Explainable and Explorable Decision Support

Tanya Braun[1][0000−0003−0282−4284]⋆ and Marcel Gehrke[2][0000−0001−9056−7673]⋆

[1] Data Science Group, University of Münster, Münster, Germany
tanya.braun@uni-muenster.de
[2] Institute of Information Systems, University of Lübeck, Lübeck, Germany
gehrke@ifis.uni-luebeck.de

Abstract. An effective decision support system requires a user's trust in its results, which are based on expected utilities of different action plans. As such, a result needs to be explainable and explorable, providing alternatives and additional information in a proactive way, instead of retroactively answering follow-up questions to a single action plan as output. Therefore, this paper presents LEEDS, an algorithm that computes alternative action plans, identifies groups of interest, and answers marginal queries for those groups to provide a comprehensive overview supporting a user. LEEDS leverages the strengths of gate models, lifting, and the switched lifted junction tree algorithm for efficient *explainable* and *explorable* decision support.

Keywords: Decision Support · Lifting · Multi Query Answering

1 Introduction

Decision support systems provide users with an action plan that is the result of considering expected utilities of various actions or learning a policy. A key challenge to decision support is building trust in the system. Often only a simple explanation is provided in the form of the action being most probable or leading to the highest utility, which may not be enough, especially if the result contradicts with a user's own assessment. Approximations may further hinder trust building, as they may be alienating to users or not good enough in applications concerning, e.g., healthcare [22]. Instead, based on exact calculations, further context, explanation, or alternatives are needed to explore the result. The problem becomes especially apparent in the health care sector. For medical professionals, understanding why they should take an action is crucial to understanding proposed actions [19]. Retroactively answering follow-up questions about groups of interest or states of patients disrupts a user's work flow. Therefore, decision support systems need to provide a full picture of alternative action plans with additional information about groups or queries registered in advance. In addition to these human-centric requirements, decision support systems need to cope with vast amounts of probabilistic, relational data and still provide results in a timely manner.

⋆ Both authors contributed equally to the paper.

Implementing such systems requires (i) a relational model compactly encoding objects, relations, and uncertainties, incorporating actions and utilities to solve a maximum expected utility (MEU) problem, asking not for a single but top-k action plans, and (ii) exact, lifted inference (using a helper structure) for accurate and timely results. Lifted inference handles groups of indistinguishable objects efficiently using representatives, which makes inference tractable regarding the number of objects [15]. Using lifted inference also enables a system to identify groups or objects of interest that act differently than the remaining objects. Using a helper structure allows for efficient answering of multiple queries, reusing calculations as much as possible.

Therefore, this paper presents Lifted Explainable and Explorable Decision Support (LEEDS). Specifically, the contributions are (i) parameterised decision gate models (PDecGMs) as the modelling formalism, incorporating parameterised utilities as well as parameterised gates for actions and (ii) LEEDS as the algorithm that outputs top-k action plans, groups of interest, and answers to marginal queries given a PDecGM, k, evidence, and possibly registered queries. LEEDS builds upon switched lifted junction tree algorithm (SLJT), which performs exact lifted inference in parameterised gate models (PGMs) [7]. PGMs represent a full joint probability distribution and consist of parameterised random variables (PRVs) that are combined by parametric factors (parfactors), which can be switched on or off using gates. We use the gates formalism [13] to efficiently model actions, explicitly encoding impacts of actions on a model by switching from one model representation to another. Modelling actions using gates also leads to fewer random variables, which has a positive effect on inference complexity. Lifting allows for exploiting relational structures during calculations [16]. SLJT performs lifted, exact inference in PGMs. A so-called first-order junction tree (FO jtree) [12,4] as an underlying helper structure enables efficient answering of multiple queries. Performance-wise, LEEDS exploits relational structures for tractable inference w.r.t. domain sizes as well as the behaviour encoded in gates, as evidenced by a small empirical case study. It proceeds adaptively to save up to 50% of its computations compared to a naive algorithm.

LEEDS as an algorithm belongs to a group of lifted algorithms aiming at performing calculations on a lifted level, using grounding only as a last resort, starting with lifted variable elimination (LVE) as the first lifted algorithm introduced [16,20,6]. The lifted junction tree algorithm (LJT) [4], first-order knowledge compilation [21], and probabilistic theorem proving [9] use a helper structure for efficient multi-query answering. Whereas the algorithms mentioned perform exact inference, approximate algorithms also exist such as lifted belief propagation [1]. LEEDS also performs lifted decision making, for which two main approaches exist: (i) finding a policy in a first-order (partially observable) Markov decision process (FO (PO)MDP) [11,18] or (ii) solving a MEU problem in a relational model that includes actions and utilities [14,2,8]. The main advantage of the first approach is its efficiency regarding decision making as it reduces to looking up the corresponding action in the policy online. The disadvantage lies in the

decision coming from a black-box policy where an explanation is hard to find, alternatives cannot be easily calculated, and an exploration of the decision and a model state would be hard to achieve. In contrast, the second approach allows for explaining and exploring a decision as the model allows for further queries to provide additional information at the expense of online calculation time. To the best of our knowledge, none of the existing algorithms tackle the combined problem of decision making and query answering needed for explainable and explorable decision support, using lifting for efficiency.

In the following, we begin by recapitulating PGMs as well as SLJT for inference on PGMs. Then, we add actions and utilities to the formalism for solving MEU problems and present LEEDS. We end with a conclusion.

2 Preliminaries

This section defines PGMs [7], which combine parameterised probabilistic models (PMs) [16] and gate models [13]. Then, it recaps SLJT.

