Learning Arithmetic Circuits

Daniel Lowd and **Pedro Domingos**
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350, U.S.A.
{*lowd,pedrod*}*@cs.washington.edu*

Abstract

Graphical models are usually learned without regard to the cost of doing inference with them. As a result, even if a good model is learned, it may perform poorly at prediction, because it requires approximate inference. We propose an alternative: learning models with a score function that directly penalizes the cost of inference. Specifically, we learn arithmetic circuits with a penalty on the number of edges in the circuit (in which the cost of inference is linear). Our algorithm is equivalent to learning a Bayesian network with context-specific independence by greedily splitting conditional distributions, at each step scoring the candidates by compiling the resulting network into an arithmetic circuit, and using its size as the penalty. We show how this can be done efficiently, without compiling a circuit from scratch for each candidate. Experiments on several real-world domains show that our algorithm is able to learn tractable models with very large treewidth, and yields more accurate predictions than a standard context-specific Bayesian network learner, in far less time.

1 INTRODUCTION

Bayesian networks are a powerful language for probabilistic modeling, capable of compactly representing very complex dependences. Unfortunately, the compactness of the representation does not necessarily translate into efficient inference. Networks with relatively few edges per node can still require exponential inference time. As a consequence, approximate inference methods must often be used, but these can yield poor and unreliable results. If the network represents manually encoded expert knowledge, this is perhaps inevitable. But when the network is learned from data, the cost of inference can potentially be greatly reduced, without compromising accuracy, by suitably directing the learning process.

Bayesian networks can be learned using local search to maximize a likelihood or Bayesian score, with operators like edge addition, deletion and reversal (Heckerman et al., 1995). Typically, the number of parameters or edges in the network is penalized to avoid overfitting, but this is only very indirectly related to the cost of inference. Two edge additions that produce the same improvement in likelihood can result in vastly difference inference costs. In this case, it seems reasonable to prefer the edge yielding the lowest inference cost. In this paper, we propose a learning method that accomplishes this, by directly penalizing the cost of inference in the score function.

Our method takes advantage of recent advances in exact inference by compilation to arithmetic circuits (Darwiche, 2003). An arithmetic circuit is a representation of a Bayesian network capable of answering arbitrary marginal and conditional queries, with the property that the cost of inference is linear in the size of the circuit. When context-specific independences are present, arithmetic circuits can be much more compact than the corresponding junction trees. We take advantage of this by learning arithmetic circuits that are equivalent to Bayesian networks with context-specific independence, using likelihood plus a penalty on the circuit size as the score function.

Previous work on learning graphical models with the explicit goal of limiting the complexity of inference falls into two main classes: mixture models with polynomial-time inference (e.g.: Meila and Jordan (2000); Lowd and Domingos (2005)) and graphical models with thin junction trees (e.g.: Srebro (2000); Chechetka and Guestrin (2008)). The former are limited in the range of distributions that they can compactly represent. The latter are computationally viable (at both learning and inference time) only for very low treewidths. Our approach can flexibly and compactly learn a wide variety of models, including models with very large treewidth, while guaranteeing efficient inference, by taking advantage of the properties of arithmetic circuits.

The prior work most closely related to ours is Jaeger et al.'s

(2006). Jaeger et al. define probabilistic decision graphs, a new language related to binary decision diagrams. In contrast, we use standard arithmetic circuits, and our models are equivalent to standard Bayesian networks. Jaeger et al. speculate that learning arithmetic circuits directly from data would be very difficult. In this paper we propose one approach to doing this.

The remainder of our paper is organized as follows. In Sections 2 and 3, we provide background on Bayesian networks and arithmetic circuits, respectively. We describe in detail our algorithm for learning arithmetic circuits in Section 4. Section 5 contains our empirical evaluation on three real-world datasets, and we conclude in Section 6.

2 BAYESIAN NETWORKS

A *Bayesian network* encodes the joint probability distribution of a set of n variables, $\{X_1, \ldots, X_n\}$, as a directed acyclic graph and a set of conditional probability distributions (CPDs) (Pearl, 1988). Each node corresponds to a variable, and the CPD associated with it gives the probability of each state of the variable given every possible combination of states of its parents. The set of parents of X_i, denoted Π_i, is the set of nodes with an arc to X_i in the graph. The structure of the network encodes the assertion that each node is conditionally independent of its non-descendants given its parents. The joint distribution of the variables is thus given by $P(X_1, \ldots, X_n) = \prod_{i=1}^{n} P(X_i|\Pi_i)$.

For discrete domains, the simplest form of CPD is a conditional probability table. When the structure of the network is known, learning reduces to estimating CPD parameters. When the structure is unknown, it can be learned by starting with an empty or prior network and greedily adding, deleting and reversing arcs to optimize some score function (Heckerman et al., 1995). The score function is usually log-likelihood plus a complexity penalty or a Bayesian score (product of prior and marginal likelihood).

The goal of inference in Bayesian networks is to answer arbitrary marginal and conditional queries (i.e., to compute the marginal distribution of a set of query variables, possibly conditioned on the values of a set of evidence variables). One common method is to construct a *junction tree* from the Bayesian network and pass messages from the leaves of this tree to the root and back. A junction tree is constructed by connecting parents of the same variable, removing arrows, and triangulating the resulting undirected graph (i.e., ensuring that all cycles of length four or more have a chord). Each node in the junction tree corresponds to a *clique* (maximal completely connected subset of variables) in the triangulated graph. Ordering cliques by the highest-ranked variable they contain, each clique is connected to a predecessor sharing the highest number of variables with it. The intersection of the variables in two adjacent cliques is called the *separator* of the two cliques. A junction tree satisfies two important properties: each variable in the Bayesian network appears in some clique with all of its parents; and if a variable appears in two cliques, it appears in all the cliques on the path between them (the *running intersection property*). The *treewidth* of a junction tree is one less than the maximum clique size. The complexity of inference is exponential in the treewidth. Finding the minimum-treewidth junction tree is NP-hard (Arnborg et al., 1987). Inference in Bayesian networks is #P-complete (Roth, 1996).

Because exact inference is intractable, approximate methods are often used, of which the most popular is *Gibbs sampling*, a form of Markov chain Monte Carlo (Gilks et al., 1996). A Gibbs sampler proceeds by sampling each non-evidence variable in turn conditioned on its Markov blanket (parents, children and parents of children). The distribution of the query variables is then approximated by computing, for each possible state of the variables, the fraction of samples in which it occurs. Gibbs sampling can be very slow to converge, and many MCMC variations have been developed, but choosing and tuning one for a given application remains a difficult, labor-intensive task. Diagnosing convergence is also difficult.

2.1 LOCAL STRUCTURE

Table CPDs require exponential space in the number of parents of the variable. A more scalable approach is to use *decision trees* as CPDs, taking advantage of context-specific independencies (i.e., a child variable is independent of some of its parents given some values of the others) (Boutilier et al., 1996; Friedman & Goldszmidt, 1996; Chickering et al., 1997). The algorithm we present in this paper learns arithmetic circuits that are equivalent to this type of Bayesian network.

In a decision tree CPD for variable X_i, each interior node is labeled with one of the parent variables, and each of its outgoing edges is labeled with a value of that variable.[1] Each leaf node is a multinomial representing the marginal distribution of X_i conditioned on the parent variable values specified by its ancestor nodes and edges in the tree.

The following two definitions will be useful in describing our algorithm.

Definition 1. *For leaf node D and k-valued variable X_j, the* split $S(D, X_j)$ *replaces D with k new leaves, each conditioned on a particular value of X_j in addition to the parent values on the path to D.*

[1] In general, each outgoing edge can be labeled with any subset of the variable's values, as long as the sets of labels assigned to all child edges include every variable value and are disjoint with each other. For simplicity, we limit our discussion to the case in which each edge has a single label, which Chickering et al. (1997) refer to as a *complete split*. For Boolean variables, as in our experiments, all types of splits are equivalent.

Definition 2. *Let D be a leaf from the tree CPD for X_i. Split $S(D, X_j)$ is* valid *iff:*

- *X_j is not a descendant of X_i in the Bayesian network*
- *No decision tree ancestor of D is labeled with X_j*

The first definition describes a structural update to the Bayesian network; the second one gives the conditions necessary for that update to be consistent and meaningful.

A Bayesian network can now be learned by greedily applying the best valid splits according to some criterion, such as the likelihood of the data penalized by the number of parameters. This is one version of Chickering et al.'s algorithm (1997). A number of other methods have also been proposed, such as merging leaves to obtain decision graphs (Chickering et al., 1997) or searching through Bayesian network structures and inducing decision trees conditioned on the global structure (Friedman & Goldszmidt, 1996).

