Efficient Multiple Query Answering in Switched Probabilistic Relational Models[⋆]

Marcel Gehrke, Tanya Braun, and Ralf Möller

Institute of Information Systems, University of Lübeck, Lübeck, Germany
{gehrke, braun, moeller}@ifis.uni-luebeck.de

Abstract. By accounting for context-specific independences, the size of a model can be drastically reduced, thereby making the underlying inference problem more manageable. Switched probabilistic relational models contain explicit context-specific independences. To efficiently answer multiple queries in switched probabilistic relational models, we combine the advantages of propositional gate models for context-specific independences and the lifted junction tree algorithm for answering multiple queries in probabilistic relational models. Specifically, this paper contributes (i) variable elimination in gate models, (ii) applying the lifting idea to gate models, defining switched probabilistic relational models, enabling lifted variable elimination in computations, and (iii) the switched lifted junction tree algorithm to answer multiple queries in such models efficiently. Empirical results show that using context-specific independence speeds up even lifted inference significantly.

Keywords: Lifting · Context-Specific Independence · Switched Models

1 Introduction

Performing inference is an important task in artificial intelligence but unfortunately, inference in general is intractable [4]. To make the underlying problem more manageable, context-specific independences help [2]. Given context-specific independences in a model, inference may require fewer calculations if parts of the model become independent given a context. One approach for Bayesian networks is to look for patterns in conditional probability tables to identify context-specific independences [2]. In such an approach, the context-specific independences are implicitly encoded in the model, which can lead to huge conditional probability tables. Another approach is to explicitly model context-specific independences in a model, avoiding the blowup of tables if implicitly encoding the independences in the tables, and providing specialised inference algorithms [7]. In this paper, we study the problem of efficient inference to answer multiple queries in models that contain explicitly encoded context-specific independences. We call such models switched models as context-specific independences lead to model parts being switched on or off.

[⋆] This research originated from the Big Data project being part of Joint Lab 1, funded by Cisco Systems Germany, at the centre COPICOH, University of Lübeck

To the best of our knowledge, the only approach for probabilistic relational models that may be used to implicitly encode context-specific independence comes from Gogate and Domingos [5] based on Markov logic networks [10]. They speed up inference by only counting worlds in which no clause evaluates to false. For example, we could use a variable A to switch between some worlds. In case $A = true$ is observed, all worlds with $A = false$ are not counted. Thereby, one can implicitly encode context-specific independences. Unfortunately, this approach may also result in large rules in a Markov logic network. Hence, efficient inference in switched probabilistic relational models is still an open problem.

Therefore, we combine lifting [9], gate models [7], and junction trees [6] to build an efficient formalism for inference in switched probabilistic relational models. Lifting allows for exploiting relational structures in a model. Gate models (GMs) provide a formalism to explicitly model context-specific independence using gates for switching, which also allows for modelling, e.g., interventions [8]. Junction trees enable efficient online query answering of multiple queries. Specifically, we use parameterised probabilistic models (PMs). PMs incorporate relational structures by parameterising random variables (randvars), called parameterised randvars (PRVs), which are then combined into parametric factors (parfactors) to model relations with uncertainties. First, we extend PMs with gates, resulting in parameterised gate models (PGMs). Then, we show that variable elimination (VE) can be used for inference with GMs as well as lifted variable elimination (LVE, Poole [9]) for inference with PGMs. Afterwards, we introduce the switched lifted junction tree algorithm (SLJT) by extending the lifted junction tree algorithm (LJT) [3], which uses LVE as a subroutine, to efficiently answer multiple queries in PGMs. Thereby, SLJT solves the problem of answering multiple queries in switched relational models efficiently. Specifically, this paper contributes (i) VE in GMs, (ii) applying the lifting idea to GMs resulting in PGMs enabling LVE in computations, (iii) building a first-order junction tree (FO jtree) for PGMs, and (iv) SLJT to reuse an FO jtree for multiple configurations and efficient multiple query answering in PGMs.

In the following, we begin by recapitulating PMs for relational models and GMs for context-specific independence. Afterwards, we parameterise GMs by leveraging the lifting idea and introduce SLJT. Then, we evaluate SLJT against implicitly modelling context-specific independences and specifying all possible submodels corresponding to different switch configurations.

2 Preliminaries

This section specifies PMs, which combine lifting and factor graphs, first introduced by Poole [9], and GMs, which combine factor graphs and context-specific independences, first introduced by Minka and Winn [7].

2.1 Parameterised Probabilistic Models

PMs combine first-order logic with probabilistic models, representing first-order constructs using logical variables (logvars) as parameters. For illustrative purposes, we use an example of an epidemic. In the example, we model an epidemic as a randvar. Further, we model persons being sick as a PRV by parameterising a randvar for sick with a logvar for persons. In the larger scheme of things, all persons are influenced in the same way when faced with an epidemic and thus are, without additional evidence, indistinguishable.

