Lazy Arithmetic Circuits

Seyed Mehran Kazemi and **David Poole**
The University of British Columbia
Vancouver, BC, V6T 1Z4
{smkazemi, poole}@cs.ubc.ca

Abstract

Compiling a Bayesian network into a secondary structure, such as a junction tree or arithmetic circuit allows for offline computations before observations arrive, and quick inference for the marginal of all variables. However, query-based algorithms, such as variable elimination and recursive conditioning, that compute the posterior marginal of few variables given some observations, allow pruning of irrelevant variables, which can reduce the size of the problem. Madsen and Jensen show how lazy evaluation of junction trees can allow both compilation and pruning. In this paper, we adapt the lazy evaluation to arithmetic circuits, allowing the best of both worlds: pruning due to observations and query variables as well as compilation while exploiting local structure and determinism.

Bayesian networks (Pearl 1988; Koller and Friedman 2009) are compact representations for probability distributions over sets of random variables. We are often interested in performing probabilistic inference over these networks: given some evidence, finding the marginal posterior probabilities for one or more query variables.

Variable elimination (Zhang and Poole 1994; Dechter 1996) and recursive conditioning (Darwiche 2001b) are two query-based inference algorithms for Bayesian networks. They allow for pruning the Bayesian network given a query variable and evidence. In order to answer queries on multiple random variables given an evidence, they can be run multiple times and gain by sharing a cache between the runs.

Clique tree algorithms (Lauritzen and Spiegelhalter 1988; Shafer and Shenoy 1990; Jensen, Lauritzen, and Olesen 1990) compile a Bayesian network into a secondary data structure called junction tree (JTree) and find the marginal probabilities for all random variables with two passes through this JTree in time linear in the size of the JTree. Another advantage of these algorithms is that they can shift some of the operations into the compilation which is done offline and only once.

There are cases where we need the probabilities for a subset (not all) of the variables. For instance in a medical setting, after observing the patient's symptoms, a doctor might be suspicious about only a few (but not all) of the diseases. In such cases, one can run a clique tree algorithm and only use the probabilities for the query variables. Traditionally, these algorithms could not prune the JTree with respect to the query variables and the evidence. Madsen and Jensen (1999) made pruning possible for JTrees by lazy evaluation of the potentials and lazy propagation of the messages. Lazy evaluation/propagation introduces some overhead, but they showed empirically that especially when there is evidence on many (but not all) random variables, their algorithm outperforms standard inference algorithms for JTrees.

It has been proved that JTrees are subsumed by a more general data structure called arithmetic circuits (Darwiche 2003; Park and Darwiche 2004). Inference on a Bayesian network can be performed by compiling the network into an arithmetic circuit and finding the probabilities for all (non-evidence) random variables given some evidence by evaluating and differentiating the circuit. Arithmetic circuits can exploit the determinism and the local structure in conditional probability tables (CPTs) of a Bayesian network and improve on clique tree algorithms when such properties appear. It has been shown that an algorithm that exploits these properties can gain exponential improvements over standard clique tree algorithms that do not exploit these properties (Darwiche 2002). Inference algorithms for arithmetic circuits do not prune the circuit based on the query variables and the evidence.

In this paper, we use the the lazy evaluation/propagation idea in (Madsen and Jensen 1999) to prune a class of arithmetic circuits that we call *lazy arithmetic circuits (LAC)* before running an inference algorithm. The size of the pruned LAC can be exponentially smaller than the size of the original LAC. Pruning can be done in time linear in the size of the *pruned* LAC and the number of irrelevant random variables. Thus, pruning can be done in time *sub-linear* in the size of the original LAC. Similar to arithmetic circuits, LACs can exploit the determinism and the local structure (in the form of context-specific independence (Boutilier et al. 1996) and beyond) to improve inference.

Following (Madsen and Jensen 1999), we focus on the inference task where the probabilities of some random variables are desired for a fixed evidence. We leave the case where queries involve different evidences as future work.

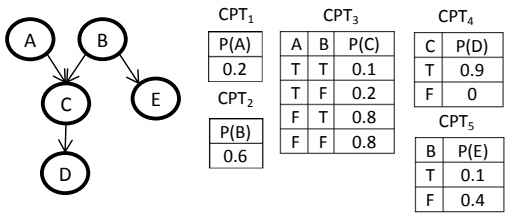

Figure 1: A Bayesian network.