3 ARITHMETIC CIRCUITS

The probability distribution represented by a Bayesian network can be equivalently represented by a multilinear function known as the *network polynomial* (Darwiche, 2003):

$$\begin{aligned} & P(X_1 = x_1, \ldots, X_n = x_n) \\ &= \sum_{\mathbf{X}} \prod_{i=1}^{n} I(X_i = x_i) P(X_i = x_i | \Pi_i = \pi_i) \end{aligned}$$

where the sum ranges over all possible instantiations of the variables, $I()$ is the indicator function (1 if the argument is true, 0 otherwise), and the $P(X_i|\Pi_i)$ are the parameters of the Bayesian network. The probability of any partial instantiation of the variables can now be computed simply by setting to 1 the indicators corresponding to the variable values in the instantiation, and to 0 the indicators for all other values of the instantiated variables. This allows arbitrary marginal and conditional queries to be answered in time linear in the size of the polynomial.

Unfortunately, the size of the network polynomial is exponential in the number of variables, but it can be more compactly represented using an *arithmetic circuit*. An arithmetic circuit is a rooted, directed acyclic graph whose leaves are numeric constants or variables, and whose interior nodes are addition and multiplication operations. The value of the function for an input tuple is computed by setting the variable leaves to the corresponding values and computing the value of each node from the values of its children, starting at the leaves. In the case of the network polynomial, the leaves are the indicators and network parameters. The arithmetic circuit avoids the redundancy present in the network polynomial, and can be exponentially more compact.

Every junction tree has a corresponding arithmetic circuit, with an addition node for every instantiation of a separator, a multiplication node for every instantiation of a clique, and a summation node as the root. Thus one way to compile a Bayesian network into an arithmetic circuit is via a junction tree. However, when the network contains context-specific independences, a much more compact circuit can be obtained. Darwiche (2003) describes one way to do this, by encoding the network into a special logical form, factoring the logical form, and extracting the corresponding arithmetic circuit.

4 LEARNING ARITHMETIC CIRCUITS

4.1 SCORING AND SEARCHING

Instead of learning a Bayesian network and then compiling it into a circuit, we induce an arithmetic circuit directly from data using a score function that penalizes circuits with more edges. The score of an arithmetic circuit C on an i.i.d. training sample T is

$$\text{score}(C, T) = \log P(T|C) - k_e n_e(C) - k_p n_p(C)$$

where the first term is the log-likelihood of the training data, $P(T|C) = \prod_{X \in T} P(X|C)$, $k_e \geq 0$ is the per-edge penalty, $n_e(C)$ is the number of edges in the circuit, $k_p \geq 0$ is the per-parameter penalty, and $n_p(C)$ is the number of parameters in the circuit. The last two allow us to easily combine our inference-cost penalty with a more traditional one based on model complexity.

We use this formulation for simplicity; our algorithm would work equally well with a Bayesian Dirichlet score (Heckerman et al., 1995), with a prior of the form $\exp(-k_e n_e(C) - k_p n_p(C))$, since the computation of the marginal likelihood would be the same as in standard Bayesian network learning. Aside from its practical utility, a prior penalizing inference cost is meaningful if we believe the inference task being modeled can be carried out quickly, for example because humans do it. Either way, the main difficulty is that the penalty (or prior) is no longer node-decomposable, and repeatedly computing it might be very expensive. Reducing this cost is one of the key technical issues addressed in this paper.

Arithmetic circuits can be learned in the same way as Bayesian networks with local structure, by starting with an empty network and greedily applying the best splits, except that candidate structures are scored by compiling them into arithmetic circuits. However, compiling an arithmetic circuit can be computationally costly, and doing so for every candidate structure would be prohibitive. A better approach

Table 1: Greedy algorithm for learning arithmetic circuits.

```
function LearnAC(T)
initialize circuit C as product of marginals
loop
  C_best ← C
  for each valid split S(D,V) do
    C' ← SplitAC(C, S(D,V))
    if score(C',T) > score(C_best,T) then
      C_best ← C'
    end if
  end for
  if score(C_best,T) > score(C,T) then
    C ← C_best
  else
    return C
  end if
end loop
```

is to incrementally compile the circuit as splits are applied. Table 1 shows pseudo-code for this algorithm.

The algorithm begins by constructing the initial arithmetic circuit C as a product of marginal distributions:

$$C = \prod_i \sum_j I(X_i = x_{ij}) P(X_i = x_{ij})$$

This initial circuit is equivalent to a Bayesian network with no edges. In each iteration, the algorithm greedily chooses and applies the best valid split, where split validity is defined according to the equivalent Bayesian network. Each split is scored by applying it to the current circuit and counting the edges and parameters.[2] Learning ends when the algorithm reaches a local optimum, where no valid split improves the score.

4.2 SPLITTING DISTRIBUTIONS

The key subroutine is SplitAC, which updates an arithmetic circuit without recompiling it from scratch. Given an arithmetic circuit C that is equivalent to a Bayesian network B and a valid split $S(D, V)$, SplitAC returns a modified circuit C' that is equivalent to B after applying split $S(D, V)$. We will use the following notation to refer to distributions, parameter nodes, and indicator nodes:

d_j**:** Parameter node corresponding to the jth probability in the multinomial distribution D.

[2]All model parameters are MAP estimates, using a Dirichlet prior with all hyperparameters $\alpha_{ijk} = 1$, where k ranges over the leaves of the decision tree for variable X_i.

Table 2: Subroutine that updates an arithmetic circuit C by splitting distribution D on variable V.

```
function SplitAC(C, S(D,V))
MA ← mutual ancestors of D and V
N ← all nodes between MA and V or D
for i ∈ Domain(V) do
  create new parameter nodes d_ij
  N_i ← copy of all nodes in N
  for each n ∈ N do
    let n_i be the copy of n in N_i
    for each child c of n do
      if c = v_i or c is inconsistent with v_i then
        skip
      else if c is some parameter node d_j then
        insert edge from n_i to d_ij
      else if c ∈ N then
        let c_i be the copy of c in N_i
        insert edge from n_i to c_i
      else
        insert edge from n_i to c
      end if
    end for
  end for
end for
for m ∈ MA do
  n_V ← child of m that is a V-ancestor
  n_D ← child of m that is a D-ancestor
  for i ∈ Domain(V) do
    n'_V ← copy of n_V in N_i
    n'_D ← copy of n_D in N_i
    n_×i ← v_i × n'_V × n'_D
  end for
  n_+ ← Σ_i n_×i
  replace m's children n_V and n_D with n_+
end for
```

D_i**:** Leaf distribution resulting from split $S(D, V)$ that replaces D when $V = i$.

d_{ij}**:** Parameter node corresponding to the jth probability in D_i.

v_i**:** Indicator node $I(V = i)$.

Table 2 contains pseudo-code for the splitting algorithm. It might at first appear that to split D on V it suffices to replace references to each d_j with a sum of products, $\sum_i d_{ij} v_i$. However, the resulting circuit would then be correct only when V is fixed to a particular value, and summing out V would produce inconsistent results. Intuitively, the circuit must maintain the running intersection property of the corresponding junction tree, so that no variable can take on different values in different subcircuits.

SplitAC accomplishes this by duplicating the relevant subcircuits and "conditioning" each copy on a different value of V. This duplication is the reason different splits can have widely different edge costs. We now describe the details of which nodes are duplicated and how they are connected.

Definition 3. *We define three special types of node in the circuit as follows:*

- *A* D-ancestor *is any leaf* d_j *corresponding to a parameter of* D*, or any parent of a* D*-ancestor.*
- *A* V-ancestor *is any leaf* v_i *corresponding to an indicator of* V*, or any parent of a* V*-ancestor.*
- *A* mutual ancestor (MA) of D and V *is a node that is both a* D*-ancestor and a* V*-ancestor, and has no child that is both a* D*-ancestor and a* V*-ancestor.*

Let N be the set of all D-ancestors and V-ancestors that are also descendants of a mutual ancestor. These are all the nodes "in between" D and V that must agree on the value of V. For each value i in the domain of V, SplitAC creates a copy N_i of the nodes in N.

Let $n_i \in N_i$ be the copy of node $n \in N$. SplitAC inserts edges from n_i to its children as follows. If n has a child $c \in N$, then it inserts an edge from n_i to the corresponding copy c_i. If n has a child $c \notin N$, then it inserts an edge from n_i to c. This minimizes node duplication by linking to existing nodes or copies whenever possible.

A few additional changes are required for N_i to properly depend on v_i. If $n_i \in N_i$ has some parameter node d_j as a child, SplitAC replaces it with d_{ij}. This is how the new leaf distributions, conditioned on V, are integrated into the circuit. Secondly, if n_i has v_i as a child, it should be omitted: every node in N_i will depend on v_i, so this is redundant. Finally, if n_i has a child that is an ancestor of some v_j but not of v_i, then that child is inconsistent with conditioning on v_i and must be removed.

Finally, SplitAC connects each mutual ancestor, m, to a sum over these copies. This relies on the properties of mutual ancestors expressed in the following lemma (see appendix).