Definition 1. *Let $\mathbf{R}$ be a set of randvar names, $\mathbf{L}$ a set of logvar names, $\varPhi$ a set of factor names, and $\mathbf{D}$ a set of constants. All sets are finite. Each logvar L has a domain $\mathcal{D}(L) \subseteq \mathbf{D}$. A* constraint *is a tuple $(\mathcal{X}, C_{\mathbf{X}})$ of a sequence of logvars $\mathcal{X} = (X_1, \dots, X_n)$ and a set $C_{\mathcal{X}} \subseteq \times_{i=1}^{n} \mathcal{D}(X_i)$. The symbol $\top$ for C marks that no restrictions apply, i.e., $C_{\mathcal{X}} = \times_{i=1}^{n} \mathcal{D}(X_i)$. A* PRV *$R(L_1, \dots, L_n), n \geq 0$ is a construct of a randvar $R \in \mathbf{R}$ possibly combined with logvars $L_1, \dots, L_n \in \mathbf{L}$. If $n = 0$, the PRV is parameterless and forms a propositional randvar. The term $\mathcal{R}(A)$ denotes the possible values (range) of a PRV A. An* event *$A = a$ denotes the occurrence of PRV A with range value $a \in \mathcal{R}(A)$. We denote a* parfactor *g by $\phi(\mathcal{A})_{|C}$ with $\mathcal{A} = (A_1, \dots, A_n)$ a sequence of PRVs, $\phi : \times_{i=1}^{n} \mathcal{R}(A_i) \mapsto \mathbb{R}^+$ a function with name $\phi \in \varPhi$, and C a constraint on the logvars of $\mathcal{A}$. A PRV A or logvar L under constraint C is given by $A_{|C}$ or $L_{|C}$, respectively. We may omit $|\top$ in $A_{|\top}$, $L_{|\top}$, or $\phi(\mathcal{A})_{|\top}$. A* PM *$G$ is a set of parfactors $\{g^i\}_{i=1}^{n}$.*

The term $lv(P)$ refers to the logvars in P, which may be a PRV, a constraint, a parfactor, or a model. The term $gr(P)$ denotes the set of all instances of P w.r.t. given constraints. An instance is an instantiation (grounding) of P, substituting the logvars in P with a set of constants from given constraints. If P is a constraint, $gr(P)$ refers to the second component $C_{\mathbf{X}}$. Given a parfactor $\phi(\mathcal{A})_{|C}$, ϕ is identical for the propositional randvars in $gr(\mathcal{A}_{|C})$.

Given $\mathbf{R} = \{Sick, Epid, Travel, Treat, Nat, Man\}$ and $\mathbf{L} = \{X, P, D, W\}$, $\mathcal{D}(X) = \{x_1, x_2, x_3\}$, $\mathcal{D}(P) = \{p_1, p_2\}$, $\mathcal{D}(D) = \{d_1, d_2\}$, and $\mathcal{D}(W) = \{w_1, w_2\}$, we can build a boolean PRV $Sick(X)$. With $C = ((X), \{(x_1), (x_2)\})$, $gr(Sick(X)_{|C}) = \{Sick(x_1), Sick(x_2)\}$. The set of $gr(Sick(X)_{|\top})$ also contains $Sick(x_3)$. Adding boolean PRVs $Epid$, $Travel(X)$, $Treat(X, P)$, $Nat(D)$, and $Man(W)$, we build a PM $G_{ex} = \{g_i\}_{i=0}^{2}$, with

- $g_0 = \phi_0(Epid, Sick(X), Treat(X, P))_{|\top}$,
- $g_1 = \phi_1(Epid, Sick(X), Travel(X))_{|\top}$, and
- $g_2 = \phi_2(Epid, Nat(D), Man(W))_{|\top}$.

Parfactors g_0, g_1, and g_2 have eight input-output pairs (omitted). Constraints are $\top$. Figure 1 depicts G_{ex} as a parfactor graph.

The semantics of a model is given by grounding and building a full joint distribution. In general, a query asks for a probability distribution of a randvar using a model's full joint distribution and given fixed events as evidence.

