# Improved Acyclicity Reasoning for Bayesian Network Structure Learning with Constraint Programming

**Fulya Trösser**[1*], **Simon de Givry**[1] and **George Katsirelos**[2]
[1]Université de Toulouse, INRAE, UR MIAT, F-31320, Castanet-Tolosan, France
[2]UMR MIA-Paris, INRAE, AgroParisTech, Univ. Paris-Saclay, 75005 Paris, France
{fulya.ural, simon.de-givry}@inrae.fr, gkatsi@gmail.com

## Abstract

Bayesian networks are probabilistic graphical models with a wide range of application areas including gene regulatory networks inference, risk analysis and image processing. Learning the structure of a Bayesian network (BNSL) from discrete data is known to be an NP-hard task with a superexponential search space of directed acyclic graphs. In this work, we propose a new polynomial time algorithm for discovering a subset of all possible cluster cuts, a greedy algorithm for approximately solving the resulting linear program, and a generalised arc consistency algorithm for the acyclicity constraint. We embed these in the constraint programming-based branch-and-bound solver CPBayes and show that, despite being suboptimal, they improve performance by orders of magnitude. The resulting solver also compares favourably with GOBNILP, a state-of-the-art solver for the BNSL problem which solves an NP-hard problem to discover each cut and solves the linear program exactly.

## 1 Introduction

Towards the goal of explainable AI, Bayesian networks offer a rich framework for probabilistic reasoning. Bayesian Network Structure Learning (BNSL) from discrete observations corresponds to finding a compact model which best explains the data. It defines an NP-hard problem with a superexponential search space of Directed Acyclic Graphs (DAG). Several constraint-based (exploiting local conditional independence tests) and score-based (exploiting a global objective formulation) BNSL methods have been developed in the past.

Complete methods for score-based BNSL include dynamic programming [Silander and Myllymäki, 2006], heuristic search [Yuan and Malone, 2013; Fan and Yuan, 2015], maximum satisfiability [Berg *et al.*, 2014], branch-and-cut [Bartlett and Cussens, 2017] and constraint programming [van Beek and Hoffmann, 2015]. Here, we focus on the latter two.

GOBNILP [Bartlett and Cussens, 2017] is a state-of-the-art solver for BNSL. It implements branch-and-cut in an integer linear programming (ILP) solver. At each node of the branch-and-bound tree, it generates cuts that improve the linear relaxation. A major class of cuts generated by GOBNILP are *cluster cuts*, which identify sets of parent sets that cannot be used together in an acyclic graph. In order to find cluster cuts, GOBNILP solves an NP-hard subproblem created from the current optimal solution of the linear relaxation.

CPBayes [van Beek and Hoffmann, 2015] is a constraint programming-based (CP) method for BNSL. It uses a CP model that exploits symmetry and dominance relations present in the problem, subproblem caching, and a pattern database to compute lower bounds, adapted from heuristic search [Fan and Yuan, 2015]. van Beek and Hoffmann showed that CPBayes is competitive with GOBNILP in many instances. In contrast to GOBNILP, the inference mechanisms of CPBayes are very lightweight, which allows it to explore many orders of magnitude more nodes per time unit, even accounting for the fact that computing the pattern databases before search can sometimes consume considerable time. On the other hand, the lightweight pattern-based bounding mechanism can take into consideration only limited information about the current state of the search. Specifically, it can take into account the current total ordering implied by the DAG under construction, but no information that has been derived about the potential parent sets of each vertex, i.e., the current domains of parent set variables.

In this work, we derive a lower bound that is computationally cheaper than that computed by GOBNILP. We give in Section 3 a polynomial-time algorithm that discovers a class of cluster cuts that provably improve the linear relaxation. In Section 4, we give a greedy algorithm for solving the linear relaxation, inspired by similar algorithms for MaxSAT and Weighted Constraint Satisfaction Problems (WCSP). Finally, in Section 5 we give an algorithm that enforces generalised arc consistency on the acyclicity constraint, based on previous work by van Beek and Hoffmann, but with improved complexity and practical performance. In Section 6, we show that our implementation of these techniques in CPBayes leads to significantly improved performance, both in the size of the search tree explored and in runtime.

## 2 Preliminaries

We give here only minimal background on (integer) linear programming and constraint programming, and refer

*Contact Author

the reader to existing literature [Papadimitriou and Steiglitz, 1998; Rossi *et al.*, 2006] for more.

**Constraint Programming**

A constraint satisfaction problem (CSP) is a tuple $\langle V, D, C\rangle$, where $V$ is a set of variables, $D$ is a function mapping variables to domains and C is a set of constraints. An assignment $A$ to $V' \subseteq V$ is a mapping from each $v \in V'$ to $D(v)$. A complete assignment is an assignment to $V$. If an assignment maps $v$ to $a$, we say it assigns $v = a$. A constraint is a pair $\langle S, P\rangle$, where $S \subseteq V$ is the *scope* of the constraint and $P$ is a predicate over $\prod_{V \in S} D(V)$ which accepts assignments to $S$ that *satisfy* the constraint. For an assignment $A$ to $S' \supseteq S$, let $A'|_S$ be the restriction of $A$ to $S$. We say that $A$ satisfies $c = \langle S, P\rangle$ if $A|_S$ satisfies $c$. A problem is satisfied by $A$ if $A$ satisfies all constraints.

For a constraint $c = \langle S, P\rangle$ and for $v \in S, a \in D(v)$, $v = a$ is generalized arc consistent (GAC) for $c$ if there exists an assignment $A$ that assigns $v = a$ and satisfies $c$. If for all $v \in S, a \in D(v)$, $v = a$ is GAC for $c$, then $c$ is GAC. If all constraints are GAC, the problem is GAC. A constraint is associated with an algorithm $f_c$, called the propagator for $c$, that removes (or *prunes*) values from the domains of variables in $S$ that are not GAC.