Lemma 1. *Every mutual ancestor of* D *and* V *is a multiplication node and has exactly one child that is a* V*-ancestor and one that is a* D*-ancestor.*

Let n_V and n_D be the children of m that are ancestors of V and D, respectively. SplitAC removes n_D and n_V as children of m and replaces them by an addition node with one child for each value of V. The ith child of the addition node is a product of v_i, the copy of n_D from N_i, and the copy of n_V from N_i. (If m was an ancestor of only certain values of V, the addition node sums only over those values.)

Intuitively, the resulting circuit represents the correct probability distribution because D has been replaced with the split distributions D_i, each conditioned on v_i, and because the circuit satisfies the running intersection property, since all nodes between V and D now depend on V.

Theorem 2. *After each iteration of LearnAC,* C *computes the network polynomial of a Bayesian network constructed by starting with an empty network and applying the same splits that were applied to* C *up to that iteration.*

Complete proofs can be found in the appendix.

4.3 OPTIMIZATIONS

We now discuss optimizations necessary to make this algorithm practical for real-world datasets with many variables.

Consider once again the high-level overview in Table 1. Scoring every possible circuit in every iteration would be very expensive. Choosing the split that leads to the best scoring circuit is equivalent to choosing the split that leads to the greatest increase in score, so we can store changes in score instead. The improvement in log-likelihood is not affected by other splits, and so this only needs to be computed once for each potential split. Unfortunately, the number of edges that a split adds to the circuit can increase or decrease due to other splits. For convenience, we will refer to the number of edges added by the application of a split as its *edge cost*.

As a simple example, consider a chain-structured junction tree of 5 variables: AB-BC-CD-DE-EF. If we add an arc from A to F, then A is added to every other cluster: AB-ABC-ACD-ADE-AEF. However, this also reduces the cost of adding an arc from A to E, since the two variables now appear together in a cluster. As a second example, suppose that we instead added an arc from B to F: AB-BC-BCD-BDE-BEF. Now the cost of adding an arc from A to F is greatly increased, since adding a variable to a larger cluster costs more edges than adding a variable to a smaller cluster.

Evaluating the edge cost of every potential split in every iteration is expensive. The number of potential splits is linear in the number of splits that have been performed so far, leading to a time complexity that is at least quadratic in the total number of splits. Further, computing the edge cost for a single candidate may be linear in the size of the current circuit. With a non-zero edge cost, circuit size tends to be linear in the number of iterations, leading to an $O(n^3)$ algorithm. While this is still polynomial, it makes learning models with thousands of splits intractable in practice.

Fortunately, most splits only change a fraction of edge costs. Determining exactly which costs need to be updated is difficult, but we can rule out many splits whose costs do not need to be updated using the following conservative rule. Applying one split may change the edge cost of another split $S(D, V)$ if the applied split changes a node that is an ancestor of D and not V, or of V and not D. This covers all nodes that lie between D or V and their mutual

ancestors, and thus all nodes that are copied by the splitting procedure. An applied split changes a node when it copies that node or reduces the number of children it has. In practice, this single heuristic lets us avoid recomputing over 95% of the edge costs.

As an alternative to this optimization, we have found a heuristic that leads to even larger speed-ups, but at the cost of no longer being perfectly greedy. We noticed that when edge costs changed, they rarely decreased. If a split's last computed edge cost was always a valid lower bound on the true value, then we could ignore any split whose total estimated score was worse than the best split found so far in this iteration. This assumption is often not valid in practice, but it lets us learn models that are nearly as effective in an order of magnitude less time.

Two other optimizations combine well with either of the above to offer further gains. First, we can reduce the number of computations by placing potential splits in order of decreasing likelihood gain, so that we consider the splits with the highest possible scores first. Since the likelihood gain is an upper bound on the score gain, once the score of the best split found so far is greater than the next likelihood gain, this split is guaranteed to be the highest-scoring one overall.

Second, we can exit the edge calculation procedure once we know that the edge cost is sufficient to make the overall score negative. It is also possible to exit once we know that the score of the current split will be worse than the best split so far, but this interferes with the other optimizations. If we only compute an upper bound on the score, we will often have to recompute the edge cost when the next iteration requires a slightly lower upper bound.

5 EXPERIMENTS

5.1 DATASETS

We evaluated our methods on three widely used real-world datasets. The KDD Cup 2000 clickstream prediction dataset (Kohavi et al., 2000) consists of web session data taken from an online retailer. Using the subset of Hulten and Domingos (2002), each example consists of 65 Boolean variables, corresponding to whether or not a particular session visited a web page matching a certain category. Anonymous MSWeb is visit data for 294 areas (Vroots) of the Microsoft web site, collected during one week in February 1998. It can be found in the UCI machine learning repository (Blake & Merz, 2000). EachMovie[3] is a collaborative filtering dataset in which users rate movies they have seen. We took a 10% sample of the original dataset, focused on the 500 most-rated movies, and reduced each variable to "rated" or "not rated". For KDD Cup and MSWeb, we used the training and test partitions provided with the datasets. For EachMovie, we randomly selected 10% of the data for the test set and used the remainder for training.

[3]Provided by Compaq at http://research.compaq.com/SRC/eachmovie/; no longer available for download, as of October 2004.

Table 3: Summary of experimental datasets.

Domain	Vars.	Train Exs.	Test Exs.	Density
KDD Cup	65	199,999	34,955	0.0079
MSWeb	294	32,711	5,000	0.0102
EachMovie	500	6,117	591	0.0581

Basic statistics for each dataset are shown in Table 3. Density refers to the fraction of non-zero entries across all examples and all variables.

5.2 LEARNING

For each dataset, we randomly split the training data into tuning and validation sets, corresponding to 90% and 10% of the training data, respectively. All parameters were tuned by training models on the tuning data and selecting the parameter sets that led to the highest log likelihood of the validation set. Finally, models were retrained using the full training set. All experiments were run on CPUs with 4 GB of RAM running at 2.8 GHz.

We used two versions of the algorithm for learning arithmetic circuits from Section 4: AC-Greedy, which guarantees that we pick the best split in each iteration, and AC-Quick, which uses a heuristic to avoid recomputing edge costs but may sometimes choose worse splits. We varied the per-edge cost k_e from 1.0 to 0.01. Not surprisingly, our models were most accurate on the validation set with low per-edge costs (0.01 or 0.02). We also tuned the per-parameter cost k_p. For KDD Cup, the best cost was 0.0; for MSWeb and EachMovie, the best costs were 1.0 for greedy ACs and 0.5 for quick ACs.

We used the WinMine Toolkit(Chickering, 2002) as a baseline. WinMine implements the algorithm for learning Bayesian networks with local structure described in Section 2 (Chickering et al., 1997), and has a number of other state-of-the-art features. We tuned WinMine's multiplicative per-parameter penalty κ; the best values were: 1 (no penalty) for KDD Cup, 0.1 for MSWeb, and 0.01 for EachMovie. We looked into using thin junction trees as a second baseline, but they do not scale to datasets of these dimensions.

A summary of the learned models appears in Table 4. For each dataset, we report the log-likelihood per example on the test data, the number of edges in the arithmetic circuit, the number of leaves across all decision trees, the average

and maximum number of parents across all variables, and the training time.

The test-set log-likelihoods of the AC learners and WinMine are very similar, with WinMine having a slight edge. This is not surprising, given that WinMine is free to choose expensive splits. Perhaps more remarkable is that this freedom translates to very little improvement in likelihood. The difference in accuracy between quick and greedy ACs is negligible except in the case of EachMovie, where the greedy AC is actually less accurate because it did not converge in the allowed time (72h).

Not surprisingly, WinMine is much faster than the AC learners. It is worth noting that the cost of learning is only incurred once, while the cost of inference is incurred many times. Also, the AC learner directly outputs an arithmetic circuit, while WinMine's Bayesian network would still have to be compiled into one, which can be very time-consuming. Finally, the quick heuristic offers up to an order-of-magnitude speedup with similar accuracy; additional heuristics might offer additional improvements.

We tried converting WinMine's networks to CNF representations and compiling them to d-DNNF with C2D[4], as described by Darwiche (2002). This process ran out of memory on all models. We tried reducing κ as an indirect way of coaxing WinMine into learning simpler models that could be compiled into circuits. Of these models, the only one we were able to compile was KDD Cup with $\kappa = 10^{-4}$. The compiled circuit had 197 million edges (over 500 times as many as our circuits) and was less accurate on the test data.

