Structured Bayesian Networks: From Inference to Learning with Routes

Yujia Shen, Anchal Goyanka, Adnan Darwiche, Arthur Choi
Computer Science Department
University of California, Los Angeles
{yujias,anchal,darwiche,aychoi}@cs.ucla.edu

Abstract

Structured Bayesian networks (SBNs) are a recently proposed class of probabilistic graphical models which integrate background knowledge in two forms: conditional independence constraints and Boolean domain constraints. In this paper, we propose the first exact inference algorithm for SBNs, based on compiling a given SBN to a Probabilistic Sentential Decision Diagram (PSDD). We further identify a tractable subclass of SBNs, which have PSDDs of polynomial size. These SBNs yield a tractable model of route distributions, whose structure can be learned from GPS data, using a simple algorithm that we propose. Empirically, we demonstrate the utility of our inference algorithm, showing that it can be an order-of-magnitude more efficient than more traditional approaches to exact inference. We demonstrate the utility of our learning algorithm, showing that it can learn more accurate models and classifiers from GPS data.

1 Introduction

Structured Bayesian Networks (SBNs) were recently proposed by (Shen, Choi, and Darwiche 2018) for representing and learning distributions over highly complex spaces, such as routes on a map. Like a Bayesian network (BN), an SBN is defined by a directed acyclic graph (DAG) and a set of conditional distributions. In contrast to a BN, each node of an SBN represents a *cluster* of variables (not just a single variable). This *cluster DAG* specifies conditional independencies between *sets* of variables, but makes no claims about the independencies between variables in the same cluster. Also in contrast to a BN, which uses tabular CPTs to represent conditional distributions, an SBN uses a conditional Probabilistic Sentential Decision Diagram, or *conditional PSDD*, to compactly represent the distribution over a cluster, conditioned on its parent clusters (Shen, Choi, and Darwiche 2018). SBNs can hence accommodate background knowledge in the form of conditional independence constraints (via the cluster DAG), as well as knowledge in the form of Boolean domain constraints (via the conditional PSDD).

Initially, (Shen, Choi, and Darwiche 2018) proposed an efficient, closed-form algorithm for estimating the parameters of an SBN from complete data. However, they left open the problem of performing inference in an SBN, once it has been learned from data. In this paper, we propose the first exact inference algorithm for SBNs, which is based on compiling them to a Probabilistic Sentential Decision Diagram (PSDD) (Kisa et al. 2014). Our algorithm is based on the compilation algorithm of (Shen, Choi, and Darwiche 2016), for compiling Bayesian networks into PSDDs.

In general, compiling an SBN to a PSDD is not always tractable. However, we identify a new sub-class of SBNs that is guaranteed to have PSDDs of *polynomial* size. These SBNs have applications in modeling distributions over routes (Zheng 2015; Choi, Shen, and Darwiche 2017). Next, we propose a new structure learning algorithm for this tractable class of SBNs. On the inference side, we show our algorithm can be an order-of-magnitude more efficient than more traditional approaches to exact inference. On the learning side, we show that we can learn more effective models for the tasks of route prediction and route classification.

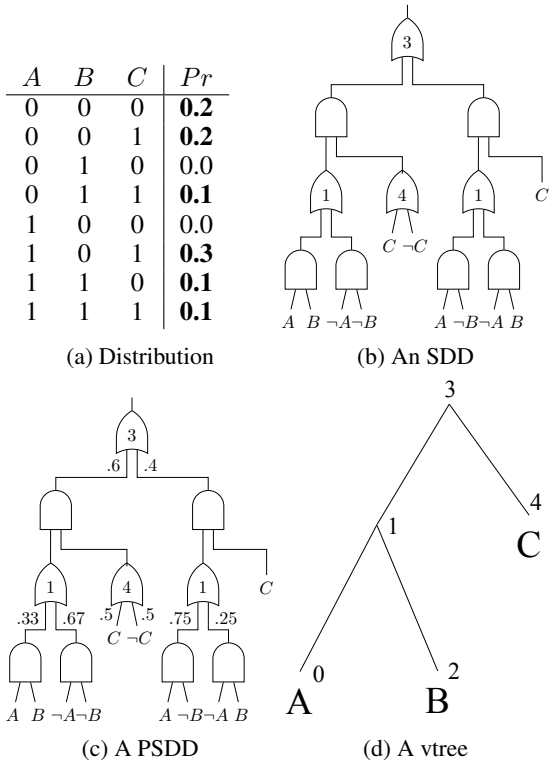
This paper is structured as follows. In Section 2, we review SBNs and PSDDs. Next, in Section 3, we propose our algorithm for compiling SBNs to PSDDs. We identify a tractable sub-class of SBNs in Section 4, whose PSDDs have polynomial size. We propose our learning algorithm in Section 5. Finally, we provide an empirical analysis in Section 6 and conclude in Section 7.

2 Structured Bayesian Networks

The Structured Bayesian network (SBN) is a recently proposed class of probabilistic graphical model which is capable of integrating background knowledge in the form of both conditional independence constraints and Boolean domain constraints (Shen, Choi, and Darwiche 2018).

To specify an SBN, one first defines a *cluster DAG*. As a Bayesian network's structure is defined by a DAG, a Structured Bayesian network's structure is defined by its cluster DAG, where each node represents a *set* of random variables (a cluster). The cluster DAG represents conditional independencies between these sets of variables: a cluster is independent of its non-descendant clusters given its parent clusters. Importantly, nothing is said in terms of independence between variables of the same cluster, which is less committing than the DAG of a Bayesian network. In a cluster DAG, we refer to a cluster $\mathbf{X}$ and its parents $\mathbf{P}$ as a *family* $\mathbf{X} \mid \mathbf{P}$.

To specify an SBN, one next annotates the cluster DAG with conditional distributions, typically using a *conditional*



(a) Distribution (b) An SDD

(c) A PSDD (d) A vtree

Figure 1: A probability distribution and its SDD/PSDD representation. The numbers annotating or-gates in (b) & (c) correspond to vtree node IDs in (d). Moreover, while the circuit appears to be a tree, the input variables are shared and hence the circuit is not a tree.

PSDD, which defines the conditional distribution of a cluster given its parent clusters. The conditional PSDD is composed of two components: (1) the logical constraints that define the feasible states of a cluster given the states of its parent clusters, and (2) the parameters that are learned from data. We review PSDDs and conditional PSDDs next.