CSPs are typically solved by backtracking search, using propagators to reduce domains at each node and avoid parts of the search tree that are proved to not contain any solutions. Although CSPs are decision problems, the technology can be used to solve optimization problems like BNSL by, for example, using branch-and-bound and embedding the bounding part in a propagator. This is the approach used by CPBayes.

**Integer Linear Programming**

A linear program (LP) is the problem of finding

$$\min\{c^T x \mid x \in \mathbb{R}^n \wedge Ax \geq b \wedge x \geq 0\}$$

where $c$ and $b$ are vectors, $A$ is a matrix, and $x$ is a vector of variables. A feasible solution of this problem is one that satisfies $x \in \mathbb{R}^n \wedge Ax \geq b \wedge x \geq 0$ and an optimal solution is a feasible one that minimizes the objective function $c^T x$. This can be found in polynomial time. A row $A_i$ corresponds to an individual linear constraint and a column $A_j^T$ to a variable. The dual of a linear program $P$ in the above form is another linear program $D$:

$$\max\{b^T y \mid y \in \mathbb{R}^m \wedge A^T y \leq c \wedge y \geq 0\}$$

where $A, b, c$ are as before and $y$ is the vector of dual variables. Rows of the dual correspond to variables of the primal and vice versa. The objective value of any dual feasible solution is a lower bound on the optimum of $P$. When $P$ is satisfiable, its dual is also satisfiable and the values of their optima meet. For a given feasible solution $\hat{x}$ of $P$, the slack of constraint $i$ is $slack_{\hat{x}}(i) = A_i^T x - b_i$. Given a dual feasible solution $\hat{y}$, $slack^D_{\hat{y}}(i)$ is the reduced cost of primal variable $i, rc_{\hat{y}}(i)$. The reduced cost $rc_{\hat{y}}(i)$ is interpreted as a lower bound on the amount that the dual objective would increase over $b^T \hat{y}$ if $x_i$ is forced to be non-zero in the primal.

An integer linear program (ILP) is a linear program in which we replace the constraint $x \in \mathbb{R}^n$ by $x \in \mathbb{Z}^n$ and it is an NP-hard optimization problem.

**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 normalised 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 $B$.

The approach which we use here for learning a BN from data is the score-and-search method. Given a set of multivariate discrete data $I = \{I_1, \ldots, I_N\}$, a scoring function $\sigma(G \mid I)$ measures the quality of the BN with underlying structure $G$. The BNSL problem asks to find a structure $G$ that minimises $\sigma(G \mid I)$ for some scoring function $\sigma$ and it is NP-hard [Chickering, 1995]. Several scoring functions have been proposed for this purpose, including BDeu [Buntine, 1991; Heckerman *et al.*, 1995] and BIC [Schwarz, 1978; Lam and Bacchus, 1994]. These functions are decomposable and can be expressed as the sum of local scores which only depend on the set of parents (from now on, *parent set*) of each vertex: $\sigma_F(G \mid I) = \sum_{v \in V} \sigma_F^v(parents(v) \mid I)$ for $F \in \{BDeu, BIC\}$. In this setting, we first compute local scores and then compute the structure of minimal score. Although there are potentially an exponential number of local scores that have to be computed, the number of parent sets actually considered is often much smaller, for example because we restrict the maximum cardinality of parent sets considered or we exploit dedicated pruning rules [de Campos and Ji, 2010; de Campos *et al.*, 2018]. We denote $PS(v)$ the set of candidate parent sets of $v$ and $PS^{-C}(v)$ those parent sets that do not intersect $C$. In the following, we assume that local scores are precomputed and given as input, as is common in similar works. We also omit explicitly mentioning $I$ or $F$, as they are constant for solving any given instance.

Let $C$ be a set of vertices of a graph $G$. $C$ is a violated cluster if the parent set of each vertex $v \in C$ intersects $C$. Then, we can prove the following property:

**Property 1.** *A directed graph $G = \langle V, E\rangle$ is acyclic if and only if it contains no violated clusters, i.e., for all $C \subseteq V$, there exists $v \in C$, such that $parents(v) \cap C = \emptyset$.*

The GOBNILP solver [Bartlett and Cussens, 2017] formulates the problem as the following 0/1 ILP:

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

$$s.t. \sum_{S \in PS(v)} x_{v,S} = 1 \qquad \forall v \in V \tag{2}$$

$$\sum_{v \in C, S \in PS^{-C}(v)} x_{v,S} \geq 1 \qquad \forall C \subseteq V \tag{3}$$

$$x_{v,S} \in \{0, 1\} \qquad \forall v \in V, S \in PS(v) \tag{4}$$

**Algorithm 1:** Acyclicity Checker

```
acycChecker (V, D)
order ← {}
changes ← true
while changes do
    changes ← false
    foreach v ∈ V \ order do
        if ∃S ∈ D(v) s.t. (S ∩ V) ⊆ order then
1           order ← order + v
            changes ← true
return order
```

This ILP has a 0/1 variable $x_{v,S}$ for each candidate parent set $S$ of each vertex $v$ where $x_{v,S} = 1$ means that $S$ is the parent set of $v$. The objective (1) directly encodes the decomposition of the scoring function. The constraint (2) asserts that exactly one parent set is selected for each random variable. Finally, the *cluster inequalities* (3) are violated when $C$ is a violated cluster. We denote the cluster inequality for cluster $C$ as $cons(C)$ and the 0/1 variables involved as $varsof(C)$. As there is an exponential number of these, GOBNILP generates only those that improve the current linear relaxation and they are referred to as *cluster cuts*. This itself is an NP-hard problem [Cussens *et al.*, 2017], which GOBNILP also encodes and solves as an ILP. Interestingly, these inequalities are facets of the BNSL polytope [Cussens *et al.*, 2017], so stand to improve the relaxation significantly.

