Learning Probabilistic Decision Graphs

Jaeger, Manfred; Dalgaard, Jens; Silander, Tomi

Published in:
Proceedings of the Second European Workshop on Probabilistic Graphical Models

Publication date:
2004

Document Version
Publisher's PDF, also known as Version of record

Learning Probabilistic Decision Graphs

Manfred Jaeger Jens D. Nielsen
Institut for Datalogi
Aalborg Universitet
Frederik Bajders Vej 7,
9220 Aalborg , Denmark
{jaeger, dalgaard}@cs.auc.dk

Tomi Silander
Complex Systems Computation Group
Helsinki Institute for Information Technology
P.O.Box 9800,
FIN-02015 HUT, Finland
tsilander@hiit.fi

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.

1 Introduction

Probabilistic decision graphs (PDGs) (Bozga & Maler 1999, Jaeger 2002) are a graphical representation language for probability distributions that is based on the representation paradigm of ordered binary decision diagrams (Bryant 1986). PDGs were originally conceived for applications in automated verification of probabilistic systems (Bozga & Maler 1999). An initial study of their potential strengths as a representation language also for AI applications was conducted in (Jaeger 2002). The main result of that study was that from a computational complexity point of view, PDGs are always as efficient as Bayesian networks, and for some types of probability distributions they are more efficient.

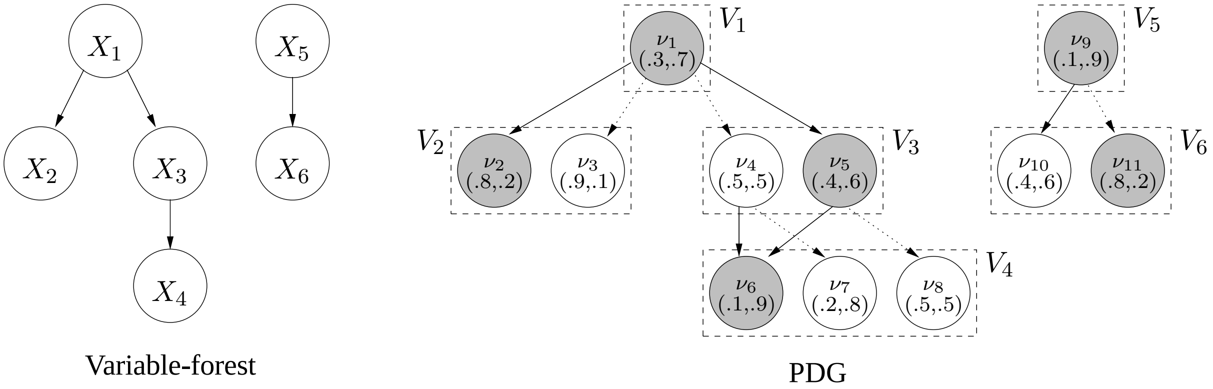
These theoretical results leave the question open, how efficient PDG representations of a given probabilistic domain can be found in practice; in particular, whether PDGs can be learned automatically from data. This question is taken up in the present paper. We here describe an approach to learning PDGs from data, and compare the PDG models with Bayesian networks learned from the same datasets. The basis for the comparison is the efficiency of probabilistic inference in the learned models.

In the following section we briefly introduce the language of PDGs, and review some of their essential properties. In section 3 we describe a learning algorithm for PDGs. Section 4 reports the results of an initial comparative study of Bayesian network and PDG models learned from data.

2 Probabilistic Decision Graphs

In this section we briefly review definitions and some results given in (Jaeger 2002, Jaeger 2004).

