# Structured Set Variable Domains in Bayesian Network Structure Learning

**Fulya Trösser**
Université Fédérale de Toulouse, ANITI, INRAE, UR 875, 31326 Toulouse, France

**Simon de Givry** ORCID
Université Fédérale de Toulouse, ANITI, INRAE, UR 875, 31326 Toulouse, France

**George Katsirelos** ORCID
Université Fédérale de Toulouse, ANITI, INRAE, MIA Paris, AgroParisTech, 75231 Paris, France

## Abstract

Constraint programming is a state of the art technique for learning the structure of Bayesian Networks from data (Bayesian Network Structure Learning – BNSL). However, scalability both for CP and other combinatorial optimization techniques for this problem is limited by the fact that the basic decision variables are set variables with domain sizes that may grow super polynomially with the number of random variables. Usual techniques for handling set variables in CP are not useful, as they lead to poor bounds. In this paper, we propose using decision trees as a data structure for storing sets of sets to represent set variable domains. We show that relatively simple operations are sufficient to implement all propagation and bounding algorithms, and that the use of these data structures improves scalability of a state of the art CP-based solver for BNSL.

**2012 ACM Subject Classification** Computing methodologies → Learning in probabilistic graphical models; Theory of computation → Discrete optimization

**Keywords and phrases** Combinatorial Optimization, Bayesian Networks, Decision Trees

**Digital Object Identifier** 10.4230/LIPIcs.CP.2022.40

**Supplementary Material** *Software (Source Code)*: `https://gkatsi.github.io/elsa-cp22.tar.gz`

**Funding** This work has been partly funded by the "Agence nationale de la Recherche" (ANR-16-CE40-0028 Demograph project and ANR-19-PIA3-0004 ANTI-DIL chair of Thomas Schiex).

**Acknowledgements** Thanks to the GenoToul (Toulouse, France) Bioinformatics platform for computational support.

## 1 Introduction

Bayesian Networks (BNs) are directed probabilistic graphical models, which can describe a normalized joint probability distribution over a potentially large set of random variables, by exploiting conditional independence to decompose the function. Learning the structure of BNs from data (the Bayesian Network Structure Learning problem, BNSL) is a challenging combinatorial optimization problem. There exist constraint-based approaches to learn BNs, which use local conditional independence tests, and score-based approaches, which use a decomposable score function to score each potential structure and aim to find the structure that minimizes this score. The former are known to be efficient, but have trouble with noisy data. The latter yield a known to be NP-hard problem [4], which additionally has proved very challenging in practice.

There exist complete methods for score-based BNSL based on dynamic programming [20], heuristic search [24, 8], maximum satisfiability [2], branch-and-cut [1] and constraint programming [22, 21]. Branch-and-cut and constraint programming have proven to be the most successful of these methods. However, scaling them up remains challenging. One challenge


has to do with the decomposition of the scoring functions: these assign a score to each potential set of parents of each vertex and the score of a specific structure is the sum of the scores of each parent set. This means that the objective function must have a term for each potential parent set, a potentially exponential number of terms. There are various methods by which this number is made manageable, but it is still among the greatest obstacles to scalability. Moreover, the best solvers, ILP-based GOBNILP [1], and CP-based ELSA [21] also explicitly have this set of parent sets in other parts of the model as well, in the case of ELSA as domains of variables.

Here, we propose exploiting the fact that these domains are structured, i.e., that each value is a set. Specifically, we show that we can represent potential parent sets as paths on decision trees and that using these decision trees we can answer queries more efficiently than by traversing a list of domain values. This feature has not been exploited in BNSL in the past and allows us to solve large instances more efficiently.

