Advances in Bayesian Network Learning using Integer Programming

Mark Barlett
Dept of Computer Science &
York Centre for Complex Systems Analysis
University of York, UK
mark.bartlett@york.ac.uk

James Cussens
Dept of Computer Science &
York Centre for Complex Systems Analysis
University of York, UK
james.cussens@york.ac.uk

Abstract

We consider the problem of learning Bayesian networks (BNs) from complete discrete data. This problem of discrete optimisation is formulated as an integer program (IP). We describe the various steps we have taken to allow efficient solving of this IP. These are (i) efficient search for cutting planes, (ii) a fast greedy algorithm to find high-scoring (perhaps not optimal) BNs and (iii) tightening the linear relaxation of the IP. After relating this BN learning problem to set covering and the multidimensional 0-1 knapsack problem, we present our empirical results. These show improvements, sometimes dramatic, over earlier results.

1 Introduction

Bayesian networks (BNs) use a directed acyclic graph (DAG) to represent conditional independence relations between random variables. Each BN defines a joint probability distribution over joint instantiations of the random variables. BNs are a very popular and effective representation for probabilistic knowledge and there is thus considerable interest in 'learning' them from data (and perhaps also prior knowledge).

In this paper we present our approach to the 'search-and-score' method of BN learning. As a score for any candidate BN we choose the log marginal likelihood with Dirichlet priors over the parameters. We choose these Dirichlet priors so that our score is the well-known BDeu score. Throughout we set the *effective sample size* for the BDeu score to be 1. The learning problem is a discrete optimisation problem: find the BN structure (DAG) with the highest BDeu score.

In Section 2 we present our integer programming (IP) encoding of the problem. Section 3 shows how we use a 'branch-and-cut' approach to solve the IP problem. Section 4 describes how we find 'cutting planes'. Section 5 describes our fast greedy algorithm for finding good but typically sub-optimal BNs. Section 6 presents our investigation into the convex hull of DAGs with 3 or 4 nodes, the facets of which turn out to be useful for learning DAGs with any number of nodes. Section 7 is more discursive in nature: pointing out interesting connections to the set covering and multidimensional knapsack problems. The central contribution of the paper—faster solving for exact BN learning—is established in Section 8 where we present our empirical results. We finish, as is the custom, with conclusions and pointers to future work.

This paper assumes familiarity with Bayesian networks and basic knowledge concerning BN learning. See [14] for a comprehensive treatment of both these topics and much more besides. The rest of the paper assumes the reader knows what an integer program is: it is a discrete optimisation problem where the objective function is a linear function of the (discrete) problem variables and where the set of feasible solutions is defined by a finite set of linear inequalities over these variables. The *linear relaxation* of the IP is obtained by removing the constraint that the problem variables take only integer values. For further reading on integer programming, consult e.g. [16]. We refer to the variables in the learned BN structure as 'nodes' rather than 'variables' to more clearly distinguish them from IP variables.

2 Integer program encoding

We encode our BN structure learning problems as an integer program (IP) so that the log marginal likelihood of any DAG is a linear function of the IP variables. This requires creating binary 'family' variables $I(W \to v)$ for each node v and candidate parent set W, where $I(W \to v) = 1$ iff W is the parent set for v. This encoding has been used in previous work [5, 13, 6].

The objective function then becomes

$$\sum_{v,W} c(v,W) I(W \to v) \tag{1}$$

where $c(v,W)$ is the 'local' BDeu score for W being the parents of v. This score is computed from the data prior to the search for an optimal BN. Evidently the number of such scores/variables increases exponentially with p, the number of nodes in the BN. However if $W \subset W'$ and $c(v,W) > c(v,W')$ then, in the absence of any user constraints on the DAG, it is obvious that W cannot be the parent set for v in any optimal BN and thus there is no need to create the unnecessary variable $I(W' \to v)$. Moreover, using a pruning technique devised by de Campos and Ji [9] it is possible to prune the search for necessary variables so that local scores for many unnecessary variables need never be computed. Nonetheless, unless p is small (e.g. ≤ 12), even with this pruning we typically have to make some restriction on the number of candidate parent sets for any node. See Section 8 for more on this.

