Learning Probabilistic Decision Graphs

Manfred Jaeger [a] Jens D. Nielsen [a] Tomi Silander [b]

[a] *Institut for Datalogi*
Aalborg Universitet
Fredrik Bajers Vej 7,
9220 Aalborg Ø, Denmark

[b] *Complex Systems Computation Group*
Helsinki Institute for Information Technology
P.O.Box 9800,
FIN-02015 HUT, Finland

Abstract

Probabilistic decision graphs (PDGs) are a representation language for probability distributions based on binary decision diagrams. PDGs can encode (context-specific) independence relations that cannot be captured in a Bayesian network structure, and can sometimes provide computationally more efficient representations than Bayesian networks. In this paper we present an algorithm for learning PDGs from data. First experiments show that the algorithm is capable of learning optimal PDG representations in some cases, and that the computational efficiency of PDG models learned from real-life data is very close to the computational efficiency of Bayesian network models.

Key words: Probabilistic models, Learning

1 Introduction

Probabilistic decision graphs (PDGs) [1,2] are a graphical representation language for probability distributions that is based on the representation paradigm of ordered binary decision diagrams [3]. PDGs were originally conceived for applications in automated verification of probabilistic systems [1]. An initial study of their potential strengths as a representation language also for AI applications was conducted in [2]. The main result of that study was that from a computational complexity

Email addresses: `jaeger@cs.aau.dk` (Manfred Jaeger), `dalgaard@cs.aau.dk` (Jens D. Nielsen), `tsilander@hiit.fi` (Tomi Silander).

point of view, PDGs are always as efficient for probabilistic inference as Bayesian networks, and for some types of probability distributions they are more efficient.

These theoretical results leave the question open, how PDG representations of a given probabilistic domain can be found in practice; in particular, whether PDGs can be learned automatically from data. This question is taken up in the present paper. We describe two approaches for learning PDGs from data. The first approach is a score-based learning procedure that constructs a PDG by a (partly randomized) search in the space of PDG structures. The second approach is a hybrid approach, in which we first learn a Bayesian network, compute its junction tree, then compile the junction tree into a PDG, and finally apply learning techniques to optimize the constructed PDG.

For both approaches we compare the PDG models with Bayesian networks learned from the same datasets. The basis for the comparison is the efficiency/accuracy trade-off of probabilistic inference in the learned models.

A comparison of learned probabilistic models in different representation frameworks raises some methodological questions. Often score functions like BIC or MDL score are used as quality measures for learned models [4]. However, neither does it seem safe to use such functions as a basis for comparison across different representation languages, nor would we want to commit to any particular score function. For this reason, we base our comparison on Size-Likelihood curves that represent the available range of possible efficiency/accuracy trade-offs in the models from the different languages.

In the following section we briefly introduce the language of PDGs, and review some of their essential properties. Section 3 describes our methodological approach of SL-curves, and its relation to ROC analysis for classifier performance. Section 4 describes our pure learning algorithm for PDGs, and experimental results comparing the learned models with learned Bayesian network models. Section 5 describes our hybrid approach for PDG learning, and presents experimental results.

2 Probabilistic Decision Graphs

In this section we briefly review by an example the basic definitions and properties of PDGs. Formal definitions can be found in [2,5].

Like a Bayesian network, a PDG is a graphical representation of a joint distribution for a set of discrete random variables. Figure 1 shows on the right an example PDG defining a distribution for binary random variables $X = X_1, \ldots, X_6$. The graphical structure of the PDG is defined in two stages: first, one defines a forest (a set of trees) over a set of nodes labeled with the given random variables. This forest is

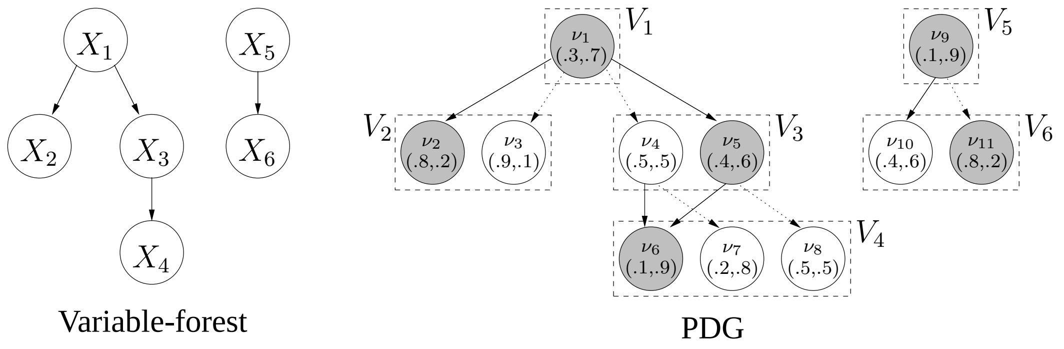


Fig. 1. Probabilistic Decision Graph with underlying forest and nodes reached by $(1, 0, 1, 1, 0, 0)$

shown in the left part of Figure 1. Then, each node X_i in the forest is expanded into a set V_i of nodes, and a node $\nu \in V_i$ is connected as follows: for each successor X_j of X_i in the variable tree containing X_i, and each possible value of X_i, there exists exactly one outgoing edge of ν leading to a node $\nu' \in V_j$. We denote by $succ(X_i)$ the set of variables that are (direct) successors of X_i, and by $desc(X_i)$ the set of descendants of X_i. The resulting structure is a rooted directed acyclic graph (rdag) for every tree in the original variable-forest. In our example all variables are $\{0, 1\}$-valued, so that each node ν contains two outgoing edges for each successor variable in the variable-forest structure. Edges corresponding to value 0 here are indicated by dotted lines, edges corresponding to value 1 by solid lines. Finally, a PDG is obtained by annotating each node $\nu \in V_i$ with a probability distribution over the possible values of X_i.

Each joint instantiation of the variables determines a sub-graph in the PDG that is a forest of the same structure as the underlying variable-forest. In figure 1 the nodes of the forest corresponding to the instantiation $X_1 = 1, X_2 = 0, X_3 = 1, X_4 = 1, X_5 = 0, X_6 = 0$ are shaded. We say that these nodes are *reached* by the given instantiation. The PDG now defines the probability of the instantiation as the product of all the probability assignments to the values of the instantiation according to the distributions at the nodes reached by the instantiation. In our example:

$$P((X_1, \ldots, X_6) = (1, 0, 1, 1, 0, 0)) = .7 \cdot .8 \cdot .6 \cdot .9 \cdot .1 \cdot .8 = 0.024192.$$