The CPBayes solver [van Beek and Hoffmann, 2015] models BNSL as a constraint program. The CP model has a parent set variable for each random variable, whose domain is the set of possible parent sets, as well as order variables, which give a total order of the variables that agrees with the partial order implied by the DAG. The objective is the same as (1). It includes channelling constraints between the set of variables and various symmetry breaking and dominance constraints. It computes a lower bound using two separate mechanisms: a component caching scheme and a pattern database that is computed before search and holds the optimal graphs for all orderings of partitions of the variables. Acyclicity is enforced using a global constraint with a bespoke propagator. The main routine of the propagator is `acycChecker` (Algorithm 1), which returns an order of all variables if the current set of domains of the parent set variables may produce an acyclic graph, or a partially completed order if the constraint is unsatisfiable. This algorithm is based on Property 1.

Briefly, the algorithm takes the domains of the parent set variables as input and greedily constructs an ordering of the variables, such that if variable $v$ is later in the order than $v'$, then $v \notin parents(v')$[1]. It does so by trying to pick a parent set $S$ for an as yet unordered vertex such that $S$ is entirely contained in the set of previously ordered vertices[2]. If all assignments yield cyclic graphs, it will reach a point where all remaining vertices are in a violated cluster in all possible graphs, and it will return a partially constructed order. If there exists an assignment that gives an acyclic graph, it will be possible by property 1 to select from a variable in $V \setminus order$ a parent set which does not intersect $V \setminus order$, hence is a subset of $order$. The value $S$ chosen for each variable in line 1 also gives a witness of such an acyclic graph.

An immediate connection between the GOBNILP and CPBayes models is that the ILP variables $x_{v,S}, \forall S \in PS(v)$ are the direct encoding [Walsh, 2000] of the parent set variables of the CP model. Therefore, we use them interchangeably, i.e., we can refer to the value $S$ in $D(v)$ as $x_{v,S}$.

[1]We treat $order$ as both a sequence and a set, as appropriate.

[2]When propagating the acyclicity constraint it always holds that $a \cap V = a$, so this statement is true. In section 3.1, we use the algorithm in a setting where this is not always the case.

## 3 Restricted Cluster Detection

One of the issues hampering the performance of CPBayes is that it computes relatively poor lower bounds at deeper levels of the search tree. Intuitively, as the parent set variable domains get reduced by removing values that are inconsistent with the current ordering, the lower bound computation discards more information about the current state of the problem. We address this by adapting the branch-and-cut approach of GOBNILP. However, instead of finding all violated cluster inequalities that may improve the LP lower bound, we only identify a subset of them.

Consider the linear relaxation of the ILP (1)– (4), restricted to a subset $\mathcal{C}$ of all valid cluster inequalities, i.e., with equation (4) replaced by $0 \leq x_{v,S} \leq 1 \forall v \in V, S \in PS(v)$ and with equation (3) restricted only to clusters in $\mathcal{C}$. We denote this $LP_{\mathcal{C}}$. We exploit the following property of this LP.

**Theorem 1.** *Let $\hat{y}$ be a dual feasible solution of $LP_{\mathcal{C}}$ with dual objective $o$. Then, if $C$ is a cluster such that $C \notin \mathcal{C}$ and the reduced cost $rc$ of all variables $varsof(C)$ is greater than 0, there exists a dual feasible solution $\hat{y}$ of $LP_{\mathcal{C}\cup C}$ with dual objective $o' \geq o + minrc(C)$ where $minrc(C) = \min_{x \in varsof(C)} rc_{\hat{y}}(x)$.*

*Proof.* The only difference from $LP_{\mathcal{C}}$ to $LP_{\mathcal{C}\cup C}$ is the extra constraint $cons(C)$ in the primal and corresponding dual variable $y_C$. In the dual, $y_C$ only appears in the dual constraints of the variables $varsof(C)$ and in the objective, always with coefficient 1. Under the feasible dual solution $\hat{y} \cup \{y_C = 0\}$, these constraints have slack at least $minrc(C)$, by the definition of reduced cost. Therefore, we can set $\hat{y} = \hat{y} \cup \{y_C = minrc(C)\}$, which remains feasible and has objective $o' = o + minrc(C)$, as required. □

Theorem 1 gives a class of cluster cuts, which we call RC-clusters, for reduced-cost clusters, guaranteed to improve the lower bound. Importantly, this requires only a feasible, perhaps sub-optimal, solution.

**Example 1** (Running example)**.** *Consider a BNSL instance with domains as shown in Table 1 and let $\mathcal{C} = \emptyset$. Then, $\hat{y} = 0$ leaves the reduced cost of every variable to exactly its primal objective coefficient. The corresponding $\hat{x}$ assigns 1 to variables with reduced cost 0 and 0 to everything else. These are both optimal solutions, with cost 0 and $\hat{x}$ is integral, so it is also a solution of the corresponding ILP. However, it is not a solution of the BNSL, as it contains several cycles, including $C = \{0, 2, 3\}$. The cluster inequality $cons(C)$ is violated in the primal and allows the dual bound to be increased.*

| Variable | Domain Value | Cost |
|---|---|---|
| 0 | $\{2\}$ | 0 |
| 1 | $\{2,4\}$ | 0 |
| | $\{\}$ | 6 |
| 2 | $\{1,3\}$ | 0 |
| | $\{\}$ | 10 |
| 3 | $\{0\}$ | 0 |
| | $\{\}$ | 5 |
| 4 | $\{2,3\}$ | 0 |
| | $\{3\}$ | 1 |
| | $\{2\}$ | 2 |
| | $\{\}$ | 3 |

Table 1: BNSL instance used as running example.