The $I(W \to v)$ variables encode DAGs, but in many cases one is more interested in *Markov equivalence classes* of DAGs [14] which group together DAGs representing the same conditional independence relations. This has motivated the *standard imset* and later *characteristic imset* representation [12]. A characteristic imset c is a zero-one vector indexed by subsets C of BN nodes such that $\mathsf{c}(C) = 1$ iff there is a node $v \in C$ such that all nodes in $C \setminus \{v\}$ are parents of v. Importantly two DAGs have the same characteristic imset representation iff they are Markov equivalent. This representation lends itself to an IP approach to BN structure learning. Existing [12] and ongoing work shows encouraging results. Connecting the characteristic imset representation to our 'family' representation we have that for any DAG and any subset of nodes C:

$$\mathsf{c}(C) = \sum_{v \in C} \sum_{W : C \setminus \{v\} \subseteq W} I(W \to v) \tag{2}$$

Returning to our own encoding, to ensure that IP solutions correspond to DAGs two families of linear constraints are used. Let V be the set of BN nodes. The convexity constraints

$$\forall v \in V : \sum_{W} I(W \to v) = 1 \tag{3}$$

simply ensure that any variable has exactly one parent set. The 'cluster' constraints

$$\forall C \subseteq V : \sum_{v \in C} \sum_{W : W \cap C = \emptyset} I(W \to v) \geq 1 \tag{4}$$

introduced by Jaakkola *et al* [13], ensure that the graph has no cycles. For each cluster C the associated constraint declares that at least one $v \in C$ has no parents in C. Since there are exponentially many cluster constraints these are added as 'cutting planes' in the course of solving.

3 Branch-and-cut algorithm

Let n be the number of $I(W \to v)$ variables so that any instantiation of these variables can be viewed as a point in $\{0,1\}^n \subset [0,1]^n \subset \mathbb{R}^n$. Let $\mathcal{P}_{cluster}$ be the polytope of all points in $\mathbb{R}^n$ which satisfy the bounds on the variables, convexity constraints (3) and cluster constraints (4). If a point in $\mathcal{P}_{cluster}$ is entirely integer-valued (i.e. is in $\{0,1\}^n$) then it corresponds to a DAG, but $\mathcal{P}_{cluster}$ also contains infinitely many non-DAG points. Due to the p convexity equations (3) this polytope is of dimension $n - p$. Note that all points in $\mathcal{P}_{cluster}$ have an objective value (1) not just those with integer values.

Jaakkola *et al* [13] consider the problem of finding a point in $\mathcal{P}_{cluster}$ with maximal objective value. They consider the dual problem and iteratively add dual variables corresponding to cluster constraints. This process is continued until the problem is solved or there are too many dual variables. In the latter case branch-and-bound is used to continue the solving process. A 'decoding' approach is used to extract DAGs from values of the dual variables.

We instead take a fairly standard 'branch-and-cut' approach to the problem. The essentials of branch-and-cut are as follows, where *LP solution* is short for 'solution of the current linear relaxation':

```
1. Let x* be the LP solution.
2. If there are valid inequalities
        not satisfied by x*
        add them and go to 1.
   Else if x* is integer-valued then
        the current problem is solved
   Else branch on a variable with
        non-integer value in x*
        to create two new sub-IPs.
```

The added valid inequalities are called 'cutting planes' since they 'cut off' the LP solution x^*. This process of cutting and perhaps branching is performed on all nodes in the search tree. If the objective value of the LP solution of a sub-IP is worse than the current best integer solution then the tree is pruned at that point. Note that the term 'branch-and-cut' is a little misleading since there is typically much cutting in the root node before any branching is done.

Where Jaakkola *et al* added dual variables to the dual problem we add cluster constraints as cutting planes (*cluster cuts*) to the original 'primal' problem. We do

a complete search for cluster cuts so that if none can be found we have a guarantee that the LP solution x^* satisfies all cluster constraints (4). Note that there is no need to find all cluster cuts to have this guarantee; in practice only a small fraction of the exponentially many cluster cuts need be found.

If no cluster cuts can be found (and the problem is not solved) we search for three other sorts of cutting planes: Gomory, strong Chvátal-Gomory (CG) and zero-half cuts. If none of these cuts can be found we branch on a fractional $I(W \to v)$ variable to create two new sub-IPs as described above. Solving the problem returns a guaranteed optimal BN. Since we are working with the primal representation there is no decoding required to extract the BN from the optimal values of the $I(W \to v)$ variables: we just return the p variables which are set to 1.

The algorithm is implemented using the the SCIP [1] callable library (`scip.zib.de`). Implementing a basic branch-and-cut algorithm with SCIP amounts to little more than setting SCIP parameter flags appropriately. We used SCIP's built-in functions to search for Gomory, strong CG and zero-half cuts. We also used SCIP's default approach to branching. See [1] and the SCIP documentation for details. However the search for cluster cuts we implemented ourselves, plugging it into the branch-and-cut algorithm as a SCIP *constraint handler*. This search is described in Section 4. Our greedy algorithm for finding good BNs is implemented as a SCIP *primal heuristic* and is described in Section 5.