The structure of a PDG encodes certain (conditional) independence relations: first, the joint distribution of the variables contained in one tree of the underlying variable-forest is independent from the joint distribution of the variables in another tree. The structure of a single rdag encodes conditional independence relations among the variables contained in the tree for this rdag. These independence relations are not characterized as for Bayesian networks in terms of subsets of variables, but in terms of *partitions* of the state space: each node set V_i defines a partition of the state space (the set of all complete instantiations) into the sets of instantiations that reach the same node in V_i. In our example, the nodes V_4 partition the state space into the sets of instantiations $\{X_3 = 1\}, \{X_1 = 0, X_3 = 0\}$ and $\{X_1 = 1, X_3 = 0\}$. Like in this example, the partition corresponding to V_i always is determined by the values of the ancestors of X_i in the variable tree of X_i. The conditional independence relations encoded by a PDG now are:

$$P(X_i \mid X \setminus \{X_i, desc(X_i)\}) = P(X_i \mid V_i) \tag{1}$$

Such partition-based independence relations can correspond to context-specific independencies in the sense of [6]. In our example, for instance, the independence relation (1) applied to $X_i = X_4$ essentially means that X_4 is independent of X_1 given that $X_3 = 1$ (because independent of the value of X_1, instantiations with $X_3 = 1$ will reach node ν_6 in V_4). However, there is no exact match between our partition-based independence relations and context-specific independencies. Furthermore, it can be shown that the class of independence relations that can be encoded with PDGs is incomparable to the class of independence relations that can be encoded with Bayesian networks, i.e. each of these two representation languages can encode independence relations that cannot be encoded by the other language. For more detailed information on independence relations encoded by PDGs the reader is referred to [5].

Based on a PDG representation some key probabilistic inference problems are solvable in linear time. This includes the computation of posterior marginal distributions for all random variables given an instantiation of some of the variables in the PDG, and the computation of the most probable explanation, i.e. the most probable full instantiation given a partial instantiation. When using BN representations, the complexity of these inference tasks is linear in the size of the junction tree constructed from the BN (see e.g. [7] for probabilistic inference using junction trees). Since in this paper we are looking at graphical representations of probability distributions mostly as computational data structures for inference (as opposed to e.g. causal models), we shall identify BNs essentially with their junction trees.

It was shown in [2, Theorem 4.1] that there is a linear transformation from junction trees into equivalent PDGs. On the other hand, there exist distributions for which a compact PDG representation, but no compact junction tree representation exists. An example for such a distribution is the joint distribution of $n + 1$ binary random variables, n of which are independently and uniformly distributed, and the $(n+1)$st represents a *parity* bit that is deterministically defined by the other variables as $X_{n+1} = \sum_{i=1}^{n} X_i \text{mod } 2$. For this distribution PDG representations of size $O(n)$ can be constructed, but all junction tree representations are exponential in n. When the set of variables is fixed, thus, PDGs are a more efficient representation language than junction trees. For the parity distribution one can also construct linear size junction tree representations by introducing suitable additional (hidden) variables. This is true in general: using suitable augmenting sets of hidden variables, one can always also define a linear transformation from PDGs to junction trees [2, Theorem 4.3].

In some sense, then, PDGs and junction trees, and hence Bayesian networks, provide computationally equally efficient representations of probability distributions.

However, the necessary introduction of hidden variables can be a major obstacle for obtaining efficient Bayesian network representations when the model is to be learned from data, since, so far, no reasonably general and effective ways of automatically learning hidden variables are known. This also indicates the challenge posed by learning PDGs: learning optimal PDGs partially subsumes the problem of learning hidden variables.

3 Method of Comparison

It is our goal to compare the computational efficiency and accuracy of PDG and BN representations when models are learned from real data. For this one has to specify what kind of inference tasks a model is expected to support. Our comparison is based on the assumption that the probabilistic model will be needed to support the exact computation of arbitrary posterior marginals, i.e. we will want to enter evidence $E_1 = e_1, \ldots, E_k = e_k$ (abbreviated $\boldsymbol{E} = \boldsymbol{e}$) for arbitrary subsets $\{E_1, \ldots, E_k\} \subseteq \{X_1, \ldots, X_n\}$ of observed variables, and then compute the posterior marginal of an unobserved variable X_j. This is the classic inference task for Bayesian networks. However, there are also more specialized tasks one can use a Bayesian network for (e.g. classification tasks, where always all but one variable are observed), or one can perform approximate inference. Such more specialized tasks would require a different method of comparison from the one we here pursue.

As explained in the previous section, computational efficiency of BNs and PDGs can be measured by the size of the BN's junction tree, respectively the size of the PDG. Ideally, one would for BNs always consider the smallest junction tree for any given BN. Since it is computationally infeasible to compute a minimal junction tree, we base our comparisons on the size of the junction trees generated by the B-course system [8], which implements the junction tree construction described in [9].

Measuring the accuracy of a model is more difficult. For a single inference, we can measure the accuracy by the discrepancy between the computed posterior marginal for X_j given $\boldsymbol{E} = \boldsymbol{e}$, and the true posterior marginal in the underlying distribution P. If P^M denotes the distribution defined by model M, and we follow the common approach of measuring discrepancy by cross-entropy (*CE*), then the in-accuracy of a computed posterior is given by $\mathit{CE}(P(X_j \mid \boldsymbol{E} = \boldsymbol{e}), P^M(X_j \mid \boldsymbol{E} = \boldsymbol{e}))$. The expected discrepancy, given that we observe variables $\boldsymbol{E}$, then is

$$\sum_{\boldsymbol{e}} P(\boldsymbol{E} = \boldsymbol{e})\mathit{CE}(P(X_j \mid \boldsymbol{E} = \boldsymbol{e}), P^M(X_j \mid \boldsymbol{E} = \boldsymbol{e})). \qquad (2)$$

It follows from well-known properties of CE that $CE(P, P^M)$ is an upper bound for (2) (e.g. [10, Theorem 2.5.3]). Thus, $\mathit{CE}(P, P^M)$ is a uniform upper bound for the expected discrepancy between computed and actual posterior marginal, independent of the set of observed variables. When learning from real data D the true underlying distribution P is not known. We therefore have to use the empirical distribution P^D defined by D (or, more often, defined by a subset of D reserved for valuation purposes) as an approximation for P. One can easily derive that

$$CE(P^D, P^M) = -H(P^D) - (1/|D|)L(M, D), \tag{3}$$

where $H(P^D)$ is the entropy of P^D and $L(M, D)$ is the log-likelihood of D under P^M. Seeing that $CE(P^D, P^M)$ is an (approximate) upper bound on expected inference in-accuracy, and the right-hand side of (3) depends on M only through $L(M, D)$, we obtain that $L(M, D)$ can be interpreted as a measure for expected accuracy for inference based on model M.

Model size and accuracy can be combined into an overall model score. Popular scores like MDL or BIC scores are just weighted combinations of size and log-likelihood measures (though the underlying philosophy for taking size into account is usually not based on measuring inference efficiency). For example, the BIC score of a model M relative to data D is given by $(1 - \lambda)L(M, D) - \lambda|M|$, where $\lambda = \textit{log} \mid D \mid / (2+ \mid D \mid)$.

