Efficient Detection of Commutative Factors in Factor Graphs

Malte Luttermann MALTE.LUTTERMANN@DFKI.DE
German Research Center for Artificial Intelligence (DFKI), Lübeck, Germany
Johann Machemer JOHANN.MACHEMER@STUDENT.UNI-LUEBECK.DE
Institute for Software Engineering and Programming Languages, University of Lübeck, Germany
Marcel Gehrke MARCEL.GEHRKE@UNI-HAMBURG.DE
Institute for Humanities-Centered Artificial Intelligence, University of Hamburg, Germany

Editors: J.H.P. Kwisthout & S. Renooij

Abstract

Lifted probabilistic inference exploits symmetries in probabilistic graphical models to allow for tractable probabilistic inference with respect to domain sizes. To exploit symmetries in, e.g., factor graphs, it is crucial to identify commutative factors, i.e., factors having symmetries within themselves due to their arguments being exchangeable. The current state of the art to check whether a factor is commutative with respect to a subset of its arguments iterates over all possible subsets of the factor's arguments, i.e., $O(2^n)$ iterations for a factor with n arguments in the worst case. In this paper, we efficiently solve the problem of detecting commutative factors in a factor graph. In particular, we introduce the *detection of commutative factors (DECOR)* algorithm, which allows us to drastically reduce the computational effort for checking whether a factor is commutative in practice. We prove that DECOR efficiently identifies restrictions to drastically reduce the number of required iterations and validate the efficiency of DECOR in our empirical evaluation.

Keywords: probabilistic graphical models; factor graphs; lifted inference.

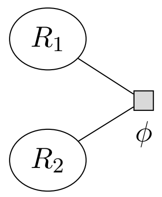
1. Introduction

Probabilistic graphical models provide a well-founded formalism to reason under uncertainty and compactly encode a full joint probability distribution as a product of factors. A fundamental task using probabilistic graphical models is to perform probabilistic inference, that is, to compute marginal distributions of random variables (randvars) given observations for other randvars. In general, however, probabilistic inference scales exponentially with the number of randvars in a propositional probabilistic model such as a Bayesian network, a Markov network, or a factor graph (FG) in the worst case. To allow for tractable probabilistic inference (e.g., probabilistic inference requiring polynomial time) with respect to domain sizes of logical variables, lifted inference algorithms exploit symmetries in a probabilistic graphical model by using a representative of indistinguishable individuals for computations (Niepert and Van den Broeck, 2014). Performing lifted inference, however, requires a lifted representation of the probabilistic graphical model, which encodes equivalent semantics to the original propositional probabilistic model. In particular, it is necessary to detect symmetries in the propositional model to construct the lifted representation. Commutative factors, i.e., factors that map to the same potential value regardless of the order of a subset of their arguments, play a crucial role when detecting symmetries in an FG and constructing a lifted representation of the FG. To detect commutative factors, the current state-of-the-art algorithm iterates over all possible subsets of a factor. In this paper, we show that the search can be guided significantly more efficient and thereby we solve the problem of efficiently detecting commutative factors in an FG.

Previous work. In probabilistic inference, lifting exploits symmetries in a probabilistic model, allowing to carry out query answering more efficiently while maintaining exact answers (Niepert and Van den Broeck, 2014). First introduced by Poole (2003), parametric factor graphs (PFGs) provide a lifted representation by combining first-order logic with probabilistic models, which can be utilised by lifted variable elimination as a lifted inference algorithm operating on PFGs. After the introduction of lifted variable elimination (LVE) by Poole (2003), LVE has steadily been refined by many researchers (De Salvo Braz et al., 2005, 2006; Milch et al., 2008; Kisyński and Poole, 2009; Taghipour et al., 2013; Braun and Möller, 2018). To perform lifted probabilistic inference, the lifted representation (e.g., the PFG) has to be constructed first. Currently, the state-of-the-art algorithm to construct a PFG from a given FG is the advanced colour passing (ACP) algorithm (Luttermann et al., 2024a), which is a refinement of the colour passing (CP) algorithm (Kersting et al., 2009; Ahmadi et al., 2013) and is able to construct a PFG entailing equivalent semantics as the initially given FG. The CP algorithm builds on work by Singla and Domingos (2008) and incorporates a colour passing procedure to detect symmetries in a graph similar to the Weisfeiler-Leman algorithm (Weisfeiler and Leman, 1968), which is commonly used to test for graph isomorphism. ACP employs a slightly refined colour passing procedure and during the course of the algorithm, ACP checks for every factor in the given FG whether the factor is commutative with respect to a subset of its arguments to determine the outgoing messages of the factor for the colour passing procedure. The check for commutativity is implemented by iterating over all possible subsets of the factor's arguments, which results in $O(2^n)$ iterations for a factor with n arguments in the worst case.

Our contributions. In this paper, we efficiently solve the problem of detecting commutative factors in an FG, thereby speeding up the construction of an equivalent lifted representation for a given FG. More specifically, we show how so-called *buckets* can be used to restrict the possible candidates of subsets that have to be considered when checking whether a factor is commutative—that is, which of its arguments are exchangeable. The idea is that within these buckets, only a subset of potentials from the original factor has to be considered. Further, argument(s), which produce certain potential(s) can be identified. Thereby, the search space can be drastically reduced. We prove that using buckets, the run time complexity is drastically reduced for many practical settings, making the check for commutativity feasible in practice. Afterwards, we gather the theoretical insights and propose the detection of commutative factors (DECOR) algorithm, which applies the theoretical insights to efficiently detect commutative factors in an FG. In addition to our theoretical results, we validate the efficiency of DECOR in our empirical evaluation.

Structure of this paper. The remainder of this paper is structured as follows. We first introduce necessary background information and notations. In particular, we define FGs as well as the notion of commutative factors and provide background information on the ACP algorithm. Afterwards, we formally define the concept of a bucket and present our theoretical results, showing how buckets can be used to efficiently restrict possible candidates of subsets that have to be considered when searching for exchangeable arguments of a factor. We then introduce the DECOR algorithm, which applies the theoretical insights to efficiently detect commutative factors in an FG in practice. Before we conclude, we provide the results of our experimental evaluation, verifying the practical efficiency of DECOR.



R_1	R_2	$\phi(R_1, R_2)$
true	true	φ_1
true	false	φ_2
false	true	φ_2
false	false	φ_3

Figure 1: A toy example for an FG consisting of two Boolean randvars R_1 and R_2 as well as one factor $\phi(R_1, R_2)$. The input-output pairs of ϕ are given in the table on the right, where each assignment of ϕ's arguments is mapped to a potential with $\varphi_1, \ldots, \varphi_3 \in \mathbb{R}^+$.

2. Preliminaries

We first introduce FGs as undirected propositional probabilistic models and afterwards define commutative factors formally. An FG compactly encodes a full joint probability distribution between randvars, where the full joint probability distribution is represented as a product of factors (Frey et al., 1997; Kschischang et al., 2001).