Table 4: Summary of Learned Models

KDD Cup	AC-Greedy	AC-Quick	WinMine
Log-likelih.	-2.16	-2.16	-2.16
Edges	382K	365K	>200M
Leaves	4574	4463	2267
Avg. parents	13.2	13.0	16.3
Max. parents	37	36	35
Time	50h	3h	3m

MSWeb	AC-Greedy	AC-Quick	WinMine
Log-likelih.	-9.85	-9.85	-9.69
Edges	204K	256K	>200M
Leaves	1353	1870	1710
Avg. parents	2.5	3.1	5.2
Max. parents	114	127	94
Time	8h	3h	2m

EachMovie	AC-Greedy	AC-Quick	WinMine
Log-likelih.	-55.7	-54.9	-53.7
Edges	155K	372K	>200M
Leaves	4070	6521	4830
Avg. parents	5.0	6.5	8.0
Max. parents	13	17	27
Time	>72h[5]	22h	3m

5.3 INFERENCE

For each dataset, we used the test data to generate queries with varied numbers of randomly selected query and evidence variables. Each query asked the probability of the configuration of the query variables in the test example conditioned on the configuration of the evidence variables in the same test example.

We estimate inference accuracy as the mean log probability of the test examples's configuration across all test examples. This is an approximation (up to an additive constant) of the Kullback-Leibler divergence between the inferred distribution and the true one, estimated using the test samples. For KDD Cup and MSWeb, we generated queries from 1000 test examples; for EachMovie, we generated queries from all 593 test examples.

For the arithmetic circuits, we used exact inference. For the Bayesian networks learned using WinMine, we used Gibbs sampling. We initialized the sampler to a random state, ran it for a burn-in period, and then collected samples to estimate the probability of the queried marginal or conditional event. All estimates were smoothed by uniformly distributing a count of 1 across all states of the query variables. Since convergence is difficult to diagnose and may take prohibitively long, we ran Gibbs sampling in three scenarios: fast (one chain, 100 burn-in iterations, 1000 sampling iterations); medium (ten chains, 100 burn-in iterations, 1000 sampling iterations); and slow (ten chains, 1000 burn-in iterations, 10,000 sampling iterations). For the camera-ready version, we plan to also run Gibbs sampling for 100,000 and one million iterations in each of 10 chains.

Figure 1 shows the relative accuracy of the different methods on each dataset. Per-variable query log-likelihood is on the y axis. In the graphs on the left, each query included 30% of the variables in the domain, conditioned on 0% to 50% of the domain variables as evidence. In the graphs on the right, the number of query variables varies from 10% to 50%, conditioned on 30% of the variables in the domain as evidence. Inference times (averaged over all queries) are listed in Table 5. Note that AC inference times are in milliseconds, while Gibbs inference times are in seconds.

The ACs were roughly one order of magnitude faster than the fastest runs of Gibbs sampling, and three orders of mag-

[4]Available at http://reasoning.cs.ucla.edu/c2d/. We also tried using the ACE package, but it does not support decision tree CPDs and, for our models, tabular CPDs would be prohibitively large.

[5]AC-Greedy did not finish running in the maximum allowed time of 72h. As a result, it has fewer edges and lower log-likelihood than AC-Quick.

Table 5: Average inference time per query.

Algorithm	KDD Cup	MSWeb	EachMovie
AC-Greedy	194ms	91ms	62ms
AC-Quick	198ms	115ms	162ms
Gibbs-Fast	1.46s	1.89s	7.22s
Gibbs-Medium	11.3s	15.6s	42.5s
Gibbs-Slow	106s	154s	452s

nitude faster than the slowest. Except when the number of query variables is very small, the ACs also easily dominate even the slowest runs of Gibbs sampling on accuracy. Because of the approximate inference, the slightly higher test-set log-likelihood of WinMine's models does not translate into higher accuracy in answering queries. Presumably, given enough time Gibbs sampling will eventually catch up with the ACs in accuracy, but by then it will be many orders of magnitude slower. Further, Gibbs sampling (like other approximate inference methods) requires tuning for best results, and we can never be sure that it has converged. In contrast, the AC inference is reliable, the time it takes is predetermined, and the time is short enough for online or interactive use.

6 CONCLUSION

In the past, work on learning and inference in graphical models has been largely separate. This has had the somewhat paradoxical result that much computational effort is often expended to learn accurate models, only to result in less accurate predictions when approximate inference becomes necessary. Our work seeks to ameliorate this by more closely integrating learning and inference. In particular, we presented an algorithm for learning arithmetic circuits by maximizing likelihood with a penalty on circuit size. This ensures efficient inference while still providing great modeling flexibility. In experiments on real-world domains, our algorithm outperformed standard Bayesian network learning on both accuracy of query answers and speed of inference.

Directions for future work include: investigating other algorithms for learning arithmetic circuits; extending our approach to handle learning with missing data and hidden variables; applying it to Markov networks, continuous domains, and relational representations; etc.

Acknowledgements

The authors wish to thank Mark Chavira, Adnan Darwiche, and Knot Pipatsrisawat for assistance in properly applying c2d to our Bayesian networks. This research was funded by a Microsoft Research fellowship awarded to the first author, DARPA contracts NBCH-D030010/02-000225, FA8750-07-D-0185, and HR0011-07-C-0060, DARPA grant FA8750-05-2-0283, NSF grant IIS-0534881, and ONR grant N-00014-05-1-0313. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, NSF, ONR, or the United States Government.

References

Arnborg, S., Corneil, D. W., & Proskurowski, A. (1987). Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic & Discrete Methods*, *8*, 277–284.

Blake, C., & Merz, C. J. (2000). *UCI repository of machine learning databases*. Dept. ICS, UC Irvine, CA. http://-www.ics.uci.edu/∼mlearn/MLRepository.html.

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

Chechetka, A., & Guestrin, C. (2008). Efficient principled learning of thin junction trees. In *NIPS 20*.

Chickering, D., Heckerman, D., & Meek, C. (1997). A Bayesian approach to learning Bayesian networks with local structure. *Proc. UAI-97* (pp. 80–89).

Chickering, D. M. (2002). *The WinMine toolkit* (Tech. Rept. MSR-TR-2002-103). Microsoft, Redmond, WA.

Darwiche, A. (2002). A logical approach to factoring belief networks. *Proc. KR-02* (pp. 409–420).

Darwiche, A. (2003). A differential approach to inference in Bayesian networks. *J. ACM*, *50*, 280–305.

Friedman, N., & Goldszmidt, M. (1996). Learning Bayesian networks with local structure. *Proc. UAI-96* (pp. 252–262).

Gilks, W. R., Richardson, S., & Spiegelhalter, D. J. (Eds.). (1996). *Markov chain Monte Carlo in practice*. Chapman and Hall.

Heckerman, D., Geiger, D., & Chickering, D. M. (1995). Learning Bayesian networks. *Mach. Learn.*, *20*, 197–243.

Hulten, G., & Domingos, P. (2002). Mining complex models from arbitrarily large databases in constant time. *Proc. KDD-02* (pp. 525–531).

Jaeger, M., Nielsen, J., & Silander, T. (2006). Learning probabilistic decision graphs. *Intl. J. Approx. Reasoning*, *42*, 84–100.

Kohavi, R., Brodley, C., Frasca, B., Mason, L., & Zheng, Z. (2000). KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, *2*, 86–98.

Lowd, D., & Domingos, P. (2005). Naive Bayes models for probability estimation. *Proc. ICML-05* (pp. 529–536).

Meila, M., & Jordan, M. (2000). Learning with mixtures of trees. *J. Mach. Learn. Research*, *1*, 1–48.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems*. Morgan Kaufmann.

Roth, D. (1996). On the hardness of approximate reasoning. *Artif. Intel.*, *82*, 273–302.

Srebro, N. (2000). Maximum likelihood Markov networks: An algorithmic approach. Master's thesis, MIT, Cambridge, MA.

A PROOFS

In this section, we present full proofs for the correctness of our learning algorithm. We do this with the help of Theorem 1 from Darwiche (2002), which says that a logical circuit satisfying certain properties and whose models are the terms of the network polynomial can be converted to an arithmetic circuit that computes the network polynomial. Since what we have is an arithmetic circuit, we will describe an inverse transformation from an arithmetic circuit to a logical circuit. By showing that this logical circuit satisfies certain properties and that its models are the terms of the network polynomial, we can apply Darwiche's Theorem 1 to conclude that the arithmetic circuit computes the network polynomial.

A.1 BACKGROUND

We begin by restating the central theorem from Darwiche, along with the necessary supporting definitions:

Theorem 3 (Theorem 1 (Darwiche, 2002))**.** *Let Δ be a smooth d-DNNF which encodes a multi-linear function f. The arithmetic circuit encoded by Δ implements the function f.*

A negated normal form (NNF) is a "rooted, directed acyclic graph in which each leaf node is labeled with a literal, *true* or *false*, and each internal node is labeled with a conjunction $\wedge$ or a disjunction $\vee$." (Darwiche, 2002) To highlight the correspondence between arithmetic circuits and NNFs, we will sometimes refer to NNFs as "logical circuits."