2.1 Parameterised Gate Models

PGMs combine first-order logic with probabilistic models, using logical variables (logvars) as parameters in random variables (randvars). For illustrative purposes, we use an example of an epidemic based on [17]. In the example, we model an epidemic as a randvar and persons being sick as a PRV by parameterising a randvar for sick with a logvar for persons. In the larger scheme, all persons are influenced in the same way in an epidemic without additional evidence and thus are indistinguishable.

Definition 1. *Let* $\mathbf{R}$ *be a set of randvar names,* $\mathbf{L}$ *a set of logvar names,* Φ *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* $\mathcal{X} = (X_1, \ldots, X_n)$ *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.,* $C_{\mathcal{X}} = \times_{i=1}^{n} \mathcal{D}(X_i)$*, and may be omitted.*

A PRV $R(L_1, \ldots, L_n), n \geq 0$ *is a syntactical construct of a randvar* $R \in \mathbf{R}$ *possibly combined with logvars* $L_1, \ldots, L_n \in \mathbf{L}$*. If* $n = 0$*, the PRV is parameterless and constitutes a propositional randvar.* $A_{|C}$ *denotes a PRV* A *under constraint* C*. The term* $\mathcal{R}(A)$ *denotes the possible values (range) of* A*. An* event $A = a$ *denotes the occurrence of* A *with value* $a \in \mathcal{R}(A)$*.*

The term $lv(\Gamma)$ refers to the logvars in some element Γ, e.g., a PRV or parfactor. The term $gr(\Gamma_{|C})$ denotes the set of instances of Γ with all logvars in Γ grounded w.r.t. constraint C. Next, we define PGMs, which consist of parfactors. A parfactor describes a function, mapping argument values to real values (potentials). Arguments of parfactors are PRVs, compactly encoding patterns, i.e., the function is identical for all instances. A parfactor can be gated, meaning that using a selector the parfactors can be turned on or off [7].

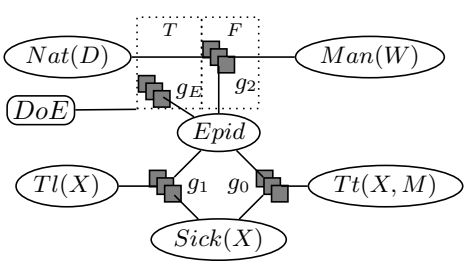

Fig. 1. Graph of M_{ex} ($Tt : Treat, Tl : Travel$)

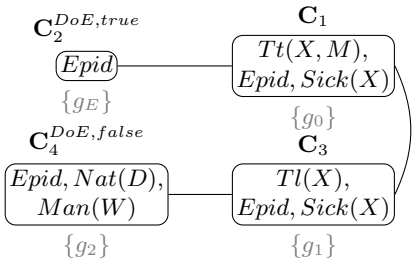

Fig. 2. FO jtree of M_{ex}

Definition 2. *We denote a* parfactor g *by* $\phi(\mathcal{A})_{|C}$ *with* $\mathcal{A} = (A_1, \ldots, A_n)$ *a sequence of PRVs,* $\phi : \times_{i=1}^{n} \mathcal{R}(A_i) \mapsto \mathbb{R}^+$ *a function with name* $\phi \in \Phi$*, and* C *a constraint on the logvars of* $\mathcal{A}$*. Given a selector* S *and parfactors* g_i *(*i *as the iterating index), a gate is denoted by* $\{g_i\}_i^{S,key}$*, which is turned on if* S *has the value key and off if* S *has any other value. Semantically, a gate represents* $(\prod_i g_i)^{\delta(s=key)}$ *with* $\delta(s = key)$ *denoting the Dirac impulse, which is* 1 *if* s *has the value key and* 0 *otherwise. An assignment to a set of selectors* $\mathbf{S}$ *is called a* configuration $\{S = s\}_{S \in \mathbf{S}}$*,* $\mathbf{s}$ *for short. A* PGM $M := \{g_k\}_k \cup \bigcup_{S \in \mathbf{S}} \{g_i\}_i^{S,key}$ *consists of non-gated parfactors* g_k *and gated parfactors* g_i *with selectors* $\mathbf{S}$*. With* Z *as the normalisation constant, the semantics of* M *given* $\mathbf{s}$ *is given by grounding and building a full joint distribution* $P_M(\mathbf{s}) = \frac{1}{Z} \prod_{s \in \mathbf{s}} (\prod_i \prod_{f \in gr(g_i)} f)^{\delta(s=key)} \prod_k \prod_{f \in gr(g_k)} f$.

Let us specify an example PGM for the epidemic scenario, which is depicted in Fig. 1 with $M_{ex}{=}\{g_i\}_{i=0}^{1} \cup \{g_2\}^{DoE,true} \cup \{g_E\}^{DoE,false}$. Parfactors g_0, g_1, and g_2 have eight input-output pairs, g_E has two (omitted). Constraints are $\top$ with some finite domains (omitted). In the graph, PRVs are shown as ellipses and parfactors as boxes with edges between them if the PRV occurs in the parfactor. Gates are depicted as dashed boxes, one gate for g_E and one gate for g_2, both with selector DoE. The gates are mutually exclusive, meaning when one gate is on, the other is off. E.g., the gate for g_2 allows for turning off the connection to causes of an epidemic, e.g., based on value of information.

A query for a PGM M asks for a (conditional) marginal distribution of a grounded PRV. Formally, a query is defined as follows.

Definition 3. *Given a query term* Q*, a configuration* $\mathbf{s}$*, and a set of events* $\mathbf{E} = \{E_j = e_j\}_{j=1}^{m}$*, the expression* $P(Q \mid \mathbf{E}, \mathbf{s})$ *denotes a* query.

Answering a query requires eliminating all instances of PRVs not occurring in the query from a PGM M. Gehrke et al. show that LVE can be used for this, which computes marginals lifted by summing out a representative as in propositional variable elimination and then factoring in isomorphic instances [7]. Given another query, LVE starts with the original model. For efficient multi-query answering, SLJT incorporates gates into the FO jtree of LJT.

