Structured Factored Inference for Probabilistic Programming

Avi Pfeffer, PhD
Charles River Analytics

Brian Ruttenberg, PhD
Charles River Analytics

William Kretschmer
MIT

Alison O'Connor
Charles River Analytics

Abstract

Probabilistic reasoning on complex real-world models is computationally challenging. Inference algorithms have been developed that work well on specific models or on parts of general models, but they require significant hand-engineering to apply to full-scale problems. Probabilistic programming (PP) enables the expression of rich probabilistic models, but inference remains a bottleneck in many applications. Factored inference is one of the main approaches to inference in graphical models, but has trouble scaling up to some hard problems expressible as probabilistic programs. We present structured factored inference (SFI), a framework that enables factored inference algorithms to scale to significantly more complex programs. Using models encoded in a PP language, SFI provides a sound means to decompose a model into submodels, apply an algorithm to each submodel, and combine results to answer a query. Our results show that SFI successfully reasons on models where standard factored inference algorithms fail due to computational complexity. SFI is nearly as accurate as exact inference and is as fast as approximate inference methods.

1 INTRODUCTION

Probabilistic modeling is at the core of many artificial intelligence (AI) applications. The complexity, richness, and diversity of models are rapidly growing as AI takes on a larger role in everyday life. As a result, the efficiency of probabilistic inference is critical for practical use of these models. Despite research into efficient algorithms, probabilistic inference remains a bottleneck in many applications.

Proceedings of the 21st International Conference on Artificial Intelligence and Statistics (AISTATS) 2018, Lanzarote, Spain. PMLR: Volume 84. Copyright 2018 by the author(s).

While there is a large body of efficient algorithms, no algorithm performs sufficiently on every model, and often there are trade-offs between algorithms. Once an algorithm proves to work well on a problem, any modification is no guarantee of continued success. Thus, AI engineers must select and configure an algorithm for each problem, a task that is often more time consuming than constructing the model itself.

One solution to reduce this burden is to automatically select an algorithm that performs well on a problem. An impediment to this approach is that the size and complexity of real-world models makes it difficult to determine the best algorithm. Further, different algorithms may be appropriate for different parts of a model, known as submodels. For example, one algorithm may work for a continuous submodel and another for a discrete submodel. As a result, one should select an algorithm for each part of a model, and combine the results to answer a query. Central to this approach is a sound method to decompose models.

The emerging field of probabilistic programming (PP) provides the opportunity to support this automated model decomposition. PP (Koller et al., 1997; Goodman, 2013) provides expressive and general purpose languages to encode probabilistic models as executable programs. PP developers can leverage the power of programming languages to create rich and complex models, and use built-in inference algorithms that operate on any model written in the language. More importantly, since the models are encoded as programs, developers can use the program structure and analysis to understand its properties before inference.

We introduce structured factored inference (SFI), a PP inference framework that uses program semantics to identify *decomposition points* within a probabilistic model that define a hierarchy of submodels. SFI's decomposition has three major benefits. First, it facilitates solving many manageable subproblems rather than one potentially intractable large problem. Since some factored algorithms scale poorly with the problem size, it can sometimes be easier to find efficient solutions to small problems and then compose them than to find a solution to the full problem. For example, Pfeffer (2000) showed that a decomposition strategy finds much better variable elimination (VE) orderings than standard heuristics on the full problem.

Second, hierarchical decomposition enables us to use dynamic programming techniques to compose submodel solutions, and re-use prior computation. For example, if the same subproblem appears again, the cost of its solution is only incurred once. This is similar to lifted inference techniques (de Salvo Braz et al., 2007).

Third, SFI's decomposition method enables any factored algorithm to solve a subproblem. The decomposition and choice of solution algorithms happen automatically. SFI can adaptively select an appropriate algorithm for a specific submodel, as opposed to applying the same algorithm to all parts or the whole problem. For example, if a problem contains small discrete subproblems, VE might be used for each of these to obtain answers that are then wrapped in an MCMC algorithm at the top level of the hierarchy. Alternatively, the subproblems may contain variables with large continuous state spaces for which MCMC is appropriate, while this is wrapped in a tractable discrete problem at the top level that can be solved by VE. In both cases, combining algorithms can yield more accurate results than if MCMC was used alone, while VE alone would have been infeasible.

We show the benefits of SFI on three models using a combination of exact and approximate inference algorithms. Our results show that probabilistic reasoning using SFI is nearly as accurate as exact inference and achieves performance equal to or better than approximate inference. Further, SFI effectively reasons on models where standard factored inference algorithms fail due to computational complexity by interleaving the construction of factors and solutions of subproblems to create more compact factors. The SFI framework is general and expandable, with potential to be the foundation for a general automated inference system.