For a node n in an NNF, $Vars(n)$ refers to the set of all propositional variables that are descendants of n, and $\Delta(n)$ refers to the formula represented by n and its descendants.

A smooth d-DNNF is an NNF satisfying the three following properties, taken directly from Darwiche (2002):

- Smoothness holds when $Vars(n_i) = Vars(n_j)$ for any two children n_i and n_j of an or-node n.
- Determinism holds when $\Delta(n_i) \wedge \Delta(n_j)$ is logically inconsistent for any two children n_i and n_j of an or-node n.
- Decomposability holds when $Vars(n_i) \cap Vars(n_j) = \emptyset$ for any two children n_i and n_j of an and-node n.

The multi-linear function encoded by Δ is a polynomial in which each term corresponds to a satisfying assignment, or model, of Δ. Each term is constructed as the product of all true variables in the assignment.

The arithmetic circuit encoded by Δ refers to the circuit obtained by replacing each conjunction with multiplication, each disjunction with addition, and each negative literal with 1.

A.2 PROPERTIES OF ARITHMETIC CIRCUITS

We now prove certain properties of the arithmetic circuits which are necessary for proper operation of the algorithm (e.g., Lemma 1) as well as other later proofs. These properties are analogous to the logical circuit properties discussed by Darwiche (2002).

From Section 3, recall that our arithmetic circuits are rooted, directed acyclic graphs in which leaf nodes are indicators or network parameters, and internal nodes are addition or multiplication operations.

We use $INodes(n, V, C)$ to denote the set of indicator nodes associated with variable V that are descended from node n in circuit C. Similarly, $PNodes(n, V, C)$ denotes the set of parameter nodes associated with variable V (possibly from many different conditional distributions) that are descended from node n in circuit C.. We further define $IVars(n, C)$ and $PVars(n, C)$ as the sets of variables corresponding to the indicator and parameter nodes descended from n:

$$IVars(n, C) = \{V | INodes(n, V, C) \neq \emptyset\}$$
$$PVars(n, C) = \{V | PNodes(n, V, C) \neq \emptyset\}$$

These function are parameterized with a circuit as well as a node in order to allow for distinctions between the properties of a node before and after a call to SplitAC.

For each of the stated NNF properties, we can define an analogous property on an arithmetic circuit, C:

- Smoothness holds when $IVars(n_i, C) = IVars(n_j, C)$ and $PVars(n_i, C) = PVars(n_j, C)$ for any two children n_i and n_j of an addition node n.
- Determinism holds when there is some V such that $INodes(n_i, V, C) \neq \emptyset$, $INodes(n_j, V, C) \neq \emptyset$, and $INodes(n_i, V, C) \cap INodes(n_j, V, C) = \emptyset$ for any two children n_i and n_j of an addition node n.
- Decomposability holds when $IVars(n_i, C) \cap IVars(n_j, C) = \emptyset$ and $PVars(n_i, C) \cap PVars(n_j, C) = \emptyset$ for any two children n_i and n_j of a multiplication node n.

We also refer to the smoothness, determinism, and decomposability of nodes. The node properties are defined in the obvious way, so that a circuit is smooth, deterministic, and

decomposable if and only if each of its nodes is smooth, deterministic, and decomposable.

Since we are using arithmetic circuits to represent probability distributions, one can intuitively look at smoothness and determinism as ensuring that our addition nodes are summing out random variables, and decomposability as ensuring that our multiplication nodes are combining the probabilities of independent groups of variables.

We will later demonstrate a correspondence between smooth, deterministic, decomposable arithmetic circuits and smooth, deterministic, decomposable NNFs, also called smooth d-DNNFs. These properties are therefore essential for later stages of the proof. Since each iteration of LearnAC implicitly depends on the operations of SplitAC, we first prove a weakened version of Lemma 1.

Lemma 4. *If C is smooth and decomposable, then every mutual ancestor (MA) of V and D is a multiplication node with one child that is a V-ancestor and one that is a D-ancestor.*

Proof. By definition, any MA n must be an ancestor of both V and D. Let n_V by a child of n that is also an ancestor of V and let n_D be a child of n that is also an ancestor of D. If n was an addition node, then n_D would be a V-ancestor, violating the condition that no child of an MA is an ancestor of both V and D. Therefore, every MA is a times node.

By decomposability, n can have at most one child that is an ancestor of D and one child that is an ancestor of V. By definition, every MA must have at least one that is an ancestor of each, and the two cannot be the same. □

Let C' be the circuit that results from calling SplitAC$(C, S(D, V))$.

Lemma 5. *If C is smooth, deterministic, and decomposable, then for each node $n \in C$:*

If $n \in C'$, then $PVars(n, C') = PVars(n, C)$; $IVars(n, C') = IVars(n, C)$; and for any domain variable U, $INodes(n, U, C') = INodes(n, U, C)$.

If n was copied, then for each copy $n' \in C'$, $PVars(n', C') = PVars(n, C)$; $IVars(n', C') = IVars(n, C) - V$; and for any domain variable $U \neq V$, $INodes(n', U, C') = INodes(n, U, C)$.

Proof. We prove this lemma by induction on the structure of C', showing that if the lemma is true for all children of n, then it is also true for n.

Base case: If $n \in C$ has no children and $n \in C'$, then n clearly has identical descendants in C' as in C, so all conditions are satisfied. If $n \notin C'$, then the lemma does not apply, since SplitAC never copies nodes without children.

Inductive step: Suppose that the lemma is true for each child of $n \in C$ and that n has one or more children, n_i.

Case 1: If $n \in C'$ and is not an MA, then it was not directly changed by SplitAC. n has the same children in C and C', and for each child n_i, $n_i \in C$. We begin by writing $IVars(n, C')$ recursively in terms of its children:

$$IVars(n, C') = \bigcup_i IVars(n_i, C')$$

By the inductive hypothesis:

$$IVars(n, C') = \bigcup_i IVars(n_i, C) = IVars(n, C)$$

An identical argument applies to show that $PVars(n, C') = PVars(n, C)$.

For an arbitrary variable, U:

$$INodes(n, U, C') = \bigcup_i INodes(n_i, U, C')$$

By the inductive hypothesis:

$$= \bigcup_i INodes(n_i, U, C) = INodes(n, U, C)$$

Case 2: If a copy of n is present in C', then let $n' \in C'$ be an arbitrary copy of n. From the operation of SplitAC, the children of n can be partitioned into three disjoint sets: N_{same}, nodes that are also children of n'; N_{copy}, nodes of which a copy is a child of n'; and N_{none}, nodes which are excluded from the children of n'. The children of n' can similarly be partitioned into two sets: N'_{copy}, copies of nodes that are children of n, and N_{same}.

We handle the $PVars/IVars$ and $INodes$ conditions as two separate sub-cases.

We prove the $INodes$ condition first. For an arbitrary variable, $U \neq V$:

$$INodes(n, U, C') = \bigcup_i INodes(n_i, U, C')$$

By the inductive hypothesis:

$$= \bigcup_i INodes(n_i, U, C) = INodes(n, U, C)$$

Addition Suppose n' is an addition node. By smoothness, $IVars(n'_i, C') = IVars(n'_j, C')$ for any two children of n'. Therefore, $IVars(n', C') = IVars(n'_i, C')$. $n \in C$ must also be an addition node, so $IVars(n, C) = IVars(n_i, C)$ for any child of n, n_i.

Let n'_i be an arbitrary child of n'. If $n'_i \in N'_{copy}$, then let $n_i \in N_{copy}$ be the node of which n'_i is a copy. By the

inductive hypothesis, $IVars(n'_i, C') = IVars(n_i, C) - V$, so by substitution, $IVars(n', C') = IVars(n, C) - V$.

Otherwise $n'_i \in N_{same}$, so by the inductive hypothesis, $IVars(n'_i, C') = IVars(n'_i, C)$. In the operation of SplitAC, children of copied nodes that are ancestors of V are also copied, so we can conclude that $IVars(n'_i, C) = IVars(n'_i, C) - V$. By substitution, $IVars(n', C) = IVars(n, C) - V$. An identical argument applies to show that $PVars(n', C) = PVars(n, C)$.

Multiplication Now suppose, instead, that n' is a multiplication node. We first prove, by contradiction, that $N_{none} = \emptyset$. Suppose, to the contrary, that there exists some $n_i \in N_{none}$. From the operation of SplitAC, there are indicator nodes v_i and v_j such that $v_j \in INodes(n_i, V, C)$ and $v_i \notin INodes(n_i, V, C)$. By decomposability and the definition of $IVars$, since $v_j \in INodes(n_i, V, C)$, $v_i \notin INodes(n_j, V, C)$ for any other child of n. Therefore, $v_i \notin INodes(n, V, C)$ and $v_j \in INodes(n, V, C)$, so n should be omitted from this copy as well. Since we originally assumed n' exists, $N_{none} = \emptyset$.