For the purpose of our comparative study, there is no good reason to commit to one particular overall score function. Instead we report our results in the form of *Size-Likelihood (SL) - curves* that show what range of possible size/likelihood combinations are obtainable by models from the different classes. These SL-curves are similar to ROC-curves [11] that are often used to report the performance of classifiers by plotting the combinations of true positive rates and false positive rates that are obtainable for a classifier through different settings of some tuning parameter. A ROC curve describes the performance of a classifier without committing in the evaluation to any particular gain/loss structure of the classification problem (which amounts to assigning different weights to true and false positive rates).

Figure 2 shows a somewhat idealized example of an SL-curve. The solid curve shows the range of size/likelihood values that are obtainable by a class of models for a fixed data set, which, when the models are generated by a learning procedure, would be the training data. However, complex models that obtain a high likelihood score on the training data will tend to overfit the training data, and hence obtain a lower likelihood score on test data. The dashed curve shows the possible development of the likelihood score of models of increasing size when evaluated over a separate test set.

In the context of ROC analysis, one obtains that, given a specific cost function, all classifiers obtaining equal expected loss l lie on on a straight line, and lines corresponding to different expected losses are parallel. Analogously, we have for SL analysis that constant BIC or MDL scores correspond to parallel straight lines in SL-space. Figure 2 indicates three of such equi-score lines for some such a score that is a weighted combination of size and likelihood. However, BIC/MDL type scores may not always be the most appropriate. Consider, for example, the situation where the model is needed for a resource-bounded or time-critical application. In that case there might be a strict upper bound on the model size, but models within these bounds would be scored only according to their accuracy (i.e. there is no bonus for staying below the upper size bound). Models obtaining equal score in such a setting are characterized by horizontal lines in SL-space that extend to the maximally allowed size.

The relevant part of SL-space is effectively bounded by two extreme points: the *independence model* models all random variables as independent. This model has minimal size (in basically every conceivable representation framework it will require for its specification n parameters, assuming the state space is generated by n binary variables), and likelihood score $L(M_{\mathit{indep}}, D)$. Models with lower likelihood score could obviously be constructed, but they would hardly have to be considered in practice. At the other extreme, one can construct a model that represents the empirical distribution of the data precisely. In most cases this cannot be done except by an explicit enumeration of the probabilities of all 2^n states, which thus gives us a model M_{emp} of size 2^n (again, this would be the same in all representation frameworks).

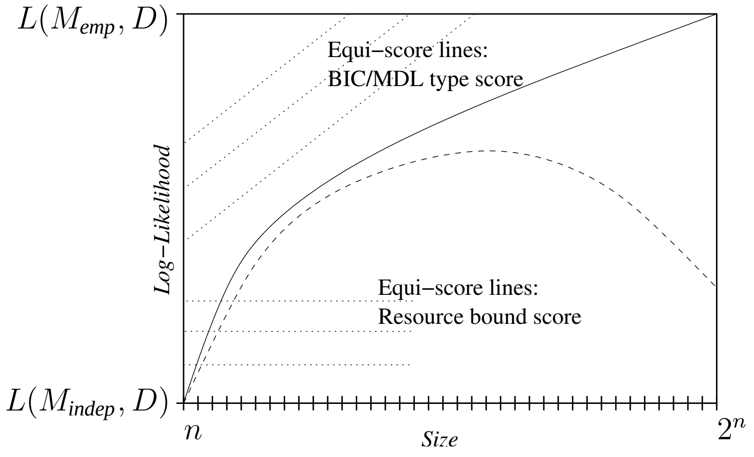


Fig. 2. SL-curves

4 Learning PDGs

4.1 The Algorithm

We use a *score-based* approach to learning PDGs from data using the generic score function:

$$S_\lambda(M) := (1 - \lambda)L(M, D) - \lambda|M| \tag{4}$$

By varying λ we learn models that yield different points in SL-space. Each setting of λ corresponds to the slope of the parallel, linear equi-score lines in SL-space.

```
Procedure Learn($D$)
1: $\mathbf{F} := \emptyset$ % Population of forest structures
2: $\mathbf{G} := \emptyset$ % Population of PDGs
3: for each testlevel $t$ do:
4:   $\mathbf{F} := \mathbf{F} \cup \{$LearnForest($D,t$)$\}$
5: for $\lambda$ in $\lambda_{max}, \ldots, \lambda_{min}$ do:
6:   for each $F \in \mathbf{F}$ do:
7:     $\mathbf{G} := \mathbf{G} \cup \{$LearnPDG($F, \lambda$)$\}$
8:   collect $\mathbf{F}_{low}$ from $\mathbf{F}$
9:   $\mathbf{F} := \mathbf{F} \backslash \{\mathbf{F}_{low}\}$
10:  output $argmax_{G \in \mathbf{G}}(S_\lambda(G))$
11:  $\mathbf{G} := \emptyset$
```

```
Procedure LearnForest($D,t$)
1: $\mathbf{X}$ :=variables from $D$
2: $F := \emptyset$
3: $H$ :=DepGraph($\mathbf{X}, t, \emptyset$)
4: for each $\mathbf{C} \in CC(H)$ do:
5:   $X_i$ :=rndVar($\mathbf{C}$)
6:   $V_i := \{\nu_i\}$
7:   $desc(X_i) := \mathbf{C} \setminus \{X_i\}$
8:   $T_i$ :=tree w. $V_i$ as root
9:   $F := F \cup \{T_i\}$
10:  repeat:
11:    Grow($T_i, t$)
12:    LearnPDG($F, \lambda_{max}$)
13:  until $T_i$ is full-grown
14: return $F$
```

```
Procedure LearnPDG($F, \lambda$)
1: $G$ :=minimal PDG for $F$
2: repeat for all trees in $G$:
3:   split nodes top-down
4:   merge nodes bottom-up
5:   redirect edges bottom-up
6: until $S_\lambda(G)$ did not change
```

```
Procedure Grow($T, t$)
1: for each leaf $V_i$ of $T$ do:
2:   $H$ :=DepGraph($desc(X_i), t, V_i$)
3:   for each $\mathbf{C} \in CC(H)$ do:
4:     $X_j$ :=rndVar($\mathbf{C}$)
5:     $V_j := \{\nu_j\}$
6:     attach $\nu_j$ below $V_i$
7:     $desc(V_j) := \mathbf{C} \setminus \{X_j\}$
```

Table 1
PDG learning procedures

Optimizing S_λ for large λ is easier than optimizing for small λ, as the strong bias towards smaller models reduces the effective size of the search-space.