4 Finding cluster cuts

Our system GOBNILP 1.3 uses a sub-IP to search for cluster cuts. The approach is essentially the same as that presented by Cussens [6]. Nonetheless we describe it here for completeness and because our current implementation has a simple but very effective optimisation that was missing from the earlier one.

To understand the sub-IP first note that, due to the convexity constraints, the constraint (4) for cluster C can be reformulated as a knapsack constraint (5):

$$\sum_{v \in C} \sum_{W : W \cap C \neq \emptyset} I(W \to v) \leq |C| - 1 \qquad (5)$$

In the sub-IP the cluster is represented by $|V| = p$ binary variables $I(v \in C)$ each with objective coefficient of -1. Now let $x^*(W \to v)$ be the value of $I(W \to v)$ in the LP solution. For each $x^*(W \to v) > 0$ create a sub-IP binary variable $J(W \to v)$ with an objective coefficient of $x^*(W \to v)$. Abbreviating $\sum_{v \in V} I(v \in C)$ to $|C|$ the sub-IP is defined to be:

$$\text{Maximise: } -|C| + \sum x^*(W \to v) J(W \to v) \qquad (6)$$

Subject to, for each $J(W \to v)$:

$$J(W \to v) \Rightarrow I(v \in C) \qquad (7)$$

$$J(W \to v) \Rightarrow \bigvee_{w \in W} I(w \in C) \qquad (8)$$

The constraints (7) and (8) are posted as SCIP `logicor` clausal constraints and the sub-IP search is set to depth-first, branching only on $I(v \in C)$ variables. Note that `logicor` constraints are a special type of linear constraint. It is a requirement that the objective value is greater than -1. For efficiency this is implemented directly as a limit on the objective value (using SCIP's `SCIPsetObjlimit` function) rather than as a linear constraint. A final constraint dictates that $|C| \geq 2$.

In any feasible solution we have that

$$-|C| + \sum x^*(W \to v) J(W \to v) > -1 \qquad (9)$$

Due to the constraints (7) and (8), for $J(W \to v)$ to be non-zero v must be in the cluster C and at least one element of W must also be in C. So (9) can be rewritten as:

$$-|C| + \sum_{v \in C} \sum_{W : W \cap C \neq \emptyset} x^*(W \to v) J(W \to v) > -1 \qquad (10)$$

It follows that the cluster C associated with a feasible solution of the sub-IP has a cluster constraint which is violated by the current LP solution x^*. Each feasible solution of the IP thus corresponds to a valid cutting plane. The sub-IP is always solved to optimality, (collecting any sub-optimal feasible solutions along the way) so it follows that if the current LP solution violates any cluster constraint then at least one will be found.

In [6] a sub-IP was also used to find cutting planes. However, there "a binary variable $J(W \to v)$ is created for **each** family variable $I(W \to v)$ in the main IP." (our emphasis). In the current approach $J(W \to v)$ variables are created *only for* $I(W \to v)$ *variables which are not zero in the LP solution*. This greatly reduces the number of variables in the sub-IP (and thus speeds up sub-IP solving) since typically most main IP variables *are* set to zero in any LP solution.

5 Sink finding algorithm

Finding a good primal solution (i.e. a BN) early on in the search is worthwhile even if it turns out to be

$I(W_{1,1} \to 1)$	$I(W_{1,2} \to 1)$	...	$I(W_{1,k_1} \to 1)$
$I(W_{2,1} \to 2)$	$I(W_{2,2} \to 2)$	...	$I(W_{2,k_2} \to 2)$
$I(W_{3,1} \to 3)$	$I(W_{3,2} \to 3)$	...	$I(W_{3,k_3} \to 3)$
...	...	...	...
$I(W_{p,1} \to p)$	$I(W_{p,2} \to p)$	...	$I(W_{p,k_p} \to p)$

Table 1: Example initial state of the sink-finding heuristic for $|V| = p$. Rows need not be of the same length.