Probabilistic Sentential Decision Diagrams

PSDDs were motivated by the need to represent probability distributions $Pr(\mathbf{X})$ with many instantiations $\mathbf{x}$ attaining zero probability, $Pr(\mathbf{x}) = 0$ (Kisa et al. 2014). Consider the distribution $Pr(\mathbf{X})$ in Figure 1a for an example. The first step in constructing a PSDD for this distribution is to construct a special Boolean circuit that captures its zero entries; see Figure 1b, which was compiled automatically from the logical constraints (the zeros). The Boolean circuit captures zero entries in the following sense. For each instantiation $\mathbf{x}$, the circuit evaluates to 0 at instantiation $\mathbf{x}$ iff $Pr(\mathbf{x}) = 0$. The second and final step of constructing a PSDD amounts to parameterizing this Boolean circuit (e.g., by learning from data), which adds a local distribution on the inputs of each or-gate; see Figure 1c.

This annotated circuit induces a distribution $Pr(\mathbf{X})$ as follows. First, the probability of a complete instantiation $\mathbf{x}$ is obtained by performing a bottom-up pass, evaluating each gate from the inputs to the output. An input evaluates to 1 or 0 depending on its value set by $\mathbf{x}$. The value of an and-gate is the product of the values of its inputs. The value of an or-gate is the weighted sum of its inputs (using the weights annotating the inputs). Finally, $\sum_{\mathbf{x}} Pr(\mathbf{x}) = 1$. For more on the semantics of PSDDs, see (Kisa et al. 2014).

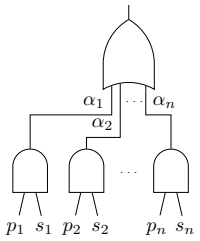


Figure 2: SDD fragment

The Boolean circuit underlying a PSDD is known as a Sentential Decision Diagram (SDD) (Darwiche 2011). An SDD circuit is constructed from the fragment in Figure 2, where or-gates can have any number of inputs, and and-gates have two inputs each. Each p_i is called a **prime** and each s_i is called a **sub.** Each SDD circuit conforms to a tree of variables called a *vtree*, which is a binary tree whose leaves are the circuit variables; see Figure 1d. The conformity is roughly as follows. For each SDD fragment with primes p_i and subs s_i, there exists a vtree node v where the variables of SDD p_i are those of the left child of v and the variables of SDD s_i are those of the right child of v. For the SDD in Figure 1b, each or-gate has been labeled with the ID of the vtree node it conforms to. For example, the top fragment conforms to the vtree root (ID=3), with its primes having variables $\{A, B\}$ and its subs having variables $\{C\}$. Finally, when the circuit is evaluated under *any* input, precisely one prime p_i of each fragment will be 1. Hence, the fragment output will simply be the value of the corresponding sub s_i.[1] A PSDD can now be obtained by annotating a distribution $\alpha_1, \ldots, \alpha_n$ on the inputs of each or-gate, where $\sum_i \alpha_i = 1$.

A *conditional PSDD* represents a *set* of distributions over variables $\mathbf{X}$, which are conditioned on the *same* set of variables $\mathbf{P}$. A conditional PSDD is motivated by the need to represent the conditional distributions $Pr(\mathbf{X} \mid \mathbf{P})$ of a cluster, in a cluster DAG, where $\mathbf{X}$ are the variables of the cluster and $\mathbf{P}$ are the variables of its parent clusters (Shen, Choi, and Darwiche 2018). A conditional PSDD can be viewed as having two components: a PSDD component that represents each conditional distribution $Pr(\mathbf{X} \mid \mathbf{p})$, and an SDD component that serves to index these distributions based on the parent instantiation $\mathbf{p}$. A conditional PSDD can also aggregate common conditional distributions, particularly when there is significant *context-specific independence* (Boutilier et al. 1996). For example, consider the conditional PSDD in Figure 3, where α and β denote two PSDDs highlighted in red, with the SDD component highlighted in blue. The PSDD for α is shared among all parent instantiations where

[1] This means that an or-gate will have at most one 1-input. Note that an SDD circuit may yield a 1-output for all possible inputs. Such circuits arise when representing strictly positive distributions.

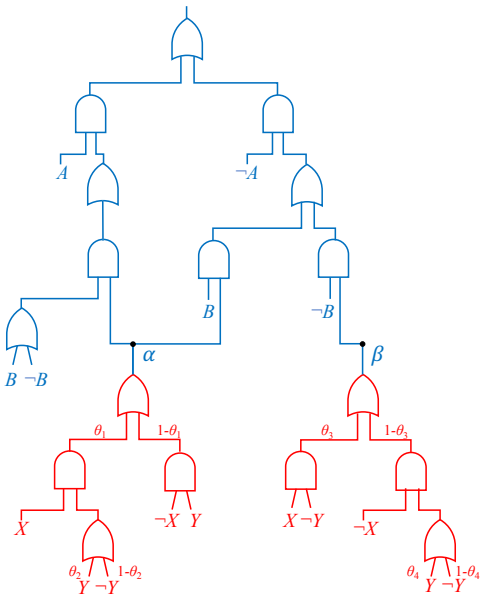


Figure 3: Conditional PSDD

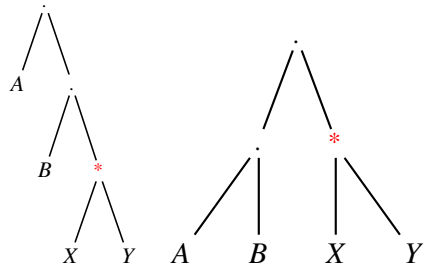


Figure 4: Conditional Vtrees

A or B is true. The PSDD for β is obtained by a single parent instantiation, where A and B are both false. For more on conditional PSDDs, see (Shen, Choi, and Darwiche 2018).

Vtrees

In the compilation algorithm that we propose next, the *conditional vtree* and the *decision vtree* will be central. First, for an internal vtree node v, we refer to v^l and v^r as the left and right children of v. We call an internal vtree node v a *Shannon node* iff its left child is a leaf node. Consider the following definition of a decision vtree, originally proposed by (Oztok, Choi, and Darwiche 2016).

Definition 1 (Decision Vtree) *A family* $\mathbf{X} \mid \mathbf{P}$ *is compatible with an internal vtree node* v *iff the family has some variables mentioned in* v^l *and some variables mentioned in* v^r*. A vtree for an SBN* N *is said to be a decision vtree for* N *iff every family in* N *is compatible with only Shannon nodes.*

Next, we consider conditional vtrees. Any conditional PSDD must conform to a conditional vtree.