Definition 1 (Factor Graph, Kschischang et al., 2001) *An* FG $G = (\boldsymbol{V}, \boldsymbol{E})$ *is an undirected bipartite graph consisting of a node set* $\boldsymbol{V} = \boldsymbol{R} \cup \boldsymbol{\Phi}$*, where* $\boldsymbol{R} = \{R_1, \ldots, R_n\}$ *is a set of variable nodes (randvars) and* $\boldsymbol{\Phi} = \{\phi_1, \ldots, \phi_m\}$ *is a set of factor nodes (functions), as well as a set of edges* $\boldsymbol{E} \subseteq \boldsymbol{R} \times \boldsymbol{\Phi}$*. The term* $\mathrm{range}(R_i)$ *denotes the possible values of a randvar* R_i*. There is an edge between a variable node* R_i *and a factor node* ϕ_j *in* $\boldsymbol{E}$ *if* R_i *appears in the argument list of* ϕ_j*. A factor is a function that maps its arguments to a positive real number, called potential. The semantics of* G *is given by*

$$P_G = \frac{1}{Z} \prod_{j=1}^{m} \phi_j(\mathcal{A}_j)$$

with Z *being the normalisation constant and* $\mathcal{A}_j$ *denoting the randvars connected to* ϕ_j*.*

Example 1 *Figure 1 displays a toy example for an FG consisting of two randvars* R_1 *and* R_2 *as well as one factor* $\phi(R_1, R_2)$*. Both* R_1 *and* R_2 *are Boolean, that is,* $\mathrm{range}(R_1) = \mathrm{range}(R_2) = \{\mathrm{true}, \mathrm{false}\}$*. The input-output pairs of* ϕ *are specified in the table on the right. Note that, in this specific example, it holds that* $\phi(\mathrm{true}, \mathrm{false}) = \phi(\mathrm{false}, \mathrm{true}) = \varphi_2$*.*

Lifted inference algorithms exploit symmetries in an FG by operating on lifted representations (e.g., PFGs), which consist of parameterised randvars and parametric factors, representing sets of randvars and factors, respectively (Poole, 2003). Symmetries in FGs are highly relevant in many real world domains such as an epidemic domain, where each person influences the probability of an epidemic in the same way—because the probability of having an epidemic depends on the number of sick people and not on individual people being sick. The probability for an epidemic is the same if there are three sick people and the remaining people in the universe are not sick, independent of whether *alice*, *bob*, and *eve* or *charlie*, *dave*, and *fred* are sick. Analogously, for movies the popularity of an actor influences the success of a movie in the same way for each actor being part of the movie.

To detect symmetries in an FG and obtain a lifted representation to speed up inference, the so-called ACP algorithm (Luttermann et al., 2024a) can be applied. The ACP algorithm builds on the colour passing algorithm (Kersting et al., 2009; Ahmadi et al., 2013) and transforms a given FG into a PFG entailing equivalent semantics as the initial FG. We provide a formal description of the ACP algorithm in Appendix A. During the course of the algorithm, ACP checks for commutative factors, which play a crucial role in symmetry detection and are defined as follows.

Definition 2 (Commutative Factor, Luttermann et al., 2024a) *Let $\phi(R_1, \dots, R_n)$ denote a factor. We say that ϕ is* commutative with respect to $\boldsymbol{S} \subseteq \{R_1, \dots, R_n\}$ *if for all events $r_1, \dots, r_n \in \times_{i=1}^{n} \mathrm{range}(R_i)$ it holds that $\phi(r_1, \dots, r_n) = \phi(r_{\pi(1)}, \dots, r_{\pi(n)})$ for all permutations π of $\{1, \dots, n\}$ with $\pi(i) = i$ for all $R_i \notin \boldsymbol{S}$. If ϕ is commutative with respect to $\boldsymbol{S}$, we say that all arguments in $\boldsymbol{S}$ are* commutative arguments *of ϕ.*

Example 2 *Consider again the factor $\phi(R_1, R_2)$ depicted in Fig. 1. Since it holds that $\phi(\mathrm{true}, \mathrm{false}) = \phi(\mathrm{false}, \mathrm{true}) = \varphi_2$, ϕ is commutative with respect to $\{R_1, R_2\}$.*

By definition, every factor is commutative with respect to any singleton subset of its arguments independent of the factor's potential mappings. Note that, in practice, we are only interested in factors that are commutative with respect to a subset $\boldsymbol{S}$ of their arguments where $|\boldsymbol{S}| > 1$ because our intention is to group indistinguishable arguments (and grouping a single element does not yield any benefit). Since all arguments in $\boldsymbol{S}$ are candidates to be grouped together, the task we solve in this paper is to compute a subset $\boldsymbol{S}$ of a factor's arguments of *maximum size* such that the factor is commutative with respect to $\boldsymbol{S}$. In many practical settings, factors are commutative with respect to a subset of their arguments, for example if individuals are indistinguishable and only the number of individuals having a certain property is of interest (as in an epidemic domain where the number of persons being sick determines the probability of an epidemic while it does not matter which specific persons are sick, and so on). We next show how commutative factors can efficiently be detected in an FG, which is crucial to exploit symmetries in the FG for lifted inference.

3. Efficient Detection of Commutative Factors Using Buckets

Detecting commutative factors is a fundamental part of lifted model construction. The current state of the art, however, applies a rather "naive" approach to compute a subset of commutative arguments of maximum size. In particular, computing a maximum sized subset of commutative arguments of a factor ϕ is currently implemented by iterating over all possible subsets of ϕ's arguments in order of descending size of the subsets. That is, for a factor $\phi(R_1, \dots, R_n)$, it is first checked whether ϕ is commutative with respect to all of its n arguments, then it is checked whether ϕ is commutative with respect to any subset consisting of $n - 1$ arguments, and so on. In the worst case, the algorithm needs $O(2^n)$ iterations to compute a maximum sized subset of commutative arguments. Even though we have to consider $O(2^n)$ subsets in the worst case, we are able to drastically prune the search space in many practical settings, as we show next. The idea is that we can partition the potential values a factor maps its arguments to into so-called *buckets*, which allow us to restrict the space of possible candidate subsets and thereby heavily reduce the number

R_1	R_2	R_3	$\phi(R_1, R_2, R_3)$	b
true	true	true	φ_1	$[3, 0]$
true	true	false	φ_2	$[2, 1]$
true	false	true	φ_2	$[2, 1]$
true	false	false	φ_3	$[1, 2]$
false	true	true	φ_4	$[2, 1]$
false	true	false	φ_5	$[1, 2]$
false	false	true	φ_5	$[1, 2]$
false	false	false	φ_6	$[0, 3]$

b	$\phi(b)$
$[3, 0]$	$\langle \varphi_1 \rangle$
$[2, 1]$	$\langle \varphi_2, \varphi_2, \varphi_4 \rangle$
$[1, 2]$	$\langle \varphi_3, \varphi_5, \varphi_5 \rangle$
$[0, 3]$	$\langle \varphi_6 \rangle$

Table 1: A factor $\phi(R_1, R_2, R_3)$ with three Boolean arguments and the corresponding mappings of potential values (left table) as well as the partition of potential values into buckets (right table). Here, ϕ is commutative with respect to $\boldsymbol{S} = \{R_2, R_3\}$.

of necessary iterations. A bucket counts the occurrences of specific range values in an assignment for a subset of a factor's arguments. Consequently, each bucket may contain multiple potential values and every potential value is part of exactly one bucket.