2 RELATED WORK

Automated algorithm selection has been a long desired goal in computer science, with possibly the first formulation by Rice (1976). As such, it has been applied to a variety of disciplines in the field, such as scientific computing (Houstis et al., 2000), game theory (Guerri and Milano, 2004), and AI (e.g., satisfiability problems (Xu et al., 2008)). Most efforts learn how to apply the best algorithm for a problem. For example, Guo (2003) uses Bayesian networks (BNs) to learn and select the best algorithm for a problem; however, neither the problems or algorithms are specific, and can be generalized to a variety of problems. SFI is complementary to this existing work, as the framework can leverage prior methods for improved algorithm selection.

Probabilistic inference is unique in some respects as the independence properties of models provides the opportunity to apply algorithms to different parts. However, there has not been significant progress that takes advantage of this. Our approach is similar in spirit to work on black box variational inference (Ranganath et al., 2013). These methods attempt to reduce the burden of configuring and applying inference to general probabilistic models, and in a sense attempt to automatically find the best configuration of the algorithm. While these approaches are promising, they only consider a single algorithm. SFI decomposition strategies also bear similarity to structured variable elimination (SVE) (Pfeffer, 2000). Like SVE, SFI enjoys the benefits of decomposition and reusing work. However, SVE applies the same algorithm to each subproblem, whereas SFI is a general framework for decomposing problems and optimizing each submodel separately.

Recently, Murray et al. (2017) provided a tractable substructure in probabilistic programs to reduce variance in Monte Carlo estimators based on Rao-Blackwellization. The solution requires that each submodel being marginalized has an analytical solution, as they do not propose a general framework for performing this marginalization otherwise. SFI does serve as a general framework in this case, and certainly can be used in conjunction with these Monte Carlo methods.

3 PROBABILISTIC PROGRAMMING

SFI has manifested in the Figaro probabilistic programming language (PPL), although it is a general approach to inference applicable to all PPLs. To present SFI, we use a minimal PPL, SimPPL. While SimPPL lacks the complexity of many PPLs, it is equally expressive.

3.1 SimPPL Language

The central concept in SimPPL is a program, which contains a sequence of random variables (RVs). Each RV r has a type T_r, which defines the set of values it can take. A program Q has a set of $free$ variables $\mathcal{F}_Q$, and consists of definitions of the form $r = e$, where e is an expression. The set of RVs defined in Q (not including $\mathcal{F}_Q$) is denoted RV_Q. An RV is *available* if it is either in $\mathcal{F}_Q$ or defined previously in Q. The set of available RVs with respect to an RV r is the set of RVs that can possibly be used when defining r, denoted A_r.

An expression defining an RV r is one of the following:

- A primitive, which directly defines a probability distribution d_r (e.g., $Uniform$) over values.

- *Apply*$(r_1, \ldots, r_n, f)$, where $r_1, \ldots, r_n$ are available RVs and f is a function $T_{r_1} \times \cdots \times T_{r_n} \rightarrow T_r$. The RVs $r_1, \ldots, r_n$ are arguments of r.
- *Chain*(r_1, f), where r_1 is an available RV and f is a function $T_{r_1} \rightarrow \mathcal{Q}$, where $\mathcal{Q}$ is the space of programs such that for each $Q' \in \mathcal{Q}$, $\mathcal{F}_{Q'} \subseteq \mathcal{A}_r$ and the final RV in Q' has type T_r. The final RV represents the return variable of the program. The RV r_1 is an argument of r.

Because an argument of an RV is always available to the RV, a program defines a directed acyclic graph over its variables.

3.2 SimPPL Semantics

The semantics of program Q can be understood by defining a generative process, that given values for the RVs in $\mathcal{F}_Q$, produces values for each of the variables defined in Q. For each variable r in Q, values for all the variables in A_r are generated before a value for r is generated. If r is defined by a primitive, it is generated by the associated probability distribution d_r. If r is defined by *Apply*$(r_1, \ldots, r_n, f)$, it is generated by applying the function f to the previously generated values of $r_1, \ldots, r_n$. If r is defined by *Chain*(r_1, f), it is generated by first applying the function f to obtain a program Q' and then generating values for each of the variables in Q'. The value of r becomes the value of the final variable in Q'. Assume that programs terminate.

For what follows, the essential point to remember about the semantics of SimPPL is that when r is defined by *Chain*(r_1, f), applying the function f to the value generated for r_1 produces a new program Q'. We call f the chain function, r_1 the parent value, and Q' the subprogram created for f and r_1. In any application of SimPPL, there is a single program called the top level program that is not a subprogram. All other programs are subprograms of this top level program.