Background and Notations

We use upper-case letters to represent random variables. An upper-case letter in bold refers to a set of variables. To simplify the descriptions, we assume variables are binary but all our ideas can be used for non-binary variables. We represent $V = true$ by v and $V = false$ by $\neg v$. A lower-case letter in bold represents a set of value assignments.

Bayesian Networks

Let $\{V_1, V_2, \ldots, V_n\}$ be n random variables. A Bayesian network (BN) is an acyclic directed graph (DAG) whose nodes are random variables and whose edges represent conditional dependencies among variables. Each variable is independent of all its non-descendants in the DAG given a value for its parents.

Let $pa(V_i)$ represent the parent nodes of random variable V_i in the DAG. The joint probability distribution over the random variables in the (BN) can be defined as:

$$P(V_1, V_2, \ldots, V_n) = \prod_{i=1}^{n} P(V_i | pa(V_i)) \quad (1)$$

Figure 1 represents an example of a BN with five random variables and the corresponding CPTs. Inference in a BN refers to answering queries on posterior probabilities of one or more target variables given some evidence. For instance in Figure 1, we might be interested in $P(C|b)$. When answering a query, a variable is called *barren* (Shachter 1986) if it is neither an evidence nor a target variable, and it only has barren descendants. These variables add no information to the given query and can be ignored. For example for the query $P(C|b)$, D and E are barren variables. Furthermore, there might be non-barren variables in the network that are independent of the query variables given the observation. These variables are *d-separated* from the query variables given the observation and can be ignored. We can find these variables efficiently in linear (in the number of network variables) time (see (Geiger, Verma, and Pearl 1990)). In order to answer a query, we refer to the barren variables and the variables d-separated from the query variables given evidence as *irrelevant* variables.

Arithmetic Circuits

Let PD be a probability distribution over random variables $\{V_1, V_2, \ldots, V_n\}$ induced by a BN. PD can be equivalently represented by a multilinear function. This function is called the *network polynomial* (Darwiche 2003) and is defined as follows:

$$f = \sum_{\mathbf{V}=\mathbf{v}} \prod_{i=1}^{n} I(V_i = v_i) P(V_i = v_i | pa(V_i) = \pi(V_i)) \quad (2)$$

where $\sum_{\mathbf{V}=\mathbf{v}}$ sums over all possible instantiations of the variables, $I(.)$ is the indicator function, $\pi(V_i)$ is the assignment in $\mathbf{v}$ projected on the parents of V_i, and $P(V_i|pa(V_i))$ are the network parameters. For simplicity, we use v and $\neg v$ in our figures instead of $I(v)$ and $I(\neg v)$.

Having the network polynomial, we can compute the probabilities for all variables given some evidence by evaluating and differentiating the polynomial function (see (Darwiche 2003) for more details). The size of this polynomial is, however, exponential in the number of variables.

Arithmetic circuits (ACs) are compact representations for network polynomials and can have a size exponentially smaller than the size of the network polynomial (exponential in treewidth). They represent the network polynomial as a rooted DAG whose internal nodes are additions and multiplications, and whose leaves are numeric constants or variables. Once the AC is constructed for a network polynomial, all the values and derivatives for some evidence can be computed in time linear in the size of the circuit. From these values and derivatives, the probabilities of all variables in the network given evidence e (as well as some other probabilities) can be computed. Figure 2 represents parts of an AC for the BN in Figure 1.

We refer to a part of an AC having a structure similar to Figure 2(a) (an OR of two AND nodes one containing e and one containing $\neg e$) as the *decision-tree (DT)* of E. We also let E_{True} refer to the AC rooted at the AND node containing the child $I(e)$ and E_{False} refer to the AC rooted at the and node containing the child $I(\neg e)$ in the DT of E.

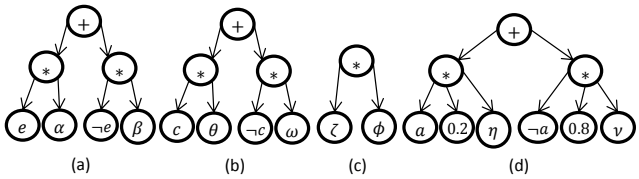

Figure 2: Parts of an AC generated for the BN in Figure 1. (a) represents the top levels of the AC, (b) represents the sub-AC denoted by α in (a), (c) represents the sub-AC denoted by θ in (b), (d) denotes the sub-AC denoted by ζ in (c). ϕ, η and ν represent other sub-ACs generated for B and D.