2.2 Switched Lifted Junction Tree Algorithm

SLJT efficiently answers *multiple queries* in a PGM by building an FO jtree of the PGM based on selectors. In the following, we examine how SLJT leverages the FO jtree for automatically handling the effects of any given configuration.

Clusters An FO jtree consists of clusters as nodes, which are sets of PRVs directly connected by parfactors. Each cluster is conditionally independent of all other clusters given the PRVs that are shared with neighbouring clusters. Each parfactor is assigned to a cluster that covers its arguments as a *local model*. For SLJT, clusters are based on selector values. Consider the FO jtree with four clusters for M_{ex} in Fig. 2. Cluster $\mathbf{C}_1$ is linked by g_0. Cluster $\mathbf{C}_3$ is linked by g_1. Clusters $\mathbf{C}_2$ and $\mathbf{C}_4$ are based on the selector DoE. $\mathbf{C}_2$ contains $Epid$, based on $DoE = true$, with g_E assigned. $\mathbf{C}_4$ contains $Epid$, $Nat(D)$, and $Man(W)$, based on $DoE = false$, with g_2 assigned. If $DoE(X) = true$, $\mathbf{C}_2$ is switched on. If $DoE(X) = false$, $\mathbf{C}_4$ is switched on. $\mathbf{C}_1$ and $\mathbf{C}_3$ can be thought of as always switched on.

At this point, local models hold state descriptions about their clus- ter PRVs, which are not available at other clusters. To answer queries on an FO jtree, SLJT first performs some preprocessing by making all necessary state descriptions available for each cluster using so-called messages. After message passing, each cluster has all descrip- tions available in its local model and received messages. To answer a query with a query term Q, LJT finds a cluster containing Q and answers $P(Q)$ on the local model and messages with LVE.

Query Answering After construction, local models hold state descriptions on their cluster PRVs not available at other clusters. SLJT uses so-called messages to efficiently distribute the descriptions for correct query answering. Specifically, SLJT takes a PGM M, a configuration $\mathbf{s}$, evidence $\mathbf{E}$, and a set of queries $\mathbf{Q}$ and proceeds as follows: 1. Construct an FO jtree J. 2. Set up $\mathbf{s}$ in J. 3. Enter evidence $\mathbf{E}$ in J. 4. Pass messages in J. 5. Answer queries in $\mathbf{Q}$ using J.

To set up $\mathbf{s}$ in J, SLJT switches clusters in J on and off based on $\mathbf{s}$. Entering $\mathbf{E}$ entails that, at each cluster covering (a part of) $\mathbf{E}$, its local model absorbs $\mathbf{E}$ in a lifted way [20]. Then, SLJT passes messages. A message m from one cluster to a neighbour $\mathbf{C}$ transports state descriptions of its local model and messages from other neighbours to $\mathbf{C}$. SLJT uses LVE to calculate m, passing on the shared PRVs as a query and the local model and respective messages as a model. Messages depend on a given configuration. If a cluster is switched on, SLJT calculates a message based on a cluster's local model and messages from neighbours. If a cluster is switched off, SLJT calculates a message based only on messages from neighbours. The information from the neighbours have to be passed on to the other clusters based on the message passing scheme. For a switched off cluster, the local model of that cluster is turned off, but the incoming messages still need to be proceeded. Thus, SLJT calculates a message solely based on messages from neighbours. Given $DoE = true$ in the FO jtree in Fig. 2, the message from $\mathbf{C}_4$ to $\mathbf{C}_3$ is empty as no other neighbour exist. With

$DoE = false$, the message from $\mathbf{C}_2$ to $\mathbf{C}_1$ is empty. Finally, SLJT answers the queries in $\mathbf{Q}$. To answer a query with a query term Q, SLJT finds a cluster containing Q and answers $P(Q)$ on the local model and messages with LVE.

SLJT allows for exploring a model state by answering queries. For decisions, we incorporate utilities and actions into PGMs and solve MEU problems.

3 Explainable and Explorable Decision Support

For explainable and explorable decision support, we add actions and utilities to PGMs, define the MEU problem in these models, and present LEEDS, which solves the MEU problem along with queries for groups forming along the way.

3.1 PDecGMs: Adding Actions and Utilities to PGMs

To model actions, we could introduce an action PRV with the actions in its range, enlarging parfactors. In contrast, PGMs allow for modelling context-specific independences, which can be seen as an action performed to change a model state externally. Thus, we model actions using gates, allowing for high expressiveness regarding the effect of actions on a model and keeping parfactors small, which has a positive effect on inference complexity. As for utilities, in PDecGMs, PRVs represent utilities, which are identical for groups of indistinguishable objects, leading to utility parfactors, defined as follows.

Definition 4. *Let Φ_u be a set of utility factor names. A parfactor that maps to a utility PRV U is a* utility parfactor g_u*, denoted by $\mu(\mathcal{A})_{|C}$ where C is a constraint on the logvars of $\mathcal{A}$ and μ is defined by $\mu : \times_{A \in \mathcal{A} \setminus \{U\}} \mathcal{R}(A) \mapsto \mathbb{R}$, with name $\mu \in \Phi_u$. The output of μ is the value of U. A* PDecGM *M is a PGM with an additional set M_u of utility parfactors. The term $rv(M_u)$ refers to all probability PRVs in M_u. Let $\{g_{u,i}\}_i^{S,key}$ denote the set of utility parfactors $g_{u,i}$ in a gate with selector S, switched on with value key, and $g_{u,k}$ be ungated utility parfactors. Given a configuration $\mathbf{s}$, M_u represents the combination of all utilities that are turned on, $U_M(\mathbf{s}) = \sum_{s \in \mathbf{s}} (\sum_i \sum_{f \in gr(g_{u,i})} f)^{\delta(s=key)} + \sum_k \sum_{f \in gr(g_{u,k})} f$.*

The semantics already shows how lifting can speed up performance: The calculations for each $f \in gr(g_{u,i})$ are identical, allowing for rewriting the sum over $f \in gr(g_{u,i})$ into a product of $|gr(g_{u,i})| \cdot f$.