**Algorithm 2:** Lower bound computation with RC-clusters

```
lowerBoundRC (V, D, C)
ŷ ← DualSolve(LP_C(D))
while True do
2     C ← V \ acycChecker(V, D^rc_{C,ŷ})
3     if C = ∅ then
          return ⟨cost(ŷ), C⟩
      C ← minimise(C)
      C ← C ∪ {C}
      ŷ ← DualImprove(ŷ, LP_C(D), C)
```

We consider the problem of discovering RC-clusters within the CP model of CPBayes. First, we introduce the notation $LP_{\mathcal{C}}(D)$ which is $LP_{\mathcal{C}}$ with the additional constraint $x_{v,S} = 0$ for each $S \notin D(v)$. Conversely, $D^{rc}_{\mathcal{C},\hat{y}}$ is the set of domains minus values whose corresponding variable in $LP_{\mathcal{C}}(D)$ has non-zero reduced cost under $\hat{y}$, i.e., $D^{rc}_{\mathcal{C},\hat{y}} = D'$ where $D'(v) = \{S \mid S \in D(v) \wedge rc_{\hat{y}}(x_{v,S}) = 0\}$. With this notation, for values $S \notin D(v)$, $x_{v,S} = 1$ is infeasible in $LP_{\mathcal{C}}(D)$, hence effectively $rc_{\hat{y}}(x_{v,S}) = \infty$.

**Theorem 2.** *Given a collection of clusters $\mathcal{C}$, a set of domains $D$ and $\hat{y}$, a feasible dual solution of $LP_{\mathcal{C}}(D)$, there exists an RC-cluster $C \notin \mathcal{C}$ if and only if $D^{rc}_{\mathcal{C},\hat{y}}$ does not admit an acyclic assignment.*

*Proof.* ($\Rightarrow$) Let $C$ be such a cluster. Since for all $x_{v,S} \in varsof(C)$, none of these are in $D^{rc}_{\mathcal{C},\hat{y}}$, so $cons(C)$ is violated and hence there is no acyclic assignment.

($\Leftarrow$) Consider once again `acycChecker`, in Algorithm 1. When it fails to find a witness of acyclicity, it has reached a point where $order \subsetneq V$ and for the remaining variables $C = V \setminus order$, all allowed parent sets intersect $C$. So if `acycChecker` is called with $D^{rc}_{\mathcal{C},\hat{y}}$, all values in $varsof(C)$ have reduced cost greater than 0, so $C$ is an RC-cluster. □

Theorem 2 shows that detecting unsatisfiability of $D^{rc}_{\mathcal{C},\hat{y}}$ is enough to find an RC-cluster. Its proof also gives a way to extract such a cluster from `acycChecker`.

Algorithm 2 shows how theorems 1 and 2 can be used to compute a lower bound. It is given the current set of domains and a set of clusters as input. It first solves the dual of $LP_{\mathcal{C}}(D)$, potentially suboptimally. Then, it uses `acycChecker` iteratively to determine whether there exists an RC-cluster $C$ under the current dual solution $\hat{y}$. If that cluster is empty, there are no more RC-clusters, and it terminates and returns a lower bound equal to the cost of $\hat{y}$ under $LP_{\mathcal{C}}(D)$ and an updated pool of clusters. Otherwise, it minimises $C$ (see section 3.1), adds it to the pool of clusters and solves the updated LP. It does this by calling $DualImprove$, which solves $LP_{\mathcal{C}}(D)$ exploiting the fact that only the cluster inequality $cons(C)$ has been added.

**Example 2.** *Continuing our example, consider the behavior of* `acycChecker` *with domains $D^{rc}_{\emptyset,\hat{y}}$ after the initial dual solution $\hat{y} = 0$. Since the empty set has non-zero reduced cost for all variables,* `acycChecker` *fails with $order = \{\}$, hence $C = V$. We postpone discussion of minimization for now, other than to observe that $C$ can be minimized to $C_1 = \{1, 2\}$. We add $cons(C_1)$ to the primal LP and set the dual variable of $C_1$ to 6 in the new dual solution $\hat{y}_1$. The reduced costs of $x_{1,\{\}}$ and $x_{2,\{\}}$ are decreased to 6 and, importantly, $rc_{\hat{y}_1}(x_{1,\{\}}) = 0$. In the next iteration of* `lowerBoundRC`*,* `acycChecker` *is invoked on $D^{rc}_{\{C_1\},\hat{y}_1}$ and returns the cluster $\{0, 2, 3, 4\}$. This is minimized to $C_2 = \{0, 2, 3\}$. The parent sets in the domains of these variables that do not intersect $C_2$ are $x_{2,\{\}}$ and $x_{3,\{\}}$, so $minrc(C_2) = 4$, so we add $cons(C_2)$ to the primal and we set the dual variable of $C_2$ to 4 in $\hat{y}_2$. This brings the dual objective to 10. The reduced cost of $x_{2,\{\}}$ is 0, so in the next iteration* `acycChecker` *runs on $D^{rc}_{\{C_1,C_2\},\hat{y}_2}$ and succeeds with the order $\{2, 0, 3, 4, 1\}$, so the lower bound cannot be improved further. This also happens to be the cost of the optimal structure.*

**Theorem 3.** *Algorithm 2 terminates but is not confluent.*

*Proof.* It terminates because there is a finite number of cluster inequalities and each iteration generates one. In the extreme, all cluster inequalities are in $\mathcal{C}$ and the test at line 3 succeeds, terminating the algorithm.