Lazy Recursive Conditioning Graphs

In this section, we describe how we use the recursive conditioning (RC) algorithm to construct a data structure that we call *lazy recursive conditioning graph (LRCGraph)*, and why this data structure is interesting. First, we define the components used in an LRCGraph.

A **node** in an LRCGraph contains a random variable and two informedEdges. An **informedEdge** for a node N with random variable V is composed of a $v \in \{True, False\}$ for

V, a (possibly empty) set of $\langle CPTIndex, Value \rangle$ pairs, and a pointer to a supernode. Let $RVs(N)$ represent the random variables in the subgraph rooted at node N. A **supernode** is a (possibly empty) set of nodes $\{N_1, N_2, \ldots, N_k\}$ such that $RVs(N_i) \cap RVs(N_j) = \emptyset$ for $i \neq j$.

An LRCGraph is a rooted, directed acyclic graph composed of *supernodes* and *informedEdges*, and can be generated for an input BN by symbolic (instead of numeric) evaluation of the RC algorithm for the input BN. Algorithm 1 gives a high level description of how an LRCGraph is generated for a BN. We use the dot notation to refer to the features and methods of an object. For instance for a supernode *sn*, *sn.nodes* gives its set of nodes. A *context* in our algorithm refers to a set of value assignments to the random variables. We describe Algorithm 1 using an example. A CPT can be evaluated if all its variables are assigned in the context.

Example 1. Consider the BN in Figure 1 and suppose we generate an LRCGraph for this BN using Algorithm 1. Initially, we call the algorithm with the BN and an empty context as input. Similar to the RC algorithm, we choose an order for the random variables and perform case analysis on the variables in that order. We put no constraints on the order of the variables, so any heuristic (e.g., see (Kjaerulff 1985; Dechter 2003; Kazemi and Poole 2014)) can be used to select an order in line 13 of the algorithm.

Suppose initially we decide to do a case analysis on E. We generate a supernode having one node for E. The node contains two informedEdges: one corresponding to $E = True$ and one corresponding to $E = False$. Since none of the CPTs of the BN can be evaluated after assigning a value to E, the informedEdges have no $\langle CPTIndex, Value \rangle$ pairs. For each informedEdge, we add the value assigned to the E to the context and call the algorithm for the BN and the new context. These operations correspond to the lines 14-22 of the algorithm. The root supernode of the LRCGraph in Figure 3(a) corresponds to the case analysis on E. We follow the algorithm for the call having $E = True$ in its context; the other call can be done similarly.

After assigning E to $True$, suppose we decide to do a case analysis on C. This case analysis can be done similarly as the case analysis on E. Let's follow the call were both E and C are assigned to $True$. In this call, the algorithm realizes that the BN is disconnected given the assignments in the context: one component has A and B and the other one has D. Therefore, the algorithm is called for each component separately, two LRCGraphs are generated, a new supernode is created having the nodes in the root supernode of each LRCGraph, and the supernode is returned as the output. These operations correspond to lines 7-12 of the algorithm. The supernode containing two nodes (one for A and one for D) in Figure 3(a) corresponds to the supernode created for this call.

Let's follow the call for the component containing A and B. Suppose we have done a case analysis for A similar to the previous random variables and we are in a branch were $A = False$ in the context. Then we do a case analysis on B. For the branch were $B = False$, CPT_2, CPT_3 and CPT_5 evaluate to 0.4, 0.8 and 0.4 respectively. We keep the index of these CPTs as well as the value they evaluate to in the

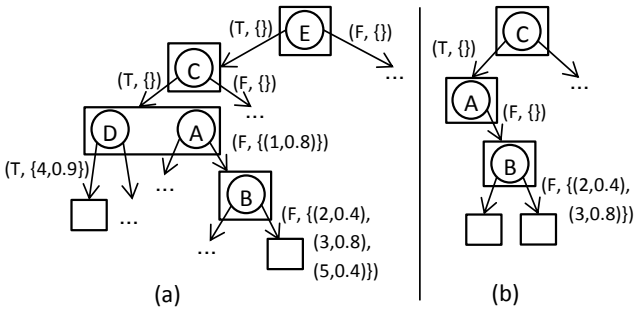

Figure 3: (a) Parts of an LRCGraph generated for the BN in Fig. 1. Circles represent nodes and rectangles represent supernodes. (b) The LRCGraph of part (a) pruned for the query $P(C|b)$.

informedEdges of the node for B. Since we keep all these values (not a product of them as is done in RC), we call our data structure a *Lazy* RCGraph. After each case analysis, we keep our computations in a cache (line 23) so we can potentially re-use them in future (lines 5 and 6). Our context forgets about the variables that are no longer needed (lines 3 and 4) to improve caching.