Definition 2. *With Z as normalising constant, a model G represents the full joint distribution $P_G = \frac{1}{Z} \prod_{f \in gr(G)} f$. The term $P(Q|\mathbf{E})$ denotes a* query *in G with Q a grounded PRV and $\mathbf{E}$ a set of events. Answering $P(Q|\mathbf{E})$ requires eliminating all randvars in G not occurring in $P(Q|\mathbf{E})$.*

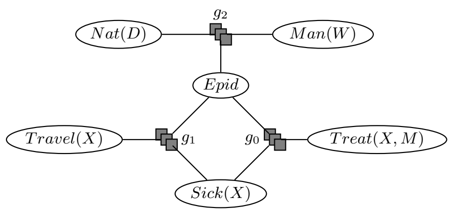

Fig. 1. Parfactor graph for G_{ex}

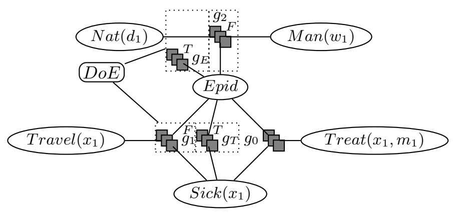

Fig. 2. Gates representation of G_{ex} for x_1

PMs allow for modelling relational aspects between objects including recurring patterns in these relations. Next, we recap GMs, which allow for explicitly modelling context-specific independence, i.e., switching, in propositional models.

2.2 Gate Models

GMs allow for representing context-specific independence [7]. A factor can be gated, meaning that using a selector the factor can be turned on or off, representing context-specific independence. Gates allows for modelling, e.g., external actions that change the state of the model or cutting off model parts depending on value of information.

To illustrate the impact of gates, Fig. 2 shows a GM representation of G_{ex} for x_1. Compared to G_{ex}, the GM has two gates (dashed boxes), one gate for g_E and g_T and one gate for g_1 and g_2, both with selector DoE. Both gates depend on the same selector DoE. Thereby, they are mutually exclusive, meaning when one gate is on, the other is off. We highlight two purposes of gates. The first purpose is *switching*. Assume only the gate for g_2 exists and we are interested in the marginal distribution of $Sick(x_1)$. The gate allows for turning off the connection to causes of an epidemic. Given observations that many people are sick, we might not care about the cause of an epidemic and cut off the cause part to not add noise or employ unnecessary computation time. But, in case the observation itself is uncertain or noisy, the cause part provides additional support, which enlarges the model and adds computations.

The second purpose is *intervention*, which uses both gates. An intervention on a randvar A, i.e., $do(A = a)$ in the *do* calculus [8], changes a model structure by eliminating the parent edges of A and setting A to a. The gates in Fig. 2 model an intervention on $Epid$, e.g., $do(Epid = true)$. The original "parent" of $Epid$ is g_2, its connection is removed upon intervention. Thus, the selector DoE is introduced, which turns off g_2 if $DoE = true$. Additionally, $Epid$ needs to be set to $true$. Setting $DoE = true$ enables g_E, which encodes the intervention value, i.e., $g_E = \phi(Epid)$ maps $true$ to 1 and $false$ to 0. Further, we might know that in case an epidemic is occurring, a travel ban will be in place. Thus, upon $DoE = true$, we also turn off g_1 and instead turn on g_T to perform inference on a smaller model, leading to fewer computations.

Additionally, GMs permit reasoning about value of information: If interested in $P(Sick(x_1))$, information about $Nat(d_1)$ has a value if and only if knowing $Nat(d_1)$ changes the marginal of $Sick(x_1)$. Thus, one could also consider setting selectors based on results of marginal distribution queries.

Next, we present switched inference on PGM as an instance of switched probabilistic relational models, specifying SLJT as an exact inference algorithm.

3 Switched Inference

We propose PGMs, leveraging lifting in GMs. Then, we show how LVE can answer queries on PGMs and adapt LJT to handle gates.

3.1 Parameterised Gate Models

Minka and Winn [7] introduce GMs for factor graphs which do not model the object/relation aspect that PMs model with logvars. Thus, we extend gates to contain not only factors but parfactors. A PM that then contains gated parfactors constitutes a PGM. Before looking at an example, we formally define PGMs including gates and introduce their semantics.

Definition 3. *A* gate *is denoted by* $(\prod_i g_i)^{\delta(s=key)}$*,* s *is the selector and* g_i *are the parfactors contained in the gate. A gate is turned off or on by raising the factors to the power of* 0 *or* 1 *respectively, which is indicated by* $\delta(s = key)$*, which is* 1 *if* s *has the value key and* 0 *otherwise. A* PGM M *consists of non-gated parfactors* g_k *and gated parfactors* g_i *with selectors* $\mathbf{S}$*. An assignment to all selectors* $\mathbf{S}$ *is called a* configuration $\{S = s\}_{S\in\mathbf{S}}$*. Given a configuration* $\mathbf{s}$*, the semantics of* M *is given by grounding and building a full joint distribution*

$$P_M = \frac{1}{Z}\prod_j (\prod_i \prod_{f\in gr(g_i)} f)^{\delta(s_j=key)} \prod_k \prod_{f\in gr(g_k)} f,$$

where Z *is the normalising constant,* j *indexes gates,* i *indexes the parfactors in* j*, and* $s_j \in \mathbf{s}$ *is the assignment to selector* S_j *for* j*. Given a query term* Q*, a set of events* $\mathbf{E}$*, and a configuration* $\mathbf{s}$*, the term* $P(Q|\mathbf{E},\mathbf{s})$ *denotes a* query *in* M*.*