# 2 Background

## 2.1 Bayesian Networks

A Bayesian Network is a directed graphical model $B = \langle G, P \rangle$ where $G = \langle V, E \rangle$ is a directed acyclic graph (DAG) called the structure of $B$ and $P$ are its parameters. A BN describes a normalized joint probability distribution. Each vertex of the graph corresponds to a random variable and presence of an edge between two vertices denotes direct conditional dependence. Each vertex $v_i$ is also associated with a Conditional Probability Distribution $P(v_i \mid parents(v_i))$. The CPDs are the parameters of $B$.

Learning a BN from a set of multivariate discrete data using the score based method uses a decomposable scoring function (such as BIC [19, 14] or BDeu [3, 12]) which assigns, based on the data, a score to each potential parent set of each vertex. The BNSL problem is the problem of finding the structure $G$ which minimizes this scoring function.

The number of candidate parent sets can in principle be exponentially large, but it is typically kept in check. For one, the BIC scoring function [19, 14] guarantees that the number of candidate parent sets grows only logarithmically with the size of the data set. Second, there exist dedicated pruning rules [7, 6] which reduce the set further. As a last resort, an upper bound can be placed on the cardinality of parent sets. This is necessary especially in larger instances, where it is necessary to limit cardinality to as low as 3 in some cases.

## 2.2 CP-based BNSL

ELSA [21] is a CP-based solver for the BNSL, based on the CPBayes solver [22]. The constraint model used in ELSA has several features that we do not discuss here. Instead, we focus on the part that is relevant to our contribution. For each random variable $X$, there exists a corresponding CSP variable $P_X$ whose domain is the set of candidate parent sets of $X$. These are unsurprisingly called parent set variables. There exists an acyclicity constraint over these which requires that their instantiation yields an acyclic graph. ELSA enforces GAC on this constraint. The central part of the GAC algorithm is algorithm 1, `acycChecker`. `acycChecker` determines in time $O(n^2 d)$ whether the current set of domains admits an acyclic solution, based on the property that in any acyclic graph, for any subset of vertices $C$, at least one of the vertices $v \in C$ has a parent set that does not intersect $C$.

In addition, ELSA computes lower bounds by approximately solving the linear relaxation

**Algorithm 1** Acyclicity checker

```
1  acycChecker (P, D)
2  order ← {}
3  changes ← true
4  while changes do
5      changes ← false
6      foreach v ∈ P \ order do
8          if ∃S ∈ D(v) s.t. S ⊆ order then
10             order ← order + v
11             changes ← true
12 return order
```

of the ILP (1), which was proposed by Bartlett and Cussens [1] for the GOBNILP solver.

$$\min \sum_{v \in \mathbf{P}, S \subseteq V \setminus \{v\}} \sigma^v(S) x_{v,S} \tag{1a}$$

$$s.t. \sum_{S \in PS(v)} x_{v,S} = 1 \qquad \forall v \in \mathbf{P} \tag{1b}$$

$$\sum_{v \in C, S \in PS^{-C}(v)} x_{v,S} \geq 1 \qquad \forall C \subseteq \mathbf{P} \tag{1c}$$

$$x_{v,S} \in \{0,1\} \qquad \forall v \in \mathbf{P}, S \in PS(v) \tag{1d}$$

This is an exponentially large ILP, but on the flip side, the constraints (1c), called *cluster constraints* are facets of the polytope [5]. Hence, following GOBNILP, ELSA starts with none of the cluster constraints in the linear relaxation and then adds only those that can improve the dual bound. This is an NP-hard problem. GOBNILP solves this NP-hard problem to find violated cluster constraints, while ELSA uses a polynomial time algorithm which can identify a strict subset of all improving cluster constraints. The central element of the algorithm used in ELSA to find cluster constraints uses algorithm 1 on the domains restricted only to values which have reduced cost 0 in the current dual solution of the linear relaxation.

Both in finding improving cluster constraints and in enforcing GAC on the acyclicity constraint, the main bottleneck is line 8 of algorithm 1, which tests whether there exists in $D(v)$ a value which is a subset of a given set. As domain sizes grow drastically faster than the number of random variables, it is crucial to optimize this step. In practical terms, even given the mitigations mentioned earlier, the average domain size can be in the thousands for larger instances.