Following all these calls gives the LRCGraph of Figure 3(a). Using this LRCGraph, we can answer many inference queries on the BN of Figure 1.

Determinism and Context-Specific Independence

LRCGraphs can exploit the determinism and context-specific independence (CSI) in the CPTs.

Example 2. In the BN in Figure 1, $P(d|\neg c) = 0$. Therefore, when we branch on D in the LRCGraph in Figure 3(a) under the branch where $C = False$, we only generate the $D = False$ branch. This could reduce the size of the LRCGraph substantially if there were many variables beneath D in the BN.

Also note that in Figure 1, when $A = False$, the probability of C being $True$ is 0.8 regardless of the value of B (i.e. C is contextually independent of B when $A = True$). In order to exploit this type of independence among variables, we can evaluate a CPT as soon as we have all required information (not when all variables are assigned in the context) and remove the CPT. Exploiting CSI has been shown to have a high impact on the efficiency of the inference (Poole and Zhang 2003; Chavira and Darwiche 2005).

The amount of determinism and CSI exploited by LRCGraphs depends on the branching order. For the BN of Figure 1, for example, if we had generated an LRCGraph by first branching on B, then A, and then C, we could not exploit C being contextually independent of B given $A = True$.

Pruning LRCGraphs

In order to answer queries on an LRCGraph, we can prune the graph based on the observation and query variables before starting the inference algorithms.

Algorithm 1 LRCGraph(Bayesian Net. *BN*, Context Con)

Input: A Bayesian network and a context.
Output: The root of an LRCGraph.

if BN has no CPTs **then**
  return an empty supernode
if $\exists\{X = val\} \in Con$ s.t. $X \notin BN$ **then**
  return LRCGraph(BN, Con $\setminus \{X = val\}$)
if $\langle\langle BN, Con\rangle, supernode\rangle \in Cache$ **then**
  return *supernode*
if $|CC = connected\_components(BN, Con)| > 1$ **then**
  Create a new supernode *sn*
  for each $cc \in CC$ **do**
    $cc\_root = LRCGraph(cc, Con)$
    $sn.add(cc\_root.nodes)$
  return *sn*
Select variable V for branching on
Create node n with variable V
for each $v \in \{True, False\}$ **do**
  Create a new InformedEdge *ie* with $ie.value = v$
  $to\_eval = CPTs.can\_be\_evaluated(Con \cup \{V = v\})$
  for each cpt $\in to\_eval$ **do**
    $ie.addPair(cpt.index, eval(cpt, Con \cup \{V = v\}))$
  $ie.child = LRCGraph(BN \setminus to\_eval, Con \cup \{V = v\})$
  $n.add(ie)$
Create supernode *sn* with one node n
$Cache.add(\langle\langle BN, Con\rangle, sn\rangle)$
return *sn*

Example 3. Consider the LRCGraph in Figure 3(a) and the query $P(C|\neg a)$. D and E are barren variables, so we remove the $\langle CPTIndex, Value\rangle$ pairs that belong to the CPT of D or E (CPT_4 and CPT_5) from the informedEdges. Once we remove these pairs, $P(C|\neg a)$ does not change if we condition on any value for D or E. So we remove these variables from the LRCGraph by replacing them with either one of the subgraphs beneath them. We can also ignore the branch that sets A to $True$ as we have observed it to be $False$. Note that after observing A, the edge going from A to C can be removed from the network, which means A is also d-separated from C after we observe it. Therefore, we can remove the $\langle CPTIndex, Value\rangle$ pairs that belong to the CPT of A (CPT_1). The resulting LRCGraph is demonstrated in Figure 3(b). Now we can run our inference algorithm on the pruned LRCGraph. Note that pruning is possible on LRCGraphs because of the lazy evaluation of the CPTs. In order to answer the same query, if we had multiplied the values of the three CPTs when branching on $B = False$ (or $B = True$), we could not replace E by one of its children because the values of the branches beneath B would depend on the value of E.