Definition 2 (Conditional Vtrees) *Let* v *be a vtree for variables* $\mathbf{X} \cup \mathbf{P}$ *which has a node* u *that contains precisely the variables* $\mathbf{X}$*. If node* u *can be reached from node* v *by only following right children, then* v *is said to be a conditional vtree for* $\mathbf{X} \mid \mathbf{P}$ *and* u *is said to be its* $\mathbf{X}$*-node.*

Figure 4 depicts two examples of conditional vtrees for $\mathbf{X} = \{X, Y\}$ and $\mathbf{P} = \{A, B\}$. The $\mathbf{X}$-nodes are starred. The vtree under the $\mathbf{X}$-node determines the circuit structure of the probabilistic (PSDD) component of a conditional PSDD. In turn, the vtree outside of the $\mathbf{X}$-node determines the circuit structure of the logical (SDD) component.

Finally, we say that two vtrees, one over variables $\mathbf{X}$ and the other over variables $\mathbf{Y}$, are compatible iff they can be obtained by projecting some other common vtree on variables $\mathbf{X}$ and $\mathbf{Y}$, respectively.

Definition 3 (Vtree Projection) *Let* v *be a vtree over variables* $\mathbf{Z}$*. The projection of* v *on variables* $\mathbf{X} \subseteq \mathbf{Z}$ *is obtained as follows. Successively remove every maximal subtree* v_0 *whose variables are outside* $\mathbf{X}$*, while replacing the parent of* v_0 *with its sibling.*

3 Exact Inference by Compilation to PSDDs

In this section, we propose the first exact inference algorithm for Structured Bayesian networks (SBNs). Our approach is based on the approach proposed by (Shen, Choi, and Darwiche 2016), for compiling Bayesian networks into PSDDs, which we summarize below:

1. pick a decision vtree v for the given Bayesian network N, using min-fill for example;
2. compile each CPT of network N into a PSDD using the vtree v projected onto the CPT's variables;
3. using the PSDD multiply operator, multiply all CPTs.

The result is a single PSDD representing the joint distribution of the given BN; we shall refer to this PSDD as the *joint PSDD*. Once we obtain the joint PSDD, we can perform exact inference efficiently: we can compute marginals or MPEs, for example, in time linear in the size of the PSDD (Kisa et al. 2014). Moreover, one can bound the size of the joint PSDD of a BN by its treewidth, by using an appropriate vtree (Shen, Choi, and Darwiche 2016).

To perform exact inference in SBNs, we also perform three similar steps, with Step (3) being the same for SBNs as it is for BNs: each conditional PSDD can be treated as a PSDD, which we can multiply together. Step (1) is also similar for SBNs, but we will need another method for picking a special type of vtrees for SBNs, which we discuss later. Step (2) is the main difference. In order to multiply two PSDDs together using the PSDD multiply operator, the vtrees of the two PSDDs must be compatible, i.e., they are the projections of the same vtree (Shen, Choi, and Darwiche 2016).[2] This is easy to ensure in a BN, as we simply convert each CPT to a PSDD using the vtree from Step (1). This is not easy to ensure in an SBN, since their conditional distributions must be specified as a conditional PSDD already, whose conditional vtrees may not be compatible.[3] Hence,

[2] Given two PSDDs with compatible vtrees, of sizes s_1 and s_2, the complexity of multiplication is $O(s_1 s_2)$.

[3] In a classical BN, a tabular CPT is learned from data, which is then easy to convert into a PSDD for the purposes of inference.

Algorithm 1 ConstructDecisionCtree(cluster DAG $\mathcal{B}$, topological ordering π)

if $\mathcal{B}$ is a single cluster $\mathbf{X}$ **then return** a leaf ctree for cluster $\mathbf{X}$
else if $\mathcal{B}$ is disconnected **then**
  $\mathcal{B}_1, \mathcal{B}_2 \leftarrow$ a disconnected partition of $\mathcal{B}$
  $\pi_1, \pi_2 \leftarrow$ sub-orders over clusters in $\mathcal{B}_1, \mathcal{B}_2$ of the total ordering π
else
  $\mathbf{X} \leftarrow$ first cluster of ordering π
  $\mathcal{B}_1, \mathcal{B}_2 \leftarrow$ root cluster $\mathbf{X}$, and cluster DAG $\mathcal{B}$ with root cluster $\mathbf{X}$ removed
  $\pi_1, \pi_2 \leftarrow$ ordering $\langle \mathbf{X} \rangle$, and ordering π with the first element $\mathbf{X}$ removed
$v^l \leftarrow$ ConstructDecisionCtree$(\mathcal{B}_1, \pi_1)$.
$v^r \leftarrow$ ConstructDecisionCtree$(\mathcal{B}_2, \pi_2)$.
return a ctree v with with left and right children v^l and v^r

for SBNs, in Step (1), we need to make sure we pick the right vtree that will allow us to, in Step (2), enforce compatibility. We discuss how to do this next.

Finding a Global Vtree

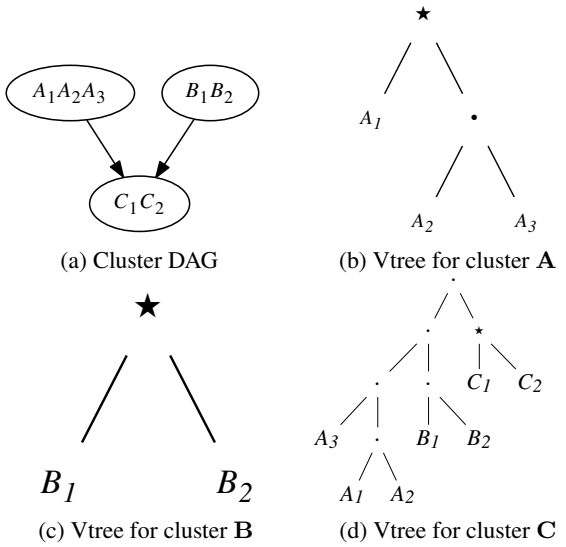
In this section, we consider analogues of vtrees (and their variations) for SBNs, where leaves are labeled by clusters of the SBN, rather than by variables; we refer to these cluster trees as *ctrees*. A ctree can be viewed as a restricted type of vtree: a ctree is a vtree where the variables $\mathbf{X}$ of a cluster appear in the same sub-vtree, but where we represent this sub-vtree with a single leaf ctree node. Note, however, that we shall leave the sub-vtree over variables $\mathbf{X}$ implicit for now, and first show how to pick a ctree.

