution with respect to the decomposition point, then compute $P(r)$ using the joint distribution and the remaining factors. Computing the joint distribution over the external variables can be accomplished by any algorithm, as can the computation of $P(r)$ once the joint distribution is computed.

So far, we have only mentioned a single decomposition point in a model. However, multiple decomposition points can be defined on a model. In Fig. 1b from the paper, for example, there are four natural decomposition points ($outcome_T^b, outcome_F^b, outcome_T^c, outcome_F^c$) that can be marginalized independently (shown as the boxes in Fig. 1b from the paper). Eqn. 3 can be reformulated for multiple points as:

$$P(r) = \frac{1}{Z} \sum_{x \in \{\mathcal{RV}_Q^{\overline{D}} \cup D\} \backslash r} \prod_{\psi \in \Psi_{\overline{D}}} \psi(x_\psi) \prod_{k=1}^{n} \psi^{\mathcal{E}^{d_k}} \tag{4}$$

where there are n decomposition points, and $\mathcal{RV}_Q^{\overline{D}}$ is the intersection of $\mathcal{RV}_Q^{\overline{d_k}}, k = 1, \ldots n$. Decomposition points can be nested inside other decomposition points, allowing inference to proceed in any hierarchical structure implied by the model.

In principle, any RV could potentially be a decomposition point. However, we would like to choose a decomposition point d that leads to a small joint factor $\psi^{\mathcal{E}^d}$ and eliminates as many variables in $\mathcal{RV}_Q$ as possible. Chains present a natural decomposition point, which have the benefit of being automatically derived from the program ands don't need to be specified by the programmer. When we apply the Chain function f to a parent value v, $f(v)$ is a program that defines a sequence of RVs, ending in a definition of a variable named "outcome". By the semantics of Chain, only the outcome RV can be used anywhere else in the program. For each Chain defined in Q, we create a decomposition point at $outcome_v$ for every value v in the support of r_1. This also implies that $\mathcal{E}^d = \mathcal{F}_{Q'}$ for a decomposition point. Thus, we know that the joint factor created at each decomposition point will only be over each "outcome" variable and free variables defined in the program Q' generated from f.

Algorithm 1 Overview of the SFI algorithm

```
    function DECOMPOSE(program $Q$, variables $\mathcal{E}$, dStrategy $DS$, iStrategy $IS$)
        $\Psi \leftarrow \emptyset$
        for $c \in Q_{chain}, v \in r_c$ do
            $Q' \leftarrow f_c(v)$
5:          $\mathcal{E}_{Q'} \leftarrow \mathcal{E}_{Q'}^{Q'.outcome} \cup Q'.outcome$
            $\Psi^{\mathcal{E}_{Q'}} \leftarrow DS(Q', \mathcal{E}_{Q'}, DS, IS)$
            $\Psi \leftarrow \Psi \cup \Psi^{\mathcal{E}_{Q'}}$
        end for
        $\psi^{\mathcal{E}} \leftarrow IS(\Psi \cup \Psi_{\overline{D}}, \mathcal{E})$
10:     return $\psi^{\mathcal{E}}$
    end function
    function SFI(program $Q$, query $q$, dStrategy $DS$, iStrategy $IS$)
        $\psi_q \leftarrow Decompose(Q, q, DS, IS)$
        return $Normalize(\psi_q)$
15: end function
```

2 USING SFI

2.1 SFI Operation

Algorithm 1 outlines inference in SFI. To query for the distribution over an RV q, a user calls the SFI function with the program Q (written in SimPPL), q, a decomposition strategy DS, and an inference strategy IS. DS and IS are functions that guide the decomposition and inference of the model, and are explained in more detail below. The SFI function calls the *Decompose* function, and the resulting factor over q is normalized to compute $P(q)$.

The *Decompose* function visits each decomposition point in Q, applies DS to the sub–model (i.e., program) defined by each point, and marginalizes out the internal variables using IS. On lines 3, SFI iterates over all Chains defined in Q and each value v of the parent variable r_c. On each iteration, it generates Q', the program created by applying the function f_c to a value v (line 4). Next, it creates the set of relevant variables to program Q' as the external variables in Q' and the "outcome" variable (line 5). It then invokes DS on the new program, which returns a set of factors that is added to the current set for Q (lines 6 and 7).

Algorithm 2 A recursive decomposition strategy

```
function RecursiveDecomposition(program $Q$, variables $\mathcal{E}$, iStrategy $IS$)
    return $Decompose(Q, \mathcal{E}, this, IS)$
end function
```

Note that a decomposition may also be recursive, as we describe below.

Once all decomposition points have been visited, the set of factors not generated from a decomposition point ($\Psi_{\overline{D}}$) is added to Ψ and IS is applied which returns a factor $\psi^{\mathcal{E}}$ over the variables in $\mathcal{E}$ (lines 9 and 10). Much of the work of the SFI framework is performed by the decomposition and inference strategies, so we explain these in detail below.