SimPPL allows conditioning and querying any top level program variable. This can encode both hard observations (only allowing a fixed value or set of values for a variable) and soft constraints (preferring some values over others). Queries take the form of computing the marginal distribution over a variable or the expectation of a function on the values of a variable.

4 STRUCTURED FACTORED INFERENCE

The intuition behind SFI is simple: If a model can be broken into smaller submodels (i.e., programs) that can solved independently (i.e., marginalizing out non-relevant variables), then different algorithms can be applied to different parts. SFI uses SimPPL semantics to identify *decomposition points* within a probabilistic model, creating an abstract hierarchy of submodels. Each submodel is independently reduced to a joint distribution over variables relevant to a query, using any inference method. Using factors to represent the joint distribution, the results are incorporated into the inference algorithms applied on other submodels. SFI also uses factors to combine information from solved (i.e., already marginalized to a relevant joint distribution) submodels to answer queries. Fundamentally, SFI is a framework for applying two types of strategies: a decomposition strategy that divides a model into submodels, and an inference strategy that applies an inference algorithm to each submodel[1].

Decompositions are based on chain variables. That is, there is a separate subproblem for each value of the chain's parent. Every subprogram that is a chain function for some parent value becomes a submodel in SFI and has an inference method applied to it. This ensures that if the inference methods applied to submodels are all exact, then SFI itself is exact. However, one of the main benefits of SFI is that it lets us mix exact and approximate algorithms for different submodels, leading to much better accuracy than using an approximate algorithm for a problem as a whole.

Consider the model (Fig. 1a), with RVs, a, b, and c, defined in Q. RVs b and c generate a value using the Q' that is generated by $f(a)$, where $outcome^b_T$ refers to the outcome for RV b when a is true. Each chain generates a program Q' for both true and false. With the exception of *outcome*, the RVs defined by Q' are not directly needed to reason about a, b, and c. That is, all of the RVs defined in Q' except *outcome* and $\mathcal{F}_{Q'}$ can be marginalized out of Q'. A joint distribution over $outcome \cup \mathcal{F}_{Q'}$ is all that is needed to reason at the top level program Q. Since this marginalization is self-contained, any inference algorithm that can compute a joint distribution can be applied to each submodel (Fig. 1b). This joint distribution is represented as a factor. Factors are then "rolled up" and used by another algorithm to answer queries on Q. This is the core operation of SFI: Given a submodel, use an algorithm to marginalize out internal variables and return a joint distribution over *outcome* and the free variables, repeating the process until the query is answered.

4.1 Factored Representation

Using factors in SFI has several advantages: It provides an interface to communicate the joint distribution of a submodel to other submodels and it makes SFI

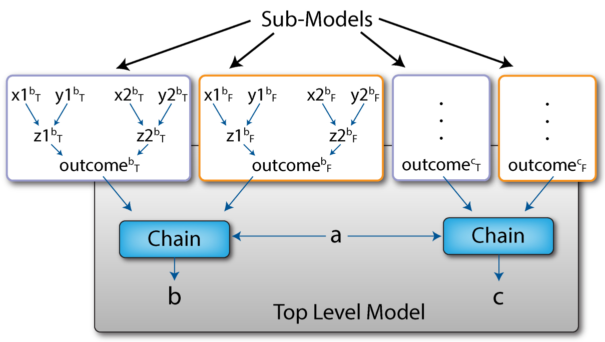
[1]Details on decomposition and inference strategies appear in Section 2 of the supplementary material.

```
a = Flip(0.6)
b = Chain(a,f)
c = Chain(a,f)

f(true) = {
  x1 = Flip(0.9)
  y1 = Flip(0.8)
  z1 = Apply(x1, y1, (b1, b2) => b1&&b2)
  x2 = Flip(0.7)
  y2 = Flip(0.8)
  z2 = Apply(x2, y2, (b1, b2) => b1&&b2)
  outcome = Apply(z1, z2, (b1, b2) => b1||b2)
}

f(false) = {
  x1 = Flip(0.1)
  y1 = Flip(0.8)
  z1 = Apply(x1, y1, (b1, b2) => b1&&b2)
  x2 = Flip(0.2)
  y2 = Flip(0.8)
  z2 = Apply(x2, y2, (b1, b2) => b1&&b2)
  outcome = Apply(z1, z2, (b1, b2) => b1||b2)
}
```

(a)



(b)

Figure 1: SimPPL code shown in (a), where $Flip(x)$ represents a Boolean distribution, with the probability of being *true* equal to x, and => denotes an anonymous function. The corresponding model is shown in (b).

algorithm-agnostic. Any algorithm that can compute a joint distribution and return a factor can be used, including factored algorithms and sampling algorithms, which can post-process a joint distribution into a factor. SimPPL's implementation of SFI uses three factor-based algorithms: VE (Koller and Friedman, 2009); belief propagation (BP) (Yedidia et al., 2003); and Gibbs sampling (GS) (Geman and Geman, 1984). In theory, primitive distributions can be defined as either discrete or continuous. However, SimPPL discretizes continuous distributions[2]. While SFI can work with both nested exact and approximate algorithms, we make no claims on the theoretical accuracy of the final solution using nested approximate algorithms. Proving bounds on SFI's accuracy with nested approximate algorithms is a target for future research.