The structure search for PDGs decomposes into two parts: the search for a variable forest, and the search for the exact PDG structure based on that variable forest. One may expect that when we obtain a high scoring PDG for some λ value, then the variable forest underlying this PDG will also support high scoring PDGs for other λ-values (this expectation has been corroborated with minor qualifications in our experiments). Together with the observation above that it is much easier to learn PDGs when scoring with large λ-values, this leads us to the following population-based approach to learning (cf. procedure `Learn` of table 1): first a population of candidate variable forests is created (`Learn`, lines 3-4). Starting with the largest λ in a set of λ-parameters, each variable forest is refined into an actual PDG using the `LearnPDG` sub-routine, which optimizes S_λ. Forests F for which `LearnPDG(`F, λ`)` yields a PDG achieving poor score are collected in set $\mathbf{F}_{low}$ and removed from the population (lines 8-9). The subroutine `LearnForest` generates the initial forests by a *constraint-based* approach that builds a forest encoding certain conditional independence relations we find in the data. We now describe the two key subroutines `LearnPDG` and `LearnForest` in greater detail. `LearnPDG(`F, λ`)` traverses the space of different PDGs over the forest F in the search for an optimal PDG, w.r.t score S_λ. Three different local operators define the traversal: *split*, *merge* and *redirect*.

The *split*-operator takes a node with $n > 1$ incoming edges, and replaces it with n nodes, one for each incoming edge. The outgoing edges of the new nodes are directed into the original successors of the eliminated node. The selection of nodes for splitting is randomized, but biased towards those nodes for which the result of splitting will lead to several new nodes that are all reached by a significant number of data items. Splitting nodes with this property affords the highest potential increase in likelihood score.

The *merge*-operator takes two nodes all of whose outgoing edges are directed to the same successor nodes, and replaces them with a single node, also having these same successors. From the number of data items reaching the original two nodes, and their local distributions, one can compute the distribution for the new node and the exact score gain obtained by the merge operation. A merge therefore always is executed iff the score gain is positive.

The *redirect*-operator is the computationally most expensive operator. It tests for every node ν in the PDG, and each of its outgoing edges leading into some $\nu' \in V_i$, whether the likelihood score can be improved by redirecting this edge into some other $\nu'' \in V_i$. This is tested by computing the likelihood score of the data-items reaching ν under the two marginal distributions defined by ν' and ν'' for the variables contained in the subtree rooted at X_i in the variable forest.

The `LearnForest` procedure constructs a variable forest incrementally. At each stage, some of the variables have been built into a variable forest. Each of the remaining variables is assigned to the descendant set ($desc(X_i)$) of some leaf X_i of an existing tree - they will be built into a subtree rooted at this leaf. Moreover, using the `LearnPDG` procedure, the partially constructed variable forest has already been expanded into a small PDG. Figure 3(a) shows this situation with three variables X_2, X_4, X_6 already built into a tree, all remaining variables assigned to $desc(X_4)$ of leaf X_4 of this tree, and a small PDG for the first three variables already constructed. In the `Grow` subroutine we first call the subroutine `DepGraph`. A call to `DepGraph(`$\mathbf{X}, t, V_i$`)` returns a dependency graph over variables in $\mathbf{X}$. Dependency tests are made conditional on the partition defined by V_i. The parameter t is a significance level for the independence tests. The use of different values for t promotes diversity in the structures in $\mathbf{F}$ (`Learn`, lines 3-4). Figure 3(b) shows the result of calling `DepGraph(`$\{X_1, X_3, X_5, X_7\}, t, V_4$`)`. Edges between variables indicate dependence between the variables. Each connected component of the resulting graph becomes a separate sub-tree under the original leaf. $CC(H)$ denotes the set of connected components in graph H. The `grow` subroutine finishes by randomly selecting from each connected component a node as the root for these new subtrees (`Grow`, line 4), and assigning the remaining variables from the connected component to this new leaf, (`Grow`, lines 5-7, figure 3(c)). One iteration

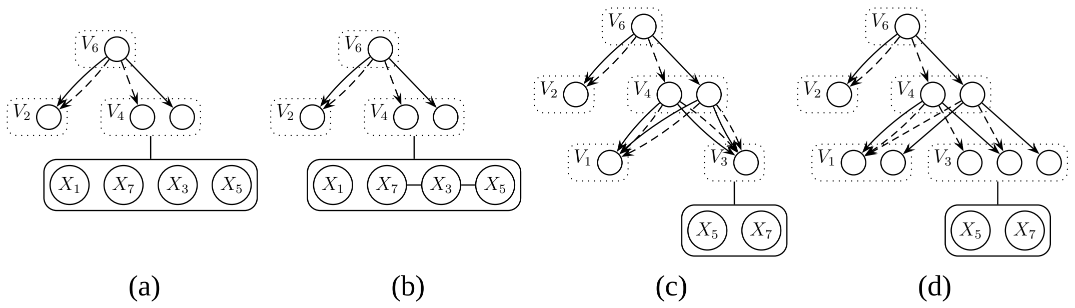


Fig. 3. Snapshots of the procedure for growing PDGs

of the `LearnForest` procedure then is completed by calling `LearnPDG` with a large parameter λ_{max} to refine the expanded forest into a small PDG, figure 3(d). `LearnForest` terminates when all leaves of all trees have empty successor sets. We then say they are full-grown.

We have implemented our PDG learning procedure in Java. The WEKA package (*http://www.cs.waikato.ac.nz/∼ml/weka/*) was used for basic data-handling routines.

As a first test of our learning algorithm we have applied it to a dataset sampled from the *parity* distribution described in section 2 with $n = 7$. The algorithm was run with a set of eight different λ-values. Figure 4(a)-(c) shows the PDGs learned for three decreasing λ-values. For the middle λ-value the learned PDG (figure 4(b)) is almost the optimal PDG for the underlying distribution. An optimal PDG would be obtained by merging the nodes 8 and 9. By avoiding this merge the algorithm here slightly overfits the data. For the smallest λ value (figure 4(c)) the overfitting is much stronger. The ability to learn the structure for the *parity* distribution demonstrates the potential of the split, merge and redirect operations for an effective PDG-structure search. The construction of the underlying variable forest here is not such a difficult problem, as any forest consisting of a single, linear tree can be used in an optimal PDG for the *parity* distribution.

4.2 Learning results: PDG vs. Bayesian networks

We applied our learning algorithm to several real-world datasets, and compared the resulting PDGs with the junction trees constructed from Bayesian networks learned from the same data. We evaluate the results using SL-curves, as described in Section 3. All datasets were split into a training set (2/3 of the data) and a test set (1/3 of the data). We consider SL-curves both for likelihood scores obtained over the training data and over the test data.