Our goal is to identify a ctree for a given SBN that will accommodate inference by PSDD multiplication. The first requirement is that the ctrees of all conditional PSDDs must be compatible with some common joint ctree. The second requirement is that these ctrees must also be conditional ctrees. The first requirement can be enforced by insisting that our ctree be a *decision ctree* with respect to our SBN. The second requirement can be enforced, in addition to using a decision ctree, by restricting the decision ctree to respect a topological ordering. In this case, the child cluster is guaranteed to appear as the right-most leaf in the ctree (which guarantees a conditional ctree). Algorithm 1 provides an algorithm for finding such a ctree, given a cluster DAG and a topological ordering of its clusters. The following proposition summarizes the result.

Proposition 1 *The ctree v returned by Algorithm 1 is a decision ctree with respect to the input cluster DAG. Moreover, for each family $\mathbf{X} \mid \mathbf{P}$ in the cluster DAG, the projection of v onto family $\mathbf{X} \mid \mathbf{P}$ is a conditional ctree for the family.*

This ctree and its implied conditional ctrees can be used to perform exact inference in an SBN via compilation. However, we must first show how to choose the sub-vtrees over the variables $\mathbf{X}$ of a cluster, which we do next.

In an SBN, conditional PSDDs represent complex conditional distributions that would otherwise have intractable representations as tables; hence, a conditional PSDD must typically be learned from data directly. In addition, when learning conditional PSDDs, we will want to learn their conditional vtrees independently (to provide the best fit to the data).



(a) Cluster DAG (b) Vtree for cluster $\mathbf{A}$

(c) Vtree for cluster $\mathbf{B}$ (d) Vtree for cluster $\mathbf{C}$

Figure 5: A cluster DAG and three conditional vtrees for clusters $\mathbf{A} = \{A_1, A_2, A_3\}$, $\mathbf{B} = \{B_1, B_2\}$, and $\mathbf{C} = \{C_1, C_2\}$. $\mathbf{X}$-nodes are labeled with a star.

Finding Local Vtrees

Typically, when learning PSDDs from data, one must also learn its vtree (Liang, Bekker, and Van den Broeck 2017; Shen, Choi, and Darwiche 2017). In an SBN, it suffices to learn the conditional PSDDs of its conditional distributions (Shen, Choi, and Darwiche 2018). To provide the best fit, we should learn the corresponding conditional vtrees independently. However, to perform inference, these conditional vtrees must be compatible, as projections of a common global vtree. We show how to achieve this next.

Consider the cluster DAG of Figure 5, and its three conditional vtrees. Here, the vtree for cluster $\mathbf{A}$ is not compatible with the vtree for cluster $\mathbf{C}$, as their projections onto variables $\mathbf{A}$ are different vtrees. More generally, consider a family $\mathbf{X} \mid \mathbf{P}$ and its conditional vtree v. Let u denote the $\mathbf{X}$-node of v. Sub-vtree u dictates the probabilistic (PSDD) component of a conditional PSDD, whereas the sub-vtree outside of u, over $\mathbf{P}$, dictates the logical (SDD) component. That is, the distribution induced by a conditional PSDD depends only on the sub-vtree u over $\mathbf{X}$. The sub-vtree over $\mathbf{P}$ impacts the size of the conditional PSDD but not its distribution. We thus propose to manipulate the conditional vtrees of an SBN so that they become compatible, but for each family $\mathbf{X} \mid \mathbf{P}$ with conditional vtree v and $\mathbf{X}$-node u, we leave the sub-vtree u fixed in v. In this case, the probabilistic (PSDD) components of each conditional PSDD will also be fixed, leaving the conditional distributions invariant.

We propose the following two-step algorithm, that takes as input an SBN whose conditional PSDDs have been learned independently from data, and outputs an SBN with an equivalent joint distribution, that further accommodates

exact inference via compilation. Our first step is to obtain a target decision ctree v, where each conditional vtree will be made compatible with v. We first run Algorithm 1 to obtain a decision ctree c for our SBN. For each family $\mathbf{X} \mid \mathbf{P}$, we then replace the leaf cluster $\mathbf{X}$ in ctree c with the $\mathbf{X}$-node of the family's conditional vtree, yielding our target vtree v.

Our second step is to adjust all conditional vtrees so that it is compatible with v. More specifically, for each family $\mathbf{X} \mid \mathbf{P}$, we adjust the logical (SDD) component of its conditional vtree/PSDD, through a process called *re-normalization*.[4] Tree rotation and swap operators were previously used to re-normalize an SDD to a new vtree (Choi and Darwiche 2013), where an operation on the vtree implied a corresponding re-factorization of the logical circuit of the SDD.[5] After re-normalization, the resulting conditional PSDDs are now all compatible with each other.

4 A Tractable Class of SBNs

We identify a tractable sub-class of SBNs, which can be compiled into joint PSDDs with only polynomial size—namely those that correspond to *binary hierarchical maps*. This class of SBNs are of practical interest, as they are inspired from an application of SBNs for modeling distributions over routes on a map, or equivalently, simple paths on a graph (Zheng 2015; Choi, Shen, and Darwiche 2017).

Consider in Figure 6, a simplified graph of neighborhoods in the Los Angeles Westside, where edges represent streets and nodes represent intersections. The nodes of the LA Westside have been partitioned into four sub-regions: Santa Monica, Westwood, Venice and Culver City. Westwood is further partitioned into two smaller sub-regions: UCLA and Westwood Village. This partitioning is an example of a *hierarchical map*, or more simply, an *hmap*. An hmap allows one to abstract the notion of a route in a map: an abstract route is a route between regions, which can be refined by recursively planning the routes in each region. For example, if we want to go from Venice to Westwood, we may first decide to use edge e_4 to go from Venice to Santa Monica, and then use edge e_1 to go from Santa Monica to Westwood. Next, we find a route in Venice to edge e_4, then a route through Santa Monica from edge e_4 to e_1, and finally a route in Westwood from e_1 to the destination (we can then recursively find routes between UCLA and Westwood Village).