$IVars(n', C')$ can be written recursively as follows:

$$IVars(n', C') = \bigcup_{n'_i \in N_{same} \cup N'_{copy}} IVars(n'_i, C')$$

As pointed out in the addition case, children of copied nodes that are ancestors of V are also copied, so for $n_i \in N_{same}$, $IVars(n_i, C) = IVars(n_i, C) - V$. Combined with the inductive hypothesis, we may conclude:

$$IVars(n', C') = \bigcup_{n_i \in N_{same} \cup N_{copy}} IVars(n_i, C) - V$$

Since $N_{none} = \emptyset$, adding it in a union changes nothing:

$$IVars(n', C') = \bigcup_{n_i \in N_{same} \cup N_{copy} \cup N_{empty}} IVars(n_i, C) - V$$

Since every child of n is an element in N_{same}, N_{copy}, or N_{none}, this is the recursive statement of $IVars(n, C) - V$:

$$IVars(n', C') = IVars(n, C) - V$$

An identical argument applies to show that $PVars(n', C) = PVars(n, C)$.

Case 3: If n is an MA for the split, then from Lemma 4 we know n is a multiplication node and its children include exactly one ancestor of D, n_D; exactly one ancestor of V, n_V; and a set of other children which we will call N_o.

From the operation of SplitAC, every $n_i \in N_o$ is unchanged by the algorithm, and therefore $n_i \in C'$. n_D and n_V, however, are replaced by an addition node, n'_+. The children of n_+ are products of some indicator node for V, v_i; a copy of n_V, $n'_{V,i}$; and a copy of n_D, $n'_{D,i}$.

We can write $PVars(n, C')$ as:

$$PVars(n, C') = PVars(n'_+, C') \cup \bigcup_{n_i \in N_o} PVars(n_o, C')$$

We describe $PVars(n'_+, C')$ in terms of n'_+'s grandchildren:

$$\bigcup_i PVars(n'_{D,i}, C') \cup PVars(n'_{V,i}, C') \cup V$$

V is included in the union since v_i is a child of the ith child of n_+. By the inductive hypothesis, since every $n'_{D,i}$ and $n'_{V,i}$ is a copy, $PVars(n'_{D,i}, C') = PVars(n_D, C)$ and $PVars(n'_{V,i}, C') = PVars(n_V, C)$. We can therefore substitute and simplify to obtain:

$$INodes(n'_+, U, C') = PVars(n_D, C) \cup PVars(n_V, C)$$

Substituting into our previous expresion for $PVars(n, C')$:

$$PVars(n, C') = PVars(n_V, C) \cup PVars(n_D, C) \cup \bigcup_{n_i \in N_o} PVars(n_o, C')$$

By the inductive hypothesis, $PVars(n_o, C') = PVars(n_o, C)$, so this reduces to the recursive description of $PVars(n, C)$.

Our procedure for proving that $INodes(n, U, C') = INodes(n, U, C)$ is nearly identical. First, we handle the case where $U \neq V$.

$$INodes(n, U, C') = INodes(n'_+, U, C') \cup \bigcup_{n_i \in N_o} INodes(n_o, U, C')$$

As before, we describe $INodes(n'_+, U, C')$ in terms of the grandchildren of n'_+:

$$\bigcup_i INodes(n'_{D,i}, U, C') \cup INodes(n'_{V,i}, U, C') \cup v_i$$

Note that we can safely ignore the v_is, since $U \neq V$. By applying the inductive hypothesis and simplifying, we obtain:

$$INodes(n'_+, U, C') = INodes(n_D, U, C) \cup INodes(n_V, U, C)$$

We can substitute this into the original expression:

$$INodes(n, U, C') = INodes(n_D, U, C) \cup INodes(n_V, U, C) \cup \bigcup_{n_i \in N_o} INodes(n_o, U, C')$$

which is equivalent to $INodes(n, U, C)$. Since $INodes(n, U, C') = INodes(n, U, C)$ for all U, it follows that $IVars(n, C') = IVars(n, C)$.

For the case where $U = V$, SplitAC explicitly includes the indicator node v_i as a grandchild of n_+ for each v_i that is a descendant of n. Therefore, the lemma holds for this case as well. □

Lemma 6. *At every iteration of LearnAC, C is smooth, decomposable, and deterministic.*

Proof. By induction.

Base case: We first show that the initial circuit is smooth, deterministic, and decomposable. The initial circuit is a product of sums of products. The leaves are indicator nodes for each value of each variable and parameter nodes for the marginal probability of each variable value. Above this are multiplication nodes, each the product of one parameter node and one indicator node, clearly satisfying decomposability. Above this are summations, each summing over the values and probabilities of a single variable. Since each child of a given plus node is a parent of values and parameters for the same variable, all addition nodes satisfy smoothness. Since the values being summed out are mutually exclusive, they also satisfy determinism. The top level multiplication is over marginal distributions for different variables, so it satisfies decomposability.

Inductive step: Let C be the circuit after the last iteration of LearnAC and let C' be the circuit that results from calling SplitAC($C, S(D, V)$). Assuming the circuit, C, was smooth, deterministic, and decomposable after the last iteration of LearnAC, we must show that C' is also smooth, deterministic, and decomposable.

We demonstrate this for each node $n' \in C'$. If $n' \in C$ and n' is not an MA, then each of its children n_i is also in C, since every path to a copied or created node leads through an MA. By Lemma 5, $PVars(n_i, C') = PVars(n_i, C)$, $IVars(n_i, C') = IVars(n_i, C)$, and $INodes(n_i, U, C') = INodes(n_i, U, C)$. By the inductive hypothesis, n' satisfied smoothness and decomposability before the split. Since $PVars$, $IVars$, and $INodes$ remain the same for all children, n must still satisfy smoothness and decomposability.

If n' is a copy of some $n \in C$, then each of its children n'_i is also a copy of a child of n, $n_i \in C$. Consider two children of n, n'_i and n'_j, and the corresponding children of n, n_i and n_j.

Suppose n' and n are addition nodes. By Lemma 5, $IVars(n'_i, C') = IVars(n_i, C) - V$. By smoothness, $IVars(n_i, C) - V = IVars(n_j, C) - V$. By Lemma 5, $IVars(n_j, C) - V = IVars(n'_j, C')$, so by transitivity $IVars(n'_i, C') = IVars(n'_j, C')$. The same argument can be applied to show that $PVars(n'_i, C') = PVars(n'_j, C')$, so n' is smooth.

Since n is deterministic, there must be some $U \in IVars(n_i, C)$ such that $INodes(n_j, U, C) \cap INodes(n_i, U, C) = \emptyset$. If $U \neq V$, then by applying Lemma 5 and transitivity we can infer that n' is deterministic. If $U = V$, then let v_i be the value of V that the particular copy n' is conditioned on during the operation of SplitAC. From determinism and our choice of V, v_i cannot be a descendant of both n_i and n_j for $n_i \neq n_j$. Since both n_i and n_j are ancestors of some indicator of V, but both are not ancestors of v_i, at most one child is copied. Therefore, n' only has one child and determinism is trivially satisfied.

This leaves the case of the MAs and newly created nodes. If n' is a newly created multiplication node, then its children are some indicator node v_i and copies $n'_{D,i}$ and $n'_{V,i}$ of two children the MA, n_D and n_V. By the inductive hypothesis, the MA satisfied decomposability before the split, so $IVars(n_V, C) \cap IVars(n_D, C) = \emptyset$ and $PVars(n_V, C) \cap PVars(n_D, C) = \emptyset$. By Lemma 5, $IVars(n'_V, C') = IVars(n_V, C) - V$, $IVars(n'_D, C') = IVars(n_D, C) - V$ (and $IVars(v_i, C') = V$). The $PVars$ are similar disjoint, demonstrating that n' satisfies decomposability.

If n' is a newly created addition node, then its children are newly created multiplication nodes, n'_i. By their construction in SplitAC:

$$IVars(n'_i, C') = V \cup IVars(n'_{D,i}, C') \cup IVars(n'_{V,i}, C')$$

From Lemma 5, since each $n'_{D,i}$ and $n'_{V,i}$ is a copy of the same n_D and n_V, their $IVars$ are identical. Therefore, $IVars(n'_i, C') = IVars(n'_j, C')$, so n' is smooth. By construction, $INodes(n'_i, V, C') = v_i$, so $INodes(n'_j, V, C') \cap INodes(n'_i, V, C') = \emptyset$ and decomposability is satisfied.

Finally, if n' is an MA node, then two of its children have been replaced by a new addition node, n'_+: $IVars(n'_+, C') = \prod_i IVars(n'_i, C')$, where each n'_i is a new multiplication node. Since we already showed that n'_+ is smooth, $IVars(n'_+, C') = IVars(n'_i, C')$. Expanding yields:

$$= V \cup IVars(n'_{D,i}, C') \cup IVars(n'_{V,i}, C')$$

Applying Lemma 5:

$$= V \cup (IVars(n_D, C) - V) \cup (IVars(n_V, C') - V)$$

Simplifying yields $IVars(n'_+, C') = IVars(n_D, C) \cup IVars(n_V, C)$. Applying the same argument demonstrates that $PVars(n'_+, C') = PVars(n_D, C) \cup PVars(n_V, C)$.