## 3 Related work

A typical approach to dealing with large domain sizes in constraint programming is to enclose the set of domain values with an underestimation and an overestimation and reason with those instead. Sometimes, this can even be achieved without any loss in strength of inference. This is the case, for example, when representing only the bounds of a variables that are only used in linear inequalities. In the case where the values of a domain are sets, the variable is called a set variable. Its domain can be represented with the subset bound scheme [9], which

underestimates by a set indicating all elements which appear in all remaining domain values and overestimates by a set indicating all elements which appear in any remaining domain value. The length-lex scheme uses lexicographic and cardinality information to get a tighter under- and over-estimation [10]. However, detecting infeasibility of the acyclicity constraint is crucial for the performance of CPBayes and even more for ELSA. Hence, over-estimating the actual domain in our case would lead to poor performance.

Hawkins et al. [11] followed an approach which is closer to our own, by using ROBDDs (reduced ordered binary decision diagrams) to represent domains. ROBDDs are diagrams like decision trees, but they require the same variable ordering in each branch and isomorphic subgraphs are merged, so that the underlying graph is a DAG rather than a tree. They can be significantly more compact than decision trees. However, Hawkins et al. used them in a setting where all constraints can be expressed as operations on ROBDDs. They do not deal with costs of the domain values, and in particular with reduced cost filtering.

## 4 Decision Trees as domain store

The set of sets that are in a domain can be seen as the set of solutions of a propositional formula, in which we have a propositional variable for each element of the universe. Therefore, knowledge compilation languages such as ROBDDs can be used to represent a domain.

There exist several queries and operations performed on the domains in ELSA, but not all are critical to optimize, as they are not performed often enough to dominate the runtime. In particular, we want to address the test in line 8, which asks whether the domain contains a set which is a subset of another given set. Therefore, the main queries that need to be supported efficiently by a domain store for our purposes are:

1. Does there exist a domain value $S$ such that $S \subseteq T$ for some $T$?
2. Does there exist a domain value $S$ with reduced cost 0 such that $S \subseteq T$ for some $T$?

And the main operations, which also have to support backtracking, are:

1. Pruning a single value $S$
2. Updating the reduced cost of a value

The main issue that disqualifies ROBDDs and other reduced representations for us is that operation 1, reduced cost filtering, may remove arbitrary values, shattering the shared suffixes that an ROBDD exploits, which means that pruning may result in increasing the size of the representation and is not even guaranteed to be in linear time. Instead, we use decision trees here, in particular binary decision trees with implied literals, inspired by a similar technique in BDDs [13]. The main use of decision trees is in machine learning for classification, but their use as a data structure for representing sets of sets (or, equivalently, a knowledge compilation language) is straightforward.

We give below definitions for the specific case of binary decision trees and binary classification, as that is all we need.

▶ **Definition 1** (Binary decision tree). *Let $A$ be a set of features $\{a_1, \ldots, a_n\}$ with Boolean domains and $C_1, C_2$ be two classes. A binary decision tree $\mathcal{T}$ over the features $A$ is a directed rooted binary tree. Each internal node $n$ of $\mathcal{T}$ is labeled with a feature $l(n) \in A$ and each arc $e$ (of the at most two outgoing arcs) from $n$ is labeled with $l(e) \in \{true, false\}$ and are mutually exclusive. Each leaf node $t$ is labeled with $l(t) \in \{C_1, C_2\}$. Given an instantiation $I$ of the features, there is a unique path from the root to a leaf $t$ so that for each arc $e = (n, c)$*

*along that path, it holds that* $I(l(n)) = l(e)$*. We say that* $\mathcal{T}$ classifies $I$ *as* $l(n)$ *and that* $I$ *and the path from the root to* $n$ *are* consistent *with each other, or simply that* $I$ *and* $n$ *are consistent with each other.*

To see how we can use binary decision trees as a data structure for a set of sets, observe that we can set the features to be the variables of the indicator function of the sets in the domain and the classes as *in-set* and *not-in-set*.