Definition 3 (Bucket, Luttermann et al., 2024b) *Let $\phi(R_1, \ldots, R_n)$ denote a factor and let $\boldsymbol{S} \subseteq \{R_1, \ldots, R_n\}$ denote a subset of ϕ's arguments such that* $\mathrm{range}(R_i) = \mathrm{range}(R_j)$ *holds for all $R_i, R_j \in \boldsymbol{S}$. Further, let $\mathcal{V}$ denote the range of the elements in $\boldsymbol{S}$ (identical for all $R_i \in \boldsymbol{S}$). Then, a* bucket *b entailed by $\boldsymbol{S}$ is a set of tuples $\{(v_i, n_i)\}_{i=1}^{|\mathcal{V}|}$, $v_i \in \mathcal{V}$, $n_i \in \mathbb{N}$, and $\sum_i n_i = |\boldsymbol{S}|$, such that n_i specifies the number of occurrences of value v_i in an assignment for all randvars in $\boldsymbol{S}$. A shorthand notation for $\{(v_i, n_i)\}_{i=1}^{|\mathcal{V}|}$ is $[n_1, \ldots, n_{|\mathcal{V}|}]$. In abuse of notation, we denote by $\phi(b)$ the multiset of potentials the assignments represented by b are mapped to by ϕ. The set of all buckets entailed by ϕ is denoted as $\mathcal{B}(\phi)$.*

Example 3 *Consider the factor $\phi(R_1, R_2, R_3)$ depicted in Table 1 and let $\boldsymbol{S} = \{R_1, R_2, R_3\}$ with* $\mathrm{range}(R_1) = \mathrm{range}(R_2) = \mathrm{range}(R_3) = \{\mathrm{true}, \mathrm{false}\}$. *Then, $\boldsymbol{S}$ entails the four buckets* $\{(\mathrm{true}, 3), (\mathrm{false}, 0)\}$, $\{(\mathrm{true}, 2), (\mathrm{false}, 1)\}$, $\{(\mathrm{true}, 1), (\mathrm{false}, 2)\}$, *and* $\{(\mathrm{true}, 0), (\mathrm{false}, 3)\}$ *(or* $[3, 0]$, $[2, 1]$, $[1, 2]$, *and* $[0, 3]$, *respectively, in shorthand notation). Following the mappings given in the table from Table 1, it holds that $\phi([3, 0]) = \langle \varphi_1 \rangle$, $\phi([2, 1]) = \langle \varphi_2, \varphi_2, \varphi_4 \rangle$, $\phi([1, 2]) = \langle \varphi_3, \varphi_5, \varphi_5 \rangle$, and $\phi([0, 3]) = \langle \varphi_6 \rangle$.*

Buckets allow us to restrict the candidates of arguments that are possibly commutative (exchangeable) and hence might be grouped. The idea is that commutative arguments can be replaced by a bucket that counts over their range values instead of listing each of the arguments separately. In particular, each combination of bucket for the commutative arguments and fixed values for the remaining arguments must be mapped to the same potential value, as illustrated in the upcoming example.

Example 4 *Take a look at Table 2, which contains a compressed representation of the factor ϕ from Table 1. Originally, ϕ maps each combination of bucket for R_2 and R_3 and fixed values for R_1 to the same potential value, e.g., R_1 =* true, *R_2 =* true, *R_3 =* false

R_1	$\#_X[R(X)]$	$\phi(R_1, \#_X[R(X)])$
true	$[2, 0]$	φ_1
true	$[1, 1]$	φ_2
true	$[0, 2]$	φ_3
false	$[2, 0]$	φ_4
false	$[1, 1]$	φ_5
false	$[0, 2]$	φ_6

Table 2: A compressed representation of the factor ϕ given in Table 1. The randvars R_2 and R_3 are now replaced by a so-called counting randvar $\#_X[R(X)]$, which uses buckets to represent R_2 and R_3 simultaneously. Note that the semantics remains unchanged while the size of the table got reduced due to grouping commutative (exchangeable) randvars.

and $R_1 = \text{true}$*,* $R_2 = \text{false}$*,* $R_3 = \text{true}$ *(that is, the combination of value* true *for* R_1 *and bucket* $[1, 1]$ *for* R_2 *and* R_3*) are mapped to the same potential value* φ_2*. Consequently, these two assignments can be represented by a single assignment* $(\text{true}, [1, 1])$ *in the compressed representation shown in Table 2. In general, it is possible to compress all commutative arguments in a factor by replacing them by a so-called counting randvar, which uses buckets as a representation. In this particular example,* ϕ *is commutative with respect to* $\boldsymbol{S} = \{R_2, R_3\}$ *and thus, we are able to group* R_2 *and* R_3 *together.*

Grouping commutative arguments does not alter the semantics of the factor and at the same time reduces the size of the table that has to be stored to encode the input-output mappings of the factor as well as the inference time. To obtain the maximum possible compression, we therefore aim to compute a *maximum sized* subset of commutative arguments.

A crucial observation to efficiently compute a maximum sized subset of commutative arguments is that the buckets of the initially given factor must contain duplicate potential values as a necessary condition for arguments to be able to be commutative. In particular, groups of identical potential values within a bucket directly restrict the possible candidates of commutative arguments, as we show next.

Theorem 4 *Let* $\phi(R_1, \ldots, R_n)$ *be a factor and let* $b \in \mathcal{B}(\phi)$ *with* $|\phi(b)| > 1$ *be a bucket entailed by* ϕ*. Further, let* $G = \{\varphi_1, \ldots, \varphi_\ell\}$ *with* $|G| > 2$ *denote an arbitrary maximal set of identical potential values in* $\phi(b)$ *and let* $(r_1^1, \ldots, r_n^1), \ldots, (r_1^\ell, \ldots, r_n^\ell)$ *denote all assignments corresponding to the potential values in* G*. Then, a subset* $\boldsymbol{S} \subseteq \{R_1, \ldots, R_n\}$ *with* $|\boldsymbol{S}| > 2$ *such that* ϕ *is commutative with respect to* $\boldsymbol{S}$ *is obtained by computing the element-wise intersection* $(r_1^1, \ldots, r_n^1) \cap (r_1^\ell, \ldots, r_n^\ell)$ *and adding all arguments* R_i *to* $\boldsymbol{S}$ *for which it holds that* $\{r_i^1\} \cap \ldots \cap \{r_i^\ell\} = \emptyset$*. Here, the element-wise intersection of two assignments* $(x_1, \ldots, x_\ell)$ *and* $(y_1, \ldots, y_\ell)$ *contains the value* x_i *at position* i *if* $x_i = y_i$*, otherwise position* i *equals* $\emptyset$*.*

Proof It holds that $\phi(r_1^1, \ldots, r_n^1) = \varphi_1, \ldots, \phi(r_1^\ell, \ldots, r_n^\ell) = \varphi_\ell$ and $\varphi_1 = \ldots = \varphi_\ell$. Moreover, let $\boldsymbol{S} \subseteq \{R_1, \ldots, R_n\}$ denote any subset of commutative arguments such that $|\boldsymbol{S}| > 2$. Recall that according to Def. 2, for all events $r_1, \ldots, r_n \in \times_{i=1}^n \text{range}(R_i)$ it holds that $\phi(r_1, \ldots, r_n) = \phi(r_{\pi(1)}, \ldots, r_{\pi(n)})$ for all permutations π of $\{1, \ldots, n\}$ with $\pi(i) = i$ for all $R_i \notin \boldsymbol{S}$. Thus, the positions of all arguments which are not in $\boldsymbol{S}$ are fixed in all permutations. Consequently, the element-wise intersection of the assignments yields non-empty sets for all positions belonging to arguments not in $\boldsymbol{S}$ because these assignments contain identical assigned values at those positions. At the same time, as we consider only buckets with at least two elements, all assignments contain at least two distinct assigned values because all assignments that contain only a single value are those that map to a bucket containing a single element. We further consider all assignments $(r_{\pi(1)}, \ldots, r_{\pi(n)})$ for all permutations π of $\{1, \ldots, n\}$ and therefore, there are at least two different assigned values for all positions belonging to arguments in $\boldsymbol{S}$, yielding an empty set when computing the element-wise intersection of the assignments for a position of any argument in $\boldsymbol{S}$. ■

Theorem 4 tells us that candidate subsets of commutative arguments can be found by first identifying groups of identical potential values within a bucket and then computing the element-wise intersection of the assignments corresponding to these potential values. The following example illustrates how Thm. 4 can be applied to identify commutative arguments.