Figure 3 shows a representation of a PGM based on G_{ex}. The parfactors g_1, g_2, g_E, and g_T are gated by the selector DoE, i.e.,

$$g_E^{\delta(DoE=T)}, g_T^{\delta(DoE=T)}, g_1^{\delta(DoE=F)}, g_2^{\delta(DoE=F)}.$$

The PGM works as described for the GM w.r.t. x_1. The two gates model an intervention of $Epid = true$.

To obtain a PGM, various approaches are possible, e.g., (i) directly specify a PGM, (ii) learn a PGM from data, or (iii) start from a GM and use, e.g., a colouring mechanism [1] to lift the GM. Next, we investigate exact algorithms for query answering in PGMs, for which we present LVE for single queries as well as SLJT for multiple queries.

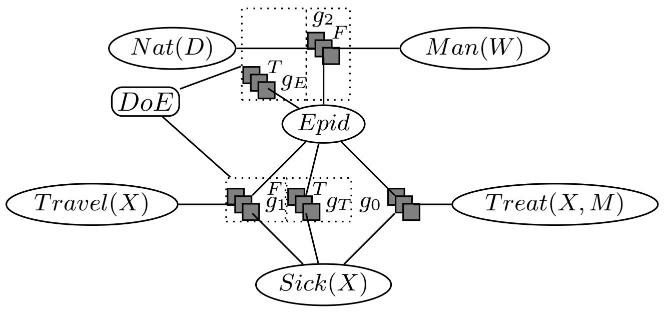

Fig. 3. Graphical representation of the PGM of G_{ex}

3.2 LVE for Query Answering

Based on the semantics, we need to define a way to answer queries for PGMs. Inference algorithms such as expectation propagation, variational message passing, and Gibbs sampling already work with GMs [14]. One well-studied inference algorithm for PMs is LVE, which performs computations in a lifted way, i.e., computes marginals by summing out a representative as in VE and then factoring in isomorphic instances. Here, we show that LVE (and as such VE) can be used for inference on PGMs (or GMs).

Proposition 1. *Given a query term Q and a configuration* $\mathbf{s}$*, VE computes* $P(Q, \mathbf{s})$ *in a GM M.*

Proof sketch. Applying a configuration $\mathbf{s}$ to a GM M leads to a plain factor graph G, which represents a full joint distribution P_G. VE is a correct algorithm to answer a query $P(Q)$ in G [15]. Given P_G, VE sums over all randvars, which are not query terms, and obtains the marginal distribution for Q, i.e., $P(Q) = \sum_v P_M$, where v are the range values of the non-query terms $\mathbf{A}$, i.e., $v \in \mathcal{R}(\mathbf{A})$. Given the factorisation in G and complying with rules of precedence and distributivity, VE computes $P(Q)$ efficiently by factoring out factors. Thereby, to compute $P(Q)$, VE avoids building the full joint distribution.

Proposition 2. *Given a query term Q and a configuration* $\mathbf{s}$*, LVE computes* $P(Q, \mathbf{s})$ *in a PGM M.*

Proof sketch. Setting a configuration $\mathbf{s}$ in a PGM M leads to a plain PM G, in which LVE computes a correct answer to a query $P(Q)$ by applying correct LVE operators to G, eliminating non-query terms [13]. The result is equivalent to one computed in $gr(G)$ with VE [13].

Given another query, LVE starts with the original input model, evidence, and configuration. Thus, we present SLJT incorporating gates into the cluster structure of LJT for efficient multi-query answering.

3.3 Switched Lifted Junction Tree Algorithm

A configuration determines the parts of a PGM that make up the full joint distribution. If we were to cluster a model based on a configuration, we could efficiently handle gates that are switched on or off. At this point, we turn to LJT [3], which uses a cluster representation of a PM for efficiently answering *multiple queries*. In the following, we introduce SLJT and examine how SLJT leverages LJT by automatically handling the effects of any given configuration on a PGM.