This allows us to further optimize the representation. Since we only care about the *in-set* class, from now on we assume that all nodes and arcs that do not appear on a path from the root to a leaf $n$ with $l(n) =$ *in-set* are removed from the decision tree.

Additionally, we can eliminate some nodes by adding *implied literals* in each node of the tree.

▶ **Definition 2** (Binary decision tree with implied literals)**.** *A binary decision tree with implied literals is a decision tree in which each node* $n$ *(internal or leaf) is additionally labeled with a set of literals* $lit(n, C_i) \subseteq \{a = v \mid a \in A, v \in \{true, false\}\}$ *for* $i \in \{1, 2\}$*. An instantiation* $I$ *is consistent with a path to a leaf* $t$ *with* $l(t) = C_i$ *if it is consistent with all the arcs it follows and all implied literal labels* $lit(n, C_i)$ *for each node* $n$ *on the path from the root to* $t$*.*

In our case, we abbreviate $lit(n,$ *in-set*$)$ to $lit(n)$, as we ignore the class *not-in-set*. Decision trees with implied literals allow us to collapse chains, i.e., paths along which every node has outdegree 1, into a single node. Hence, they are not more compact than those without implied literals by more than a linear factor, but they have almost no overhead and, in preliminary experiments, we found them to provide some performance improvement.

In machine learning, the objective is not only to construct models that perform well on the training set, but that also generalize. Hence, it is not only acceptable, but also desirable to misclassify some samples in training sets, if that means a smaller and hence more general decision tree. In our setting, however, where we use decision trees to model a Boolean function, we accept no error. So no two sets that belong to different classes, i.e., one in *in-set* and one in *not-in-set*, are allowed to both be consistent with the same leaf node.

We place an additional constraint on the decision trees we construct, which is that each leaf node must be consistent with exactly one positive instantiation. This ensures that there exists a bijection between leaves of the tree and values in the domain. This is not as significant a constraint as it might seem at first. A leaf node $n$ that is consistent only with positive instantiations but more than one of them is expanded into a full binary tree of depth $k$, where $k$ is the number of variables (features) which have not appeared on the path from the root to $n$. However, for the queries that we care about, this means only that the corresponding algorithm will have to traverse an additional $k$ nodes before answering, and, crucially, will only arrive at this point when it is guaranteed that it will give a positive answer. Even that overhead can be eliminated with some care. Indeed, while traversing the decision tree, we can determine that we have reached such a node $n$ if the number of possible instantiations that are consistent with $n$ is equal to the number of leaves reachable from $n$. The former is $2^{n-lvl}$, where $lvl$ is the distance from the root to $n$. The latter can be computed on construction and updated as values are removed. If these are equal, we know that the subtree contains all possible subsets and we can answer our query without more search. We give more detail later.

### Constructing decision trees.

Constructing a minimum decision tree is NP-hard with respect to several metrics [15]. We use the *information gain* heuristic [17] to choose which variable to branch on in each node.

It is a natural side effect of computing the information gain that we learn how many of the sets that are consistent with a node $n$ contain the literals $a = true$ and $a = false$ for all $a \in A$. If either of these is 0, then its negation is added to the implied literal label for $n$ and $a$ is not considered as a candidate for branching. We also experimented with optimizing the in-memory layout for better cache behavior. Compared to the van-Emde Boas layout [23], a depth-first, false-child first layout performed better.

### Maintaining a decision tree during search

It is fairly straightforward to update a decision tree for a pruning. In order to prune a value, we remove the unique leaf node that corresponds to it. Once we remove a leaf, its parent may no longer be able to reach any more leaves, hence we propagate this removal upwards. We associate each removed node with the decision level in which it was removed, so on backtracking we add them back to restore the tree to its correct state.