4.2 Evolution of Chain Construction

Since we are using a factored framework, the representation of factors at the junction between a model and its submodels is critical. Without SFI, the representation of these factors is unnecessarily expensive. One of the major side benefits of SFI is that, by eliminating subproblems first, these factors can be made much simpler in many cases. We explain three constructions of chain factors consecutively to demonstrate in a logical manner how SFI improves the overall inference process.

The original chain construction did not take advantage of SFI. We describe this construction, using the variable X generated by $Chain(Y, f)$. For each value y_i of the variable Y, we expanded the factor graph to include all the variables in the program $f(y_i)$. From the definition of $f(y_i)$, we knew that if the value of Y was y_i, then the value of the variable X was equal to the value of the final variable in $f(y_i)$, which we call Z_i. In other words, X was a deterministic function of $Y, Z_1, \ldots, Z_n$, where X was equal to the value of Z_i if $Y = y_i$. We add a new variable W defined by the expression $Apply(Y, Z_1, , Z_n, g)$ to capture this logic.

This original construction was inefficient. The factor created for W contained an entry for every combination of values of $X, Y, Z_1, \ldots, Z_n$. This is exponential in n (the number of values of the parent Y of the chain). Since a variable can have many values, this is an unacceptable size. Therefore, we introduced a second construction based on the decomposition of the chain factor. According to this second construction, when Y took on value y_i, only the value of Z_i was relevant to determining the factor entry. Specifically, the factor entry was 1 if $z_i = x$ and 0 otherwise. All the other Z_j were irrelevant. For a specific i, we captured the cases where Z_j was relevant or irrelevant with a factor over X, Y, Z_i representing the function f_i defined by:

$$f_i(x, y_j, z_i) = \begin{cases} 1 \text{ if } j \neq i \text{ or } z_i = x \\ 0 \text{ otherwise} \end{cases}$$

By defining $f(x, y, z_1, \ldots, z_n) = \prod_{i=1}^{n} f_i(x, y, z_i)$, where f is the function that constructs the factor that defines W, we see that each factor in the decomposition only mentions three variables and the number of factors is linear in n. This second construction is exponentially more compact than the original construction.

[2]Details on factored representation appear in Section 1.1.1 of the supplementary material

However, this second chain construction can still cause problems when there are a lot of parent values. Consider the case where all the produced subprograms have the same set of values for the last variable in the program, and let the number of these values be N. Let the number of parent values be P. Then the total size of the chain factors produced is $O(N^2P^2)$. On the other hand, if all the subprograms have a disjoint set of N values, the total size of the chain factors is $O(N^2P^3)$.

To avoid this issue, we introduced a third (and final) construction, that takes advantage of SFI, resulting in a single factor whose size is $O(NP)$ when all subprograms produce the same set of values, and $O(NP^2)$ when they produce disjoint sets of values. This is a quadratic savings in the size of the factors, which in turn leads to quadratic or better savings in the cost of inference.

The reason for the expensive original construction was because the variables representing the outcome of a subprogram might be linked to other variables in the model, and the produced factors had to preserve all these linkages. However, in SFI, all the variables in a subprogram, except for the final variable, are completely eliminated before solving a higher level program. As long as the subprogram uses no free variables, there will be no connections between the final variable in the subprogram and other variables in the model.

The final construction is for cases where all the subprograms created for different parent values of a *Chain* expression use no free variables. This can easily be detected by examining the factors produced after solving the subprograms. If none of those factors mention free variables, the final construction can be applied. Our experience shows that this is the most common case.

For a definition $X = Chain(Y, f)$ that satisfies this property, the final construction creates a factor over the two variables Y and X. The values of Y are all the parent values, while the values of X are the union of all the values of the final variables of the subprograms created for this *Chain* expression. Therefore, if all the subprograms produce the same set of N values for each of the P parent values, the total number of entries in the factor is NP, while if the subprograms produce disjoint sets of N values, for a total of NP values, the number of entries in the factor is NP^2.