An LRCGraph can be pruned by following these steps:

- Given the query variables and the evidence, find all the irrelevant variables in the original BN.
- Replace any node in the LRCGraph having an irrelevant variable with either one of its children.
- Remove all the $\langle CPTIndex, Value\rangle$ pairs from informedEdges where *CPTIndex* belongs to an irrelevant variable.
- For the evidence variables, only keep the informedEdge whose value is consistent with the evidence.
- During pruning, whenever the root supernode contains more than one node, remove the nodes (and their subgraphs) from the root having only non-query variables in their subgraph.

From LRCGraphs to ACs

LRCGraphs can be compiled into ACs and the same pruning can be run on the ACs. Let *sn* be a supernode in the LRCGraph with nodes $\{n_1, n_2, \ldots, n_k\}$. In order to compile this supernode into an AC, we generate a *product node* with k children, where the i-th child is generated from compiling n_i into AC. In order to compile a node n with variable V, we generate a DT for n: a *summation node* having two product nodes as children, one for V being $True$ and one for V being $False$. The product node for the $True$ branch has these children: $I(v)$[1], the values in $\langle CPTIndex, Value\rangle$ pairs with CPTIndex stored in them as extra information, and the AC generated from compiling the child supernode that the informedEdge for V being $True$ is pointing to. The children of the product node for the $False$ branch are similar. Once the LRCGraph has been compiled into an AC, all the inference algorithms for ACs can be used for them.

Example 4. Let's compile the LRCGraph in Figure 3(a) into an AC. First, we look at the root supernode and generate the DT in Figure 2(a). α and β in this figure correspond to the ACs generated for the two branches going out of the node in the root supernode. Following the branch where $E = True$, we reach to a supernode containing a node for C. For this supernode, we generate the AC in Figure 2(b), where θ and ω represent the ACs generated for each of the two branches going out of the node for C. Following the branch where $C = True$, we will reach to a supernode with two nodes inside it. For this supernode, we generate a product node having two children representing the ACs for the two nodes inside this supernode. This has been represented in Figure 2(c) where ζ and ϕ correspond to the ACs generated for the two nodes in this supernode respectively. We follow the same procedure for all the branches and generate an AC from the LRCGraph. Note that we can simplify this AC by replacing θ with two children: ζ and ϕ.

ACs generated from compiling an LRCGraph have a special structure. They consist of alternating rows of *products* and *summations*, where each summation node is a DT for a random variable. Due to the caching procedure when generating the LRCGraphs, the product nodes in the generated AC only have one summation parent. This structure of the ACs corresponds to the decision-DNNFs of Huang and Darwiche (2007) with network parameters added to the AC.

Note that the product nodes of ACs generated from compiling LRCGraphs might have more than one children having a numeric constant value. Having these values in separate nodes enables pruning. Since we postpone multiplying

[1] If $I(v)$ has been already generated and is in the circuit, we just use it and avoid creating a new node.

these numbers to the inference (instead of compile) time, we refer to these ACs as *lazy* ACs.

Definition 1. An arithmetic circuit that has been generated for a BN by compiling an LRCGraph is called a *lazy arithmetic circuit (LAC)*.

When we compile an LRCGraph into an AC, we can also take advantage of equal parameters in a CPT that are not due to CSI. When two parameters in a CPT are equal, we have two informedEdges in our LRCGraph containing the same pair $\langle CPTIndex, Value \rangle$. At compile time, we generate only one node in the AC for these two pairs. Algorithm 1 can be modified slightly to compile a BN directly into an LAC.

Pruning LACs

LACs can be pruned similar to LRCGraphs as follows:

- Given the query variables and the evidence, find all the irrelevant variables in the original BN.
- For any irrelevant variable A, replace the DT of A with either A_{True} or A_{False}.
- Remove all LAC nodes having a CPTIndex (as extra information) belonging to the CPT of an irrelevant variable.
- For any evidence $C = Val$, replace DT of C with C_{Val} and remove the nodes in the LAC containing c or $\neg c$.
- Replace any product or summation node having only one child with its child node.
- During pruning, whenever the root of the LAC is a product node, remove the child nodes (and their sub-ACs) only having non-query variables in their sub-AC.