Example 5 *Consider again the factor $\phi(R_1, R_2, R_3)$ depicted in Table 1. For the sake of the example, let us take a look at the bucket $[2, 1]$. We have two groups of identical values, namely $G_1 = \{\varphi_2, \varphi_2\}$ and $G_2 = \{\varphi_4\}$. Since G_2 contains only a single potential value, there are no candidates of commutative arguments induced by G_2. However, G_1 contains the potential value φ_2 two times and the corresponding assignments are $(\mathrm{true}, \mathrm{true}, \mathrm{false})$ and $(\mathrm{true}, \mathrm{false}, \mathrm{true})$. The element-wise intersection of those assignments is then given by $(\mathrm{true}, \mathrm{true}, \mathrm{false}) \cap (\mathrm{true}, \mathrm{false}, \mathrm{true}) = (\mathrm{true}, \emptyset, \emptyset)$. As the element-wise intersection is empty at positions two and three, the set $\{R_2, R_3\}$ is a possible subset of commutative arguments.*

By looking for *maximal* sets G of identical potential values, we ensure that we actually find maximum sized candidate subsets of commutative arguments. Moreover, the number of positions having an empty element-wise intersection directly corresponds to the size of every candidate subset $\boldsymbol{S}$, that is, the number of positions having an empty element-wise intersection is equal to $|\boldsymbol{S}|$. As all permutations of these $|\boldsymbol{S}|$ positions are mapped to the same potential value, we know that if there is a subset of commutative arguments of size $|\boldsymbol{S}|$, there must be at least $|\boldsymbol{S}|$ duplicate potential values in every bucket of size at least two.

Corollary 5 *Let $\phi(R_1, \ldots, R_n)$ be a factor and let $\boldsymbol{S} \subseteq \{R_1, \ldots, R_n\}$ be a subset of arguments such that ϕ is commutative with respect to $\boldsymbol{S}$. Then, in every bucket $b \in \mathcal{B}(\phi)$ with $|\phi(b)| > 1$, there exists a potential value φ that occurs at least $|\boldsymbol{S}|$ times in $\phi(b)$.*

Example 6 *Consider again the factor $\phi(R_1, R_2, R_3)$, which is commutative with respect to $\boldsymbol{S} = \{R_2, R_3\}$, depicted in Table 1. Therefore, ϕ must map the buckets $[2, 1]$ and $[1, 2]$ (the only buckets that are mapped to at least two elements by ϕ) to at least $|\boldsymbol{S}| = 2$ identical potential values. Here, φ_2 occurs twice in $\phi([2, 1])$ and φ_5 occurs twice in $\phi([1, 2])$.*

As we know for every bucket b containing more than one element that there must exist a potential occurring at least $|\boldsymbol{S}|$ times in b to allow for a subset $\boldsymbol{S}$ of commutative arguments to exist, we also know that there cannot be more commutative arguments than the minimum number of duplicate potential values over all buckets containing more than one element.

Corollary 6 *Let $\phi(R_1, \ldots, R_n)$ denote a factor. Then, the size of any subset of commutative arguments of ϕ is upper-bounded by*

$$\min_{b \in \{b \mid b \in \mathcal{B}(\phi) \wedge |\phi(b)| > 1\}} \max_{\varphi \in \phi(b)} \mathrm{count}(\phi(b), \varphi),$$

where $\text{count}(\phi(b), \varphi)$ *denotes the number of occurrences of potential* φ *in* $\phi(b)$*.*

Example 7 *The factor* $\phi(R_1, R_2, R_3)$ *given in Table 1 maps* $[2, 1]$ *to* $\langle \varphi_2, \varphi_2, \varphi_4 \rangle$ *and* $[1, 2]$ *to* $\langle \varphi_3, \varphi_5, \varphi_5 \rangle$ *and thus, the upper bound for the size of any subset of commutative arguments of* ϕ *is two because* ϕ *maps the buckets* $[2, 1]$ *and* $[1, 2]$ *to two identical potential values each.*

The bounds implied by Cors. 5 and 6 show that the number of possible candidate subsets of commutative arguments can be heavily reduced in many practical settings. In particular, we are now able to exploit the insights from Thm. 4 as well as Cors. 5 and 6 to drastically restrict the search space of possible candidate subsets of commutative arguments, thereby avoiding the naive iteration over all $O(2^n)$ subsets of a factor's arguments and thus making the check for commutativity feasible in practice.

We next incorporate our theoretical findings into a practical algorithm, called DECOR, to efficiently compute subsets of commutative arguments of maximum size in practice.

4. The DECOR Algorithm

We are now ready to gather the theoretical results to obtain the DECOR algorithm, which efficiently computes a maximum sized subset of commutative arguments of a given factor. Algorithm 1 presents the whole DECOR algorithm, which undertakes the following steps on a given input factor $\phi(R_1, \ldots, R_n)$ to compute subsets of commutative arguments.

First, DECOR initialises a set C of possible candidate subsets of commutative arguments, which contains the whole argument list of ϕ at the beginning. DECOR then iterates over all buckets entailed by ϕ, where all buckets containing only a single element are skipped because they do not restrict the search space. DECOR then partitions the values in every bucket into maximal groups $G_1, \ldots, G_\ell$ of identical potential values, where each group must contain at least two identical potential values as we are only interested in subsets of commutative arguments of size at least two. If no such group is found, DECOR returns an empty set as there are no candidates for a subset of commutative arguments of size at least two. Afterwards, DECOR applies the insights from Thm. 4 and iterates over all groups of identical potential values to compute the element-wise intersection of the assignments corresponding to the potential values within a group. For every group G_i of identical potential values, DECOR then builds a set C_i containing all arguments whose position in the element-wise intersection over their corresponding assignments is empty. DECOR then adds the set C_i of arguments that are commutative according to the current bucket to the set C' of candidate subsets for the current bucket if C_i is not subsumed by any other candidate subset in C'. Thereafter, as each bucket further restricts the search space for possible commutative arguments, DECOR computes the intersection of all candidate subsets collected so far in C and all candidate subsets of the current bucket in C'. DECOR keeps all candidate subsets of size at least two after the intersection that are not already subsumed by another candidate subset after the intersection. In case there are no candidate subsets of size at least two left after considering a specific bucket, DECOR returns an empty set as a result. Finally, if there are still candidate subsets left after iterating over all buckets of size at least two, DECOR returns the set of all candidate subsets of commutative arguments of maximum size. A maximum sized subset of commutative arguments can then be obtained by selecting any candidate subset in C that contains the most elements.

Algorithm 1: Detection of Commutative Factors (DECOR)

```
Input: A factor $\phi(R_1, \dots, R_n)$.
Output: All subsets $\boldsymbol{S} \subseteq \{R_1, \dots, R_n\}$ with $|\boldsymbol{S}| \geq 2$ such that $\phi$ is commutative
        with respect to $\boldsymbol{S}$, or $\emptyset$ if no such subset $\boldsymbol{S}$ exists.
1  $C \leftarrow \{\{R_1, \dots, R_n\}\}$
2  foreach $b \in \mathcal{B}(\phi)$ do
3      if $|\phi(b)| < 2$ then
4          continue
5      $G_1, \dots, G_\ell \leftarrow$ Partition $\phi(b)$ into maximal groups of identical values such that
         $|G_i| \geq 2$ for all $i \in \{1, \dots, \ell\}$
6      if $\ell < 1$ then
7          return $\emptyset$
8      $C' \leftarrow \emptyset$
9      foreach $G_i \in \{G_1, \dots, G_\ell\}$ do
10         $P_1, \dots, P_n \leftarrow$ Intersection of positions corresponding to all potentials in $G_i$
11         $C_i \leftarrow \{R_i \mid i \in \{1, \dots, n\} \land P_i = \emptyset\}$
12         if $\nexists C_j \in C' : C_i \subseteq C_j$ then
13             $C' \leftarrow C' \cup \{C_i\}$
14     $C_\cap \leftarrow \emptyset$
15     foreach $C_i \in C$ do
16         foreach $C_j \in C'$ do
17             if $|C_i \cap C_j| \geq 2 \land \nexists C'' \in C_\cap : C_i \cap C_j \subseteq C''$ then
18                 $C_\cap \leftarrow C_\cap \cup \{C_i \cap C_j\}$
19     $C \leftarrow C_\cap$
20     if $C = \emptyset$ then
21         return $\emptyset$
22 return $C$
```