Given another child of n', n'_i, from Lemma 5 we know that $IVars(n'_i, C') = IVars(n_i, C)$. Therefore, we can write the intersection $IVars(n'_i, C') \cap IVars(n'_+)$ as:

$$IVars(n_i, C) \cap (IVars(n_D, C) \cup IVars(n_V, C))$$

By the distributive law:

$$\begin{aligned}=&(IVars(n_i, C) \cap IVars(n_D, C)) \\ &\cup (IVars(n_i, C) \cap IVars(n_V, C))\end{aligned}$$

Applying the inductive hypothesis that n is decomposable, we can conclude:

$$IVars(n_i', C') \cap IVars(n_+') = \emptyset \cup \emptyset = \emptyset$$

The same result holds for $PVars$. Given n_i' and n_j' where neither is n_+', Lemma 5 and the inductive hypothesis show that their respective $IVars$ and $PVars$ are also disjoint. Therefore, n' satisfies decomposability. □

A.3 PROPERTIES OF THE LOGICAL IMAGE

For an arithmetic circuit, C, that represents some Bayesian network over the variables $X_1, \ldots, X_n$, the *logical image* of C, $\mathcal{L}(C)$, is obtained by replacing addition with disjunction and multiplication with conjunction. In order to make the different values of each variable mutually exclusive, we replace indicator nodes v_i with conjunctions of v_i and the negation of every other v_j for $j \neq i$. We make the conditional probability parameters for each variable mutually exclusive with each other using an analagous transformation. We use $\mathcal{L}(n, C)$ to refer to the node in $\mathcal{L}(C)$ that replaces $n \in C$. For non-leaf $n' \in \mathcal{L}(C)$, we use $\mathcal{L}^-1(n', \mathcal{L}(C))$ to refer to the node in C that n' replaced.

If we take the logical image of C and replace conjunction with multiplication, disjunction with addition, and negated variables with 1, then we recover an arithmetic circuit that is equivalent to C. Therefore, from our earlier definitions, $\mathcal{L}(C)$ encodes the arithmetic circuit C.

Recall that $\Delta(n)$ refers to the logical formula represented by n and its descendants. We modify this notation as $\Delta(n, L)$, to specify the logical circuit L from which this subformula is drawn. For notational convenience, we will abbreviate $\Delta(\mathcal{L}(n, C))$ as $\Delta(n, C)$, where C is an arithmetic circuit.

Lemma 7. *If $\mathcal{L}(C)$ is smooth, then $\Delta(n, C)$ is false unless:*

- *For each $V \in IVars(n, C)$, the literal for exactly one indicator v_i of V is true, and $v_i \in INodes(n, V, C)$*
- *For each $V \in PVars(n, C)$, the literal for exactly one parameter d_i of V is true, and $d_i \in PNodes(n, V, C)$*

Proof. Suppose that $V \in IVars(n, C)$, and that in a particular truth assignment, more than one literal for V is true or no $v_i \in INodes(n, V, C)$ is true. By induction over the structure of C and $\mathcal{L}(C)$, we show that $\Delta(n, C))$ must be false. The exact same argument can be applied for the second condition, substituting $PVars$ and $PNodes$ for $IVars$ and $INodes$.

Base case: If node n has no children and $V \in IVars(n, C)$, then n must be an indicator node v_i. Recall that $\mathcal{L}(v_i, C)$ is a conjunction over the literals for each value of variable V, where only v_i appears non-negated. By the pigeon-hole principle, if more than one literal of V is true, then one of them must be negated in the conjunction, so $\Delta(v_i, C)$ is false. In the second case, if no $v_j \in INodes(v_i, V, C)$ is true, then v_i is false. Since $\mathcal{L}(v_i, C)$ is a conjunction that includes v_i, this implies that $\Delta(v_i, C)$ is false.

Inductive step: Suppose the lemma holds for all children of n. Consider first the case where $\mathcal{L}(n, C)$ is a conjunction. Since $V \in IVars(n, C)$, n must have some child n_i such that $V \in IVars(n_i, C)$. If no $v_i \in INodes(n, V, C)$ is true, then no $v_i \in INodes(n_i, V, C)$ is true either. By the inductive hypothesis, $\Delta(n_i, C)$ is false, so the conjunction $\Delta(n, C)$ is false. If more than one literal for V is true, then by the inductive hypothesis, $\Delta(n_i, C)$ is false, and hence $\Delta(n, C)$ is false.

Otherwise, $\mathcal{L}(n, C)$ is a disjunction. Since $V \in IVars(n, C)$ and $\mathcal{L}(C)$ is smooth, for every child n_i, $V \in IVars(n_i, C)$. If more than one literal for V is true, then by the inductive hypothesis, every child $\Delta(n_i, C)$ is false, so the disjunction $\Delta(n, C)$ is false. If no literal $v_i \in INodes(n, V, C)$ is true, then no $v_i \in INodes(n_i, V, C)$ is true either. By the inductive hypothesis, every $\Delta(n_i, C)$ is false, so $\Delta(n, C)$ is also false. □

Lemma 8. *The logical image of a smooth, deterministic, and decomposable arithmetic circuit is a smooth d-DNNF.*

Proof. Given an arithmetic circuit, C, $\mathcal{L}(C)$ is clearly an NNF. To show that it is a smooth d-DNNF, we must show that $\mathcal{L}(C)$ satisfies smoothness, determinism, and decomposability.

Every conjunction or disjunction $n' \in \mathcal{L}(C)$ is a replacement for some node $n \in C$. For each $n \in C$, we will demonstrate that the replacement $n' \in \mathcal{L}(C)$ satisfies smoothness, determinism, and decomposability.

If $n \in C$ has fewer than two children, then $n' \in \mathcal{L}(C)$ must either be a conjunction of literals representing all indicator or parameter nodes for a particular variable, or it must be a disjunction or conjunction with fewer than two children. In the former case, no two children represent the same literal so decomposability is satisfied and neither smoothness nor determinism applies. In the latter case, smoothness, determinism, and decomposability are all trivially satisfied by n'.

Otherwise, let n_i and n_j be two arbitrary children of n, and let n_i' and n_j' be the corresponding replacements in $\mathcal{L}(C)$.

Suppose that an indicator v_k (or parameter d_k) of variable V is in $Vars(n'_i)$, the set of all literals in the subgraph rooted at n'_i. n'_i must be a parent or ancestor of a parent of v_k (or d_k). Parents of literal nodes are conjunctions that were handled earlier, in the case where n has fewer than two children. Therefore, n'_i is an ancestor of a parent of v_k's literal (or d_k's), which is a conjunction that replaced some v_l (or d_l) in C. In the original circuit, n_i is therefore an ancestor of v_l (or d_l). We use this fact as a starting point for proving decomopsability, smoothness, and determinism.

If n is a multiplication node, then by decomposability, $V \notin IVars(n_j)$ (or $PVars(n_j)$), since $V \in IVars(n_i)$ (or $IVars(n_i)$). From how the logical image is constructed, v_k (or d_k) $\notin Vars(n'_j)$, where v_k (d_k) is any literal for an indicator (parameter) of V. Since the child nodes and descendant of n'_i were arbitrary, $Vars(n'_i) \cap Vars(n'_j) = \emptyset$, and n' satisfies decomposability.

Otherwise, n is an addition node. By smoothness, $V \in IVars(n_j)$ (or $PVars(n_j)$). From the construction of the logical image, n'_i and n'_j must both be ancestors of every indicator (or parameter) node for V. Since the child node was arbitrary, $Vars(n'_i) \subset Vars(n'_j)$. Since n'_i and n'_j are arbitrary, by symmetry $Vars(n'_j) \subset Vars(n'_i)$. Therefore, $Vars(n'_i) = Vars(n'_j)$, so n' satisfies smoothness.

From determinism, there must be some variable V such that n_i and n_j are both ancestors of one or more indicator nodes for V, but no indicator node v_k is the descendant of both. Suppose n'_i is true for a particular assignment of truth values to logical variables. By Lemma 7 and determinism, some descendant v_k of n_i must be true, where v_k is not a descendant of c_j. By Lemma 7, n'_j can only be true when exactly one indicator variable of V is true. Since we have selected v_k to be true and v_k is not a descendant of n_j, by Lemma 7 n'_j is false. Therefore, n'_j is false whenever n'_i is true, so n'_i and c'_j are logically inconsistent. Since the child nodes were arbitrary, n' satisfies determinism. □

Lemma 9. *After each iteration of LearnAC, the models of $\mathcal{L}(C)$ are the terms of the network polynomial for a Bayesian network constructed by starting with an empty network and applying the same splits that were applied to C up to that iteration.*

For a logical formula Δ, we will use the notation $\Delta[x/y]$ to refer to the logical formula obtained by replacing every occurrence of x in Δ with y.