To see that it is not confluent, consider an example with 3 clusters $C_1 = \{v_1, v_2\}, C_2 = \{v_2, v_3\}$ and $C_3 = \{v_3, v_4\}$ and assume that the minimum reduced cost for each cluster is unit and comes from $x_{2,\{4\}}$ and $x_{3,\{1\}}$, i.e., the former value has minimum reduced cost for $C_1$ and $C_2$ and the latter for $C_2$ and $C_3$. Then, if minimisation generates first $C_1$, the reduced cost of $x_{3,\{1\}}$ is unaffected by $DualImprove$, so it can then discover $C_3$, to get a lower bound of 2. On the other hand, if minimisation generates first $C_2$, the reduced costs of both $x_{2,\{4\}}$ and $x_{3,\{1\}}$ are decreased to 0 by $DualImprove$, so neither $C_1$ nor $C_3$ are RC-clusters under the new dual solution and the algorithm terminates with a lower bound of 1. □

**Related Work.** The idea of performing propagation on the subset of domains that have reduced cost 0 has been used in the VAC algorithm for WCSPs [Cooper *et al.*, 2010]. Our method is more light weight, as it only performs propagation on the acyclicity constraint, but may give worse bounds. The bound update mechanism in the proof of theorem 1 is also simpler than VAC and more akin to the "disjoint core phase" in core-guided MaxSAT solvers [Morgado *et al.*, 2013].

### 3.1 Cluster Minimisation

It is crucial for the quality of the lower bound produced by Algorithm 2 that the RC-clusters discovered by `acycChecker`

are minimised, as the following example shows. Empirically, omitting minimisation rendered the lower bound ineffective.

**Example 3.** *Suppose that we attempt to use* `lowerBoundRC` *without cluster minimization. Then, we use the cluster given by* `acycChecker`*,* $C_1 = \{0, 1, 2, 3, 4\}$*. We have* $minrc(C_1) = 3$*, given from the empty parent set value of all variables. This brings the reduced cost of* $x_{4,\{\}}$ *to 0. It then proceeds to find the cluster* $C_2 = \{0, 1, 2, 3\}$ *with* $minrc(C_2) = 2$ *and decrease the reduced cost of* $x_{3,\{\}}$ *to 0, then* $C_3 = \{0, 1, 2\}$ *with* $minrc(C_3) = 1$*, which brings the reduced cost of* $x_{1,\{\}}$ *to 0. At this point,* `acycChecker` *succeeds with the order* $\{4, 3, 1, 2, 0\}$ *and* `lowerBoundRC` *returns a lower bound of 6, compared to 10 with minimization. The order produced by* `acycChecker` *also disagrees with the optimum structure.*

Therefore, when we get an RC-cluster $C$ at line 2 of algorithm 2, we want to extract a minimal RC-cluster (with respect to set inclusion) from $C$, i.e., a cluster $C' \subseteq C$, such that for all $\emptyset \subset C'' \subset C'$, $C''$ is not a cluster.

Minimisation problems like this are handled with an appropriate instantiation of QuickXPlain [Junker, 2004]. These algorithms find a minimal subset of constraints, not variables. We can pose this as a constraint set minimisation problem by implicitly treating a variable as the constraint "this variable is assigned a value" and treating acyclicity as a hard constraint.

However, the property of being an RC-cluster is not monotone. For example, consider the variables $\{v_1, v_2, v_3, v_4\}$ and $\hat{y}$ such that the domains restricted to values with 0 reduced cost are $\{\{v_2\}\}, \{\{v_1\}\}, \{\{v_4\}\}, \{\{v_3\}\}$, respectively. Then $\{v_1, v_2, v_3, v_4\}$, $\{v_1, v_2\}$ and $\{v_3, v_4\}$ are RC-clusters. but $\{v_1, v_2, v_3\}$ is not because the sole value in the domain of $v_3$ does not intersect $\{v_1, v_2, v_3\}$. We instead minimise the set of variables that does not admit an acyclic solution and hence *contains* an RC-cluster. A minimal unsatisfiable set that contains a cluster is an RC-cluster, so this allows us to use the variants of QuickXPlain. We focus on RobustXPlain, which is called the deletion-based algorithm in SAT literature for minimising unsatisfiable subsets [Marques-Silva and Mencía, 2020]. The main idea of the algorithm is to iteratively pick a variable and categorise it as either appearing in all minimal subsets of $C$, in which case we mark it as necessary, or not, in which case we discard it. To detect if a variable appears in all minimal unsatisfiable subsets, we only have to test if omitting this variable yields a set with no unsatisfiable subsets, i.e., with no violated clusters. This is given in pseudocode in Algorithm 3. This exploits a subtle feature of `acycChecker` as described in Algorithm 1: if it is called with a subset of V, it does not try to place the missing variables in the order and allows parent sets to use these missing variables. Omitting variables from the set given to `acycChecker` acts as omitting the constraint that these variables be assigned a value. The complexity of `MinimiseCluster` is $O(n^3 d)$, where $n = |V|$ and $d = \max_{v \in V} |D(v)|$, a convention we adopt throughout.

**Algorithm 3:** Find a minimal RC-cluster subset of $C$

```
MinimiseCluster (V, D, C)
N = ∅
while C ≠ ∅ do
    Pick c ∈ C
    C ← C \ {c}
    C' ← V \ acycChecker(N ∪ C, D)
    if C' = ∅ then
        N ← N ∪ {c}
    else
        C ← C' \ N
return N
```

## 4 Solving the Cluster LP

Solving a linear program is in polynomial time, so in principle $DualSolve$ can be implemented using any of the commercial or free software libraries available for this. However, solving this LP using a general LP solver is too expensive in this setting. As a data point, solving the instance `steel_BIC` with our modified solver took 25,016 search nodes and 45 seconds of search, and generated $5,869$ RC-clusters. Approximately 20% of search time was spent solving the LP using the greedy algorithm that we describe in this section. CPLEX took around 70 seconds to solve $LP_{\mathcal{C}}$ with these cluster inequalities once. While this data point is not proof that solving the LP exactly is too expensive, it is a pretty strong indicator. We have also not explored nearly linear time algorithms for solving positive LPs [Allen-Zhu and Orecchia, 2015].