An hmap induces a cluster DAG as follows; see Figure 6. Each node of this cluster DAG represents a region, and each edge of the cluster DAG indicates that the child is a sub-region of the parent. Each node of the cluster DAG is associated with the map edges that are used at that level; for

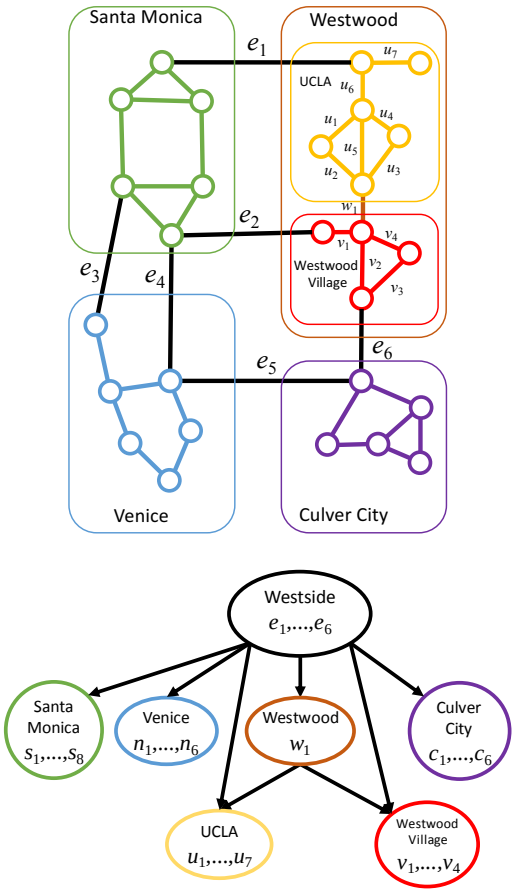


Figure 6: A hierarchical map and its cluster DAG.

an internal node, they are the edges that are used to cross between the child sub-regions. Here, the root cluster represents the LA Westside and its variables represent the edges $e_1, \ldots, e_6$ that are used to cross between its four (immediate) sub-regions. Finally, each leaf cluster represents the edges that are strictly contained in that sub-region. Consider the conditional independencies implied by a cluster DAG: given the edges used to enter a region, the route that we take inside of a region is independent of the route taken outside the region. We make an additional assumption, due to (Choi, Shen, and Darwiche 2017), which we also exploit: the route taken inside of a region must also be a simple path.

In a *binary hmap*, regions are recursively split into two sub-regions. Such maps have three key properties, that lead to its tractability: (1) the simple-path constraints for interior nodes are trivial to compile,[6] (2) the number of parent instantiations that we need to consider in a conditional PSDD is quadratic in the number of edges crossing into the region,[7] and (3) if the simple-path constraints of a leaf region are too

[4] For a family $\mathbf{X}|\mathbf{P}$, let w be its conditional PSDD, and let v' be the projection of decision vtree v onto $\mathbf{X}|\mathbf{P}$. First, we extract the SDD circuit of a conditional PSDD. For both vtrees w and v', we replace the leaf cluster $\mathbf{X}$ with a dummy leaf vtree; we also replace the corresponding PSDD nodes of the conditional PSDD with dummy terminals. We then re-normalize the resulting (algebraic) SDD so that it conforms to v' instead of w, and replace the dummy terminals with the original PSDD nodes.

[5] To re-normalize an SDD, it also suffices to re-construct the SDD bottom-up for the new vtree, using the SDD's `apply` operator (Darwiche 2011), which we did in our experiments.

[6] A path consists of crossing from one region to another using a single edge, or not at all, because of the simple path assumption (in a simple path, we cannot visit the same region twice).

[7] Due to the simple-path constraint, a path cannot re-enter a region once it has exited it. Hence there are only a quadratic number of way to enter and exit a region to consider (at most two incident

hard to compile, we can make the map deeper until they are compilable. With a binary hmap, we obtain the following *polynomial* bound on the size of its joint PSDD.

Theorem 1 *Consider a binary hmap with t nodes, where k is the maximum number of edges assigned to a cluster and n is the maximum number of edges that cross into a region. Let m denote the size of the largest PSDD of any leaf region. The size of the joint PSDD is $O(t \cdot n^2 \cdot (m + k))$.*

A proof is included in the Appendix.

5 Learning Binary Hierarchical Maps

In this section, we consider how to learn a binary hmap, and hence, the structure of an SBN.

Random Binary Hmaps. Consider the following simple algorithm for inducing a random binary hmap, based on recursively decomposing a map into two regions. First, pick two seed nodes a and b of a map, where a will belong to one region and b will belong to the other. Each region alternates between absorbing a neighboring node into their region (if possible), until all nodes are absorbed. A deeper hmap can then be produced by recursing on the sub-regions. We typically recurse until each leaf region is small enough to be compilable to an SDD.

Learning Hmaps from Data. We next propose a heuristic for learning a binary hierarchical map from a dataset consisting of routes. Consider a popular road on a map which is used by many routes in the data. A random partitioning of the map may put one intersection (node) of the road in one region, the next intersection in another region, and the third intersection in the same region as the first. Hence, a route on this road would exit the first region, enter the second, and then re-enter the first. Such a route is not simple relative to an hmap as it visits the same region twice. This assumption was introduced by (Choi, Shen, and Darwiche 2017), and is important for guaranteeing tractability as we discussed in the previous section.

Hence, we propose a heuristic that tries to avoid situations like the above, for learning a binary hmap from data. Our approach is bottom-up.[8] Intuitively, we want to cluster nodes together if they are commonly used by the same route. First, we assign each node to its own region. Next, we have an edge between regions if there is a street connecting them, and we give that edge a weight based on the number of routes in the data that crosses from one region to the other. We then find a maximum weight matching[9] and merge each of the paired regions. We update the scores between regions and repeat, until we obtain a single cluster. Next, from the clustering, we want to extract an hmap whose leaf regions are as large as possible, given an upper limit. Hence, to obtain our final hmap, we navigate the clustering in depth-first fashion until we find the first node under our limit which we take as an hmap leaf. We then backtrack and continue until we have picked all of our leaves.

edges can be used).

[8]Essentially, learning a binary hmap is a type of hierarchical clustering. See, e.g., (Murphy 2012) which discusses both bottom-up (agglomerative) and top-down (divisive) clustering.

[9]We used the `networkx` python module.

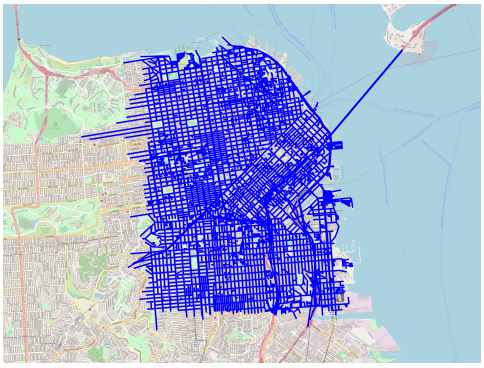


Figure 7: The road network of downtown SF.