Proof. (By induction.)

Base case: The initial circuit is a product of marginal distributions, equivalent to a Bayesian network with no arcs.

Inductive step: Suppose that, after k iterations, the models of $\mathcal{L}(C)$ are the terms of the network polynomial for some Bayesian network, B. In the $k+1$st iteration, $LearnAC$ selects and applies some split $S(D, V)$. Let B' and C' be the resulting Bayesian network and arithmetic circuit. The network polynomial of B' is identical to that of B except that in every term containing an indicator of V, v_i, and a parameter of distribution D, d_j, the parameter d_j is replaced by $d_{i,j}$. We will demonstrate that the models of $\mathcal{L}(C')$ are exactly these terms.

Let m be a mutual ancestor of D and V in C. From Lemma 1 and the logical image transformation,

$$\Delta(m, C) = \Delta(n_D, C) \wedge \Delta(n_V, C) \wedge \left(\bigwedge_{o \in O} \Delta(o, C)\right)$$

where n_D is the D-ancestor, n_V is the V-ancestor, and O is the set of all other children. Now consider how m is changed by calling $SplitAC(C, S(D, V))$. By construction,

$$\Delta(m, C') = \bigvee_j (\Delta(v_j, C') \wedge \Delta(n'_{D_j}, C') \wedge \Delta(n'_{V_j}, C')) \wedge \left(\bigwedge_{o \in O} \Delta(o, C')\right)$$

where $n'_{D,j}$ and $n'_{V,j}$ are copies of n_D and n_V, respectively. Since neither $o \in O$ nor any descendent of o is changed by the split, $\bigwedge_{o \in O} \Delta(o, C') = \bigwedge_{o \in O} \Delta(o, C)$. We will abbreviate this disjunction as $\Delta(O)$.

From Lemma 7, the literal of exactly one indicator for V is true in every model of $\Delta(m, C)$ and $\Delta(m, C')$. Let us restrict m to the case where a particular v_i is true. Note that we only need to apply the substitution to ancestors of v_i:

$$\Delta(m, C)[v_i/\top] = \Delta(O) \wedge \Delta(n_D, C) \wedge \Delta(n_V, C)[v_i/\top]$$

$$\Delta(m, C')[v_i/\top] = \Delta(O) \wedge \bigvee_j (\Delta(v_j, C')[v_i/\top] \wedge \Delta(n'_{D_j}, C') \wedge \Delta(n'$$

Since the indicators of V are mutually exclusive, the latter case simplifies as follows:

$$\mathcal{L}(m, C') = \Delta(O) \wedge \Delta(n'_{D_i}, C') \wedge \Delta(n'_{V_i}, C')$$

From the operation of SplitAC, the subtree rooted at n'_{V_i} is a copy of the subtree rooted at n_V, excluding v_i and every node that is an ancestor of some v_j but not v_i. When we apply the substitution $[v_i/\top]$, v_i becomes redundant. From Lemma 7, the other excluded nodes are all false when v_i is true. From decomposability, the other excluded nodes are never children of conjunctions or the parent conjunction would also be excluded. Therefore, the other excluded nodes are false children of disjunctions, which means they can be ignored when v_i is true. Since every excluded node is either redundant or irrelevant when v_i is true, $\Delta(n'_{V_i}, C') = \Delta(n_V, C)[v_i/\top]$.

By construction $\Delta(n_{D_i}, C')$ is identical to $\Delta(n_D, C)$ except with the parameter nodes of D replaced by those of

D_i. Since n_D is the only child of m that is an ancestor of D, we can conclude:

$$\begin{aligned}\Delta(m, C')[v_i/\top] &= \Delta(O) \wedge \Delta(n'_{D_i}, C') \wedge \Delta(n_V, C)[v_i/\top] \\ &= \Delta(m, C)[v_i/\top, d_j/d_{i,j}]\end{aligned}$$

Since every path from the root to D goes through some m, and since only mutual ancestors and their descendants were changed by SplitAC,

$$\mathcal{L}(C')[v_i/\top] = \mathcal{L}(C)[v_i/\top, d_j/d_{i,j}]$$

Since some v_i is true in every model of $\mathcal{L}(C)$ and $\mathcal{L}(C')$ (by Lemmas 7,6, and 8), the models of $\mathcal{L}(C')$ are the terms of the updated network polynomial. □

Proof of Theorem 2. By Lemma 6, C' is smooth, deterministic, and decomposable. From Lemma 8, we know that its logical image $\mathcal{L}(C')$ is a smooth d-DNNF. By Lemma 9, the models of $\mathcal{L}(C')$ are the terms of the updated network polynomial. It follows from the previous two statements and Theorem 1 from Darwiche (2002) that C' computes the network polynomial of B with the split $S(D, V)$. □

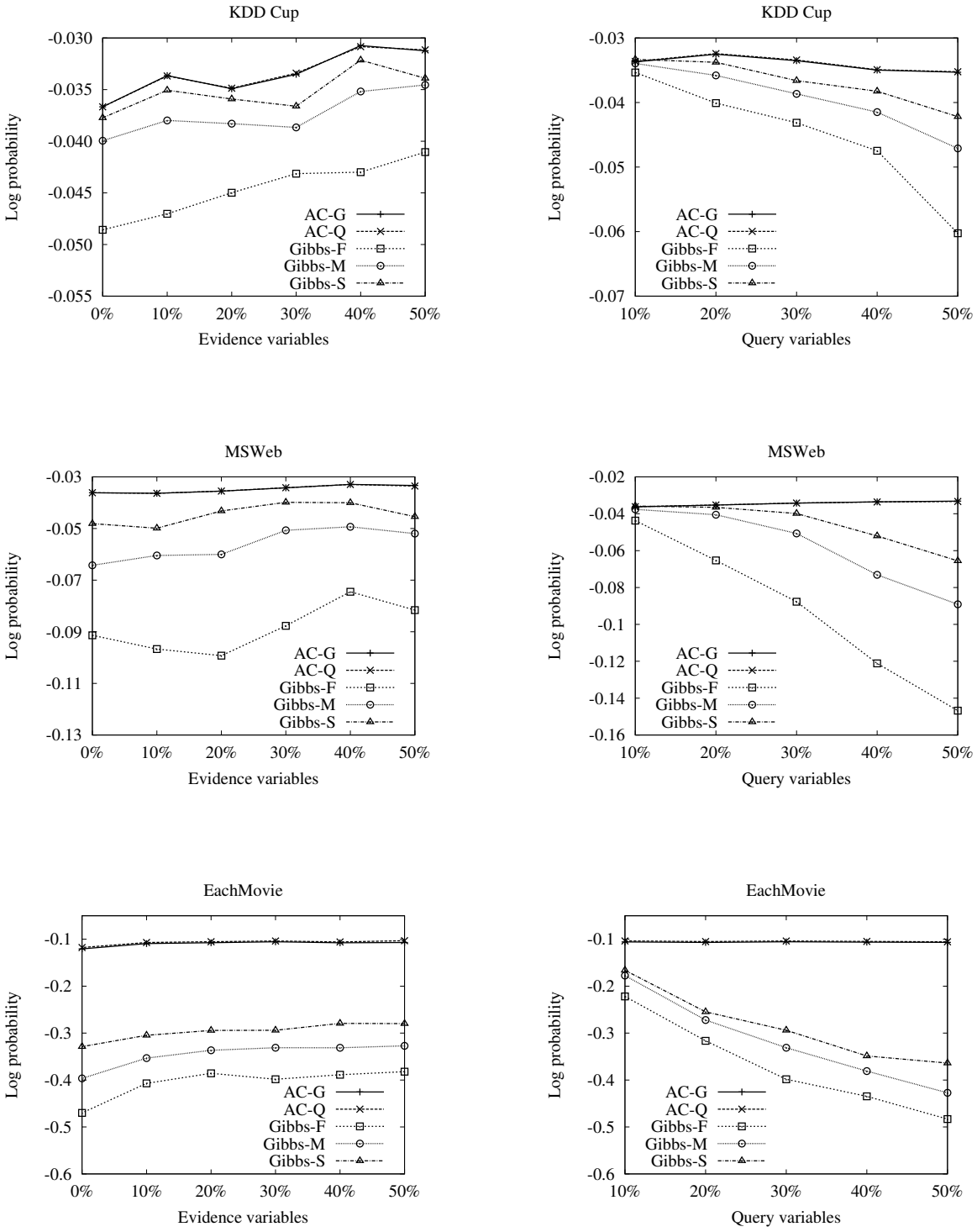

Figure 1: Conditional log probability per query variable, per query.