Figure 3 shows a PDecGM based on M_{ex}. Compared to the original M_{ex}, we replace the PRV $Treat(X, M)$ with two action gates and a selector $DoT(X)$ to model treatments as actions with different effects. The two actions for the gates with the selector $DoT(X)$ are *treat patient*, T_1, and *do nothing*, T_2. Additionally, the PDecGM contains two action gates with a selector $DoB(X)$, a utility $Util$ (grey diamond), and four utility parfactors (crossed boxes). The two actions for the gates with the selector $DoB(X)$ are *travel ban*, B_1, and *do nothing*, B_2. As $DoB(X)$ and $DoT(X)$ select the executed action (action selector), we call the assignment of $DoB(X)$ and $DoT(X)$ an *action configuration*. Given an action configuration, either parfactors g_{U2}, g_1 or parfactor g_{U1} is on as well as either

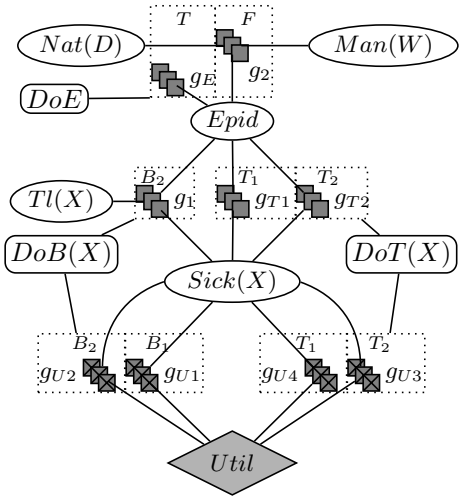

Fig. 3. Graph of the PDecGM M_{ex}

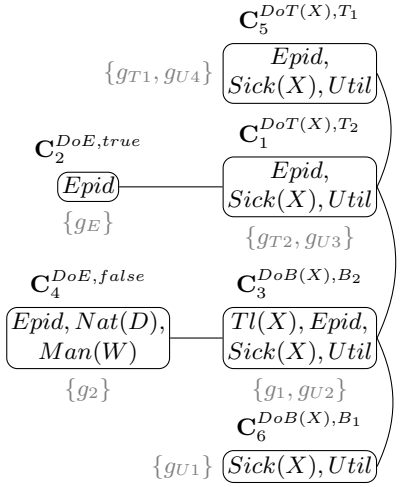

Fig. 4. FO jtree for the PDecGM M_{ex}

parfactors g_{U3}, g_0 or parfactors g_{U4}, g_T are on. All four actions influence the utility, albeit differently based on the condition of patients as represented by the utility parfactors. For example, sick people can infect other people. Hence, the condition of people, i.e., how likely it is that a person is sick, influences the overall utility. Sick people traveling can infect people in other areas, which increases the probability of an epidemic. Further, travelling can also worsen the condition of persons. In case many people are sick, an action is a travel ban. However, a travel ban limits people's freedom. Thus, a travel ban may also negatively influence the utility. In case a person is sick, a medical professional can treat that person. However, doctors have limited time to treat people. Additionally, treating a healthy person could also cause more harm than good. Therefore, the utility values after treating people also have to be chosen carefully.

3.2 Maximum Expected Utility

To define the MEU problem on a PDecGM, we need to define expected utilities in a PDecGM.

Definition 5. *Given a PDecGM M, a query term Q, a configuration* $\mathbf{s}$*, events* $\mathbf{E}$*, the expression $P(Q \mid \mathbf{E}, \mathbf{s})$ denotes a* probability query *for $P_M(\mathbf{s})$. The expression $U(Q, \mathbf{E}, \mathbf{s})$ refers to a* utility *for U_M. Given* $\mathbf{s}$ *and* $\mathbf{E}$*, the* expected utility *of M is defined by*

$$eu(\mathbf{E}, \mathbf{s}) = \sum_{v \in \times_{r \in rv(M)} \mathcal{R}(r)} P(v \mid \mathbf{E}, \mathbf{s}) \cdot U(v, \mathbf{E}, \mathbf{s}) \tag{1}$$

LVE already answers probability queries efficiently in PGMs, which extends to PDecGMs as utility parfactors are ignored when answering a probability query.

LVE also allows for exactly computing an expected utility in a PDecGM based on Eq. (1). The inner product in Eq. (1) calculates a belief state $P(v \mid \mathbf{E}, \mathbf{s})$ and combines it with corresponding utilities $U(v, \mathbf{E}, \mathbf{s})$. By summing over the range of all PRVs of M, one obtains a scalar representing an expected utility. Equation (1) also allows for analysing which constants are associated with high utilities and computing marginal queries for these constants. For groups of indistinguishable constants, one representative query is sufficient, which lifting renders possible. Next, we show that LVE correctly computes expected utilities.

Proposition 1. *Given a configuration* $\mathbf{s}$ *and evidence* $\mathbf{E}$*, LVE correctly computes* $eu(\mathbf{E}, \mathbf{s})$ *in a PDecGM* M*.*

Proof. Computing $eu(\mathbf{E}, \mathbf{s})$ requires LVE to eliminate all non-utility PRVs, i.e., utility PRVs basically form the query terms. Eliminating PRVs in parfactors is correctly implemented through LVE operators [20]. As utilities are PRVs, LVE correctly handles them when applying operators. After eliminating all non-utility PRVs, the remaining parfactor holds the expected utility.