6 Empirical Evaluation

We now evaluate the inference and learning algorithms we proposed for SBNs. First, we compare inference using PSDDs with a baseline using jointree inference with sparse tables. Next, we evaluate the quality of SBNs learned using our proposed heuristic, in a route prediction task. Finally, we highlight the utility of SBN models for route classification.

Efficiency of Exact Inference. First, we compare the efficiency of our exact inference algorithm for SBNs, with jointree message-passing using sparse tables (Larkin and Dechter 2003).[10] We evaluate these inference algorithms on an SBN induced from a hmap, as done by (Choi, Shen, and Darwiche 2017). We obtained public map data of San Francisco (SF) from openstreetmap.org, and selected 7 increasingly larger regions of SF. We induced a random binary hmap from each map, as described in Section 5. For each map size, we generated $5 \times 5 \times 5 = 125$ problem instances: (1) 5 random hmaps, (2) 5 random parameterizations of the SBN, and (3) 5 MPE queries (what is the most likely route between randomly chosen source/destination pairs). In Table 1, we report averages and standard deviations for the times to (1) compile the joint PSDD (offline step), (2) evaluate the PSDD (online query step), and (3) run the jointree algorithm. For each region, the graph size reported is the number of edges (road segments) in the map, and depth is the average depth of the binary hmap.

The largest map considered had $10,500$ edges; its graph is highlighted in Figure 7. On average, the corresponding SBN had 1.7M parameters and the joint PSDD was of size 8.9M (edges). Note that this map is over 26 times larger than the map considered by (Choi, Shen, and Darwiche 2017),[11] which highlights the scalability of our approach. Next, observe that as we increase the size of the map, evaluation time of the joint PSDD is increasingly more efficient than the

[10]We first reduce our SBN to a flat factor graph, and induce sparse factors from each conditional PSDD. We pick a jointree structure that reflects the hmap that was used; this allows it to exploit the significant determinism in the CPTs. Otherwise, jointree inference would be intractable for such models.

[11]For inference and learning, (Choi, Shen, and Darwiche 2017) conjoined the SDDs of each region, and estimated parameters on the joint SDD to obtain a PSDD, which is much less efficient (statistically) for learning (Shen, Choi, and Darwiche 2017).

Table 1: Compilation versus jointree inference by map size. Size is # of edges. Improvement is jointree over evaluation time.

map size	depth	compilation time (s)			evaluation time (s)			jointree time (s)			improvement
910	6.2	47.422	±	8.708	2.223	±	0.572	7.374	±	1.595	3.317 ×
2,103	7.4	290.919	±	70.580	9.156	±	3.298	27.044	±	2.809	2.953 ×
3,241	8.6	785.877	±	171.873	13.458	±	3.905	55.011	±	2.447	4.087 ×
5,202	9.6	2,003.242	±	424.589	21.492	±	2.783	135.941	±	41.475	6.325 ×
7,621	10.6	3,938.633	±	558.427	26.315	±	5.111	249.658	±	41.422	9.487 ×
9,290	10.8	8,508.526	±	2,778.961	51.645	±	17.204	519.567	±	201.134	10.060 ×
10,500	11.2	12,333.635	±	3,086.726	60.136	±	14.517	607.294	±	173.474	10.098 ×

jointree algorithm, and over one order-of-magnitude more efficient in the largest graph evaluated. This is in part due to the ability of PSDDs to exploit context-specific independence as well as determinism, whereas sparse tables only take advantage of determinism. While binary hmaps are tractable, we expect the gap between compilation and jointree inference to grow for general SBNs. Finally, we note that compilation time is non-trivial, although this is a one-time cost that is spent offline. When many queries are performed online, the time savings obtained by using joint PSDDs will be a considerable advantage.

Learning Hierarchical Maps. We now evaluate the algorithm proposed in Section 5 for learning binary hmaps, using a route prediction task that we describe next. We first took the region of SF covering 910 edges from Table 1. We took the `cabspotting` dataset of GPS traces collected from taxicab routes in SF (Piorkowski, Sarafijanovoc-Djukic, and Grossglauser 2009). Using the map-matching API of the `graphhopper` package, we projected GPS traces onto the map. We used 8,196 routes from this dataset to learn the structure and parameters of our SBNs (using Laplace smoothing).[12] We learned a binary hmap for an SBN using our proposed heuristic and also using a random binary hmap, both described in Section 5. We used another 128 routes as test routes to perform route prediction: given a source, destination and a partial trip so far (half the trip), what is the most likely completion? This is an MPE query on a PSDD, which we computed using the inference algorithm based on the PSDD multiply operator, proposed in Section 3.

For each route in the test set, we measure its similarity with the route predicted from the PSDD, using three metrics: (1) dissimilarity in segment number (DSN), which counts the proportion of non-common road segments between the true and predicted route, (2) Hausdorff distance (Groves, Nunes, and Gini 2014), which matches all points in one path with the closest point in the other path, and reports the largest such match (normalized by the true trip length), and (3) the difference between trip lengths (normalized by the true trip length). Note that each metric has its own advantages and disadvantages. Further, we expect better hmaps to provide more accurate route predictions.[13]

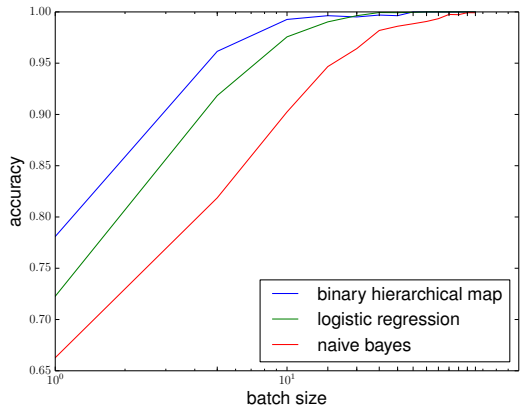


Figure 8: Route classification.