To define this factor F, we must specify the entry associated with each parent value y_i and each outcome value x_i. If x_i is not a possible value of the final variable in the subprogram created for parent value y_i, the entry is 0 because x_i cannot be associated with y_i. Otherwise, let $F_1, \ldots, F_k$ be the factors produced after eliminating variables in the subprogram created for parent value y_i. By assumption, this subprogram does not use any free variables, so these factors mention only the variable x_i. The entry associated with y_i and x_i in F is then the product of entries associated with x_i in $F_1, \ldots, F_k$.

5 EXPERIMENTS

Our implementation of SFI optimized between VE, BP, and GS. We chose VE, BP, and GS as example solvers because they are well understood and widely used. First, the optimizer decided whether to use an exact inference algorithm for a submodel by first choosing an elimination order using standard heuristics and then comparing the cost of VE with this order to the original size of the factors. If the cost was less than some threshold, we used VE, otherwise, BP or GS. To decide between BP and GS, we used the amount of determinism in the submodel, invoking BP with more determinism. Determinism is defined as the fraction of deterministic variables compared to all other variables[3].

We tested SFI using three models. First, we encoded a modified QMR medical diagnosis model (Jaakkola and Jordan, 1999). Like the standard model, our model is a BN of diseases associated with observable symptoms. However, we insert a layer of intermediate illnesses between the disease and symptom layer. The illnesses are conditioned on the diseases, and the symptoms conditioned on the illnesses. The number of diseases and parents per symptom vary so the BN is constructed by randomly connecting nodes between each layer. In each test, we observed a random number of diseases and symptoms and queried a random subset of diseases.

Second, we used a mixed directed-undirected model. The undirected portion is an Ising model (Glauber, 1963), where each Boolean variable v in an $n \times n$ grid has a potential to its four neighbors. The prior over each variable is modeled as a BN conditioned on a causal variable c_n. This model can be viewed as n^2 BNs joined in a top level Ising model. SFI was used for the directed portion and the undirected portion was treated jointly. We varied n in testing, but for each test, a random 20% of the c_n variables are observed as either true or false. All queries were over the posterior distributions of a random subset of Boolean variables.

Finally, we built a model to solve a desktop activity recognition problem from the DARPA Perceptive Assistant that Learns (PAL) Program. Given a sequence of user event types, we determined the associated workflow, instance, and position of each event. Our model is a dynamic BN where each point in time captures the current event (i.e., type, workflow, instance, and position). See Fig. 2 for an illustration of the model.

[3]Details on how to choose between algorithms and the integration of GS with SFI appear in Sections 2.3.1 and 3 respectively in the supplementary material.

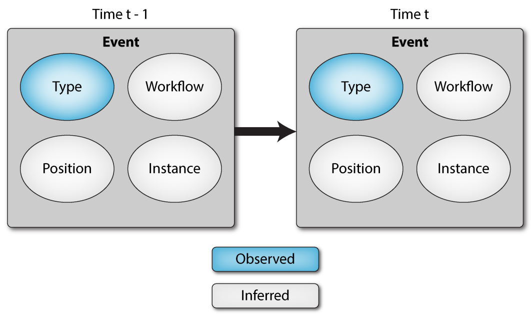


Figure 2: Desktop activity recognition model

These three models are well suited for SFI because they contain a significant amount of structure that can be decomposed: the QMR model's diseases and symptoms are a series of chain RVs; each Boolean variable in the Ising model is a chain that uses each c_n as a parent; and the desktop model represents each transition between states as a chain that depends on the previous state. Further, the desktop model is a realistic model used for a practical application. All non-SFI factored inference algorithms failed to solve the desktop model due to computational complexity.

5.1 Results

First, we tested how different decomposition strategies affect inference using flat and hierarchical strategies with VE and BP on the QMR model (see Fig. 3). In general, a flat strategy represents typical inference, where factors are created for all variables and all non-query variables are marginalized out in a flat operation. A hierarchical strategy represents a recursive inference process in which a large model is decomposed until no more decomposition points are found and each decomposition point in a model is visited in a depth-first traversal. Once a program is reached with no decomposition, inference is performed to return a joint distribution over the external variables and outcome.

For VE, the hierarchical strategy is generally faster. Mathematically, both strategies perform the same operations. However, the hierarchical strategy imposes a partial elimination order; an elimination order is found for each VE instance, but the order that the decomposition points are visited is fixed. The flat strategy uses a heuristic to find the best elimination order given all the factors in the model. From this, it appears that the structure imposed by programmer (i.e., by using chains) finds a better elimination order than the heuristics used to solve this NP-hard problem (Pearl, 2014). These results are consistent with previous work on SVE (Koller and Pfeffer, 1997).