The MEU problem asks for the action configuration leading to the *maximum* expected utility as in Eq. (1), defined as follows:

Definition 6. *Given a PDecGM* M *with fixed non-action selectors* $\mathbf{s}$ *and events* $\mathbf{E}$*, the MEU problem is given by*

$$meu[M \mid \mathbf{E}, \mathbf{s}] = (\arg\max_{\mathbf{a}} eu(\mathbf{E}, \mathbf{s}, \mathbf{a}), \max_{\mathbf{a}} eu(\mathbf{E}, \mathbf{s}, \mathbf{a})) \tag{2}$$

Equation (2) suggests a naive algorithm for calculating an MEU, namely by iterating over all possible action configurations, solving Eq. (1) for each configuration. Since the gates in M are parameterised, the complexity of computing Eq. (2) is no longer exponential in the number of ground actions, enabling tractable inference in terms of domain sizes [15]. Instead of the domain sizes, the complexity is exponential in the number of groups forming due to evidence, which is usually very much lower than the number of constants. Assume that we observe whether a person is sick. Evidence for instances can be in the range of $Sick(X)$: boolean. Thus, X can be split into three groups, with observed values of $true$, of $false$, or no observation. For each group individualy, $DoB(X)$ can be set to either B_1 or B_2. The same holds for $DoT(X)$. Thus, in our example, we need to iterate over 4^1 to 4^3 action configurations depending on evidence.

Since LVE can compute expected utilities, one can also use LVE to solve an MEU problem in a PDecGM. For an exact solution, one constructs all possible action configurations $\mathbf{S}_a$ depending on splits due to evidence and then computes the expected utility of each action configuration. The action configuration that maximises the expected utility is selected. As the utility value is a scalar, we can easily rank them to get the top-k action configurations.

When a configuration is changed to a new one, only parts of a model are turned on or off, while the structure of the remaining model is unchanged. Therefore, while solving the MEU problem, a naive algorithm eliminates the

unchanged structure twice, which is very costly. With more actions, unchanged structures are eliminated even more often. Therefore, we introduce LEEDS as a means to efficiently handle changing configurations.

3.3 LEEDS

LEEDS provides lifted explainable and explorable decision support by outputting top-k action plans and answers to marginal queries for groups of interest. Algorithm 1 shows an outline. Inputs are a PDecGM M, a number k to set the k in top-k, a configuration for non-action selectors $\mathbf{s}$, evidence $\mathbf{E}$, and queries $\mathbf{Q}$. Our example PDecGM in Fig. 3 would be M and k could be 3. There exists one non-action selector, DoE, which we could set to $false$. Evidence may be that two people x_1 and x_2 are travelling, i.e., $Travel(X') = true$ for $\mathcal{D}(X') = \{x_1, x_2\}$. Providing queries allows for registering PRVs of interest in advance, such as $Epid$ or $Sick(X)$. Then, LEEDS can answer marginal queries for propositional randvars like $Epid$ or answer representative queries for PRVs like $Sick(X)$ based on groups and constants identified by LEEDS because of evidence or high influence in expected utilities of Eq. (1).

Given these inputs, LEEDS begins by constructing an FO jtree for M based on selectors comparable to SLJT. Then, LEEDS enters $\mathbf{E}$ and sets up $\mathbf{s}$. LEEDS proceeds with solving the MEU problem, which also includes answering $\mathbf{Q}$. To solve the problem and answer queries respectively, message passing is necessary. Compared to SLJT, LEEDS has to handle utilities in its messages to pass around not only the state descriptions of its standard PRVs but also to distribute information about its utility PRV. LEEDS also has to handle changing action configurations efficiently. Therefore, we first look into how LEEDS handles utilities in FO jtrees. Then, we present how LEEDS adapts to action configurations. Last, we detail how LEEDS proceeds to compile the outputs of top-k action plans and answers to marginal queries for groups of interest.

Utilities and Messages LEEDS builds an FO jtree based on selectors for non-action and action selectors as before. Given a configuration for both types of gates, a message from one cluster to a neighbour is calculated based on incoming messages alone or together with its local model depending on whether the cluster is switched off or on. An open question regards how to handle utilities.

Utilities are modelled as PRVs, which means they can be treated as such in an FO jtree: Clusters are sets of PRVs, including utility PRVs, possibly accompanied by a selector S and a value key. Local models may also contain utility parfactors. Consider the FO jtree in Fig. 4 for the example PDecGM. Compared to the FO jtree of the PGM, the FO jtree of the PDecGM contains two additional clusters $\mathbf{C}_5$ and $\mathbf{C}_6$ for two gates, with $DoB(X) = true$ and $DoT(X) = true$ associated. Additionally, now $\mathbf{C}_1$ and $\mathbf{C}_3$ are also gated. The local model of $\mathbf{C}_6$ contains only a utility parfactor, while the local model of $\mathbf{C}_5$ contains both types of parfactors. Another difference is that $\mathbf{C}_1$ and $\mathbf{C}_3$ now also include utility parfactors. As $DoB(X)$ and $DoT(X)$ can be set to one action for

Algorithm 1 Lifted Explainable and Explorable Decision Support

```
1: function LEEDS(PDecGM M, number k, configuration s, evidence E, queries Q)
2:     Build an FO jtree J for M
3:     Enter evidence E into J
4:     Set up the (non-action) configuration s in J
5:     Construct a Gray sequence of action configurations S_a
6:     · ← empty sequence
7:     Sorted list meu of length k with tuples ( · , 0, ∅) indexed 0 ... k − 1
8:     for each a ∈ S_a do
9:         Adapt the selectors in J to a
10:        Adapt messages in J
11:        Calculate expected utility u
12:        if u > eu(meu[k − 1]) then
13:            V ← Answer Q for groups in J
14:            Update meu with (a, u, V)
15:    return meu
```

some instances and to the other action for the remaining instances, based on splits due to evidence, the four clusters can be switched on at the same time.