Note that DECOR returns *all* candidate subsets of commutative arguments that are not subsumed by another candidate subset and could be replaced by a so-called counting randvar in the lifted representation of the factor. As we aim for the most compression possible, we are mostly interested in the maximum sized subset of commutative arguments. However, it might be conceivable that there are settings in which counting over two or more candidate subsets of commutative arguments yields a higher compression than counting over a single maximum sized subset. We therefore keep DECOR as a general algorithm that is able to return all candidate subsets of commutative arguments that are not subsumed by another candidate subset and leave the decision of which subset(s) to choose to the user.

Example 8 *Let us take a look at how DECOR computes a maximum sized subset of commutative arguments for the factor $\phi(R_1, R_2, R_3)$ from Table 1. Initially, DECOR starts with the set $C = \{\{R_1, R_2, R_3\}\}$. After skipping the bucket $[3, 0]$, DECOR finds the group $G_1 = \{\varphi_2, \varphi_2\}$ for the bucket $[2, 1]$. Other groups contain less than two elements and are thus ignored. DECOR then computes the element-wise intersection of the assignments corresponding to the potential values in G_1, which is given by* $(\text{true}, \text{true}, \text{false}) \cap (\text{true}, \text{false}, \text{true}) =$

(true, $\emptyset, \emptyset$). *As the element-wise intersection is empty at positions two and three, DECOR adds the set $C_i = \{R_2, R_3\}$ to the set C' of candidate subsets for the bucket* $[2, 1]$. *Afterwards, DECOR computes the intersection of $\{R_1, R_2, R_3\}$ (as $C = \{\{R_1, R_2, R_3\}\}$) and $\{R_2, R_3\}$ (as $C' = \{\{R_2, R_3\}\}$), which yields $C_\cap = \{\{R_2, R_3\}\}$ and thus also $C = \{\{R_2, R_3\}\}$ for the next iteration. DECOR next finds the group $G_1 = \{\varphi_5, \varphi_5\}$ for the bucket* $[1, 2]$ *and computes the element-wise intersection of the assignments corresponding to the potential values in G_1, which is given by* $(\text{false}, \text{true}, \text{false}) \cap (\text{false}, \text{false}, \text{true}) = (\text{false}, \emptyset, \emptyset)$. *Again, positions two and three are empty and thus, DECOR adds the set $C_i = \{R_2, R_3\}$ to the set C' of candidate subsets for the bucket* $[1, 2]$. *Thereafter, DECOR computes the intersection of $\{R_2, R_3\}$ (as $C = \{\{R_2, R_3\}\}$) and $\{R_2, R_3\}$ (as $C' = \{\{R_2, R_3\}\}$), which yields $C = C_\cap = \{\{R_2, R_3\}\}$. Finally, as the next bucket* $[0, 3]$ *is skipped again and there are no other buckets left, DECOR returns the set $\{\{R_2, R_3\}\}$ containing a single candidate subset of commutative arguments, which is at the same time a maximum sized subset.*

We remark that the number of candidate subsets considered by DECOR now directly depends on the number of groups of identical potential values in the buckets of the factor. In most practical settings, potential values are not identical unless the factor is commutative with respect to a subset of its arguments. Therefore, DECOR heavily restricts the search space for possible candidate subsets of commutative arguments in most practical settings.

Recall that the "naive" algorithm iterates over $O(2^n)$ subsets of arguments and then has to consider all $|\mathcal{R}|^n$ (where $\mathcal{R}$ denotes the range of the arguments and n is the total number of arguments) potential values in the table of potential mappings during each iteration to check whether a subset of arguments is commutative. While DECOR also has to consider all $|\mathcal{R}|^n$ potential values during the course of the algorithm (in fact, every algorithm searching for commutative arguments has to take a look at every potential value at least once), DECOR considers every potential value just twice (in Lines 5 and 10).

Theorem 7 *Let $\phi(R_1, \ldots, R_n)$ denote a factor and let* $\mathcal{R} = \text{range}(R_1) = \ldots = \text{range}(R_n)$. *The worst-case time complexity of the "naive" algorithm to compute a maximum sized subset of commutative arguments is in $O(2^n \cdot |\mathcal{R}|^n \cdot n)$.*

Proof The "naive" algorithm iterates over all subsets of ϕ's arguments and then iterates for every subset over all buckets induced by that subset to check whether each bucket is mapped to a unique potential value by ϕ. To compute the buckets, all n positions in all $|\mathcal{R}|^n$ assignments are considered. The iteration over all values in the buckets requires time $O(|\mathcal{R}|^n)$, as every potential value is looked at once. In the worst case, the "naive" algorithm iterates over $O(2^n)$ subsets of arguments and thus has a complexity of $O(2^n \cdot |\mathcal{R}|^n \cdot n)$. ■

Theorem 8 *Let $\phi(R_1, \ldots, R_n)$ denote a factor and let* $\mathcal{R} = \text{range}(R_1) = \ldots = \text{range}(R_n)$. *The expected worst-case time complexity of the DECOR algorithm to compute a maximum sized subset of commutative arguments is in $O(|\mathcal{B}(\phi)| \cdot k \cdot \binom{n}{n/2} + |\mathcal{R}|^n \cdot n)$, where k denotes the maximum number of potential values within a bucket entailed by ϕ.*

Proof The partitioning of $\phi(b)$ into maximal groups of identical potential values (Line 5) requires time linear in the number of potential values in the bucket b and as all $|\mathcal{R}|^n$ potential values are looked at exactly once over all buckets, the partitioning requires time $O(|\mathcal{R}|^n)$

in total for all buckets. DECOR finds at most $\lfloor |\mathcal{R}|^n/2 \rfloor$ groups of identical potential values (if every group contains exactly two identical potential values) over all buckets. The loop in Line 9 thus needs $O(|\mathcal{R}|^n)$ iterations in the worst case for all buckets entailed by ϕ. Computing the intersection over all positions is done in time $O(n)$ and subset testing can be done in expected time $O(n)$, thereby yielding an expected time complexity of $O(|\mathcal{R}|^n \cdot n)$ for the loop in Line 9. For each bucket, DECOR adds at most $k/2$ candidate subsets to C' and as there are at most $\binom{n}{n/2}$ candidate subsets in C (there are $\binom{n}{j}$ subsets of size j and the binomial coefficient reaches its maximum at $j = n/2$), the loops in Lines 15 and 16 run in time $O(k \cdot \binom{n}{n/2})$ per bucket, yielding a total run time of $O(|\mathcal{B}(\phi)| \cdot k \cdot \binom{n}{n/2} + |\mathcal{R}|^n \cdot n)$. ■

There are $|\mathcal{B}(\phi)| = \binom{n+|\mathcal{R}|-1}{n}$ buckets for a factor $\phi(R_1, \ldots, R_n)$ where $\mathcal{R} = \mathrm{range}(R_1) = \ldots = \mathrm{range}(R_n)$[1]. For example, a factor with n Boolean arguments ($|\mathcal{R}| = 2$) entails $\binom{n+1}{n} = n+1$ buckets. Of these buckets, DECOR only has to consider $n-1$ buckets, as two map to one potential. Note that the worst-case complexity of DECOR is overly pessimistic as it is impossible to have the maximum number of groups $G_1, \ldots, G_\ell$ and the maximum number of candidate subsets C simultaneously. The binomial coefficient is a generous upper bound nearly impossible to reach in practice, which is also confirmed in our experiments.