This guarantees that the tree representation of a domain only shrinks down a branch of the branch and bound tree. Hence, the tree can remain static and we only mask nodes that do not lead to any leaves that correspond to unpruned values, which is simple to implement.

### Reduced costs

ELSA solves the linear relaxation (1) from scratch at every node, and then strengthens it by discovering new violated cluster inequalities using the acyclicity checker (algorithm 1). Both these algorithms require an efficient implementation of the subset query on the subset of values which have reduced cost 0. In contrast to the domain itself, however, this set is reset to the empty set at the beginning of every node and grows monotonically until it admits an acyclic solution. Here again, the fact that there exists a bijection between values and leaves of the tree allows us to represent the set of 0-cost values as a subset of the full decision tree. Every time the reduced cost of a value reaches 0, the unique leaf it corresponds to, as well as all its parents, are added to the set of visible nodes for these queries. This is implemented as an additional mask on top of that which hides pruned values.

### Subset queries

To answer the query "does the domain contain a value $S$ such that $S \subseteq T$?", we perform a depth first traversal of the tree. At each node $n$, we check $l(n)$. If $l(n) \notin T$, we only allow DFS to follow the outgoing arc labeled with false. If $l(n) \in T$, we allow DFS to follow both outgoing arcs. If the label $lit(n)$ contains a literal $p \notin T$, we backtrack. If we reach a leaf, we stop and report success. If we exhaust the search without reaching a leaf, we report failure.

When this procedure reaches a node which is the root of a complete subtree of depth $k$, with no additional implied literal labels, it is guaranteed to terminate after visiting exactly $k$ nodes and report success. Indeed, since this is a complete subtree, one of the outgoing arcs is always available to the depth first search, and it will reach a leaf after $k$ more steps.

This procedure can be used to answer subset queries either on the entire domain, masking away only pruned values, or on those values which have reduced cost 0, masking away both pruned values and those whose reduced cost is greater than 0.

## 5 Experimental Results

We implemented decision trees as the domain representation on top of ELSA. The default implementation of a subset query in ELSA iterates over all domain values and returns if

it finds one that is a subset of $T$. We replaced this by the depth-first traversal described in section 4 and denote this solver [1] ELSA$^{IG}$. We compare against the previous version of ELSA[2], GOBNILP[3], and CPBayes[4].

The datasets come from the UCI Machine Learning Repository[5], the Bayesian Network Repository[6], and the Bayesian Network Learning and Inference Package[7]. We have 51 medium datasets with $|V| < 64$, and 18 large datasets with $64 \leq |V| < 128$.

Local scores were computed from the datasets using B. Malone's code[8]. BDeu and BIC scores were used for medium datasets (less than 64 variables) and only BIC score for large datasets (above 64 variables). The maximum number of parents was limited to 5 for large datasets (except for `accidents.test` with maximum of 8), a high value that allows learning even complex structures [18]. For example, `jester.test` has 100 random variables, a sample size of $4,116$ and $770,950$ parent set values. For medium datasets, no restriction was applied except for some BDeu scores, where we limit sets to 6 or 8 to complete the computation of the local scores within 24 hours of CPU-time [16].

For the experiments, we modified the C++ source of CPBayes v1.1 just to allow us to run it with datasets having more than 64 variables. All computations were performed on a single core of Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz and 1 TB of RAM with a 1-hour CPU time limit for the 51 medium datasets, as well as 3 of the large datasets: `kdd.ts`, `kdd.test`, and `kdd.valid`. For the remaining 15 large datasets, we had a 10-hour CPU time limit. For the preprocessing phase, we used two different settings depending on problem size $n = |V|$: $l_{min} = 20, l_{max} = 26, r_{min} = 50, r_{max} = 500$ if $n \leq 64$, else $l_{min} = 20, l_{max} = 20, r_{min} = 15, r_{max} = 30$, where $l_{min}, l_{max}$ are partition lower bound sizes and $r_{min}, r_{max}$ are the number of restarts for the local search.