Instead of using real-world data, one might also consider using synthetic data sam-

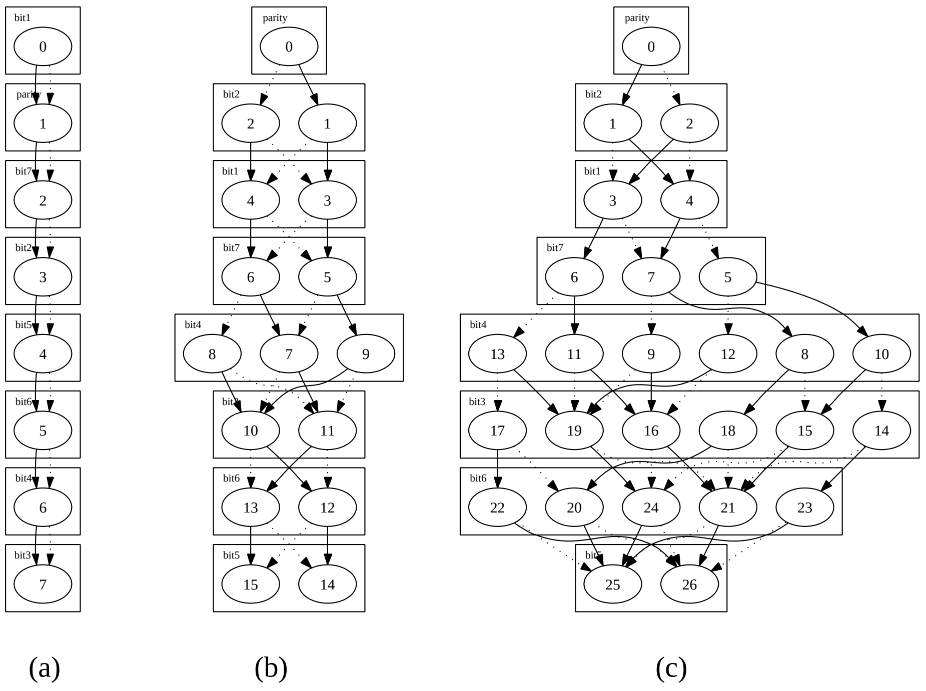


Fig. 4. Learned PDGs from parity data

pled from some distribution P. This approach avoids the difficulty of having to approximate the true distribution P with an empirical distribution P^D, and the accuracy of a model M can be evaluated directly via $CE(P^M, P)$. However, this approach is problematic in our context, where we aim to compare different representation frameworks: the representation used for the generating distribution P can easily bias the results of the comparison in favor of that representation framework which is more closely related to the generating model. If, for example, we generate data with a Bayesian network, then the data can be expected to contain independence structures that are more easily expressible with Bayesian networks than with PDGs. The converse holds if we sample data from a PDG.

By optimizing (4) we attempt to learn models that yield optimal size/likelihood trade-offs, i.e. models that are not dominated in SL-space by any other models of the same representation language (one model dominates another in SL-space, if its SL-coordinates are to the left and above the other model's coordinates). When we compare the achieved SL-values for different types of models, then two major factors will influence our results: the first factor is the existence of small, accurate models for the given real-world distributions in the respective representation frameworks; the second factor is our ability to find the best possible models with our learning methods. Ideally, one would investigate these two different factors separately. On the one hand, one would determine the SL-curves defined by the optimal models available in different representations. On the other hand, one would have to investigate how close to optimal the models are that we obtain from our learning methods. In our experiments, we cannot separate these two issues. From a practical point of view, however, one can argue that the mere existence of effi-

Data	#variables	size	Description	Source
Adult	15	45.222	Census data.	*UCI*
Letter	17	20000	Recognition of handwritten letters.	*UCI*
Hall of fame	17	1320	Major League Baseball hall of fame data.	*StatLib*
Yeast	9	1446	Prediction of Cellular Localization Sites of Proteins.	*UCI*
Supreme	8	4052	Prediction of action taken based on supreme court data from legal cases.	*StatLib*

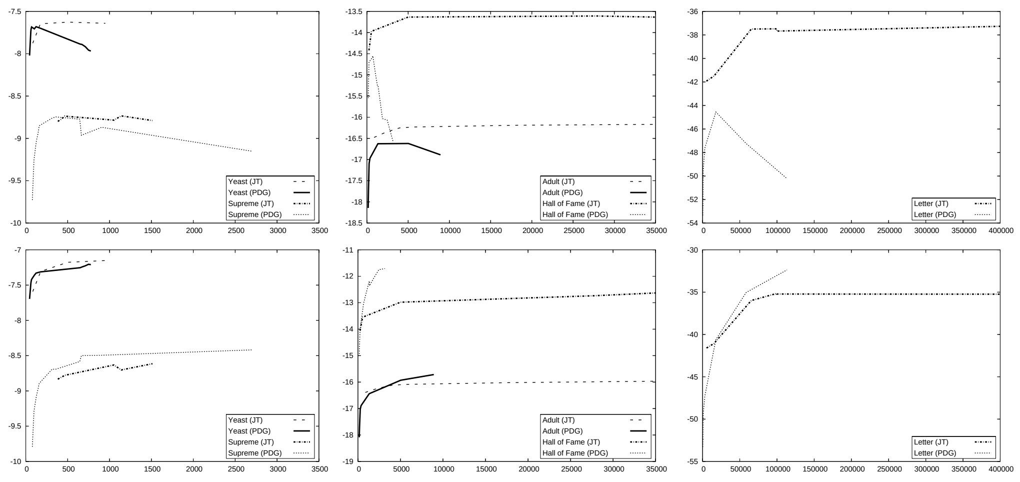
Table 2
Datasets used. Sources are the UCI-repository (*http://kdd.ics.uci.edu/*) and the StatLib site (*http://lib.stat.cmu.edu/*)

cient models in a given representation language is of little value if we are unable to learn these models from data. The 'practical efficiency' of a representation language, then, would be measured in the size and accuracy of models we are actually able to learn from data – which is what we do in our experiments.

For Bayesian network learning we use the B-course algorithm [8]. This is a score-based learning algorithm that performs structure search by local arc insertion, deletion and reversal operations. We use it with the generic score function (4) and various λ-values. The search in B-course continues to explore for better models until a timeout, always memorizing the best model found so far. In our experiments we set the timeout to 1 hour for every λ value. Bayesian networks were learned for 6-8 different λ-values, giving a total runtime for B-course of approximately 6-8 hours per dataset. The search in our PDG learner, on the other hand, terminates when no score improvement has been found within a certain number of iterations. The total runtime of the PDG learner proved to be highly dependent on the size of the datasets, because the local structure changing operations require quite frequent parameter re-estimations, and hence expensive data-reads. To learn models for all the given λ-values our algorithm needed in between 15 minutes for the smallest datasets, and 12 hours for the Adult dataset. To reduce overfitting, both learning procedures apply parameter smoothing methods to the model learned from optimization of S_λ.

The data used for the experiments are displayed in table 2. The preprocessing for all datasets consisted of removing cases with missing values and discretization of continuous variables into uniform intervals. Figure 5 shows the SL-curves obtained by the BN and PDG learners on our five datasets. The figure contains both the SL-curves obtained on the training data (lower row) and those for the test data (upper row). Recall that for Bayesian networks, the reported size is that of the generated junction tree. The likelihood scores are per-instance, i.e. equal to $L(M, D)/\mid D\mid$.