We evaluate the quality of routes predicted by each of the two SBNs in the following table:

hmap	DSN	Haus.	trip length	# bad routes
random	0.300	0.120	0.147	3,833 / 59
heuristic	**0.250**	**0.089**	**0.076**	**2,791 / 41**

Each entry is an average over 10 runs with randomly sampled training and testing sets (of size 8,196 and 128) from the 31,175 cabspotting routes inside the region. We see that for all three metrics, our heuristic learns a binary hmap with much higher predictive accuracy. For example, the predictions from the heuristic hmap had half the error compared to a random hmap, in terms of trip length. In the last column, we consider how many simple routes become invalid in the binary hmap, as discussed in Section 5. Invalid routes visit the same region twice in the hmap—such routes have probability zero in the SBN. We separately report the number of invalid routes in the training and testing sets. The random binary hmap has 137% more invalid routes, indicating that our heuristic is effective at lowering the number of invalid routes that result in a hierarchical decomposition.

[12] To learn the parameters of an SBN, we independently learn the parameters of its conditional PSDDs, which can be done in closed form (Shen, Choi, and Darwiche 2018). A route dataset however may have paths that are not simple, or paths that do not respect the binary hmap (i.e., visits the same region twice). We can still utilize such routes for training, which helped in our experiments. First, we project each route in the training set onto each family of the SBN—multiple paths through the same region becomes a set of independent paths. A projected route may still not be simple; in this case, we segment it further into sub-paths that are simple.

[13] Note that standard metrics based on test-set likelihood are difficult to apply. Our model assumes paths are simple across regions, so routes violating this assumption have zero probability. Prior empirical comparisons with SBNs considered, instead of likelihood, domain-specific tasks such as next-turn prediction (Choi, Shen, and Darwiche 2017; Krumm 2008).

Route Classification. We report results on route classification in Figure 8. We consider two classes of routes from two datasets: (1) the `cabspotting` dataset (Taxi), and (2) a simulated dataset collected by querying Google Maps Directions API with source/destination pairs (Google).[14] We took $2^{15} = 32,768$ and $2^{12} = 4,096$ routes from each dataset for training/testing. We took a map of size 5,374 edges and learned a binary hmap, as in Section 5; the SBN had $737,928$ parameters. We trained two sets of SBN parameters (using Laplace smoothing), one for each dataset, yielding a (structured) naive Bayes classifier (Choi, Tavabi, and Darwiche 2016).[15] The Taxi dataset was collected in 2008, which predates the proliferation of GPS navigators. Hence, we view our Google vs. Taxi classifier as discriminating between drivers with/without GPS navigation, or alternatively, Uber drivers (with) vs Taxi drivers (without). Figure 8 summarizes our results, where we compare against (1) an (unstructured) naive Bayes classifier (2) and a logistic regression classifier. Both uses each edge as a binary feature (each edge is either present or absent). On the x-axis, we provide each classifier with a batch of routes of increasing size, from 1 to 60. Each batch of size x is a set of x routes of the same type—the idea is that the one can better distinguish a driver as an Uber or a taxi driver, as more routes from the same driver are provided. As we give each classifier more routes from the same type of driver, we achieve higher accuracy (as expected), converging to 100% accuracy. Clearly, our SBN (using a binary hmap) is superior to logistic regression, which in turn is superior to naive Bayes.

7 Conclusion

In this paper, we proposed the first exact inference algorithm for structured Bayesian networks (SBNs), based on compiling SBNs to PSDDs. We highlighted the importance of vtrees for the purposes of inference and separately for learning. Next, we identified a tractable sub-class of SBNs based on binary hierarchical maps, for learning route distributions. We also proposed an algorithm to learn the structure of binary hierarchical maps from data. Empirically, we showed that inference based on compilation can be an order-of-magnitude more efficient compared to jointree inference. In our experiments, we demonstrated the practical utility of our algorithms using route distributions learned from real-world GPS data. We showed that our inference algorithm can scale to much larger maps than those previously considered, and that our learning algorithm can learn structures with improved route-prediction performance. Finally, we demonstrated the utility of binary hierarchical maps for the purposes of route classification.

[14] In particular, we initially took the map of size 5,202 from Table 1, and from the `cabspotting` dataset, we took all 172,265 routes strictly contained in the map. For each route, we took the source, destination, day-of-week and time-of-day, which we used to request a corresponding route from Google Maps.

[15] If a train/test route is invalid (visits the same region twice), then it has probability zero in the SBN. In this case, we segment the route into multiple valid routes, as in Footnote 6, for training. For testing, we observe each route segment as a feature in the structured naive Bayes classifier.

Acknowledgments We thank John Stucky and Yaacov Tarko for comments and discussions on this paper. This work has been partially supported by NSF grant #IIS-1514253, ONR grant #N00014-18-1-2561 and DARPA XAI grant #N66001-17-2-4032.

A Proof of Theorem 1

We provide a construction of the joint PSDD in this section.

For a given region, we refer to its *external* edges as those edges that have one endpoint inside the region and the other endpoint outside the region. External edges are the edges that are used to enter and exit a region. Further, a path is *simple* if it does not visit the same node twice. We say that a path is *hierarchically simple* if it does not visit the same region twice, in the hmap. In an SBN of a binary hmap, all paths must be hierarchically simple; otherwise, they have probability zero (Choi, Shen, and Darwiche 2017).

Consider a leaf node c in a binary hmap. We want the PSDDs representing routes inside this region. Under the hierarchical simple-path assumption, at most two external edges will be used—we cannot visit the same region twice. At the most, we can enter and exit a region.

When exactly two external edges e_1 and e_2 are used, we want a PSDD over all simple paths that connect to both e_1 and e_2 in the region c. We refer to this PSDD as $\mathsf{non\text{-}terminal}_c(e_1, e_2)$, because all paths must pass through the region c. When exactly one external edge e is used, we want a PSDD over all simple paths that start at edge e and then end inside region c. We refer to this PSDD as $\mathsf{terminal}_c(e)$, because all paths terminate in region c. Finally, if no external edge is used, we want a PSDD over all simple paths strictly contained inside the region. We refer to this PSDD as $\mathsf{internal}_c$. In PSDDs $\mathsf{terminal}_c(e)$ and PSDDs $\mathsf{non\text{-}terminal}_c(e_1, e_2)$ that connect to the same endpoint inside the region, we allow for an empty path inside the region. Further, we assume that all PSDDs $\mathsf{non\text{-}terminal}_c(e_1, e_2)$, $\mathsf{terminal}_c(e)$, and $\mathsf{internal}_c$ have a bounded size. This can be ensured by using a binary hmap that is deep enough to have small enough regions. In our experiments, we used the GRAPHILLION package[16] to compile regions into ZDDs, which are then converted to SDDs.