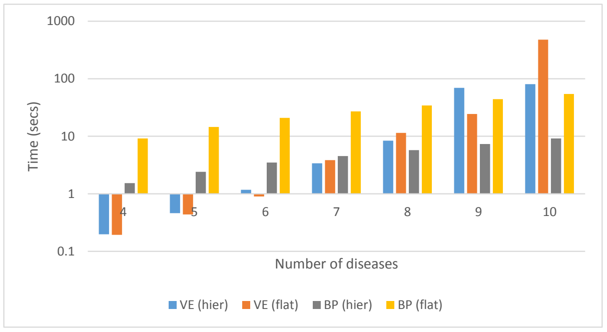
For hierarchical BP, each BP instance ran for 10 iterations, whereas flat BP ran once for 100 iterations. The hierarchical strategy is consistently faster. This comparison is not exact though as it is hard to determine how many iterations in a flat strategy "equals" hierarchical iterations. However, looking at the accuracy of the methods in Fig. 3, the hierarchical method is consistently more accurate. Thus, even if the flat BP is run for more iterations to improve its accuracy (assuming it has not converged), it is already dominated in runtime and accuracy by the hierarchical method.

Next, we applied our three-algorithm (i.e., VE, BP, and GS) hybrid strategy with hierarchical decomposition to determine if we can improve the speed and/or accuracy using multiple algorithms as compared to one. The results for the QMR model for runtime and accuracy are shown in Fig. 4. The inference strategy only chose to run VE and BP, so we only compare to those.

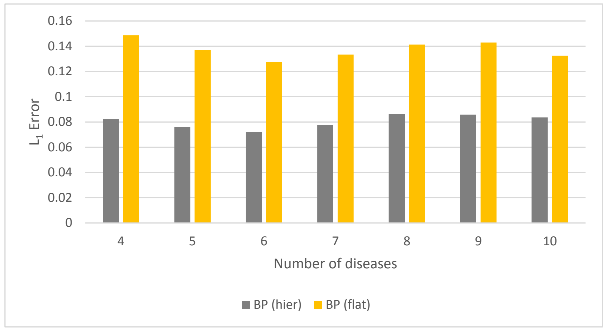
For runtime, VE is competitive until the number of diseases reaches nine; at 10 diseases, we start to see the effect of exponential growth. Comparing BP to the combined VE/BP method directly is difficult. However, combined with the accuracy results in Fig. 4, we can analyze the relationship between runtime and accuracy for all methods. BP(10) and BP(50) have comparable accuracy. The hybrid methods, however, are much more accurate. VE/BP(10) is nearly four times more accurate than BP alone. Further, VE/BP(50) has nearly zero error. The runtime for the hybrid methods are both faster than their "respective" BP version (i.e., comparing BP(10) to hybrid VE/BP(10)). While VE/BP(50) has a longer runtime than the single BP(10) iteration strategy, it is nearly as accurate as VE with significantly less runtime.

The results on the mixed model are shown in Fig. 5. The hybrid strategy only ran VE and GS. Similar to the QMR model, VE has the best performance until the model becomes large, at which point the hybrid strategy has the best runtime. The accuracy of the hybrid methods is better than the GS approaches. Overall, the hybrid approach dominates the GS approach in terms of accuracy and time. However, as more GS iterations are performed, the performance gap decreases, as the runtimes of both approaches are dominated by applying GS to the undirected portion of the model.

In testing the desktop model, we evaluated how well SFI using the final chain construction scaled to complex problems compared to non-SFI factored algorithms and SFI using the original chain construction. To do this, we incrementally increased the number of possible events in the model. The smallest version of the problem had only 11 possible events, while the full problem required we perform inference in an event space that had 162 possible events.

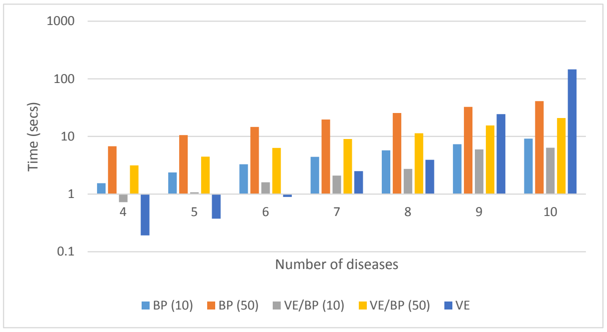


(a)

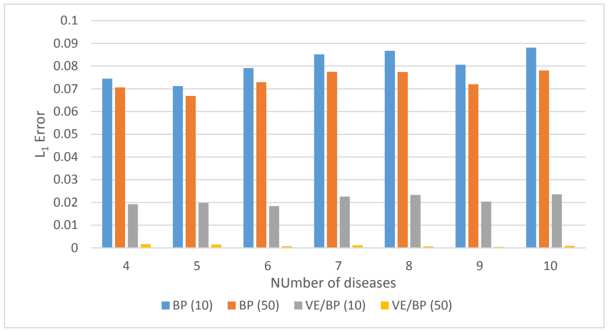


(b)

Figure 3: Comparing flat and hierarchical strategies in terms of (a) runtime and (b) accuracy to VE.