In Table 1, we show the time needed to find the optimal solution and prove optimality for all these solvers. We see that, while the use of decision trees has little effect, either positive or negative, for the smaller instances, it makes a great difference in the larger instances. In particular, ELSA$^{IG}$ is the only solver that can prove optimality for the `baudio.test` and `jester.valid` datasets. For the only instances where ELSA is significantly worse than CPBayes, `bnetflix.ts`, `bnetflix.test`, and `bnetflix.valid`, ELSA$^{IG}$ either closes the gap back down (`bnetflix.valid`) or is faster yet than CPBayes (`bnetflix.ts` and `bnetflix.test`). However, ELSA$^{IG}$ regresses with respect to ELSA in the `accidents` dataset and in `plants.test`. Part of the reason for this is that the benefit of the decision trees in terms of the reduction of the cost in answering the subset queries is comparatively reduced, hence the other overheads of decision trees dominate. For example, in `bnetflix.ts`, where ELSA$^{IG}$ significantly outperforms ELSA, ELSA looks at an average of 3315 values to answer each subset test, while ELSA$^{IG}$ visits just 90 nodes of the decision tree. On the other hand, in `accidents.test`, ELSA looks at an average of 80 values to answer each subset test, while ELSA$^{IG}$ visits 20 nodes of the decision tree. This difference is not enough to overcome other overheads.

With respect to GOBNILP, ELSA$^{IG}$ mostly outperforms it, but there are some instances

[1] Available at `https://gkatsi.github.io/elsa-cp22.tar.gz`
[2] Available at `https://gkatsi.github.io/elsa-ijcai21.tar.gz`
[3] Version 1.6.3 with CPLEX 12.7.1
[4] Retrieved from `http://cs.uwaterloo.ca/~vanbeek/Publications/CPBayes.zip`
[5] `http://archive.ics.uci.edu/ml`
[6] `http://www.bnlearn.com/bnrepository`
[7] `https://ipg.idsia.ch/software.php?id=132`
[8] `http://urlearning.org`