Further, we argue that $n/2$, instead of the binomial coefficient, is still a generous upper bound in practice for the maximum number of groups and the maximum number of candidate subsets. In case there are hardly any symmetries within the factor, there are also hardly any duplicate potentials in a bucket. With hardly any duplicate potentials in a bucket, the number of candidate sets is small. In case the model is symmetric in general and there are symmetries within a factor, then there is a good chance that in each bucket at least one potential occurs quite often. This case leads to few but large candidate sets. Further, the candidate sets across all buckets are likely to be rather identical. Thus, having $n/2$ as an upper bound for both the maximum number of groups and the maximum number of candidate subsets is still pessimistic in practice, as DECOR most likely either finds few (larger) sets or hardly any sets at all. Overall, our arguments why DECOR finishes fast in practice are: In case the model is highly symmetric, symmetries within a factor are expected to be over larger sets. Thus, DECOR has just few sets to consider. In case there are hardly any symmetries within a factor and the model itself, DECOR has to check few small candidate sets and might even end up with an empty candidate set after few buckets.

Finally, before we demonstrate the practical efficiency of DECOR in our experimental evaluation, we remark that DECOR can handle factors with arguments having arbitrary ranges. During the course of this paper, we consider identical ranges of all arguments for brevity, however, it is possible to apply DECOR to factors with arguments having various ranges $\mathcal{R}_1, \ldots, \mathcal{R}_k$ by considering the buckets for all arguments with range $\mathcal{R}_i$ separately for all $i \in \{1, \ldots, k\}$ as only arguments having the same range can be commutative at all.

5. Experiments

In addition to our theoretical results, we conduct an experimental evaluation of DECOR to assess its performance in practice. We compare the run times of DECOR to the "naive" approach, i.e., iterating over all possible subsets of a factor's arguments in or-

1. The number of buckets is given by the number of weak compositions of n into $|\mathcal{R}|$ parts.

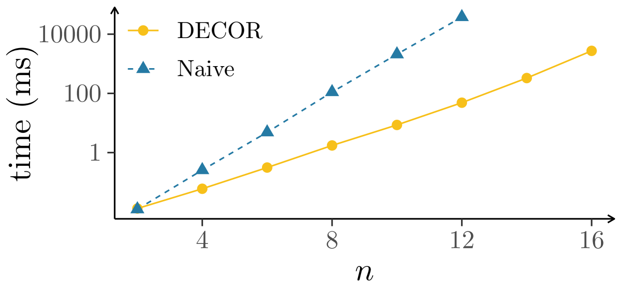


Figure 2: Average run times of DECOR and the "naive" algorithm for factors with different numbers of arguments n. For each choice of n, we report the average run time over all choices of numbers of commutative arguments k, where $k \in \{0, 2, \lfloor \frac{n}{2} \rfloor, n-1, n\}$.

der of descending size of the subsets. For our experiments, we generate factors with $n \in \{2, 4, 6, 8, 10, 12, 14, 16\}$ Boolean arguments, of which $k \in \{0, 2, \lfloor \frac{n}{2} \rfloor, n-1, n\}$ arguments are commutative. We run each algorithm with a timeout of five minutes per instance and report the average run time over all instances for each choice of n.

Figure 2 displays the average run times of the algorithms on a logarithmic scale. While both DECOR and the "naive" algorithm are capable of solving small instances with less than ten arguments in under a second, the "naive" algorithm struggles to solve instances with increasing n. More specifically, at $n = 8$, DECOR is already faster than the "naive" algorithm by a factor of 100 and the factor drastically increases with an increasing size of n. After $n = 12$, the "naive" algorithm runs into a timeout, which is not surprising as it iterates over $O(2^n)$ subsets. DECOR, on the other hand, solves all instances within the specified timeout and is able to handle large instances in an efficient manner. We remark that the "naive" algorithm is able to solve instances with $n > 12$ if $k = n-1$ or $k = n$, which is due to its iterations being performed in order of descending size of the subsets. However, as the "naive" algorithm runs into a timeout for all remaining choices of k, it is not possible to compute a meaningful average run time. We therefore provide additional experimental results in Appendix B, where we report run times for each choice of k separately.

6. Conclusion

We present the DECOR algorithm to efficiently detect commutative factors in FGs. DECOR makes use of buckets to significantly restrict the search space and thereby efficiently handles factors even with a large number of arguments where previous algorithms fail to compute a solution within a timeout. We prove that the number of subsets to check for commutative arguments is upper-bounded depending on the number of duplicate potential values in the buckets. By exploiting this upper bound, DECOR drastically reduces the number of subsets to check for commutative arguments compared to previous algorithms, which iterate over $O(2^n)$ subsets for a factor with n arguments in the worst case. Additionally, DECOR returns *all* candidate subsets of commutative arguments that are not subsumed by another candidate subset, allowing to compress the model even further. Overall, DECOR drastically increases the efficiency to identify commutative factors and further reduces the model.

Acknowledgments

This work is funded by the BMBF project AnoMed 16KISA057.

References

B. Ahmadi, K. Kersting, M. Mladenov, and S. Natarajan. Exploiting Symmetries for Scaling Loopy Belief Propagation and Relational Training. *Machine Learning*, 92:91–132, 2013.

T. Braun and R. Möller. Parameterised Queries and Lifted Query Answering. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-2018)*, pages 4980–4986. IJCAI Organization, 2018.

R. De Salvo Braz, E. Amir, and D. Roth. Lifted First-Order Probabilistic Inference. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-2005)*, pages 1319–1325. Morgan Kaufmann Publishers Inc., 2005.

R. De Salvo Braz, E. Amir, and D. Roth. MPE and Partial Inversion in Lifted Probabilistic Variable Elimination. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI-2006)*, pages 1123–1130. AAAI Press, 2006.

B. J. Frey, F. R. Kschischang, H.-A. Loeliger, and N. Wiberg. Factor Graphs and Algorithms. In *Proceedings of the Thirty-Fifth Annual Allerton Conference on Communication, Control, and Computing*, pages 666–680. Allerton House, 1997.

K. Kersting, B. Ahmadi, and S. Natarajan. Counting Belief Propagation. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI-2009)*, pages 277–284. AUAI Press, 2009.

J. Kisyński and D. Poole. Constraint Processing in Lifted Probabilistic Inference. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI-2009)*, pages 293–302. AUAI Press, 2009.

F. R. Kschischang, B. J. Frey, and H.-A. Loeliger. Factor Graphs and the Sum-Product Algorithm. *IEEE Transactions on Information Theory*, 47:498–519, 2001.

M. Luttermann, T. Braun, R. Möller, and M. Gehrke. Colour Passing Revisited: Lifted Model Construction with Commutative Factors. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence (AAAI-2024)*, pages 20500–20507. AAAI Press, 2024a.

M. Luttermann, J. Machemer, and M. Gehrke. Efficient Detection of Exchangeable Factors in Factor Graphs. In *Proceedings of the Thirty-Seventh International Florida Artificial Intelligence Research Society Conference (FLAIRS-2024)*. Florida Online Journals, 2024b.