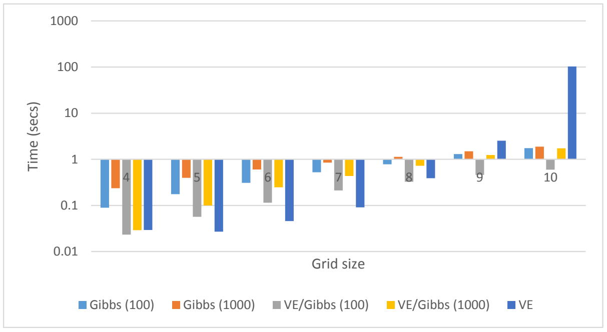


(a)

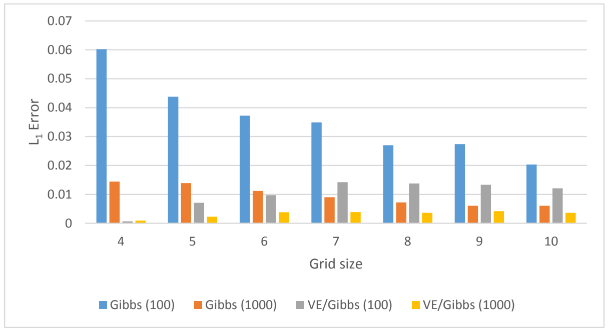


(b)

Figure 4: Comparing the hybrid inference strategy in terms of (a) runtime, and (b) accuracy to VE.



(a)

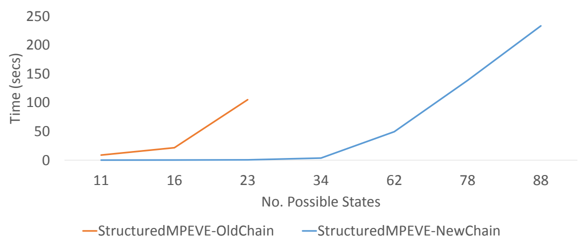


(b)

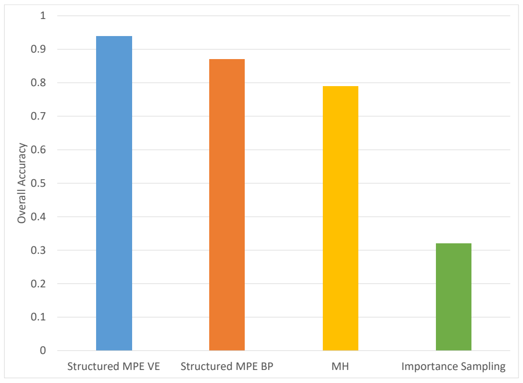
Figure 5: Comparing the hybrid inference strategy in terms of (a) runtime, and (b) accuracy to VE.

Even with 11 possible events, all non-SFI factored algorithms failed due to computational complexity. SFI with the original chain construction lasted through the first three levels: 11, 16, and 23 possible events, but failed to complete inference by the next level, 34. With the creation of more compact factors using the final chain construction, SFI worked for the entire problem and resulted in quadratic time savings. We show this savings up to 88 events in Fig. 6a. In the final version of the problem with 162 events, we see the first large jump in runtime, of around 45 minutes.

For the 88-event model, we compared SFI to two sampling algorithms, Metropolis-Hastings (MH) (Chib and Greenberg, 1995) and importance sampling (see Fig. 6b). When run for the same length of time as VE, MH yielded a 79% overall accuracy rate compared to VE's 94% overall accuracy rate, where overall accuracy refers to the number of events with the correctly identified workflow, instance, and position divided by the total number events. When run for a 50% longer time than VE, importance sampling only yielded a 32% overall accuracy rate. Despite its longer runtime, BP had the



(a)



(b)

Figure 6: Comparing (a) old vs new chain runtime, and (b) accuracy between VE and BP.

second highest overall accuracy rate, 87%. VE's average runtime was more than eight times faster than BP (233 seconds versus 2,007 seconds per user). VE was consistently faster than BP for every model version.