Example 5. Consider the BN in Figure 4 and suppose we have generated the LAC on the right of the figure for this network (for brevity, we have ignored the nodes containing numeric constants). In order to answer the query $P(E|c)$, we can see that F is a barren variable, so we can only keep either F_{True} or F_{False}. Let's assume we keep F_{True}. The root node is a product node whose left sub-AC contains only non-query variables, so we prune it and keep the AC rooted at the node denoted by 1 in Figure 4. Since we have observed $C = True$, we only keep the C_{True} and remove the node for c. Since C_{True} only has one child, we can replace it by its child an get the AC rooted at the node denoted by 2. B is d-separated from the query given the observation, thus we can prune it similarly. Then we have two separated LACs, where the first one only contains non-query variables. Therefore, we can only keep the second one (the triangle containing DE). We can now perform inference on a small LAC containing only D and E which is more efficient compared to the standard algorithms that do not prune the AC and calculate the probabilities for all variables.

Analyses and Comparisons

We compare inference algorithms based on standard ACs, LACs, and (lazy) JTrees considering the amount of time required for compilation, time complexity of online inference, the efficiency of inference algorithms, exploiting determinism and local structure (or succinctness), and the ability to be pruned based on the query and observation.

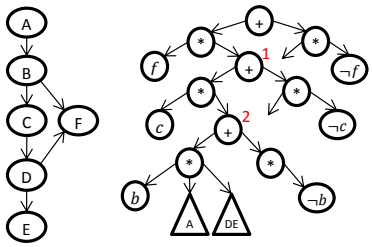

Figure 4: A BN used as motivating example in (Madsen and Jensen 1999) and parts of an LAC generated for that.

Compiling Time

In order to exploit as much as the determinism and local structure in CPTs as possible and generate succinct ACs, algorithms for compiling BNs into ACs (e.g., (Chavira and Darwiche 2005)) typically encode the network as a *conjunctive normal form (CNF)*, simplify the CNF based on the determinism and local structure, compile the CNF into a structure called d-DNNF (Darwiche 2001a), take the minimum cardinality of the d-DNNF, and substitute *OR*s and *AND*s in the d-DNNF with summations and products respectively. Compiling a BN into LACs or JTrees does not require some of these steps (such as CNF encoding/conversion or taking the minimum cardinality) and involves less overhead. So the compile time for LACs and JTrees is potentially less than the aforementioned algorithms for ACs.

Complexity of Online Inference

The inference algorithms for lazy JTrees and LACs spend some time on pruning the structure. For lazy JTrees, pruning takes a linear time in the number of the variables. For LACs, pruning take a time linear in the size of the *pruned LAC* and irrelevant variables (*sub-linear* in the size of the original LAC). For standard ACs, there is no pruning step. Once the pruning is done for lazy JTrees and LACs, however, these two algorithms do inference on a data structure which is potentially exponentially smaller than that of standard ACs.

Note that the inference algorithm for LACs involves some overhead compared to standard ACs, because the multiplication of some numeric constants is postponed to the inference (instead of compile) time. But this time is negligible if each product node of the LAC has only a few children with numeric constants. Lazy JTrees suffer more from this overhead since they postpone the multiplication of the CPTs (or potentials) to the inference time.

Inference Algorithm Efficiency

A JTree can be compiled into an AC with an specific structure: the AC alternates between summation and product nodes and each product node has a single parent. This is quite similar to the structure of the LACs. It has been shown in (Park and Darwiche 2004) that inference on an AC with such a structure is more efficient than inference on standard ACs. Thus, inference can be done more efficiently on JTrees and LACs than on standard ACs.

Determinism and Local Structure

The standard inference algorithms for JTrees cannot exploit determinism and local structure. For LACs, we can exploit determinism, context-specific independence (CSI), and also take advantage of the equal parameters in the same CPT that are not due to the CSI. Therefore, LACs can be more succinct that JTrees. However, the amount of local structure exploited by LACs depends on the branching order. Standard ACs can also exploit determinism and take advantage of the equal values in the same CPTs. Compilers for standard ACs typically use logical reasoning techniques, allowing them to (potentially) exploit more local structure than our compiler for LACs. Furthermore, LACs impose restrictions on the structure of the final AC. Therefore, standard ACs can be more succinct than LACs.