B. Milch, L. S. Zettlemoyer, K. Kersting, M. Haimes, and L. P. Kaelbling. Lifted Probabilistic Inference with Counting Formulas. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-2008)*, pages 1062–1068. AAAI Press, 2008.

M. Niepert and G. Van den Broeck. Tractability through Exchangeability: A New Perspective on Efficient Probabilistic Inference. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI-2014)*, pages 2467–2475. AAAI Press, 2014.

D. Poole. First-Order Probabilistic Inference. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-2003)*, pages 985–991. Morgan Kaufmann Publishers Inc., 2003.

P. Singla and P. Domingos. Lifted First-Order Belief Propagation. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-2008)*, pages 1094–1099. AAAI Press, 2008.

N. Taghipour, D. Fierens, J. Davis, and H. Blockeel. Lifted Variable Elimination: Decoupling the Operators from the Constraint Language. *Journal of Artificial Intelligence Research*, 47:393–439, 2013.

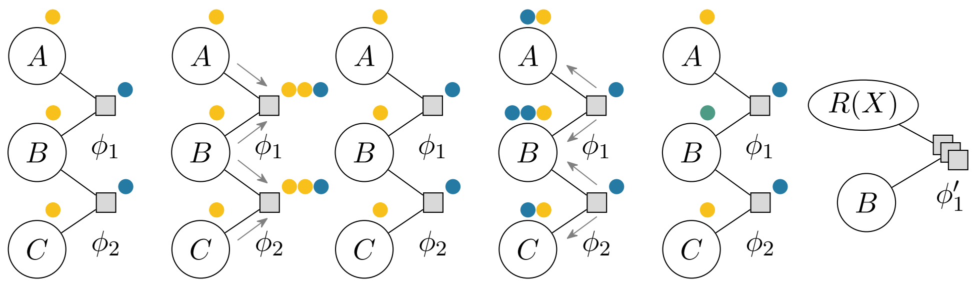
B. Weisfeiler and A. A. Leman. The Reduction of a Graph to Canonical Form and the Algebra which Appears Therein. *NTI, Series*, 2:12–16, 1968. English translation by Grigory Ryabov available at https://www.iti.zcu.cz/wl2018/pdf/wl_paper_translation.pdf.

Appendix A. Formal Description of the Advanced Colour Passing Algorithm

The advanced colour passing (ACP) algorithm introduced by Luttermann et al. (2024a) builds on the colour passing algorithm (Kersting et al., 2009; Ahmadi et al., 2013) and solves the problem of constructing a lifted representation—more specifically, a so-called parametric factor graph (PFG)—from a given factor graph (FG). The idea of ACP is to first find symmetries in a propositional FG and then group together symmetric subgraphs. ACP looks for symmetries based on potentials of factors, on ranges and evidence of random variables (randvars), as well as on the graph structure by passing around colours. Algorithm 2 provides a formal description of the ACP algorithm, which proceeds as follows.

Algorithm 2: Advanced Colour Passing (Luttermann et al., 2024a)

```
Input: An FG $G$ with randvars $\boldsymbol{R} = \{R_1, \dots, R_n\}$ and factors $\boldsymbol{\Phi} = \{\phi_1, \dots, \phi_m\}$,
       as well as a set of evidence $\boldsymbol{E} = \{R_1 = r_1, \dots, R_k = r_k\}$.
Output: A lifted representation $G'$ in form of a PFG entailing equivalent
        semantics to $G$.
1  Assign each $R_i$ a colour according to range($R_i$) and $\boldsymbol{E}$
2  Assign each $\phi_i$ a colour according to order-independent potentials and rearrange
    arguments accordingly
3  repeat
4      foreach factor $\phi \in \boldsymbol{\Phi}$ do
5          $signature_{\phi} \leftarrow [\,]$
6          foreach randvar $R \in neighbours(G, \phi)$ do
               // In order of appearance in $\phi$
7              $append(signature_{\phi}, R.colour)$
8          $append(signature_{\phi}, \phi.colour)$
9      Group together all $\phi$s with the same signature
10     Assign each such cluster a unique colour
11     Set $\phi.colour$ correspondingly for all $\phi$s
12     foreach randvar $R \in \boldsymbol{R}$ do
13         $signature_R \leftarrow [\,]$
14         foreach factor $\phi \in neighbours(G, R)$ do
15             if $\phi$ is commutative w.r.t. $\boldsymbol{S}$ and $R \in \boldsymbol{S}$ then
16                 $append(signature_R, (\phi.colour, 0))$
17             else
18                 $append(signature_R, (\phi.colour, p(R, \phi)))$
19         Sort $signature_R$ according to colour
20         $append(signature_R, R.colour)$
21     Group together all $R$s with the same signature
22     Assign each such cluster a unique colour
23     Set $R.colour$ correspondingly for all $R$s
24 until grouping does not change
25 $G' \leftarrow$ construct PFG from groupings
```



A	B	$\phi_1(A,B)$
true	true	φ_1
true	false	φ_2
false	true	φ_3
false	false	φ_4

C	B	$\phi_2(C,B)$
true	true	φ_1
true	false	φ_2
false	true	φ_3
false	false	φ_4

$R(X)$	B	$\phi'_1(R(X),B)$
true	true	φ_1
true	false	φ_2
false	true	φ_3
false	false	φ_4

Figure 3: A visualisation of the steps undertaken by the ACP algorithm on an input FG with only Boolean randvars and no evidence (left). Colours are first passed from variable nodes to factor nodes, followed by a recolouring, and then passed back from factor nodes to variable nodes, again followed by a recolouring. The colour passing procedure is iterated until convergence and the resulting PFG is depicted on the right. This figure is reprinted from (Luttermann et al., 2024a).

ACP begins with the colour assignment to variable nodes, meaning that all randvars that have the same range and observed event are assigned the same colour. Thereafter, ACP assigns colours to factor nodes such that factors representing identical potentials are assigned the same colour. Two factors represent identical potentials if there exists a rearrangement of one of the factor's arguments such that both factors have identical tables of potentials when comparing them row by row (Luttermann et al., 2024b).

After the initial colour assignments, ACP passes the colours around. ACP first passes the colours from every variable node to its neighbouring factor nodes and afterwards, every factor node ϕ sends its colour plus the position $p(R, \phi)$ of R in ϕ's argument list to all of its neighbouring variable nodes R. Note that factors being commutative with respect to a subset $\boldsymbol{S}$ of their arguments omit the position when sending their colour to a neighbouring variable node $R \in \boldsymbol{S}$, and therefore, ACP has to compute for each factor a maximum sized subset of its arguments such that the factor is commutative with respect to that subset. In its original form, ACP iterates over all possible subsets of arguments, in descending order of their size, to compute the maximum sized subset.