Message passing follows the same idea as before: If a cluster is switched off in a given configuration, messages are calculated without its local model. The difference lies in the messages themselves, which also transport information about utilities. Thus, LEEDS calculates a probability query over shared PRVs with its neighbour (as before) and possibly a utility as given in Def. 5 over shared utility PRVs. After such a message pass, LEEDS could already answer marginal queries $P(Q \mid \mathbf{E}, \mathbf{s}, \mathbf{a})$ or expected utility queries based on the current configurations $\mathbf{s}, \mathbf{a}$, and evidence $\mathbf{E}$. LEEDS answers $P(Q \mid \mathbf{E}, \mathbf{s}, \mathbf{a})$ by finding a cluster containing Q and eliminating all terms, except Q from the local model and messages, ignoring utility parfactors. For expected utilities, LEEDS answers a conjunctive query over utilities and sums out all non-utility PRVs from corresponding local models and received messages to compute Eq. (1). In our example, we only have one utility PRV, $Util$, and thus, a single term query.

Next, we show how LEEDS solves the MEU problem adaptively.

Solving the MEU Problem Solving the MEU problem requires maxing over action configurations in a PDecGM. Again, naively, one would construct all possible action configurations $\mathbf{S}_a$, which LEEDS also has to do to obtain the top-k action configurations, over the groups elicited by evidence. The above setting of evidence in the form of $Travel(X') = true$ for $\mathcal{D}(X') = \{x_1, x_2\}$ leads to two groups of X constants in the underlying model. Given the four possible actions B_1, B_2, T_1, T_2 in gates with X as a parameter, there exist $4^2 = 16$ possible action configurations. For each configuration $\mathbf{a} \in \mathbf{S}_a$, LEEDS would set up $\mathbf{a}$ in the FO jtree, pass messages, and ask for the expected utility. At the end, LEEDS would output those $\mathbf{a}$ with their expected utilities that have the k highest expected utilities among all configurations in $\mathbf{S}_a$.

This method would compute each expected utility value for different configurations from scratch. But, we can arrange the configurations in $\mathbf{S}_a$ s.t. only one selector for one group has a changed assignment, similar to Gray codes in coding theory [10]. Let us call such a sequence a *Gray sequence*. If a configuration changes incrementally, many messages are still valid. Performing query answering with changing inputs falls under adaptive inference. For inputs that change incrementally, adaptive inference aims at performing inference more efficiently than starting from scratch. The adaptive LJT (aLJT) performs adaptive inference, handling changes in model or evidence, by adapting an FO jtree to changes in a model and performing evidence entering and message passing adaptively [5]. We use the adaptive message passing scheme of aLJT to re-calculate only those messages necessary based on the clusters that changed their status of being on or off. As selector assignments change in only one place from one configuration to the next, the changes in the FO jtree are locally restricted, which means that only messages outbound from these clusters need adapting.

In line 5 of Alg. 1, LEEDS generates a Gray sequence of action configurations. LEEDS executes the configurations in the order of the sequence in the following for-loop, adapting selectors and messages accordingly. During the first iteration, LEEDS has to set the action configuration for the first time and thus performs a full message pass as no previous setting and pass exist to adapt. In line 10, LEEDS calculates the expected utility given the current action configuration $\mathbf{a}$ and then proceeds to test whether $\mathbf{a}$ belongs to the current top-k configurations, which includes compiling further outputs for $\mathbf{a}$ if part of the top-k.

Next, we look into how LEEDS performs this compilation of outputs.

Compiling the Outputs The outputs of LEEDS are the top-k action plans, i.e., action configurations, their expected utilities as a form of explanation, and the answers to the registered query PRV for groups or specific constants of interest. To collect the outputs, a helper variable *meu* stores the current top-k MEU solutions, their corresponding expected utilities, and answers to registered marginal queries. The queries are instantiated with representatives for groups of indistinguishable constants, occurring due to evidence. Algorithmically, the variable is a list of triples $(\mathbf{a}, u, \mathbf{V})$ of an action configuration $\mathbf{a}$, an expected utility u, and a set of answers $\mathbf{V}$. The list is sorted to easily check if a new action configuration has an expected utility in the top-k.

The body of the if-statement in line 11 in Alg. 1 concerns the compilation of outputs in *meu*. It compiles the outputs here as the FO jtree is prepared accordingly. Given the current action configuration $\mathbf{a}$ with expected utility u, LEEDS tests whether u is higher than the lowest utility at the last position of the sorted list. If so, LEEDS calculates marginals for the registered propositional randvars and marginals for specific constants as well as groups occurring in $\mathbf{a}$ or with large influence in u. To do so, LEEDS has to look at the constraints in $\mathbf{a}$ to find groups and analyse Eq. (1) for calculating u, the result could be $\{x_1, x_2\}$ and $\{x_3\}$ in our example as a result of evidence. Given the groups identified in this way and the registered queries for $Epid$ and $Sick(X)$, LEEDS answers the queries $Epid$, $Sick(x_1)$ as a representative of the group $\{x_1, x_2\}$,

and $Sick(x_3)$. As the FO jtree is prepared to answer any marginal query given the current configuration and evidence, no additional pre-processing has to be performed to answer these queries and LEEDS is able to answer each query on one of the clusters (instead of the complete model). LEEDS stores the answers to the queries in a set $\mathbf{V}$. The next step regards updating *meu* with the new triple $(\mathbf{a}, u, \mathbf{V})$. Updating *meu* entails finding the position i to store the triple at, deleting the last element $meu[k-1]$, and adding $(\mathbf{a}, u, \mathbf{V})$ at i.