Figure 1 shows on the right an example PDG defining a joint probability distribution for binary random varibles $X_1, \ldots, X_6$. 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 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 $\nu' \in V_j$. 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



Variable-forest

PDG

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

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 induces 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. This is a very strong independence property, and therefore we will mostly encounter PDGs whose underlying variable-forest consists of a single 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 $\{X_1 = 1, X_3 = 0\}$. The conditional independence relations encoded by a PDG now are:

$$P(X_i \mid \textit{nonsucc}(X_i)) = P(X_i \mid V_i) \qquad (1)$$

where *nonsucc* stands for the set of variables that are not successors of X_i in the variable-forest. Such partition-based independence relations can correspond to context-specific independencies in the sense of (Boutilier, Friedman, Goldszmidt & Koller 1996). 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 $X_3 = 1$. 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 (Jaeger 2004).

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. Inference in Bayesian networks, on the other hand, is linear only in the size of the junction tree constructed from the Bayesian network. To compare the complexity of probabilistic inference based on Bayesian network and PDG representations, one thus has to compare the sizes of a PDG representation with the size of the junction tree generated from a Bayesian network. More precisely, this is the pertinent comparison when one is interested in exact computation of arbitrary posterior marginal distributions. For more specialized inference tasks (e.g. classification with all attributes instantiated), or approximate inference procedures, inference in Bayesian networks can become linear already in the size of the Bayesian network.

It was shown in (Jaeger 2002) 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 $X_{n+1} = \sum_{i=1}^{n} X_i \text{mod } 2$. 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 (Jaeger 2002).

In some sense, then, PDGs and Junction Trees, and hence Bayesian networks, provide computationally equally efficient representations of probability distributions[1]. 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 Learning PDGs

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

$$S_\lambda(M) := (1-\lambda)L(M,D) - \lambda|M| \qquad (2)$$

where $L(M,D)$ is the log-likelihood of the data given the model M, $|M|$ is the size of the model (measured by its total number of parameters), and $0 \leq \lambda \leq 1$ is a parameter that can be set for the learning problem at hand. Optimizing (2) with large λ-values will lead to small and not very accurate models, whereas small λ-values give higher scores for larger and more accurate models. Setting λ to *log* $\mid D \mid /(2+ \mid D \mid)$, for example, makes this score function equivalent to BIC score (Friedman & Goldszmidt 1999). One can also use the generic score function (2) to learn models for resource-bounded applications by first setting λ to a large value, and then decrementing until the learned model exceeds a given size constraint. The last model learned is then the most accurate model we can find under the given constraints.

Optimizing S_λ for large λ is easier than optimizing for small λ, as the strong bias towards smaller models efficiently reduces the 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 (because capturing certain conditional independence relations among variables in the forest structure will lead to smaller models and hence higher scores for all λ values). 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. table 1):

[1] Strictly speaking, to establish the equal efficiency of Bayesian networks one also has to show that for every distribution there exists a Bayesian network from which a junction tree can be constructed that has minimal size (up to a linear factor) among all junction trees for the given distribution.

first a population of candidate variable forests is created (line 3). Starting with the largest λ in a set L of λ-parameters, then each variable forest is refined into an actual PDG using the `LearnPDG` sub-routine, which optimizes S_λ. Forests achieving a poor score in this optimization are then removed from the population (lines 6-7), and the process is repeated with the next smaller λ-value. The subroutine `LearnForest` that generates the initial forests is a *constraint-based* approach that builds a forest encoding certain conditional independence relations we find in the data. It requires a parameter t that determines the level of conditional independence tests inside the routine. The use of different test-levels promotes diversity in the structures in F (lines 2-3).

```
0: Procedure Learn
1: $L$ :=set of decreasing $\lambda$-values
2: $T$ :=set of different test-levels
3: $F := \{$LearnForest($t$)$|t \in T\}$
4: for each $\lambda \in L$ do:
5:     $G := \{$LearnPDG($a, \lambda$)$|a \in F\}$
6:     compute $F_{low}$
7:     $F := F \setminus \{a | a \in F_{low}\}$
8:     output highest scoring $g \in G$
```

Table 1: Main PDG-learning loop

We now describe the two main subroutines in greater detail. `LearnPDG(`a, λ`)` (see table 2) traverses the space of different PDGs over the forest a in the search for an optimal PDG, w.r.t 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. If the probability distributions at the two nodes are sufficiently similar, thent the size reduction obtained by the merge will outweigh the loss in likelihood score, and hence lead to an overall score improvement. It is possible to compute the exact change in score from the local information at the two nodes, so that merges are performed systematically for pairs of nodes for which the score change is positive.