Pruning Based on Query and Observation

As discussed earlier, inference algorithms on standard ACs find the probabilities for all random variables given evidence. Therefore, they do not prune the AC if we only need to query a subset of the variables. LACs have been developed with the motivation of being pruned based on the query variables. All barren variables and those d-separated from the query variables given the observation can be pruned from the LAC. For JTrees, one can use the lazy propagation technique to be able to prune the JTree based on the query.

Conclusion

We introduced *lazy recursive conditioning graphs (LRC-Graphs)* as a target data structure for compiling BNs into and performing inference on them. We showed how LRC-Graphs can be pruned given the query and the observation to speed-up inference. We also identified a class of arithmetic circuits (ACs) called *lazy arithmetic circuits (LACs)* that can be obtained by compiling LRCGraphs into ACs. LACs are capable of exploiting the determinism and the local structure inherent in the conditional probabilities, and they can be pruned based on the observation and query in time sublinear in the size of the original LAC. In the future, we will test LACs empirically on real-data and compare their runtime and efficiency with inference algorithms for standard ACs as well as (lazy) junction trees.

References

Boutilier, C.; Friedman, N.; Goldszmidt, M.; and Koller, D. 1996. Context-specific independence in Bayesian networks. In *Proceedings of UAI*, 115–123.

Chavira, M., and Darwiche, A. 2005. Compiling Bayesian networks with local structure. In *IJCAI*.

Darwiche, A. 2001a. On the tractability of counting theory models and its application to belief revision and truth maintenance. *Journal of Applied Non-Classical Logics* 11(1-2):11–34.

Darwiche, A. 2001b. Recursive conditioning. *Artificial Intelligence* 126(1-2):5–41.

Darwiche, A. 2002. A logical approach to factoring belief networks. In *Proceedings of International Conference on Knowledge Representation and Reasoning*, 409–420.

Darwiche, A. 2003. A differential approach to inference in Bayesian networks. *Journal of the ACM (JACM)* 50(3):280–305.

Dechter, R. 1996. Bucket elimination: a unifying framework for probabilistic inference. In *Proceedings of the twelfth international conference on uncertainty in artificial intelligence (UAI)*, 211–219. Morgan Kaufmann.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann, San Fransisco, CA.

Geiger, D.; Verma, T. S.; and Pearl, J. 1990. d-separation: From theorems to algorithms. In *Uncertainty in Artificial Intelligence*, 139–148.

Huang, J., and Darwiche, A. 2007. The language of search. *Journal of Artificial Intelligence Research (JAIR)* 29:191–219.

Jensen, F. V.; Lauritzen, S. L.; and Olesen, K. G. 1990. Bayesian updating in causal probabilistic networks by local computations. *Computational statistics quarterly 4* 269–282.

Kazemi, S. M., and Poole, D. 2014. Elimination ordering in first-order probabilistic inference. In *Proc. of Association for the Advancements of Artificial Intelligence (AAAI)*.

Kjaerulff, U. 1985. Triangulation of graphs–algorithms giving small total state space. Technical report, Deptartment of Mathematics and Computer Science, Aalborg University, Denmark.

Koller, D., and Friedman, N. 2009. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, Cambridge, MA.

Lauritzen, S. L., and Spiegelhalter, D. J. 1988. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society. Series B (Methodological)* 157–224.

Madsen, A. L., and Jensen, F. V. 1999. Lazy propagation: a junction tree inference algorithm based on lazy evaluation. *Artificial Intelligence* 113(1):203–245.

Park, J., and Darwiche, A. 2004. A differential semantics for jointree algorithms. *Artificial Intelligence* 156(2):197–216.

Pearl, J. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaumann.

Poole, D., and Zhang, N. L. 2003. Exploiting contextual independence in probabilistic inference. *Journal of Artificial Intelligence Research* 18:263–313.

Shachter, R. D. 1986. Evaluating influence diagrams. *Operations research* 34(6):871–882.

Shafer, G. R., and Shenoy, P. P. 1990. Probability propagation. *Annals of Mathematics and Artificial Intelligence* 2(1-4):327–351.

Zhang, N. L., and Poole, D. 1994. A simple approach to Bayesian network computations. In *Proceedings of the 10th Canadian Conference on AI*, 171–178.