Our greedy algorithm is derived from theorem 1. Observe first that $LP_{\mathcal{C}}$ with $\mathcal{C} = \emptyset$, i.e., only with constraints (2) has optimal dual solution $\hat{y}_0$ that assigns the dual variable $y_v$ of $\sum_{S \in PS(v)} x_{v,S} = 1$ to $\min_{S \in PS(v)} \sigma^v(S)$. That leaves at least one of $x_{v,S}, S \in PS(v)$ with reduced cost 0 for each $v \in V$. $DualSolve$ starts with $\hat{y}_0$ and then iterates over $\mathcal{C}$. Given $\hat{y}_{i-1}$ and a cluster $C$, it sets $\hat{y}_i = \hat{y}_{i-1}$ if $C$ is not an RC-cluster. Otherwise, it increases the lower bound by $c = minrc(C)$ and sets $\hat{y}_i = \hat{y}_{i-1} \cup \{y_C = c\}$. It remains to specify the order in which we traverse $\mathcal{C}$.

We sort clusters by increasing size $|C|$, breaking ties by decreasing minimum cost of all original parent set values in $varsof(C)$. This favours finding non-overlapping cluster cuts with high minimum cost. In section 6, we give experimental evidence that this computes better lower bounds.

$DualImprove$ can be implemented by discarding previous information and calling $DualSolve(LP_{\mathcal{C}}(D))$. Instead, it uses the RC-cluster $C$ to update the solution without revisiting previous clusters.

In terms of implementation, we store $varsof(C)$ for each cluster, not $cons(C)$. During $DualSolve$, we maintain the reduced costs of variables rather than the dual solution, otherwise computing each reduced cost would require iterating over all cluster inequalities that contain a variable. Specifically, we maintain $\Delta_{v,S} = \sigma^v(S) - rc_{\hat{y}}(x_{v,S})$. In order to test whether a cluster $C$ is an RC-cluster, we need to compute $minrc(C)$. To speed this up, we associate with each stored cluster a *support pair* $(v, S)$ corresponding to the last minimum cost found. If $rc_{\hat{y}}(v, S) = 0$, the cluster is not an RC-cluster and is skipped. Moreover, parent set domains are sorted by increasing score $\sigma^v(S)$, so $S \succ S' \iff \sigma^v(S) > \sigma^v(S')$. We also maintain the maximum amount of cost transferred to the lower bound, $\Delta_v^{max} = \max_{S \in D(v)} \Delta_{v,S}$

**Algorithm 4:** GAC propagator for acyclicity

```
Acyclicity-GAC (V, D)
O ← acycChecker(V, D)
if O ⊊ V then
    return Failure
foreach v ∈ V do
    changes ← true
    i ← O^{-1}(v)
    prefix ← {O_1, ..., O_{i-1}}
4   while changes do
        changes ← false
        foreach w ∈ O \ (prefix ∪ {v}) do
            if ∃S ∈ D(w) s.t. S ⊆ prefix then
                prefix ← prefix ∪ {w}
                changes ← true
    Prune {S | S ∈ D(v) ∧ S ⊈ prefix}
return Success
```

for every $v \in V$. We stop iterating over $D(v)$ as soon as $\sigma^v(S) - \Delta_v^{max}$ is greater than or equal to the current minimum because $\forall S' \succ S, \sigma^v(S') - \Delta_{v,b} \geq \sigma^v(S) - \Delta_v^{max}$. In practice, on very large instances $97.6\%$ of unproductive clusters are detected by support pairs and $8.6\%$ of the current domains are visited for the rest[3].

To keep a bounded-memory cluster pool, we discard frequently unproductive clusters. We throw away large clusters with a productive ratio $\frac{\#productive}{\#productive+\#unproductive}$ smaller than $\frac{1}{1{,}000}$. Clusters of size 10 or less are always kept because they are often more productive and their number is bounded.

## 5 GAC for the Acyclicity Constraint

Previously, van Beek and Hoffmann[van Beek and Hoffmann, 2015] showed that using `acycChecker` as a subroutine, one can construct a GAC propagator for the acyclicity constraint by probing, i.e., detecting unsatisfiability after assigning each individual value and pruning those values that lead to unsatisfiability. `acycChecker` is in $O(n^2d)$, so this gives a GAC propagator in $O(n^3d^2)$. We show here that we can enforce GAC in time $O(n^3d)$, a significant improvement given that $d$ is usually much larger than $n$.

Suppose `acycChecker` finds a witness of acyclicity and returns the order $O = \{v_1, \ldots, v_n\}$. Every parent set $S$ of a variable $v$ that is a subset of $\{v' \mid v' \prec_O v\}$ is supported by $O$. We call such values consistent with $O$. Consider now $S \in D(v_i)$ which is inconsistent with $O$, therefore we have to probe to see if it is supported. We know that during the probe, nothing forces `acycChecker` to deviate from $\{v_1, \ldots, v_{i-1}\}$. So in a successful probe, `acycChecker` constructs a new order $O'$ which is identical to $O$ in the first $i-1$ positions and in which it moves $v_i$ further down. Then all values consistent with $O'$ are supported. This suggests that instead of probing each value, we can probe different orders.

`Acyclicity-GAC`, shown in Algorithm 4, exploits this insight. It ensures first that `acycChecker` can produce a valid order $O$. For each variable $v$, it constructs a new order $O'$ from $O$ so that $v$ is as late as possible. It then prunes all parent set values of $v$ that are inconsistent with $O'$.