Clusters: LJT builds a cluster representation of a PM called an FO jtree, whose nodes are *clusters*. Intuitively, a cluster is a set of PRVs that are directly connected by parfactors. Each cluster has the parfactors that connect its PRVs as a *local model* assigned. For SLJT, clusters are based on selectors and their assignments. Consider the FO jtree with four clusters in Fig. 4 derived from the example PGM. Cluster $\mathbf{C}_1$ contains $Epid, Sick(X), Treat(X, M)$, linked by g_0. Clusters $\mathbf{C}_2$, $\mathbf{C}_3$ and $\mathbf{C}_4$ are based on the selector DoE. $\mathbf{C}_2$ contains $Epid$, $Sick(X)$, based on $DoE = true$, with g_E and g_T assigned. $\mathbf{C}_3$ contains $Epid$, $Sick(X)$, $Travel(X)$, based on $DoE = false$, with g_1 assigned. $\mathbf{C}_4$ contains $Epid$, $Nat(D)$, $Man(W)$, based on $DoE = false$, with g_2 assigned. If $DoE(X) = true$, $\mathbf{C}_2$ is switched on. If $DoE(X) = false$, $\mathbf{C}_3$ and $\mathbf{C}_4$ are switched on. $\mathbf{C}_1$ does not have a selector associated, it can be thought of as always switched on.

Query Answering: To answer queries on an FO jtree, LJT performs some preprocessing using local models. A local model holds state descriptions about its cluster PRVs, which is not available at another cluster. During preprocessing, LJT makes all necessary state descriptions available for each node through messages. A message m from one cluster to a neighbour $\mathbf{C}_j$ transports state descriptions of its local model and messages from other neighbours to $\mathbf{C}_j$. LJT uses LVE to calculate m, passing on the shared PRVs as a query and the local model and respective messages as a model. Without considering the selectors in the FO jtree in Fig. 4, LJT passes messages from $\mathbf{C}_2$ and $\mathbf{C}_4$ to $\mathbf{C}_1$ and back. With selectors present, message calculation changes: If a cluster is switched on, LJT calculates a message based on a cluster's local model and messages from neighbours. If a cluster is switched off, LJT calculates a message based only on messages from neighbours. Given a configuration of $DoE = true$ in the FO jtree in Fig. 4, the messages from $\mathbf{C}_3$ and $\mathbf{C}_4$ are empty without the local models and no other incoming message. With $DoE = false$, the message from $\mathbf{C}_2$ is empty.

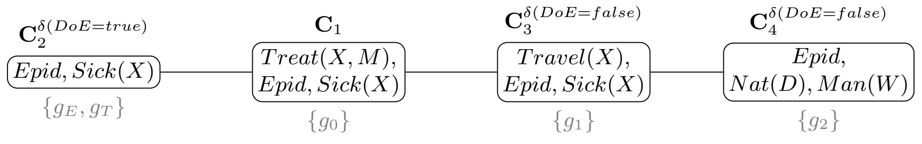
$\mathbf{C}_2^{\delta(DoE=true)}$
$Epid, Sick(X)$
$\{g_E, g_T\}$
$\mathbf{C}_1$
$Treat(X, M)$, $Epid, Sick(X)$
$\{g_0\}$
$\mathbf{C}_3^{\delta(DoE=false)}$
$Travel(X)$, $Epid, Sick(X)$
$\{g_1\}$
$\mathbf{C}_4^{\delta(DoE=false)}$
$Epid$, $Nat(D), Man(W)$
$\{g_2\}$

Fig. 4. An FO jtree for the PGM of G_{ex} in Fig. 3

After message passing, each cluster has all necessary state descriptions of the model under the current configuration available in its local model and received messages. To answer a query with a query term Q, LJT finds a cluster that contains Q and answers $P(Q)$ on the local model and messages with LVE.

The original FO jtree construction of LJT does not account for selectors as it is designed for PMs. Thus, we extend the FO jtree construction to handle gates.

FO Jtree Construction: Algorithm 1 outlines how to build an FO jtree of a PGM M. The guiding idea is to cluster M based on selector-key pairs. First, SLJT partitions M based on keys and builds an FO jtree J for each partition. An FO jtree is a cycle-free graph. The clusters are sets of PRVs from the input model and the arguments of each parfactor of the model appear in one cluster. A valid FO jtree also fulfils the *running intersection property* (RI), which says that a PRV appearing in two clusters must appear in all clusters on the path between them [6]. LJT constructs such an FO jtree for a given input model.