| | n | sum \|D\| | GOBNILP | CPBayes | ELSA | ELSA$^{IG}$ |
|---|---|---|---|---|---|---|
| carpo100_BIC | 60 | 423 | **0.5** | 76.7 (27.5) | 52.6 (0.1) | 52.5 (0.0) |
| insurance1000_BIC | 27 | 506 | **0.6** | 31.6 (0.0) | 32.8 (0.0) | 37.2 (0.0) |
| spectf_BIC | 45 | 610 | 1.4 | 4.2 (3.5) | **0.8 (0.0)** | 1.0 (0.1) |
| sponge_BIC | 45 | 618 | **1.6** | 5.1 (3.3) | 1.8 (0.0) | 2.1 (0.0) |
| insurance1000_BDe | 27 | 792 | **0.6** | 34.8 (0.0) | 34.3 (0.0) | 39.2 (0.0) |
| alarm1000_BIC | 37 | 1002 | **1.3** | 191.1 (159.1) | 34.4 (1.0) | 37.9 (1.9) |
| flag_BDe | 29 | 1324 | 4.0 | 16.6 (15.6) | **1.0 (0.2)** | 1.3 (0.2) |
| autos_BIC | 26 | 2391 | **11.9** | 18.4 (0.0) | 19.2 (0.0) | 19.9 (0.1) |
| soybean_BIC | 36 | 5926 | **48.9** | 51.9 (1.7) | 50.8 (3) | 49.6 (0.0) |
| wdbc_BIC | 31 | 14613 | 86.3 | 459.4 (398.0) | **56.0 (2.4)** | 61.7 (1.7) |
| autos_BDe | 26 | 25238 | 1005.2 | 239.5 (0.1) | **145.8 (0.8)** | 177.1 (0.3) |
| kdd.ts | 64 | 43584 | **508.8** | † | 1452.3 (274.6) | 1355.2 (141.3) |
| steel_BIC | 28 | 93026 | † | 1265.6 (1196.1) | 124.2 (71.8) | **100.6 (45.7)** |
| kdd.test | 64 | 152873 | 3178.0 | † | 1594.3 (224.4) | **1519.6 (48.9)** |
| mushroom_BDe | 23 | 438185 | † | 167.0 (4.9) | 182.6 (58.9) | **150.1 (16.7)** |
| bnetflix.ts | 100 | 446406 | † | 1086.9 (876.3) | 2103.1 (1900.9) | **557.9 (358.4)** |
| plants.test | 111 | 520148 | † | † | **28049.6 (26312.9)** | 35961.7 (33712.7) |
| jester.ts | 100 | 531961 | † | † | 21550.5 (21003.7) | **7951.4 (7301.6)** |
| accidents.ts | 100 | 568160 | **1932.2** | † | 2302.2 (930.0) | † |
| plants.valid | 111 | 684141 | † | † | **17801.6 (14080.2)** | 19819.2 (14547.9) |
| jester.test | 100 | 770950 | † | † | 30186.8 (29455.0) | **9644.5 (8742.8)** |
| baudio.test | 100 | 1016403 | † | † | † | **31077.1 (29028.1)** |
| bnetflix.test | 100 | 1103968 | † | 5794.5 (5486.2) | 10333.1 (10096.5) | **1448.8 (1137.7)** |
| bnetflix.valid | 111 | 1325818 | † | **998.1 (451.0)** | 10871.7 (10527.7) | 1476.5 (1041.5) |
| accidents.test | 100 | 1425966 | 14453.1 | † | **3641.7 (680.7)** | 8434.1 (4723.0) |
| jester.valid | 100 | 1463335 | † | † | † | **31949.5 (30624.2)** |
| accidents.valid | 100 | 1617862 | **27730.5** | † | † | † |

**Table 1** Comparison of GOBNILP, CPBayes, ELSA, and ELSA$^{IG}$ in terms of total running (and search) time in seconds. Time limit for the datasets above the line is 1 hour, and for the rest it is 10 hours. Datasets are sorted by increasing total domain size for each time limit category. For CPBayes as well as all variants of ELSA we report in parentheses time spent in search, after preprocessing finishes. † indicates a timeout.

where neither ELSA nor ELSA$^{IG}$ can match it. It seems, however, that ELSA$^{IG}$ is overall the best performer.

## 6 Conclusion

We have shown that, in the BNSL problem, we can exploit the structure of domains to get a significant speedup in learning the structure of BNs of larger datasets. Specifically, we have shown that by treating domains as sets of sets instead of sets of values, and using decision trees to represent these sets, we can answer subset queries significantly faster. This is unlike the typical approach to handling large domains in CP, which uses over- and under-approximations. Although the current implementation shows some significant improvements, answering subset queries is still the most time consuming operation performed by the solver. Moreover, the fact remains that decision trees as a knowledge compilation language are fairly weak in terms of conciseness. It remains an open question whether we can overcome the issues with ROBDDs or even DNNFs to achieve even more significant speedups.

## References

1 Mark Bartlett and James Cussens. Integer linear programming for the bayesian network structure learning problem. *Artificial Intelligence*, pages 258–271, 2017.

2 Jeremias Berg, Matti Järvisalo, and Brandon Malone. Learning optimal bounded treewidth bayesian networks via maximum satisfiability. In *Artificial Intelligence and Statistics*, pages 86–95. PMLR, 2014.

3 Wray Buntine. Theory refinement on bayesian networks. In *Proc. of UAI*, pages 52–60. Elsevier, 1991.

4 David Maxwell Chickering. Learning bayesian networks is NP-Complete. In *Proc. of Fifth Int. Workshop on Artificial Intelligence and Statistics (AISTATS)*, pages 121–130, Key West, Florida, USA, 1995. URL: `https://doi.org/10.1007/978-1-4612-2404-4_12`, `doi:10.1007/978-1-4612-2404-4\_12`.