**Theorem 4.** *Algorithm 4 enforces GAC on the Acyclicity constraint in $O(n^3d)$.*

*Proof.* Let $v \in V$ and $S \in D(v)$. Let $O = \{O_1, \ldots, O_n\}$ and $Q = \{Q_1, \ldots, Q_n\}$ be two valid orders such that $O$ does not support $S$ whereas $Q$ does. It is enough to show that we can compute from $O$ a new order $O'$ that supports $S$ by pushing $v$ towards the end. Let $O_i = Q_j = v$ and let $O_p = \{O_1, \ldots, O_{(i-1)}\}$, $Q_p = \{Q_1, \ldots, Q_{(j-1)}\}$ and $O_s = \{O_{i+1}, \ldots, O_n\}$.

Let $O'$ be the order $O_p$ followed by $Q_p$, followed by $v$, followed by $O_s$, keeping only the first occurrence of each variable when there are duplicates. $O'$ is a valid order: $O_p$ is witnessed by the assignment that witnesses $O$, $Q_p$ by the assignment that witnesses $Q$, $v$ by $S$ (as in $Q$) and $O_s$ by the assignment that witnesses $O$. It also supports $S$, as required.

Complexity is dominated by repeating $O(n)$ times the loop at line 4, which is a version of `acycChecker` so has complexity $O(n^2d)$ for a total $O(n^3d)$. □

## 6 Experimental Results

### 6.1 Benchmark Description and Settings

The datasets come from the UCI Machine Learning Repository[4], the Bayesian Network Repository[5], and the Bayesian Network Learning and Inference Package[6]. Local scores were computed from the datasets using B. Malone's code[7]. BDeu and BIC scores were used for medium size instances (less than 64 variables) and only BIC score for large instances (above 64 variables). The maximum number of parents was limited to 5 for large instances (except for `accidents.test` with maximum of 8), a high value that allows even learning complex structures [Scanagatta *et al.*, 2015]. For example, `jester.test` has 100 random variables, a sample size of $4,116$ and $770,950$ parent set values. For medium instances, no restriction was applied except for some BDeu scores (limit sets to 6 or 8 to complete the computation of the local scores within 24 hours of CPU-time [Lee and van Beek, 2017]).

We have modified the C++ source of CPBayes v1.1 by adding our lower bound mechanism and GAC propagator. We call the resulting solver ELSA and have made it publicly available. For the evaluation, we compare with GOBNILP v1.6.3 using SCIP v3.2.1 with cplex v12.7.0. All computations were performed on a single core of Intel Xeon E5-2680 v3 at 2.50 GHz and 256 GB of RAM with a 1-hour (resp. 10-hour) CPU time limit for medium (resp. large) size instances. We used default settings for GOBNILP with no approximation in branch-and-cut (`limits/gap = 0`). We used the same settings in CPBayes and ELSA for their preprocessing phase (partition lower bound sizes $l_{min}, l_{max}$ and local search number of restarts $r_{min}, r_{max}$). We used two

[3] See the supplementary material for more.

[4] http://archive.ics.uci.edu/ml

[5] http://www.bnlearn.com/bnrepository

[6] https://ipg.idsia.ch/software.php?id=132

[7] http://urlearning.org

| Instance | $\|V\|$ | $\sum \|ps(v)\|$ | GOBNILP | CPBayes | ELSA | ELSA \ GAC | ELSA $^{chrono}$ |
|---|---|---|---|---|---|---|---|
| carpo100_BIC | 60 | 424 | **0.6** | 78.5 (29.7) | 40.6 (0.0) | 40.7 (0.0) | 40.6 (0.0) |
| alarm1000_BIC | 37 | 1003 | **1.2** | 204.2 (172.9) | 27.8 (0.7) | 28.8 (1.5) | 29.9 (2.7) |
| flag_BDe | 29 | 1325 | 4.4 | 19.0 (18.1) | **0.9** (0.1) | 0.9 (0.1) | 1.3 (0.5) |
| wdbc_BIC | 31 | 14614 | 99.8 | 629.8 (576.6) | **48.9** (1.6) | 49.1 (1.7) | 50.3 (3.1) |
| kdd.ts | 64 | 43584 | **327.6** | † | 1314.5 (158.2) | 1405.4 (239.5) | 1663.2 (512.4) |
| steel_BIC | 28 | 93027 | † | 1270.9 (1218.9) | **98.0** (49.2) | 99.2 (50.1) | 130.0 (81.2) |
| kdd.test | 64 | 152873 | 1521.7 | † | **1475.3** (120.6) | 1515.9 (128.5) | 1492.4 (109.5) |
| mushroom_BDe | 23 | 438186 | † | 176.4 (56.0) | 135.4 (33.7) | 137.0 (35.0) | **133.7** (31.9) |
| bnetflix.ts | 100 | 446406 | † | **629.0** (431.4) | 1065.1 (878.4) | 1111.4 (931.0) | 1132.4 (936.3) |
| plants.test | 69 | 520148 | † | † | **18981.9** (17224.0) | 30791.2 (29073.0) | † |
| jester.ts | 100 | 531961 | † | † | **10166.0** (9697.9) | 14915.9 (14470.1) | 23877.6 (23325.7) |
| accidents.ts | 111 | 568160 | **1274.0** | † | 2238.7 (904.5) | 2260.3 (986.1) | 2221.1 (904.8) |
| plants.valid | 69 | 684141 | † | † | **12347.6** (8509.7) | 19853.1 (15963.1) | † |
| jester.test | 100 | 770950 | † | † | **17637.8** (16979.2) | 21284.0 (20661.9) | † |
| bnetflix.test | 100 | 1103968 | † | **3525.2** (3283.8) | 8197.7 (7975.6) | 8057.3 (7841.4) | 7915.0 (7686.3) |
| bnetflix.valid | 100 | 1325818 | † | **1456.6** (1097.0) | 9282.0 (8950.3) | 10220.5 (9898.4) | 9619.7 (9257.4) |
| accidents.test | 111 | 1425966 | 4975.6 | † | **3661.7** (641.5) | 4170.1 (1213.6) | 3805.2 (687.6) |