2.2 Strategies for Decomposition

A decomposition strategy DS is a method that defines how a program should be decomposed. It is a function that receives a program Q' and set of relevant variables $\mathcal{E}_{Q'}$, and returns a set of factors $\Psi^{\mathcal{E}_{Q'}}$ over *at least* $\mathcal{E}_{Q'}$. The simplest DS is what we call "raising": For a point d, return the set of factors over all variables defined in Q'. This strategy performs no inference, and as a result, all of the factors from $\mathcal{RV}_{Q'}$ are "rolled up" to the top–level. If each d is raised, we get a "flat" strategy. This is how typical inference works; factors are created for all variables and all non–query variables are marginalized out in a flat operation.

To take advantage of different inference algorithms, it is clearly beneficial to have a DS that actually reduces the number of variables in the returned factor set $\Psi^{\mathcal{E}_{Q'}}$. As such, we define a recursive strategy as one that will recursively apply the *Decompose* function until no more decomposition points are found, shown in Alg. 2.

Here, each decomposition point in a model is recursively visited in a depth–first traversal. Once a program is reached with no decompositions, IS is applied to the factors in the program, and a joint distribution over the external variables and outcome is returned, and the process is repeated. This is referred to as hierarchical inference in SFI.

More complex and sophisticated strategies can also be applied. For example, a strategy could decompose only if $\mathcal{E}$ is at most n variables (n would have to be specified at compile time). If the number of external variables is greater than n, then the function returns all of the factors defined for the program without running any inference strategy. Otherwise, it calls *Decompose* again to continue the recursive decomposition.

2.3 Strategies for Inference

A strategy for inference applies an inference algorithm to a set of factors defined by program Q and returns a joint distribution over $\mathcal{E}$, the set of external variables in the factors. While SFI uses factors communicate the joint distribution to other programs, there is no restriction that an algorithm operate on factors. As long as the algorithm can ingest factors from other decompositions and output a joint factor over $\mathcal{E}$, then any algorithm can be used.

SimPPL's implementation of SFI uses factor–based algorithms. There are three algorithms available: Variable elimination (VE) (Koller and Friedman, 2009), belief propagation (BP) (Yedidia et al., 2003) and Gibbs sampling (GS) (Geman and Geman, 1984). VE and BP are standard implementations of these algorithms on factors, and as such we do not provide any details. GS is implemented on a set of factors, but integrating it into SFI is not trivial. Much of the effort is due to the determinism frequently found in PPLs. Our implementation uses automated blocking schemes to ensure proper convergence of the Markov chain. Details on GS can be in Section 3.

2.3.1 Choosing an Algorithm

SFI provides the opportunity to develop schemes that dynamically select the best inference algorithm for a decomposition point, serving as the foundation for an automated inference framework. At the application of the inference strategy, there is opportunity to analyze and estimate the complexity of various algorithms applied to the factors, and choose the one with the smallest cost (e.g., speed or accuracy). For example, methods that estimate the running time of various inference algorithms on a model (Lelis et al., 2013) can be encoded into an inference strategy, and the algorithm with the lowest estimated running time can be chosen.

We created a simple heuristic to choose an inference algorithm, but yet still demonstrates the potential of the approach. As VE is an exact algorithm, it is always preferred over other algorithms if it is not too expensive, but unfortunately is impractical on most problems. We therefore have a heuristic to use VE on a set of factors. We first compute an elimination order, O, to marginalize to the external variables. The cost of eliminating a single variable is the increase in cost between the new factor involving the variable and the existing factors, and the cost of VE is the maximum over all variables, using O. If the cost is less than some threshold we use VE, otherwise, BP or GS.

To choose between BP and GS, we also use another heuristic. As the degree of determinism in a model strongly correlates with the convergence rate of BP (Ihler et al., 2005), we use the amount of determinism in the model as a choice between using BP or GS. We mark a variable as deterministic if, when using GS, we must create a block of more than one variable. If the fraction of deterministic variables (as compared to all variables) in the model is greater than a threshold, then we invoke BP and otherwise GS. While these strategies are heuristics, they do demonstrate the proof of concept for automated inference, and the results presented in the next section show that they are effective.

3 GIBBS SAMPLING ON FACTORS

We designed a blocked Gibbs sampler that operates on factor graphs and is compatible with our structured factored inference interface.

3.1 Automatic Blocking

Blocked Gibbs sampling is necessary because variables in SimPPL can have deterministic dependencies. Failing to sample deterministically related variables jointly via Gibbs sampling results in a reducible Markov Chain over the state space of the variables that cannot converge to the correct result. Since the algorithm only receives a set of factors and basic information about the variables in the factors, it must automatically determine a set of blocks over which to perform Gibbs sampling, where each block is a set of variables. We block variables together by classifying them as either stochastic or deterministic. For deterministic variables, we maintain a list of its parents (i.e., the set of variables that uniquely determine its value). Stochastic variables can be thought of as variables with no deterministic parents. We begin by placing each stochastic variable in its own block. Then, we recursively add each deterministic variable to every block that contains one of its parents.