The *redirect*-operator is the compuationally 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 dataitems 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.

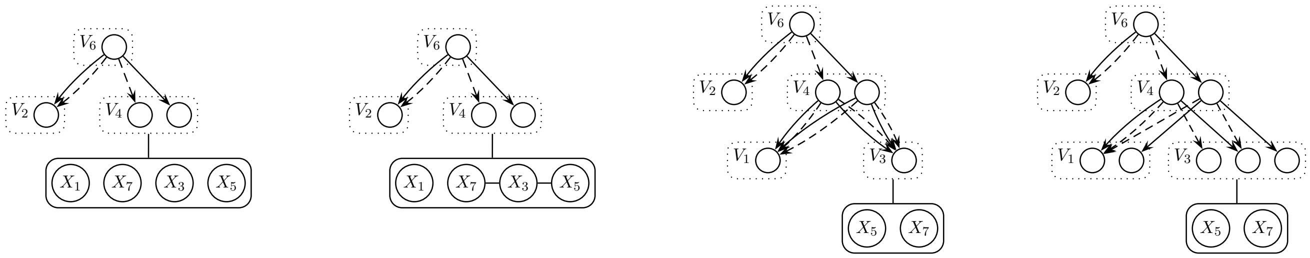
```
Procedure LearnPDG($a, \lambda$):
repeat:
    split nodes top-down
    merge nodes bottom-up
    redirect edges bottom-up
until no improvement in $S_\lambda$
```

Table 2: LearnPDG procedure

```
Procedure LearnForest($t$)
Initialize($F, t$)
for each $a \in F$ do:
    repeat:
        grow($a, t$)
        LearnPDG($a, \lambda_{max}$)
    until $a, t$ is complete
```

Table 3: LearnForest procedure

The `LearnForest` procedure is sketched in table 3. The procedure constructs the variable forest incrementally. At each stage, some of the variables have been built into a variable forests. Each of the remaining variables is assigned to some leaf of an existing tree, which means that 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



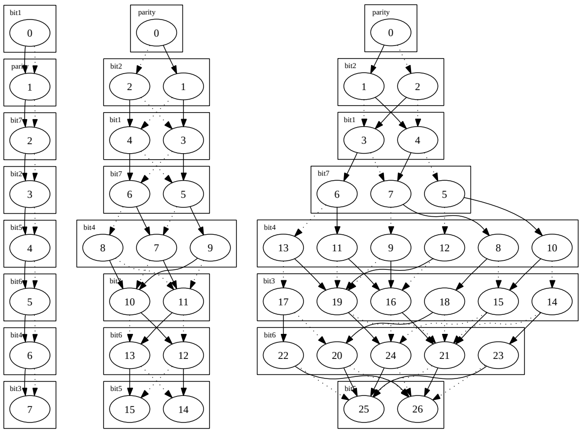
(a) (b) (c) (d)

Figure 2: Snapshots of the procedure for growing PDGs

into a small PDG. Figure 2(a) shows this situation with three variables X_2, X_4, X_6 already built into a tree, all remaining variables assigned to leaf X_4 of this tree, and a small PDG for the first three variables already constructed. In the `grow` subroutine we first perform a χ^2 independence test for all pairs of variables assigned to the same leaf, where the test is conditional on the partition defined by the leaf. Variables for which the test indicates dependence are connected by an edge, figure 2(b). Each connected component of the resulting graph becomes a separate sub-tree under the original leaf. In the first stage of the algorithm variables are not yet assigned to an existing leaf, and the connected components determine the partitioning of the variables into the separate trees of the forest. The `grow` subroutine finishes by randomly selecting from each connected component a node as the root for these new subtrees and assigning the remaining variables from the connected component to this new leaf, figure 2(c). One iteration 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 2(d).