Consider an internal node c in a binary hmap, which has a left child region l and a right child region r. Each internal node c is itself a binary hmap, rooted at c. It represents a map consisting of all nodes and edges inside the corresponding binary hmap. As we did previously, we will construct PSDDs $\mathsf{non\text{-}terminal}_c(e_1, e_2)$, $\mathsf{terminal}_c(e)$ and $\mathsf{internal}_c$ over the different types of routes implied by the selected external edges. These PSDDs can be specified using the PSDDs of its left and right child sub-regions.

Suppose we have external edges $e_1, \ldots, e_n$, and the edges $m_1, \ldots, m_k$ that cross between the left and right sub-regions. Let empty_c denote the PSDD for c containing no routes (all edges must be set to false). Consider the PSDD $\mathsf{internal}_c$. An internal route is either (1) strictly contained in the left region, (2) strictly contained in the right region,

[16] https://github.com/takemaru/graphillion

or (3) it crosses the two regions using exactly one edge m_i. Thus, the corresponding SDD has the elements:

$$\begin{aligned}&\mathsf{internal}_l, \mathsf{empty}_r\\ &\mathsf{empty}_l, \mathsf{internal}_r\\ &\mathsf{terminal}_l(m_1), \mathsf{terminal}_r(m_1)\\ &\vdots\\ &\mathsf{terminal}_l(m_k), \mathsf{terminal}_r(m_k)\end{aligned}$$

By the hierarchical simple-path assumption, the path between the left and right regions must consist of a single edge.

Consider the PSDDs $\mathsf{terminal}_c(e)$. Suppose that e connects to a node in the left child region (the case for the right child is symmetric). There are two cases: (1) the path stays in the left, or (2) the path crosses into the right using one edge m_i. This PSDD has an SDD with the elements:

$$\begin{aligned}&\mathsf{terminal}_l(e), \mathsf{empty}_r\\ &\mathsf{non\text{-}terminal}_l(e, m_1), \mathsf{terminal}_r(m_1)\\ &\vdots\\ &\mathsf{non\text{-}terminal}_l(e, m_k), \mathsf{terminal}_r(m_k)\end{aligned}$$

Consider the PSDDs $\mathsf{non\text{-}terminal}_c(e_1, e_2)$. Suppose that e_1 connects to the left child region and that e_2 connects to the right child region (the reverse case is symmetric). Here, the path must cross from the left to the right using one edge m_i. This PSDD has an SDD with the elements:

$$\begin{aligned}&\mathsf{non\text{-}terminal}_l(e_1, m_1), \mathsf{non\text{-}terminal}_r(m_1, e_2)\\ &\vdots\\ &\mathsf{non\text{-}terminal}_l(e_1, m_k), \mathsf{non\text{-}terminal}_r(m_k, e_2)\end{aligned}$$

If e_1 and e_2 both connect to the same region, say the left one, then the path must stay inside the left region. We have an SDD with a single element:

$$\mathsf{non\text{-}terminal}_l(e_1, e_2), \mathsf{empty}_r.$$

To count the total size of the joint PSDD, we count the number of PSDD nodes that we constructed, and also count the size of each node (number of elements). Moreover, we do not count elements with a false sub in our PSDD, which are not needed to represent the distribution. First, for a leaf node in the binary hmap, if n is the number of its external edges, then we have $O(n^2)$ distinct PSDDs, each having a bounded size m. If there are t nodes in the binary hmap, then there are $O(t)$ leaf nodes. Hence, the total size of the leaf PSDDs is $O(t \cdot n^2 \cdot m)$. Second, for an internal node in the binary hmap, if n is the number of its external edges, then we have $O(n^2)$ PSDD nodes. If k is the number of edges that cross between the left and right sub-regions, then the PSDD node has $O(k)$ elements. Thus, the number of PSDD nodes for internal nodes in the binary hmap is $O(t \cdot n^2)$ and their aggregate size is $O(t \cdot n^2 \cdot k)$. Thus the total size of the joint PSDD is $O(t \cdot n^2 \cdot m + t \cdot n^2 \cdot k)$.

References

Boutilier, C.; Friedman, N.; Goldszmidt, M.; and Koller, D. 1996. Context-specific independence in Bayesian networks. In *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, 115–123.

Choi, A., and Darwiche, A. 2013. Dynamic minimization of sentential decision diagrams. In *Proceedings of the 27th Conference on Artificial Intelligence (AAAI)*.

Choi, A.; Shen, Y.; and Darwiche, A. 2017. Tractability in structured probability spaces. In *NIPS*, 3480–3488.

Choi, A.; Tavabi, N.; and Darwiche, A. 2016. Structured features in naive Bayes classification. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI)*.

Darwiche, A. 2011. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of IJCAI*, 819–826.

Groves, W.; Nunes, E.; and Gini, M. L. 2014. A framework for predicting trajectories using global and local information. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, 1–10.

Kisa, D.; Van den Broeck, G.; Choi, A.; and Darwiche, A. 2014. Probabilistic sentential decision diagrams. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR)*.

Krumm, J. 2008. A Markov model for driver turn prediction. Technical report, SAE Technical Paper.

Larkin, D., and Dechter, R. 2003. Bayesian inference in the presence of determinism. In *AISTATS*.

Liang, Y.; Bekker, J.; and Van den Broeck, G. 2017. Learning the structure of probabilistic sentential decision diagrams. In *Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence (UAI)*.

Murphy, K. P. 2012. *Machine Learning: A Probabilistic Perspective*. MIT Press.

Oztok, U.; Choi, A.; and Darwiche, A. 2016. Solving PP^{PP}-complete problems using knowledge compilation. In *Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 94–103.

Piorkowski, M.; Sarafijanovoc-Djukic, N.; and Grossglauser, M. 2009. A Parsimonious Model of Mobile Partitioned Networks with Clustering. In *The First International Conference on COMmunication Systems and NETworkS (COMSNETS)*.

Shen, Y.; Choi, A.; and Darwiche, A. 2016. Tractable operations for arithmetic circuits of probabilistic models. In *Advances in Neural Information Processing Systems 29 (NIPS)*, 3936–3944.

Shen, Y.; Choi, A.; and Darwiche, A. 2017. A tractable probabilistic model for subset selection. In *Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence (UAI)*.

Shen, Y.; Choi, A.; and Darwiche, A. 2018. Conditional PSDDs: Modeling and learning with modular knowledge. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, 6433–6442.

Zheng, Y. 2015. Trajectory data mining: An overview. *ACM Transaction on Intelligent Systems and Technology*.