Now, SLJT has $|\mathbf{P}|$ valid FO jtrees with corresponding selector-key pairs assigned. To combine the FO jtrees into one valid FO jtree, SLJT takes a first FO jtree J, at random or an ungated FO jtree if available. Then, SLJT iteratively connects J to the remaining FO jtrees J_i by adding an edge from one cluster of J to a cluster of J_i. For the edge, SLJT chooses the two clusters with the largest overlap in PRVs. Combining two FO jtrees in such a fashion may violate RI. As keys may be mutually exclusive, RI only has to hold on valid paths. A valid path is a path between two clusters that are both switched on at the same time by any configuration. Therefore, SLJT extends clusters with PRVs until RI holds again on valid paths. After connecting all remaining FO jtrees to J, SLJT returns J.

To construct an FO jtree for the PGM G_{ex} in Fig. 3, SLJT first groups the parfactors. Here, each parfactor gets assigned its own group, as none of them share the same selector-key pair. Then, SLJT builds an FO jtree for each group. In this case, each FO jtree consists of one cluster, i.e., one FO jtree consisting of $\mathbf{C}_1$, one of $\mathbf{C}_2$, and one of $\mathbf{C}_3$ and $\mathbf{C}_4$. $\mathbf{C}_1$ is not gated and therefore selected as a starting point. Now, SLJT selects either $\mathbf{C}_2$ or $\mathbf{C}_3$ and $\mathbf{C}_4$ at random, e.g.,

Algorithm 1 FO jtree Construction

```
function SFOJT(PGM $M$)
    Let $\mathbf{P}$ be a partitioning of $M$ based on keys
    for each partition $P_i \in \mathbf{P}$ do
        Build FO jtree $J_i$ of $P_i$ and add to $\mathbf{F}$
    Take an FO jtree $J$ out from $\mathbf{F}$    ▷ Choose $J$ s.t. $P$ without a key or at random
    while $\mathbf{F}$ not empty do
        Take an FO jtree $J_i$ out from $\mathbf{F}$
        Connect $J_i$ to $J$                ▷ Edge between clusters sharing most PRVs
        while RI does not hold on valid paths do
            Extend clusters with PRVs
    return $J$
```

$\mathbf{C}_2$. SLJT connects $\mathbf{C}_1$ and $\mathbf{C}_2$. With only two clusters, RI still holds. Lastly, $\mathbf{C}_3$ and $\mathbf{C}_4$ are added to the FO jtree. As $\mathbf{C}_3$ overlaps with both $\mathbf{C}_1$ and $\mathbf{C}_2$ with $Epid$ and $Sick(X)$, SLJT chooses one at random, e.g., $\mathbf{C}_1$. Adding an edge between $\mathbf{C}_1$ and $\mathbf{C}_3$ leads to the FO jtree depicted in Fig. 4. In the resulting FO jtree, RI still holds on all paths and all paths are valid paths.

Theorem 1. *The FO jtree construction of SLJT is sound.*

Proof sketch. The initial FO jtrees built are valid. Their clusters contain PRVs from the input model and the arguments of each parfactor appear in some cluster. By combining one node of a cycle-free graph with exactly one node from another cycle-free graph the result is again a cycle-free graph. Adding edges may only violate RI, which SLJT systematically restores by extending clusters with PRVs. Thus, Alg. 1 produces a valid FO jtree.

Algorithm Description: SLJT takes a PGM M, a configuration $\mathbf{s}$, evidence $\mathbf{E}$, and a set of queries $\mathbf{Q}$. Algorithm 2 shows an outline of SLJT. SLJT constructs an FO jtree J as in Alg. 1 and then switches clusters in J on and off based on $\mathbf{s}$, followed by entering $\mathbf{E}$ into the clusters: At each cluster that contains the evidence randvars, the local model absorbs $\mathbf{E}$ in a lifted way (cf. Taghipour et al. [13]). Then, SLJT passes messages as described above. Finally, SLJT answers the queries in $\mathbf{Q}$ or starts processing incoming queries online.

Theorem 2. *SLJT is sound, i.e, calculates correct answers to queries on a PGM M and a configuration $\mathbf{s}$.*

Proof sketch. SLJT constructs a valid FO jtree based on Theorem 1, which allows for local computations for messages and queries [12]. To answer queries correctly, SLJT has to distribute state descriptions of local models through the FO jtree. Therefore, SLJT uses the massage passing scheme of LJT, which coincides with the scheme by Shafer and Shenoy, which they show to be sound [11]. Additionally, SLJT includes the local model of the current cluster only if the selector of the cluster is on. In case the selector is off, the cluster only uses the information of received messages, if there are any, to calculate the outgoing message. Thus, after a message pass, each cluster holds all necessary state descriptions under a given configuration and can answer queries about its PRVs. Hence, as LJT is sound and SLJT calculates the same answers LJT would on an FO jtree with only the parfactors, which are turned on, SLJT is sound.