3.2 Gibbs Sampling with Factor Operations

After block creation, the algorithm can proceed with Gibbs sampling on one block at a time. It must first generate a consistent sample for each variable. If no variables have hard evidence, the algorithm can generate an initial sample by performing forward sampling according to the factors. Otherwise, a WalkSAT-like procedure is necessary to produce a sample consistent with the evidence. To compute the joint distribution of a block conditioned on its Markov blanket, observe that the Markov blanket of a block consists of the set of variables that share a factor with at least one variable in the block. Thus, an iteration of Gibbs sampling on a block proceeds as follows:

1. Take the set of factors defined over at least one variable in the block.
2. Condition these factors on the current assignment of the block's Markov blanket.
3. Marginalize each factor to the variables in the block.
4. Compute the joint distribution over the block by taking the product of the conditioned and marginalized factors.
5. Sample from the normalized joint distribution over the block.
6. Update the current assignment of variables in the block.

A single iteration of Gibbs sampling completes when this procedure has been applied once to every block, following which we may record a sample. Unfortunately, this naive implementation is generally exponentially slow in the size of the block. In the SimPPL implementation, factors are stored as a map from an assignment of the factor's variables to a real-valued weight. Consequently, conditioning and marginalizing generally takes time proportional to the number of entries in a factor. Additionally, computing a product of two factors with distinct variables takes time proportional to the product of the number of entries in the two factors. This fails to take advantage of the fact that variables in the blocks are deterministically related. Indeed, for any assignment of a single value to a variable in the block, there is only one consistent assignment of values for the remaining variables in the block when given the Markov blanket. This is to say that if the single non-deterministic variable in a block has a set of outcomes of size n, a better implementation should take time linear in n to compute the joint distribution given the Markov blanket.

3.3 Performance Optimizations

We describe several optimizations that allow our factored Gibbs sampler to perform comparably to a traditional Gibbs sampler.

3.3.1 Conditioning and Marginalizing

We begin by changing the choice of data structure for factors in a block. We take each factor and partition its variables into two sets: those that belong to the block, B, and those that belong to the block's Markov blanket, M. We then group the rows in the factor by assignment to variables in M. We create a new factor F which maps a variable assignment over M to a factor over B. Thus, F is a modified factor which stores a factor instead of a real–valued weight. This effectively allows us to perform the conditioning and marginalizing steps above in constant time by looking up a row in F.

3.3.2 Computing the Joint Distribution

We improve the computation speed of the joint distribution over a block by taking advantage of sparse factors. Sparse factors in our language are factors that store only nonzero rows. Such sparse factors are useful because computing a product of two sparse factors usually takes linear time in the number of entries when the product contains no more nonzero entries than either of the two factors used to compute it. Intuitively, if we can preserve the "sparseness" when taking this product, then the product can be computed efficiently. Based on our blocking scheme, we expect the joint distribution over a conditioned block to be extremely sparse because of the deterministic relationships between variables in a block. To compute this distribution quickly, it is crucial that every intermediate factor used in computing the product is as sparse as possible. We accomplish this with a priority queue that orders factors according to their sparseness. We define the sparseness S of a factor over v variables with e nonzero entries by:

$$S(v, e) = e^{\frac{1}{v}}$$

We choose this function because if each variable can take on $n > 1$ possible values and every row in the factor is nonzero (e.g., if every variable in the factor is independent), then the sparseness is exactly n. Otherwise, the sparseness is strictly less than n. For example, if all of the variables are perfectly correlated and can take on only one possible value (i.e., $e = 1$), the sparseness is exactly 1. Thus, we favor factors with a smaller sparseness. Computing the product of a set of factors proceeds as follows:

1. Insert all of the factors in a priority queue according to S.

2. Dequeue the two factors for which S is smallest and compute their product.

3. Insert the product of these factors into the priority queue according to S.

4. Repeat (2) and (3) until the priority queue contains only a single factor.

3.3.3 Caching

Our final optimization is to optionally cache the most recently computed joint distributions over a block according to the corresponding assignment to the block's Markov blanket. The Markov chain used in Gibbs sampling will frequently revisit recently visited states, so storing these distributions allows saving some computation time at the cost of memory. By default, we store up to 1000 of the most recently used distributions.

References

S. Geman and D. Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):721–741, 1984.

A. T. Ihler, J. Iii, and A. S. Willsky. Loopy belief propagation: Convergence and effects of message errors. In *Journal of Machine Learning Research*, pages 905–936, 2005.

D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

L. H. Lelis, L. Otten, and R. Dechter. Predicting the size of depth-first branch and bound search trees. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 594–600. AAAI Press, 2013.

J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. *Exploring artificial intelligence in the new millennium*, 8:236–239, 2003.