Figure 3 illustrates the ACP algorithm on an example FG (Ahmadi et al., 2013). In this example, A, B, and C are Boolean randvars with no evidence and thus, they all receive the same colour (e.g., yellow). As the potentials of ϕ_1 and ϕ_2 are identical, ϕ_1 and ϕ_2 are assigned the same colour as well (e.g., blue). The colour passing then starts from variable nodes to factor nodes, that is, A and B send their colour (yellow) to ϕ_1 and B and C send their colour (yellow) to ϕ_2. ϕ_1 and ϕ_2 are then recoloured according to the colours they received from their neighbours to reduce the communication overhead. Since ϕ_1 and ϕ_2

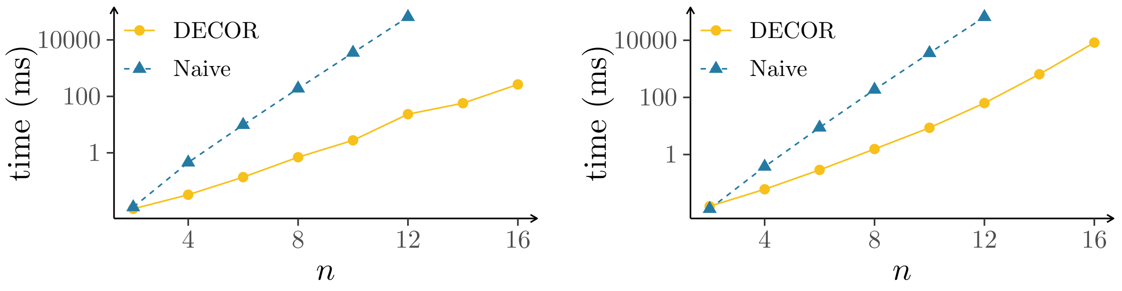


Figure 4: Run times of DECOR and the “naive” algorithm for factors with different numbers of arguments n, of which $k = 0$ (left) and $k = 2$ (right) arguments are commutative, respectively. Both plots use a logarithmic scale on the y-axis.

received identical colours (two times the colour yellow), they are assigned the same colour during recolouring. Afterwards, the colours are passed from factor nodes to variable nodes and this time not only the colours but also the position of the randvars in the argument list of the corresponding factor are shared because none of the factors is commutative with respect to a subset of its arguments having size at least two. Consequently, ϕ_1 sends a tuple $(\text{blue}, 1)$ to A and a tuple $(\text{blue}, 2)$ to B, and ϕ_2 sends a tuple $(\text{blue}, 2)$ to B and a tuple $(\text{blue}, 1)$ to C (positions are not shown in Fig. 3). As A and C are both at position one in the argument list of their respective neighbouring factor, they receive identical messages and are recoloured with the same colour. B is assigned a different colour during recolouring than A and C because B received different messages than A and C. The groupings do not change in further iterations and hence the algorithm terminates. The output is the PFG shown on the right in Fig. 3, where both A and C as well as ϕ_1 and ϕ_2 are grouped.

More details about the colour passing procedure and the grouping of nodes can be found in (Luttermann et al., 2024a). The authors also provide extensive results demonstrating the benefits of constructing and using a lifted representation for probabilistic inference.

Appendix B. Additional Experimental Results

In addition to the experimental results provided in Sec. 5, we provide further experimental results for individual scenarios in this section. In particular, the plot given in Fig. 2 displays average run times over multiple runs for different choices of the number of commutative arguments k. We now showcase separate plots for each of the choices of the number of commutative arguments k to highlight the influence of k on the performance on the algorithms. Again, we compare the run times of the “naive” algorithm to the run times of detection of commutative factors (DECOR) algorithm on factors with $n \in \{2, 4, 6, 8, 10, 12, 14, 16\}$ Boolean arguments and set a timeout of five minutes per instance.

Figure 4 shows the run times of both algorithms on factors with $k = 0$ (left plot) and $k = 2$ (right plot) commutative arguments on a logarithmic scale. The performance of the “naive” algorithm is similar for $k = 0$ and $k = 2$. As in Fig. 2, the “naive” algorithm is able to solve instances with less than ten arguments in under a second but then faces serious scalability issues, eventually leading to timeouts for all factors with $n > 12$. DECOR

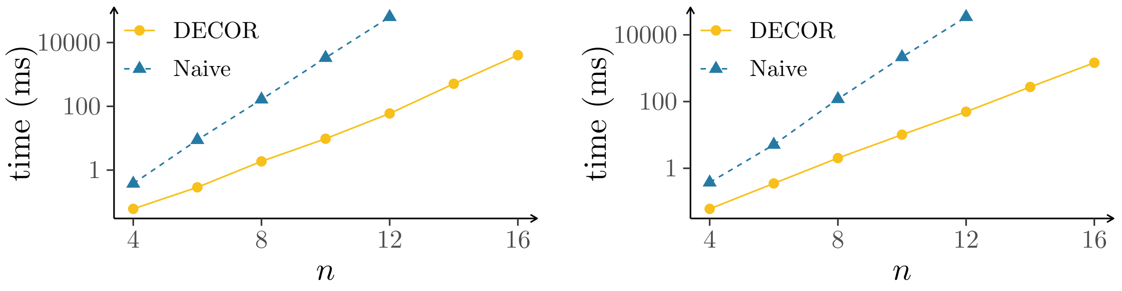


Figure 5: Run times of DECOR and the "naive" algorithm for factors with different numbers of arguments n, of which $k = \lfloor \log_2(n) \rfloor$ (left) and $k = \lfloor \frac{n}{2} \rfloor$ (right) arguments are commutative, respectively. Both plots use a logarithmic scale on the y-axis.

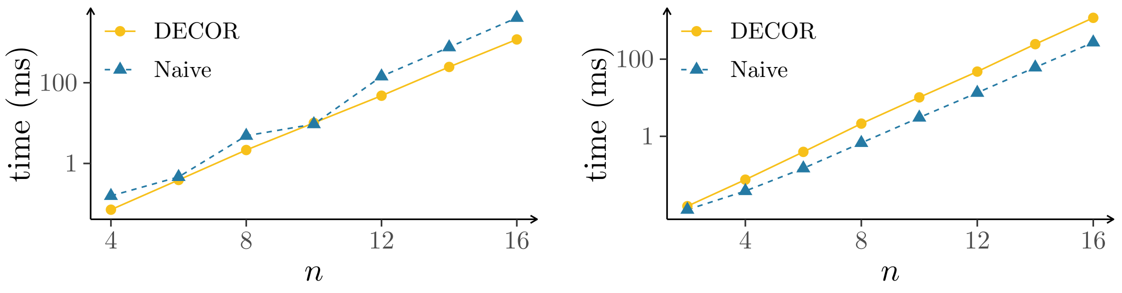


Figure 6: Run times of DECOR and the "naive" algorithm for factors with different numbers of arguments n, of which $k = n - 1$ (left) and $k = n$ (right) arguments are commutative, respectively. Both plots use a logarithmic scale on the y-axis.

solves all instances within the specified timeout and while the run times of DECOR for $k = 2$ are similar to those in Fig. 2, DECOR is especially fast on instances where $k = 0$ of the arguments are commutative. This can be explained by the fact that all instances with $k = 0$ are generated such that each potential value is unique to ensure that there are no commutative arguments. Consequently, if there are only unique values in each bucket, DECOR has no candidates to check and returns immediately (the run time still increases the larger n gets because computing the buckets becomes more costly).

Figure 5 presents the run times of the algorithms on factors with $k = \lfloor \log_2(n) \rfloor$ (left plot) and $k = \lfloor \frac{n}{2} \rfloor$ (right plot) commutative arguments on a logarithmic scale. The general pattern of the run times is again similar to Fig. 2 and we therefore move on to take a look at Fig. 6, which depicts the run times of the algorithms on factors having $k = n - 1$ (left plot) and $k = n$ (right plot) commutative arguments, again on a logarithmic scale.

The left plot of Fig. 6 shows that the "naive" algorithm is now able to solve all instances within the specified timeout and its run times are similar to the run times of DECOR. As $k = n - 1$, the "naive" algorithm is able to find a subset of commutative arguments in few iterations due to its approach of starting with the largest subsets first. Consequently, the "naive" algorithm performs well on the right plot where $k = n$ as it considers exactly one subset and immediately finishes afterwards. Therefore, DECOR is even slightly slower than the “naive” algorithm in this particular extrem case—however, the difference is only marginal. In conclusion, we find that the “naive” approach is generally not able to scale to numbers of arguments larger than 12 (only in extrem cases) whereas DECOR is able to handle large n efficiently in every evaluated scenario.