5 James Cussens, Matti Järvisalo, Janne H Korhonen, and Mark Bartlett. Bayesian network structure learning with integer programming: Polytopes, facets and complexity. *Journal of Artificial Intelligence Research*, 58:185–229, 2017.

6 Cassio P de Campos, Mauro Scanagatta, Giorgio Corani, and Marco Zaffalon. Entropy-based pruning for learning bayesian networks using BIC. *Artificial Intelligence*, 260:42–50, 2018.

7 Cassio Polpo de Campos and Qiang Ji. Properties of bayesian dirichlet scores to learn bayesian network structures. In *Proc. of AAAI-10*, Atlanta, Georgia, USA, 2010.

8 Xiannian Fan and Changhe Yuan. An improved lower bound for bayesian network structure learning. In *Proc. of AAAI-15*, Austin, Texas, 2015.

9 Carmen Gervet. Conjunto: Constraint logic programming with finite set domains. In Maurice Bruynooghe, editor, *Logic Programming, Proceedings of the 1994 International Symposium, Ithaca, New York, USA, November 13-17, 1994*, pages 339–358. MIT Press, 1994.

10 Carmen Gervet and Pascal Van Hentenryck. Length-lex ordering for set CSPs. In *Proceedings of AAAI*, 2006.

11 Peter Hawkins, Vitaly L. Lagoon, and Peter J. Stuckey. Solving set constraint satisfaction problems using ROBDDs. *Journal of Artificial Intelligence Research*, 24:109–156, 2005.

12 David Heckerman, Dan Geiger, and David M Chickering. Learning bayesian networks: The combination of knowledge and statistical data. *Machine learning*, 20(3):197–243, 1995.

13 Y. Lai, D. Liu, and S. Wang. Reduced ordered binary decision diagram with implied literals: A new knowledge compilation approach. *Knowledge and Information Systems*, 35(3):665–712, 2013.

14 Wai Lam and Fahiem Bacchus. Using new data to refine a bayesian network. In *Proc. of UAI*, pages 383–390, 1994.

15 Hyafil Laurent and Ronald L Rivest. Constructing optimal binary decision trees is NP-complete. *Information processing letters*, 5(1):15–17, 1976.

16 Colin Lee and Peter van Beek. An experimental analysis of anytime algorithms for bayesian network structure learning. In *Advanced Methodologies for Bayesian Networks*, pages 69–80, 2017.

17 J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–107, 1986.

18 Mauro Scanagatta, Cassio P de Campos, Giorgio Corani, and Marco Zaffalon. Learning bayesian networks with thousands of variables. *Proc. of NeurIPS*, 28:1864–1872, 2015.

19 Gideon Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, 6(2):461–464, 1978.

20 Tomi Silander and Petri Myllymäki. A simple approach for finding the globally optimal bayesian network structure. In *Proc. of UAI'06*, Cambridge, MA, USA, 2006.

21 Fulya Trösser, Simon de Givry, and George Katsirelos. Improved acyclicity reasoning for bayesian network structure learning with constraint programming. In *Proceedings of IJCAI*, pages 4250–4257, 2021.

348 **22** Peter van Beek and Hella-Franziska Hoffmann. Machine learning of bayesian networks using
349 constraint programming. In *Proc. of International Conference on Principles and Practice of*
350 *Constraint Programming*, pages 429–445, Cork, Ireland, 2015.

351 **23** Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings*
352 *of the 16th Annual Symposium on Foundations of Computer Science*, pages 75–84, 1975.

353 **24** Changhe Yuan and Brandon Malone. Learning optimal bayesian networks: A shortest path
354 perspective. *J. of Artificial Intelligence Research*, 48:23–65, 2013.