After iterating over all action configurations in $\mathcal{S}_a$, *meu* contains the top-k action configurations including their expected utility and the answers to the registered query PRV for groups or specific constants of interest depending on the corresponding configuration. At the end, LEEDS returns *meu*, which contains the technical output for explainable and explorable decision support. Specifically, *meu* is a list of length k of triples, each containing an action plan, an expected utility, and further probability distributions. Together, the triples provide a complete picture by providing k alternative action plans with high expected utility as well as information about the context in terms of probability distributions about PRVs that the user deems crucial for a decision.

Output Interpretation LEEDS produces a list of length k of triples, each containing an action plan, an expected utility, and further probability distributions. Together, the triples provide a complete pic- ture by providing k alternative action plans with high expected utility as well as information about the context in terms of probability distri- butions about PRVs that the user deems crucial for a decision. Given an output, one can start interpreting the results.

For our running example with $k = 3$, $DoE = false$, and $Travel(X') = true$ for $\mathcal{D}(X') = \{x_1, x_2\}$, *meu* may contain the following items with exemplary values, with X referring to the remaining X constants and x' and x to representative instances of X' and X, respectively (per item with expected utility u: action plan, answers):

- Position 0: expected utility of 100
  - $DoB(X') = B_2$, $DoB(x_3) = B_1$, $DoT(X') = T_2$, $DoT(X) = T_1$
  - $(Epid = true \to 0.3, Epid = false \to 0.7)$,
    $(Sick(x') = true \to 0.2, Sick(x') = false \to 0.8)$,
    $(Sick(x) = true \to 0.7, Sick(x) = false \to 0.3)$
- Position 1: expected utility of 95
  - $DoB(X') = B_1$, $DoB(x_3) = B_1$, $DoT(X') = T_2$, $DoT(X) = T_1$
  - $(Epid = true \to 0.2, Epid = false \to 0.8)$,
    $(Sick(x') = true \to 0.1, Sick(x') = false \to 0.9)$,
    $(Sick(x) = true \to 0.8, Sick(x) = false \to 0.2)$
- Position 2: expected utility of 40
  - $DoB(X') = B_1$, $DoB(x_3) = B_1$, $DoT(X') = T_1$, $DoT(X) = T_1$
  - $(Epid = true \to 0.2, Epid = false \to 0.8)$,
    $(Sick(x') = true \to 0.3, Sick(x') = false \to 0.7)$,
    $(Sick(x) = true \to 0.8, Sick(x) = false \to 0.2)$

The first two plans do not vary greatly in their expected utility, whereas the third plan has an expected utility less than half of the expected utility of the preceding plan. Since the first two plans have similar expected utilities, one may want to compare the plans and realise that they only differ in the action for one group, namely, the travel ban for $\{x_1, x_2\}$, which only the second plan proposes (B_1). Given the results, a user could now consider additional information beyond the data encoded in the PDecGM for deciding which action plan to execute. E.g., on the one hand, a travel ban for all X may be easier to implement than a travel ban only for subgroups. On the other hand, since x_1 and x_2 are observed to be travelling, a travel ban may hit this group especially hard. Taking into consideration the results of the additional queries, the probability of $Epid = true$ is lower with the second plan, the same can be said about the probabilities of $Sick(X) = true$ for both x' and x_3. Without investing effort to finding top-k action plans and answering additional queries based on the result of an action plan, only the first action plan with an expected utility of 100 would be returned, providing few information and no alternatives.

Beyond the information compiled in meu, a user could still ask follow-up queries, from different queries for specific top-k configurations to action configurations apart from those in the top-k. The result in meu also allows for analysing which information is actually used under an action configuration. Given a travel ban, i.e., $DoB(X) = B_1$ for all groups of X, evidence on $Travel(X)$ is not used.

4 Empirical Case Study

We have implemented a prototype of LEEDS, based on the LVE[3] and the LJT[4] implementations available. Since LEEDS uses an FO jtree and LVE for its calculations, LEEDS outperforms grounded versions as well as LVE for a set of queries as shown in [20,3]. Thus, we concentrate on the following two claims: First, LEEDS performs inference faster in a PDecGM, modelling actions with gates, than LJT in a PM, modelling actions using PRVs. Second, since LEEDS ensures a minimum number of calculations to solve MEU problems and answering probability queries, by only adapting the model to changing configurations and thereby reusing as many computations as possible, LEEDS is faster than solving the MEU problem in a non-adaptive way.

We consider the running example PDecGM as input with domain sizes of W and D fixed to 50 and the domain size of X set to 10, 100, and 1000. We consider two groups in the X constants. As registered queries, we consider $Epid$ and $Sick(X)$. Since there are two groups in the X constants and four possible actions, there are 16 action configurations to consider. Computing the same MEU grounded would require 4^{10} to 4^{1000} action configurations to consider instead of 4^2. We compare runtimes of implementations of LEEDS (`leeds`), of LEEDS without adaptation (`nogray`), and of LJT on a PM with action PRVs (`actprvs`). The last one requires providing the action configurations as evidence to LJT.