We have implemented our PDG learning procedure in Java. For basic data-handling routines we made use of the WEKA package (*http://www.cs.waikato.ac.nz/∼ml/weka/*).

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 setting L to a set of eight different λ-values. Figure 3(a)-(c) shows the PDGs learned for three decreasing λ-values. For the middle λ-value the learned PDG 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 3(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.



(a) (b) (c)

Figure 3: Learned PDGs from parity data

4 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. Main objective of these experiments is to empirically evaluate the relative inferential efficiency of PDG and Bayesian network representations for real-world distributions. Ideally, we would like to find optimal (i.e. as small as possible) PDG and junction tree models for the exact distribution that generated the data. This, of course, is impossible, since the precise distribution is unknown, and all we have is a sample drawn from the distribution. Thus, all models that we learn can only be approximations of the true underlying distribution. We use the log-likelihood of a test dataset under a model as a measure for how accurately the model approximates the underlying distribution. For this purpose all datasets have been split into a training set (2/3 of the data) and a test set (1/3 of the data).

One might avoid some of these problems by learning from synthetic data sampled from some model. The known model can then be used to evaluate the accuracy of the learned models precisely. This, however, is problematic in our context, where we aim to compare different representation frameworks: the representation used for the generating model 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.

We try to learn models that are as accurate and as small as possible. The success of this learning task will be determined by two factors: the existence of small and accurate models for the given real-world distributions in the respective representation frameworks, and our ability to find the best possible models with our learning methods. Our experiments, thus, confound two issues that, from a theoretical point of view, one would like to investigate separately. From a practical point of view, however, one can argue that the mere existence of efficient 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.

In our experiments we do not focus only on learning a single best approximate model, but we investigate the size of models over a range of different accuracies. This is of interest, because it might be the case that for a resource-bounded application we are not necessarily interested in the most accurate model, but in a model that is as accurate as possible within a given size bound. With our experiments we aim at obtaining a full picture of the different size/accuracy tradeoffs that can be realized in the different representation frameworks. In the case of PDGs we obtain a spectrum of models representing different size/accuracy tradeoffs in a single run of our algorithm which records the best model found for each $\lambda \in L$.

For Bayesian network learning we use the B-course algorithm (Myllymaki, Silander, Tirri & Uronen 2002). 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 (2) 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 4. The preprocessing for all datasets consisted of removing cases with missing values and continuous variables discretized into uniform intervals.

Figure 4 shows the size and log-likelihood values

Data	#variables	size	Description	Source
Adult	15	45.222	Census data.	UCI
Space	7	3197	Geographical analysis spatial data.	StatLib
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 4: Datasets used. Data sources: *http://kdd.ics.uci.edu/* (UCI) and *http://lib.stat.cmu.edu/* (StatLib).

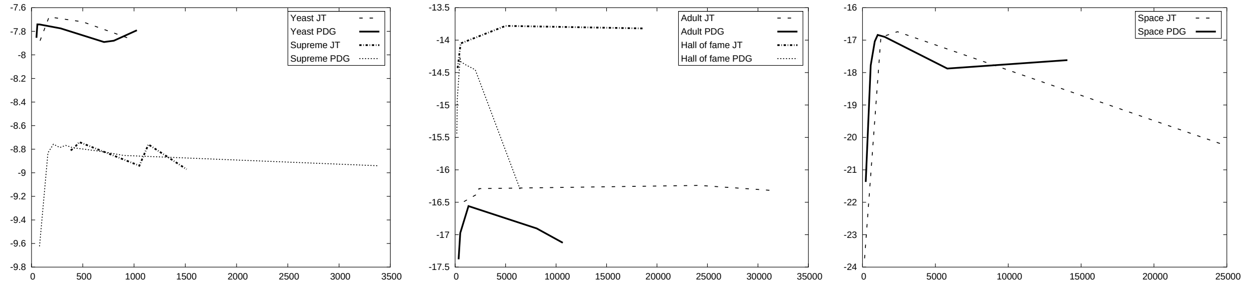


Figure 4: Size/accuracy tradeoff-curves for PDGs and BNs learned from yeast and supreme data (leftmost plot) adult and halloffame data (middle) and space data (rightmost). X-axis is Size and Y-axis is log-likelihood.