As expected, the curves for the training data are monotonically increasing. On most datasets we obtain with PDGs a somewhat higher likelihood score with models of the same or smaller size than with BNs. However, when we turn to the SL-curves for the test data we find that PDG models suffer to a greater extent from overfitting, so that the SL-curves here tend to decrease after attaining a maximum. This effect can also be observed on some datasets for the BN models (visible only for the



(a) (b) (c)

Fig. 5. SL-curves for PDGs and BNs learned from the datasets in table 2. Upper plots shows performance on test data and lower plots shows performance on training data. Plots are for the following data: yeast and supreme (a), adult and hall of fame (b) and letter (c).

Supreme data), but here is not nearly as strong. The reason why this overfitting effect is stronger for PDGs appears to be the following: when we have learned a PDG of size 1000, for example, then we will fit in the parameter learning phase 500 free parameters (assuming all variables are binary). A Junction tree of size 1000, on the other hand, would have been induced by a Bayesian network that contains only a much smaller number of free parameters. Since the parameter learning is done on the Bayesian network, not the junction tree, we have far fewer parameters to fit, and hence are less liable to overfit the training data.

We obtain the following general picture: on all datasets the SL-curves of PDGs and BNs show a surprisingly similar behavior. One might have expected that for some datasets one representation framework would clearly outperform the other, because of independence structures in the data that are more easily expressed in one of the two frameworks. However, no really big discrepancies in the results have been found [1]. The results for PDGs tend to be better than the results for junction trees when the evaluation is over the training data. PDGs here allow to fit the empirical distribution closely using smaller models than the junction trees generated from BNs. However, the models for which this difference becomes pronounced overfit

[1] To obtain a better intuition for the magnitude in likelihood differences, consider the following: suppose that the test data defines a distribution on binary variables $X_1, \ldots, X_{n+1}$ such that variable X_{n+1} is deterministically determined by the values of $X_1, \ldots, X_n$. Consider two models M_1, M_2 for the data that agree with respect to the marginal distribution of $X_1, \ldots, X_n$, but M_1 correctly identifies the functional dependence of X_{n+1}, whereas M_2 models X_{n+1} as independent from the other variables, with probability 1/2 for both its values. Then the difference in per-instance log-likelihood score for these two models will be equal to 1.

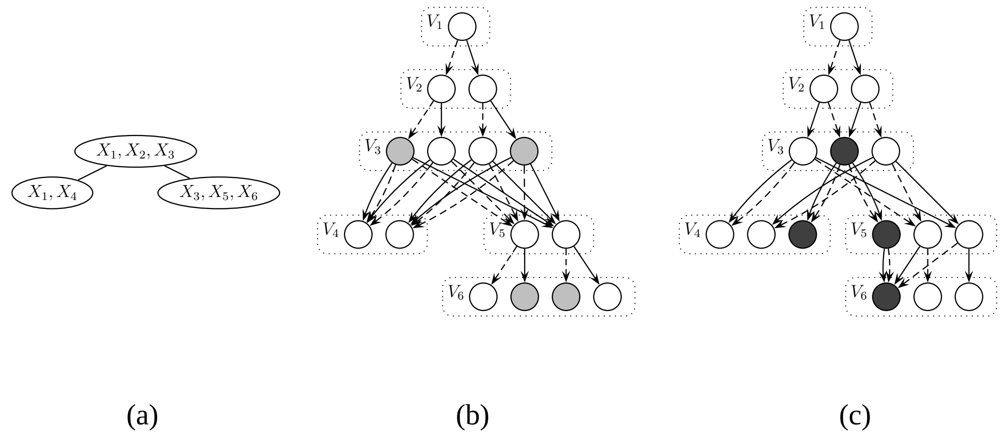


Fig. 6. The first steps in the *hybrid*-procedure. (a) shows a junction tree and (b) shows the PDG that result from compilation of the junction tree. Nodes with zero data-support are shaded light-gray. By garbage-collection we get the PDG shown in (c) - garbage nodes shaded dark-gray.

the data, so that the PDGs advantage is canceled, or even reversed, when evaluated over the test data.

5 Hybrid learning: combining BN and PDG-learning

As previously stated, there exists a linear transformation from junction trees into equivalent PDGs [5]. This naturally suggests another way of learning PDGs: one can first learn a Bayesian network from data, and then compile its junction tree into a PDG. The PDG so constructed can then be used as a starting point in our PDG learning procedure. A potential advantage of this approach is that the Bayesian network learning methods might be more successful in identifying independence relations among the variables, which would then be reflected in the tree structure of the compiled PDG. Thus, we would mainly hope to optimize the learned PDG forest structure by using this approach.

Figure 6 (a) shows a simple junction tree , and (b) the result of compiling it into a PDG using the method described in [5]. The compiled PDG is composed of several complete binary trees (in the case of binary variables). When the junction tree was learned from data, then it will typically contain in its clique potential tables many zero entries, corresponding to combinations of values that were not encountered in the data. In the compiled PDG these configurations with zero data support correspond to PDG nodes that are not reached by any data items. In Figure 6 (b) these zero-nodes are indicated by a grey shading. A first way to compress the size of the PDG representation without any loss of likelihood score on the underlying training data, is to eliminate the zero nodes.

We perform this elimination by collapsing all zero nodes in a node set V_i into a single "garbage-node". Such garbage nodes are then connected to form for each branch of the variable tree a garbage-path, and are initialised with parameters of uniform distribution. Figure 6 (c) shows the result of this operation with the new garbage nodes indicated by a dark shading.

The example illustrated in figure 6 does not show any gain (size reduction) from this garbage-collection. However, in our experiments we typically gain a considerable size reduction in this way. Starting with the PDG obtained from compilation and garbage collection, we then perform a series of *merge*-operations to further reduce the size. The rationale behind focusing on merge operations is that besides the zero nodes, the compiled PDG may also contain numerous equivalent non-zero nodes that may be merged without any likelihood loss. Specifically, context-specific independencies [6] in the underlying distribution would lead to the existence of such equivalent, mergeable nodes. For the merge operation we use the merge subroutine from our general learning procedure. This merge operation is parameterized with the λ parameter of our generic score function, and, depending on the λ value, will also merge nodes that are only approximately equivalent. The higher the λ-value, the more merge operations will be performed, leading to smaller and less accurate models. Apart from *merge* operations one might also transform the initial PDG using the *split* and *redirect* operations of `LearnPDG`. However, since the initial PDG obtained from a junction tree tends to be rather large already, we currently only use the size reducing *merge* operation.