Algorithm 2 Switched Lifted Junction Tree Algorithm

```
procedure SLJT(PGM $M$, configuration $\mathbf{s}$, evidence $\mathbf{E}$, queries $\mathbf{Q}$)
    FO jtree $J \leftarrow$ SFOJT($M$)
    Enter evidence $\mathbf{E}$ into $J$
    Pass messages on $J$
    for each query $Q \in \mathbf{Q}$ do
        Answer $Q$ on a cluster in $J$
```

SLJT allows for building an FO jtree for a PGM and then reuse the FO jtree for multiple queries and configurations. Next, we evaluate the performance gain by using the context-specific independences.

4 Evaluation

To evaluate SLJT, we use a variation of G_{ex} with 3 selectors. We compare SLJT against implicitly specifying the context-specific independences in parfactors and against specifying a model for each configuration. 3 selectors result in 8 configurations, leading to 8 small models. Thus, we compare SLJT with a PGM against LJT with a model containing an implicit encoding of the switches as well as LJT with 8 models corresponding to configurations. For the evaluation, we compare the runtimes w.r.t. message passing to prepare an FO jtree for query answering as well as the runtime for answering two queries, namely $P(Epid)$ and $P(Sick(x_1))$. Additionally, we evaluate the runtimes for $|\mathcal{D}(X)| \in \{10, 100, 1000\}$. One claim investigated in this evaluation is that it is advantageous to use explicit context-specific independences also in the lifted case. Another claim is that SLJT requires about the same runtime for query answering as LJT does on the models corresponding to configurations.

Figure 5 shows the runtimes for message passing in ms and Fig. 6 shows the runtimes of each of the two queries. The runtimes are the average of 10 runs. In both figures, the x-axis shows different configurations. Thus, for $x = A$ the runtimes for the first configuration are shown, for $x = B$ the runtimes for the second configuration are shown, and so on.

In Fig. 5, we can see that message passing on the large model with an encoding of the switches in parfactors takes the longest. The runtimes are about the same for all configuration as the configuration is passed to LJT as evidence leading to absorbing the variables used to encode the switching. Hence, the variables used for encoding switching behaviour can be thought of as eliminated after evidence entering. Nonetheless, LJT still needs to perform a message pass on a rather large model. Therefore, the runtimes for this model are the upper bound in our evaluation. For the small models, we can see that runtimes for message passing increase with the different configurations and that the runtimes are bounded by the implicit encoding. The increase is incidental due to the sort-

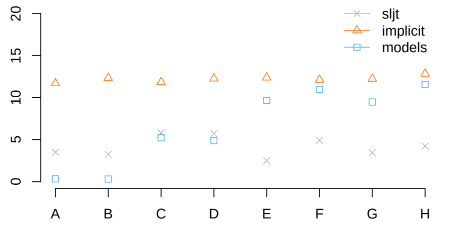

Fig. 5. Message passing runtimes [ms] for $|\mathcal{D}(X) = 100|$, x-axis: configurations

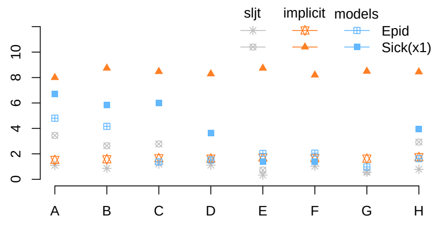

Fig. 6. Query answering runtimes [ms] for $|\mathcal{D}(X) = 100|$, x-axis: configurations

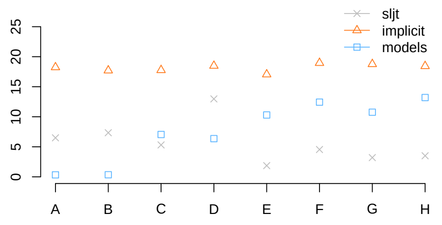

Fig. 7. Message passing runtimes [ms] for $|\mathcal{D}(X) = 1000|$, x-axis: configurations

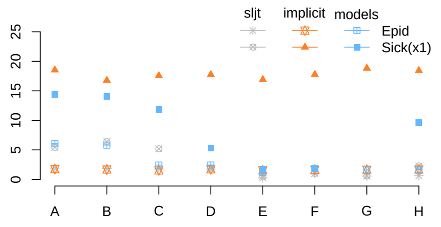

Fig. 8. Query answering runtimes [ms] for $|\mathcal{D}(X) = 1000|$, x-axis: configurations

ing of the configurations. Further, we can see that for configuration A and B, the model results in an FO jtree with one parcluster as LJT does not spend any time on message passing, but relatively long on query answering as can be seen in Fig. 6. For SLJT, we can see that message passing only slightly variates between the different configurations. SLJT always needs to compute the same number of messages, as the FO jtree always remains the same. However, which parcluster and thereby which parfactors are turned on and off depends on the configuration leading to slight variations in the runtimes.