obtained by the BN and PDG learners by optimizing S_λ for a range of λ-values. For Bayesian networks, the reported size is that of the generated junction tree. The likelihood scores are per-instance, i.e. equal to $L(M, D)/\mid D\mid$. The results show a surprisingly close correspondence for the two frameworks. 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. Overall, the results for Bayesian networks are slightly better than for PDGs. The absolute differences in likelihood score are rather small, however[2].

5 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 (Darwiche 2000, Darwiche 2002). 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). In (Darwiche 2002) some empirical results are given which show that circuit representations can be much smaller than junction tree representations.

The most closely related work about learning PDG-related models is work on learning probability estimation trees (PETs)(Provost & Domingos 2003) and decision graphs for CPT representations in a Bayesian network (CPT-DG)(Chickering, Heckerman & Meek 1997, Friedman & Goldszmidt 1999). Both of these frameworks serve only for the representation of a distribution of a single variable, con-

[2] Logarithms here are base 2. 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

ditional 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 condtional 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)*(Fujita, McGeer & Yang 1997) 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 labelling of nodes in the PDG with variables follows much stricter rules than imposed in a PET or CPT-DG. Nevertheless, (Chickering et al. 1997) use in the structure search for CPT-DGs split and merge operations that somewhat resemble our split and merge operations. However, Chickering et al. 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.

6 Conclusion

We have developed and implemented a method for learning probabilistic decision graphs from data. Applying the method to the artificial *parity* dataset we found that we are able to learn PDG models which are more efficient than any Bayesian network model for the same data. Similar advantages, so far, have not been found for real-life datasets. However, there is still much potential for improving the PDG learning method. In combination with a better understanding for what types of data might be most appropriate for PDG modelling, this may still lead to applications where learned PDGs have a clear advantage over learned Bayesian networks. Furthermore, more specialized applications for PDGs (e.g. in classification) should be investigated.

References

Boutilier, C., Friedman, N., Goldszmidt, M. & Koller, D. (1996), Context-specific independence in Bayesian networks, *in* ‘Proceedings of UAI–96’, pp. 115–123.

Bozga, M. & Maler, O. (1999), On the representation of probabilities over structured domains, *in* ‘Proceedings of CAV-99’, number 1633 *in* ‘Lecture Notes in Computer Science’.

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

Chickering, D. M., Heckerman, D. & Meek, C. (1997), A Bayesian approach to learning Bayesian networks with local structure, *in* ‘Proceedings of UAI–97’.

Darwiche, A. (2000), A differential approach to inference in Bayesian networks, *in* ‘Proceedings of UAI–2000’.

Darwiche, A. (2002), A logical approach to factoring belief networks, *in* ‘Proceedings of KR-2002’.

Friedman, N. & Goldszmidt, M. (1999), Learning bayesian networks with local structure, *in* M. I. Jordan, ed., ‘Learning in Graphical Models’, MIT Press.

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

Jaeger, M. (2002), Probabilistic decision graphs: Combining verification and AI techniques for probabilistic inference, *in* ‘Proceedings of the first European Workshop on Probabilistic Graphical Models (PGM)’.

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

Myllymaki, P., Silander, T., Tirri, H. & Uronen, P. (2002), ‘B-course: A web-based tool for Bayesian and causal data analysis’, *International Journal on Artificial Intelligence Tools* **11**(3), 369–387.

Provost, F. & Domingos, P. (2003), ‘Tree induction for probability-based ranking’, *Machine Learning* **52**, 199–215.