Table 2: Comparison of ELSA against GOBNILP and CPBayes. Time limit for instances above the line is 1h, for the rest 10h. Instances are sorted by increasing total domain size. For variants of CPBayes we report in parentheses time spent in search, after preprocessing finishes. † indicates a timeout.

different settings depending on problem size $|V|$: $l_{min} = 20, l_{max} = 26, r_{min} = 50, r_{max} = 500$ if $|V| \leq 64$, else $l_{min} = 20, l_{max} = 20, r_{min} = 15, r_{max} = 30$.

### 6.2 Evaluation

In Table 2 we present the runtime to solve each instance to optimality with GOBNILP, CPBayes, and ELSA with default settings, without the GAC algorithm and without sorting the cluster pool (leaving clusters in chronological order, rather than the heuristic ordering presented in Section 4). For the instances with $\|V\| \leq 64$ (resp. $> 64$), we had a time limit of 1 hour (resp. 10 hours). We exclude instances that were solved within the time limit by GOBNILP and have a search time of less than 10 seconds for CPBayes and all variants of ELSA. We also exclude 8 instances that were not solved to optimality by any method. This leaves us 17 instances to analyse here out of 69 total. More details are given in the supplemental material, available from the authors' web pages.

**Comparison to GOBNILP.** CPBayes was already proven to be competitive to GOBNILP [van Beek and Hoffmann, 2015]. Our results in Table 2 confirm this while showing that neither is clearly better. When it comes to our solver ELSA, for all the variants, all instances solved within the time limit by GOBNILP are solved, unlike CPBayes. On top of that, ELSA solves 9 more instances optimally.

**Comparison to CPBayes.** We have made some low-level performance improvements in preprocessing of CPBayes, so for a more fair comparison, we should compare only the search time, shown in parentheses. ELSA takes several orders of magnitude less search time to optimally solve most instances, the only exception being the `bnetflix` instances. ELSA also proved optimality for 8 more instances within the time limit.

**Gain from GAC.** The overhead of GAC pays off as the instances get larger. While we do not see either a clear improvement nor a downgrade for the smaller instances, search time for ELSA improves by up to $47\%$ for larger instances compared to ELSA \ GAC.

**Gain from Cluster Ordering.** We see that the ordering heuristic improves the bounds computed by our greedy dual LP algorithm significantly. Compared to not ordering the clusters, we see improved runtime throughout and 3 more instances solved to optimality.

## 7 Conclusion

We have presented a new set of inference techniques for BNSL using constraint programming, centered around the expression of the acyclicity constraint. These new techniques exploit and improve on previous work on linear relaxations of the acyclicity constraint and the associated propagator. The resulting solver explores a different trade-off on the axis of strength of inference versus speed, with GOBNILP on one extreme and CPBayes on the other. We showed experimentally that the trade-off we achieve is a better fit than either extreme, as our solver ELSA outperforms both GOBNILP and CPBayes. The major obstacle towards better scalability to larger instances is the fact that domain sizes grow exponentially with the number of variables. This is to some degree unavoidable, so our future work will focus on exploiting the structure of these domains to improve performance.

## Acknowledgements

We thank the GenoToul (Toulouse, France) Bioinformatics platform for its support. This work has been partly funded by the "Agence nationale de la Recherche" (ANR-16-CE40-0028 Demograph project and ANR-19-PIA3-0004 ANTIDIL chair of Thomas Schiex).

## References

[Allen-Zhu and Orecchia, 2015] Zeyuan Allen-Zhu and Lorenzo Orecchia. Nearly-linear time positive LP solver

with faster convergence rate. In *Proc. of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC'15, page 229–236, New York, NY, USA, 2015.

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

[Berg *et al.*, 2014] 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.

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

[Chickering, 1995] 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.

[Cooper *et al.*, 2010] Martin C Cooper, Simon de Givry, Martı Sánchez, Thomas Schiex, Matthias Zytnicki, and Tomas Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8):449–478, 2010.

[Cussens *et al.*, 2017] 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.

[de Campos and Ji, 2010] Cassio Polpo de Campos and Qiang Ji. Properties of bayesian dirichlet scores to learn bayesian network structures. In *Proc. of AAAI-00*, Atlanta, Georgia, USA, 2010.

[de Campos *et al.*, 2018] 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.

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

[Heckerman *et al.*, 1995] 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.

[Junker, 2004] Ulrich Junker. Preferred explanations and relaxations for over-constrained problems. In *Proc. of AAAI-04*, pages 167–172, San Jose, California, USA, 2004.

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

[Lee and van Beek, 2017] 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.

[Marques-Silva and Mencía, 2020] João Marques-Silva and Carlos Mencía. Reasoning about inconsistent formulas. In Christian Bessiere, editor, *Proc. of IJCAI-2020*, pages 4899–4906, 2020.

[Morgado *et al.*, 2013] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints An Int. J.*, 18(4):478–534, 2013.

[Papadimitriou and Steiglitz, 1998] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[Rossi *et al.*, 2006] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[Scanagatta *et al.*, 2015] 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.

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

[Silander and Myllymäki, 2006] 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.

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

[Walsh, 2000] Toby Walsh. SAT vs CSP. In *Proc. of the Sixth International Conference on Principles and Practice of Constraint Programming*, pages 441–456, 2000.

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