Our algorithms significantly outperformed the DARPA PAL Program's reference model (a logical Hidden Markov model). For the 162-event model, the PAL reference model had a workflow accuracy of 45%, where workflow accuracy refers to the number of correct workflows identified divided by the total number events. Conversely, our model using SFI VE had a workflow accuracy of 81%.

These results demonstrate that SFI is able to solve realistic practical problems that cannot be solved by other factored algorithms.

6 CONCLUSION

We introduced a new framework for inference in probabilistic modeling, structured factored inference. Leveraging the capabilities of PP, we have shown a semantically sound method to decompose a model into submodels. We demonstrated that SFI can be used to implement a basic automated inference scheme that is nearly as accurate as exact inference methods and solves problems where non-SFI algorithms fail due to computational complexity.

SFI is a general framework that can incorporate any solver that operates on factors. Our goal in designing SFI was to extract the best performance when using any set of such solvers. We chose VE, BP, and GS as example solvers because they are well understood and widely used. We do not claim that SFI coupled with these algorithms outperforms specialized advanced inference techniques. Rather, we demonstrate that automatic structured decomposition improves the performance of factor-based inference solvers. We anticipate that if specialized techniques were incorporated into the SFI framework, it would perform even better. We also emphasize that we used simple heuristics that provided benefits. We anticipate that with better heuristics, the results would be even better.

This work serves as a starting point for a more robust automated inference framework, but several capabilities still must be developed. First, more intelligent algorithm selection methods must be developed. Recent work provides a starting point for this (Flerova et al., 2012; Lelis et al., 2013), but new methods that leverage the analyzability of PP can make the estimation of complexity more accurate (Hur et al., 2014). New decomposition points also must be developed to enable more sophisticated decomposition. While our chain decomposition is effective, user-defined or object-oriented decomposition points may be more effective at decomposing a model to facilitate faster inference. We are currently working on an extension to SFI, called lazy structured factored inference (LSFI), that enables SFI to be used for models that do not terminate in all cases. Our hope is that the LSFI framework will be the catalyst for future research in these areas.

Acknowledgments

This material is based upon work supported by the United States Air Force (USAF) under Contract No. FA8750-14-C-0011. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the USAF.

References

S. Chib and E. Greenberg. Understanding the metropolis-hastings algorithm. *The american statistician*, 49(4):327–335, 1995.

R. de Salvo Braz, E. Amir, and D. Roth. 1 lifted first-order probabilistic inference. *Statistical relational learning*, page 433, 2007.

N. Flerova, R. Marinescu, and R. Dechter. Preliminary empirical evaluation of anytime weighted and/or best-first search for map. In *Proceedings of 4th NIPS workshop on Discrete Optimization in Machine Learning*. Citeseer, 2012.

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.

R. J. Glauber. Time-dependent statistics of the ising model. *Journal of mathematical physics*, 4(2):294–307, 1963.

N. D. Goodman. The principles and practice of probabilistic programming. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 399–402. ACM, 2013.

A. Guerri and M. Milano. Learning techniques for automatic algorithm portfolio selection. In *ECAI*, volume 16, page 475, 2004.

H. Guo. A bayesian approach for automatic algorithm selection. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI03), Workshop on AI and Autonomic Computing, Acapulco, Mexico*, pages 1–5, 2003.

E. N. Houstis, A. C. Catlin, J. R. Rice, V. S. Verykios, N. Ramakrishnan, and C. E. Houstis. Pythia-ii: a knowledge/database system for managing performance data and recommending scientific software. *ACM Transactions on Mathematical Software (TOMS)*, 26(2):227–253, 2000.

C.-K. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel. Slicing probabilistic programs. In *ACM SIGPLAN Notices*, volume 49, pages 133–144. ACM, 2014.

T. S. Jaakkola and M. I. Jordan. Variational probabilistic inference and the qmr-dt network. *Journal of artificial intelligence research*, pages 291–322, 1999.

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

D. Koller and A. Pfeffer. Object-oriented bayesian networks. In *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*, pages 302–313. Morgan Kaufmann Publishers Inc., 1997.

D. Koller, D. McAllester, and A. Pfeffer. Effective bayesian inference for stochastic programs. In *AAAI/IAAI*, pages 740–747, 1997.

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.

L. M. Murray, D. Lundén, J. Kudlicka, D. Broman, and T. B. Schön. Delayed sampling and automatic rao-blackwellization of probabilistic programs. *arXiv preprint arXiv:1708.07787*, 2017.

J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 2014.

A. J. Pfeffer. *Probabilistic reasoning for complex systems*. Stanford University, 2000.

R. Ranganath, S. Gerrish, and D. M. Blei. Black box variational inference. *arXiv preprint arXiv:1401.0118*, 2013.

J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of Artificial Intelligence Research*, pages 565–606, 2008.

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.