Figure 7 contains plots showing the performance of our *hybrid* procedure for learning PDGs. The procedure was invoked on all the junction trees obtained from the BN learning as described in section 4.2. The solid lines depict the SL-curves of the initial junction trees; they are exactly the same as the plots for junction trees presented in figure 5 with the exception that in figure 7 we use a logarithmic scale for size. Each execution of the hybrid procedure gives us a sequence of PDGs obtained by iterated merge operations with increasing λ-parameter. The dotted lines in the plots are the SL-curves obtained for the models in one execution of the hybrid procedure. The temporal direction of the dotted lines are from right to left, i.e. from large towards smaller models. The first (rightmost) point on any dotted line represents the PDG that is obtained after compilation and elimination of zero nodes. The rest of the points each corresponds to additional *merge*-operations with increased λ-value.

Looking at the bottom plots of figure 7, depicting performance on training data, the first observation that can be made is that PDGs generally represent the empirical distribution with the same or better accuracy as the BN-model, while using fewer parameters. Secondly, we observe that the junction tree compilation does not always produce a PDG with exactly the same likelihood as the original BN. The reason for this is the same as we already encountered in section 4.2: the likelihood scores reported for the junction trees are those that are obtained by the underly-

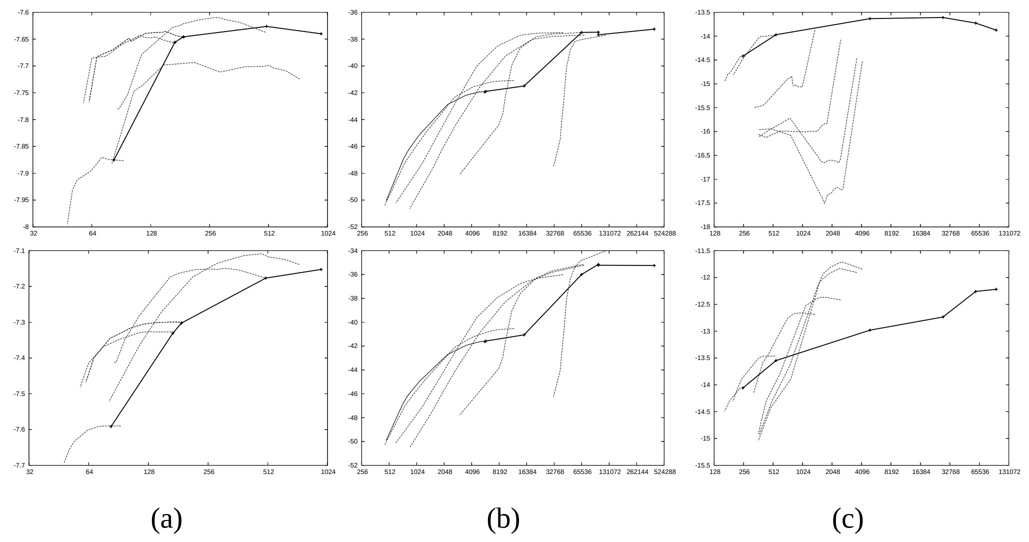


Fig. 7. Performance plots of the hybrid-procedure. Solid lines depicts SL-curves of junction trees, and dotted lines depicts SL-curves obtained by using the junction trees as starting point for the hybrid procedure. Plots are for yeast (a), letter (b) and hall of fame (c) datasets. Performance on test data are displayed in the upper plots and on training data below. Please note that the X-axis (Size) is logarithmic.

ing Bayesian network model, because parameters are fitted on that model. The Bayesian network model usually dictates a stricter independence model than the generated junction tree (which manifests itself in a smaller number of free parameters). The independence model represented by the PDG obtained from compiling the junction tree (before eliminating zero nodes) is the same as the independence model of the junction tree, and so it, too, imposes fewer independence constraints than the original Bayesian network. When we relearn parameters for the PDG-structure retrieved from the junction tree, we potentially exploit some additional degrees of freedom offered by the PDG-structure. This effect is clearly visible for the hall of fame data (c), and can also be noticed to a lesser degree for the other datasets.

Turning to the upper plots of figure 7, depicting the performance on test data, we see that for the yeast and letter data (a,b) the behavior is similar for the test data as for the training data: the initial compilation and the first one or two steps of the merge procedure produce models that have nearly the same likelihood score as the initial junction tree, but with a size reduced by about a factor 2. The results for the two remaining datasets from table 2 (Adult and Supreme) are similar to the results for yeast and letter, and are here omitted. For the Hall of fame dataset (c) the results look somewhat different. Here we clearly over-fit the training data also with our hybrid procedure, just like the overfitting problem was already most pronounced for this dataset in Figure 5 (not surprisingly: this dataset contains a relatively small number of cases, but has a relatively large state space with 17 variables). The gain in likelihood score on the training data, thus corresponds to a loss in likelihood score on the test data. The SL-curves for the Hall of fame test data show a somewhat strange, irregular behavior. We do not have a completely satisfactory explanation for the sudden drop in likelihood score, followed by a partial recovery, that we here observe as model size decreases. Most likely, this is due to the fact that our parameter smoothing routine may sometimes on larger models be a more effective instrument against overfitting than on somewhat smaller models.

Comparing the results of the hybrid-procedure to the results of the direct PDG learning algorithm reported in section 4.2 figure 5, we obtain more or less equivalent results in terms of Size/Likelihood trade-off on training data. For both Yeast and Hall of fame (Fig. 7(a)-(c)) we get equivalent size of the models with highest likelihood score for the training data, but for smaller model sizes the hybrid-procedure outperforms direct learning. For Letter-data (Fig. 7(b)) we do not obtain quite as good likelihood scores with the larger models as with direct learning, but again for smaller models hybrid learning performs better. For all the datasets, we improve performance on test sets - even for the dataset on which we experience over-fitting of training data (Hall of Fame, Fig. 7(c)).

The main lesson to take from these first experiments with the hybrid-procedure is that PDGs can offer a compact representation by compiling junction trees into PDGs. The compilation only ensures that the size of the PDG is within a factor 2 of the original junction tree [5, Theorem 5.1]. However, these experiments show that in practice we can reduce the size of the PDGs considerably by simple procedures, with limited or no loss in accuracy. The factor by which we can perform lossless compression of the PDG compared to the junction tree in practice seems closer to $1/2$ than 2.

6 Related Work

A related approach to making representations of probability distributions more compact and thereby speeding up probabilistic inference is the work by Darwiche on arithmetic circuit representations [12,13]. The key difference between arithmetic circuit representations and PDGs is that the former are not a dedicated representation framework for probability distributions, i.e. the subclass of circuits that represent distributions is not characterized by a simple syntactic criterion. As a consequence, it would appear very difficult to learn arithmetic circuits directly from data, as the search space of possible models is not well circumscribed. Consequently, Darwiche envisages arithmetic circuits mostly as a secondary representation that has to be obtained by compilation from some primary representation (e.g. a polynomial or a junction tree representation). Empirical results in [13] show that compiled circuit representations can be much smaller than junction tree representations. The compilation technique of Darwiche is related to the first phase of our hybrid learning procedure. Since arithmetic circuits have fewer structural constraints than

PDGs, one would, in fact, expect that by pure compilation smaller arithmetic circuit than PDG representations can be obtained. However, PDGs have the advantage that compilation can be combined with parameter re-estimation from the data, so that we can always fit optimal parameters to the structure of the compiled model.