In Fig. 6, we can see that answering the query about *Epid* is always faster compared to $Sick(x_1)$ because $Sick(x_1)$, x_1 needs to be split from X. Implicitly encoding the switching behaviour in the model leads to largest runtimes for answering $Sick(x_1)$. Regarding both queries, implicitly encoding the behaviour leads to runtimes very close to each other over different configurations as described above. Regarding the models based on configurations, LJT saves effort during query answering with increasing effort during message passing. SLJT is the fastest approach for both queries. SLJT always answers the queries on an FO jtree with many rather small parclusters. Having small parclusters is really advantageous for query answering and explains why the runtimes of SLJT are often even slightly below LJT for the constructed small model corresponding to the configuration. Overall, we can see that using context-specific independences has a huge impact on runtimes.

Figures 7 and 8 shows runtimes for $|\mathcal{D}(X)| = 1000$, the programs exhibiting the same behaviour compared to each other as with $|\mathcal{D}(X)| = 100$. The setting $|\mathcal{D}(X)| = 10$ also shows the same behaviour (omitted here). In summary, answering queries on an FO jtree with small parclusters is advantageous. Additionally, specifying a model for each configuration is cumbersome, always incurring an overhead for constructing an FO jtree, a step which we did not evaluate here. Overall, compared to the other two methods, SLJT efficiently uses context-specific independence to significantly speed up inference.

5 Conclusion

To make inference more manageable, we investigate multiple queries in switched probabilistic relational models, which explicitly handle context-specific independence. By leveraging lifting principles for GMs, which allows for representing context-specific independence using gates, and then extending LJT to efficiently handle switching behaviour, SLJT allows for efficient answering of multiple queries in switched probabilistic relational models. Empirical results show that using context-specific independence speeds up lifted inference significantly.

Future work focusses on including causal inference [14] and counterfactual reasoning. Further, we look into decision support as gates with context-specific independences seems to be an ideal formalism to model different actions.

References

1. Ahmadi, B., Kersting, K., Mladenov, M., Natarajan, S.: Exploiting Symmetries for Scaling Loopy Belief Propagation and Relational Training. Machine learning **92**(1), 91–132 (2013)
2. Boutilier, C., Friedman, N., Goldszmidt, M., Koller, D.: Context-specific independence in Bayesian networks. In: Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence. pp. 115–123. Morgan Kaufmann Publishers Inc. (1996)
3. Braun, T., Möller, R.: Lifted Junction Tree Algorithm. In: Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz). pp. 30–42. Springer (2016)
4. Cooper, G.F.: The computational complexity of probabilistic inference using Bayesian belief networks. Artificial intelligence **42**(2-3), 393–405 (1990)
5. Gogate, V., Domingos, P.M.: Probabilistic Theorem Proving. In: UAI 2011, Proceedings of the Twenty-Seventh Conference on Uncertainty in Artificial Intelligence, Barcelona, Spain, July 14-17, 2011. pp. 256–265. AUAI Press (2011)
6. Lauritzen, S.L., Spiegelhalter, D.J.: Local Computations with Probabilities on Graphical Structures and their Application to Expert Systems. Journal of the Royal Statistical Society. Series B (Methodological) **50**(2), 157–224 (1988)
7. Minka, T., Winn, J.: Gates. In: Advances in Neural Information Processing Systems. pp. 1073–1080 (2009)
8. Pearl, J.: Causality. Cambridge university press (2009)
9. Poole, D.: First-order probabilistic inference. In: Proceedings of IJCAI. vol. 3, pp. 985–991 (2003)
10. Richardson, M., Domingos, P.: Markov Logic Networks. Machine learning **62**(1), 107–136 (2006)
11. Shafer, G.R., Shenoy, P.P.: Probability Propagation. Annals of Mathematics and Artificial Intelligence **2(1)**, 327–351 (1990)
12. Shenoy, P.P., Shafer, G.R.: Axioms for Probability and Belief-Function Propagation. Uncertainty in Artificial Intelligence 4 **9**, 169–198 (1990)
13. Taghipour, N., Fierens, D., Davis, J., Blockeel, H.: Lifted Variable Elimination: Decoupling the Operators from the Constraint Language. Journal of Artificial Intelligence Research **47(1)**, 393–439 (2013)
14. Winn, J.: Causality with gates. In: Artificial Intelligence and Statistics. pp. 1314–1322 (2012)
15. Zhang, N.L., Poole, D.: A Simple Approach to Bayesian Network Computations. In: Proceedings of the 10th Canadian Conference on Artificial Intelligence. pp. 171–178. Springer (1994)