[3] `https://dtai.cs.kuleuven.be/software/gcfove`
[4] `https://www.ifis.uni-luebeck.de/index.php?id=518`

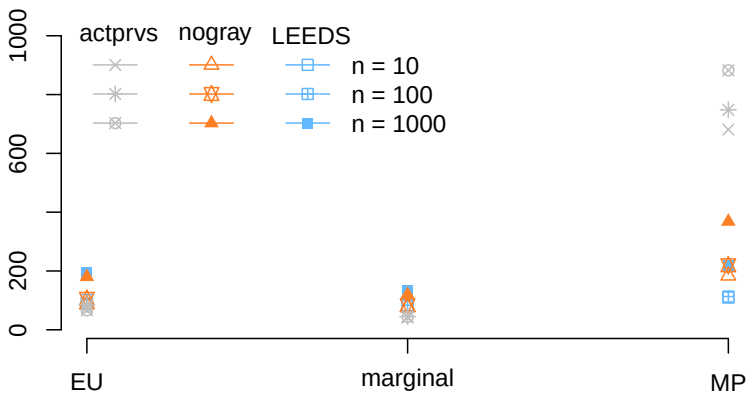

Fig. 5. Runtimes in miliseconds

Runtimes are collected and averaged over 10 runs on a virtual machine with 16GB working memory and no additional load on the machine.

Figure 5 shows runtimes for `leeds`, `nogray`, and `actprvs`. In the figure, we can see that answering expected utility queries and marginal queries roughly take the same time to answer across the different approaches. The runtimes are roughly the same as each of the approaches answers the queries on a single par-cluster, which should have roughly the same amount of PRVs leading to similar runtimes. Additionally, we can see that as to be expected for lifted algorithms, increasing domain sizes does not have an exponential influence on runtimes. The main difference between the approaches is during message passing. `Nogray` roughly takes about twice as long as `leeds`. With the Gray code, `leeds` roughly reuses 50% of the messages while changing action assignments leading to the speed up in comparison to `nogray`. In comparison to `actprvs`, even for this small model, `leeds` achieves a speed up of nearly one order of magnitude due to adaptive inference and a more compactly represented model.

5 Conclusion

We present LEEDS to provide effective decision support beyond simple explanations. To support a user in their decision, LEEDS outputs top-k action plans and answers to representative marginal queries for groups of interest. To this end, we define PDecGMs as the modelling formalism, incorporating parameterised utilities as well as parameterised gates for actions. LEEDS then takes a PDecGMs, a number k, evidence, and possibly registered queries as inputs. LEEDS solves the MEU problem in PDecGMs exactly and in a lifted way, reusing computations under varying configurations, and answers registered queries. Areas such as health care can benefit from the lifting idea for many patients and the decision support beyond explanations.

Using LEEDS as a basis, we are looking into finding attributable approximations for further speed-up. Furthermore, we are working on extending LEEDS to the temporal case to support lifted sequential decision making under uncertainty.

Acknowledgements The research of MG was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2176 'Understanding Written Artefacts: Material, Interaction and Transmission in Manuscript Cultures', project no. 390893796. The research was conducted within the scope of the Centre for the Study of Manuscript Cultures (CSMC) at Universität Hamburg.

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. Apsel, U., Brafman, R.I.: Extended Lifted Inference with Joint Formulas. In: Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence. pp. 11–18. AUAI Press (2011)
3. Braun, T.: Rescued from a Sea of Queries: Exact Inference in Probabilistic Relational Models. Ph.D. thesis, University of Lübeck (2020)
4. 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)
5. Braun, T., Möller, R.: Adaptive Inference on Probabilistic Relational Models. In: Proceedings of the 31st Australasian Joint Conference on Artificial Intelligence. Springer (2018)
6. Braun, T., Möller, R.: Parameterised Queries and Lifted Query Answering. In: Proceedings of IJCAI 2018. pp. 4980–4986 (2018)
7. Gehrke, M., Braun, T., Möller, R.: Efficient Multiple Query Answering in Switched Probabilistic Relational Models. In: Proceedings of AI 2019: Advances in Artificial Intelligence. Springer (2019)
8. Gehrke, M., Braun, T., Möller, R., Waschkau, A., Strumann, C., Steinhäuser, J.: Lifted Maximum Expected Utility. In: Artificial Intelligence in Health. pp. 131–141. Springer International Publishing (2019)
9. 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)
10. Gray, F.: Pulse Code Communication (1953), u.S. Patent 2,632,058
11. Joshi, S., Kersting, K., Khardon, R.: Generalized First Order Decision Diagrams for First Order Markov Decision Processes. In: IJCAI. pp. 1916–1921 (2009)
12. 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)
13. Minka, T., Winn, J.: Gates. In: Advances in Neural Information Processing Systems. pp. 1073–1080 (2009)
14. Nath, A., Domingos, P.: A language for relational decision theory. In: Proceedings of the International Workshop on Statistical Relational Learning (2009)

15. Niepert, M., Van den Broeck, G.: Tractability through exchangeability: A new perspective on efficient probabilistic inference. In: AAAI. pp. 2467–2475 (2014)
16. Poole, D.: First-order probabilistic inference. In: Proceedings of IJCAI. vol. 3, pp. 985–991 (2003)
17. de Salvo Braz, R., Amir, E., Roth, D.: Lifted First-order Probabilistic Inference. In: IJCAI-05 Proc. of the 19th International Joint Conference on AI (2005)
18. Sanner, S., Kersting, K.: Symbolic Dynamic Programming for First-order POMDPs. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence. pp. 1140–1146. AAAI Press (2010)
19. Stalnaker, R.: Knowledge, Belief and Counterfactual Reasoning in Games. Economics & Philosophy **12**(2), 133–163 (1996)
20. 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)
21. Van den Broeck, G., Taghipour, N., Meert, W., Davis, J., De Raedt, L.: Lifted Probabilistic Inference by First-order Knowledge Compilation. In: IJCAI-11 Proceedings of the 22nd International Joint Conference on Artificial Intelligence. pp. 2178–2185. IJCAI Organization (2011)
22. Wemmenhove, B., Mooij, J.M., Wiegerinck, W., Leisink, M., Kappen, H.J., Neijt, J.P.: Inference in the Promedas Medical Expert System. In: Conference on Artificial Intelligence in Medicine in Europe. pp. 456–460. Springer (2007)