Another recent framework related to PDGs and arithmetic circuits are the *case-factor diagrams* of Collins et al. [14]. Like PDGs, case-factor diagrams are inspired by binary decision diagrams, and support linear time probabilistic inference. The learnability of case-factor diagrams has not been investigated yet.

The most closely related work about learning PDG-related models is work on learning probability estimation trees (PETs)[15] and decision graphs for CPT representations in a Bayesian network (CPT-DG)[16,17]. Both of these frameworks serve only for the representation of a distribution of a single variable, conditional on values of other variables. In case of PETs this is the distribution of the class variable given attribute values; in the case of CPT-DGs this is the distribution of a network variable conditional on its parents. More fundamental than this difference, however, is the fact that both PETs and CPT-DGs follow the *multi-terminal binary decision diagram (MTBDD)*[18] paradigm of function representation: the internal nodes of the representations only serve to determine the argument for the function; they do not as in PDGs already contain numerical information from which the function value (i.e. a probability) is incrementally constructed while descending through the tree or graph. As a result, such representations always require as many leaves as there are different function values, whereas in the case of PDGs the number of function values only induces a lower bound on the number of paths through the graph.

The structure search for good PETs or CPT-DGs on the one hand, and PDGs on the other hand, has to focus on somewhat different problems: for the former types of representations one main question is which variables to include in the graph or tree, so as to obtain an informative case-distinction for the distribution of the target variable at the leaves. For PDGs, the set of variables is given, and the labeling of nodes in the PDG with variables follows much stricter rules than imposed in a PET or CPT-DG. Nevertheless, [16] use in the structure search for CPT-DGs split and merge operations that somewhat resemble our split and merge operations. However, Chickering et al. [16] apply their split and merge operations only at leaf nodes. Moreover, their application of split and merge operations is purely random, and not based on any score improvement heuristics as in our algorithm.

De Campos and Huete [19] describes an algorithm that directly learns a junction tree, rather than a BN, through independence tests. This work is related to ours in that it allows to score a candidate model directly in terms of its efficiency for probabilistic inference. No experimental results are reported in [19].

7 Conclusion

We have developed and implemented a method for learning probabilistic decision graphs from data. The results obtained from applying the method to the *parity* dataset show that our structure search procedure can identify optimal or near optimal PDG structures in at least some non-trivial problems. Using our method, we have learned PDG models for a number of real-life datasets, and on the basis of Size-Likelihood curves compared the learned models with junction tree representations obtained from Bayesian network learning. The results here indicate a better ability of the PDGs to fit the training data exactly, which gives a higher likelihood score on the training data, but leads to overfitting. Combining Bayesian network learning with the merge-subroutine of PDG learning, we developed a hybrid learning method that improves both on pure BN learning and pure PDG learning.

At this point it is still unclear to what extent the results we obtained in PDG learning were limited by the representation language as such, i.e. the (non-)availability of small, accurate PDG models, or by our learning method, i.e. the (non-)ability to find good PDG structures for a given dataset.

Future work should be directed at refining the structure search methods in PDG learning, the experimental exploration of further datasets in order to identify types of distributions for which PDG representations are best suited, and at adapting PDGs for more specialized inference tasks like classification.

References

[1] M. Bozga, O. Maler, On the representation of probabilities over structured domains, in: Proceedings of CAV-99, no. 1633 in Lecture Notes in Computer Science, 1999.

[2] M. Jaeger, Probabilistic decision graphs: Combining verification and AI techniques for probabilistic inference, in: Proceedings of the first European Workshop on Probabilistic Graphical Models (PGM), 2002, pp. 81 – 88.

[3] R. E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Transactions on Computers 35 (8) (1986) 677–691.

[4] M. I. Jordan (Ed.), Learning in Graphical Models, MIT Press, 1999.

[5] M. Jaeger, Probabilistic decision graphs - combining verification and AI techniques for probabilistic inference, Int. J. of Uncertainty, Fuzziness and Knowledge-based Systems 12 (2004) 19–42.

[6] C. Boutilier, N. Friedman, M. Goldszmidt, D. Koller, Context-specific independence in Bayesian networks, in: Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI–96), 1996, pp. 115–123.

[7] R. G. Cowell, A. P. Dawid, S. L. Lauritzen, D. J. Spiegelhalter, Probabilistic Networks and Expert Systems, Springer, 1999.

[8] P. Myllymaki, T. Silander, H. Tirri, P. Uronen, B-course: A web-based tool for Bayesian and causal data analysis, International Journal on Artificial Intelligence Tools 11 (3) (2002) 369–387.

[9] C. Huang, A. Darwiche, Inference in belief networks: A procedural guide, International Journal of Approximate Reasoning 15 (1996) 225–263.

[10] T. M. Cover, J. Thomas, Elements of information theory, Wiley, 1991.

[11] F. Provost, T. Fawcett, Analysis and visualization of classifier performance: Comparison under imprecise class and cost distribution, in: Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97), 1997, pp. 43–48.

[12] A. Darwiche, A differential approach to inference in Bayesian networks, in: Proceedings of the Sixteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–2000), 2000, pp. 123–132.

[13] A. Darwiche, A logical approach to factoring belief networks, in: Proceedings of the Eighth International Conference on Principles and Knowledge Representation and Reasoning (KR-2002), 2002, pp. 409–420.

[14] M. Collins, D. McAllester, F. Pereira, Case-factor diagrams for structured probabilistic modeling, in: Proceedings of the Twentieth Annual Conference on Uncertainty in Artificial Intelligence (UAI–2004), 2004, pp. 382–391.

[15] F. Provost, P. Domingos, Tree induction for probability-based ranking, Machine Learning 52 (2003) 199–215.

[16] D. M. Chickering, D. Heckerman, C. Meek, A Bayesian approach to learning Bayesian networks with local structure, in: Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–97), Morgan Kaufmann Publishers, San Francisco, CA, 1997, pp. 80–89.

[17] N. Friedman, M. Goldszmidt, Learning bayesian networks with local structure, in: M. I. Jordan (Ed.), Learning in Graphical Models, MIT Press, 1999.

[18] M. Fujita, P. C. McGeer, J.-Y. Yang, Multi-terminal binary decision diagrams: an efficient data structure for matrix representation, Formal Methods in System Design 10 (1997) 149–169.

[19] L. M. de Campos, J. F. Huete, Algorithms for learning decomposable models and chordal graphs, in: Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI–97), 1997, pp. 46–53.