~~$I(W_{1,1} \to 1)$~~	$I(W_{1,2} \to 1)$	...	$I(W_{1,k_1} \to 1)$
$\mathbf{I(W_{2,1} \to 2)}$	~~$I(W_{2,2} \to 2)$~~	~~...~~	~~$I(W_{2,k_2} \to 2)$~~
$I(W_{3,1} \to 3)$	~~$I(W_{3,2} \to 3)$~~	...	$I(W_{3,k_3} \to 3)$
...	...	...	...
~~$I(W_{p,1} \to p)$~~	~~$I(W_{p,2} \to p)$~~	...	$I(W_{p,k_p} \to p)$

Table 2: Example intermediate state of the sink-finding heuristic.

suboptimal. High scoring solutions allow more effective branch-and-bound and may help in the root node due to *root reduced cost strengthening* [1, §7.7]. If a problem cannot be solved to optimality having a good, albeit probably suboptimal, solution is even more important.

We have a 'sink-finding' algorithm which proposes primal solutions using the current LP solution. The algorithm is based on two ideas: (i) that there might be good primal solutions 'near' the current LP solution and (ii) that an optimal BN is easily found if we can correctly guess an optimal total ordering of BN nodes. The first idea is common to all *rounding heuristics*. SCIP has 6 built-in rounding heuristics and we allow SCIP to run those that are fast and they do sometimes find high scoring BNs. The second idea has been exploited in dynamic programming approaches to exact BN learning [15].

To understand how the algorithm works consider Table 1. The table has a row for each node and there are $p = |V|$ rows. The $I(W \to v)$ variables for each node are ordered according to their objective coefficient $c(v, W)$, so that, for example, $W_{1,1}$ is the 'best' parent set for node 1 and W_{1,k_1} the worst. The objective coefficients $c(v, W)$ play no role in the sink-finding algorithm other than to determine this ordering. Since the number of available parent sets may differ between nodes the rows will typically not be of the same length.

On the first iteration of the sink finding algorithm, for each child variable the 'cost' of selecting its best-scoring parent set is computed. This cost is $1 - x^*(W_{v,1} \to v)$, where $x^*(W_{v,1} \to v)$ is the value of $I(W_{v,1} \to v)$ in the LP solution.

Denote the child variable chosen as v_p. In order to ensure that the algorithm generates an acyclic graph a total order is also generated. This is achieved by setting $I(W \to v)$ to 0 if $v_p \in W$. As a result v_p will be a sink node of the BN which the algorithm will eventually construct.

Suppose that it turned out that $v_p = 2$ and that node 2 was a member of parent sets $W_{1,1}$, $W_{3,2}$, $W_{p,1}$ and $W_{p,2}$. The state of the algorithm at this point is illustrated in Table 2. In the next iteration $I(W_{3,1} \to 3)$ remains available as the 'best' parent set for node 3 but for node 1 the best parent set now is $I(W_{1,2} \to 1)$. In the second and subsequent iterations the algorithm continues to choose the best available parent set for some node according to which choice of node has minimal cost. However in these non-initial iterations cost is computed as $(\sum_{W \in \text{ok}(v)} x^*(W \to v)) - x^*(W_{v,\text{best}} \to v)$, where $I(W_{v,\text{best}} \to v)$ is the best scoring remaining choice for v and $\text{ok}(v)$ is the set of remaining parent set choices for v. After each such selection, parent set choices for remaining nodes are updated just like for the first iteration. Note that the node selected at any iteration will be a sink node (in the final DAG) for the subset of nodes available at that point.

There is an added complication if some $I(W \to v)$ are already fixed to 1 when the sink-finding algorithm is run. This can happen either due to user constraints or due to branching on $I(W \to v)$. Trying to rule out such a variable leads the algorithm to abort.

Due to its greedy nature the sink finding algorithm is very fast and so we can afford to run it after solving *every* LP relaxation. For example, in one of our bigger examples, `Diabetes_100`, the sink-finding algorithm was called 9425 times taking only 30s in total. Note that each new batch of cutting planes produces a new LP and thus a new LP solution. In this way we use the LP to help us move around to search for high-scoring BNs.

6 Tightening the LP relaxation

Given a collection of n $I(W \to v)$ variables, each feasible DAG corresponds to a (binary) vector in $\mathbb{R}^n$. Consider now $\mathcal{P}$, the *convex hull* of these points and recall $\mathcal{P}_{cluster}$, the polytope defined in Section 3 containing all points satisfying the variable bounds, the convexity constraints (3) and all cluster constraints (4). As Jaakkola *et al* [13] note, $\mathcal{P}$ is strictly contained within $\mathcal{P}_{cluster}$, except in certain special cases. GOBNILP uses SCIP to add Gomory, strong CG and zero-half cutting planes in addition to cluster cuts. This produces a linear relaxation which is tighter than $\mathcal{P}_{cluster}$ and, as the results presented in Section 8 show, typically im-

proves overall performance, although strong CG cuts are generally not helpful. Denote the polytope defined by adding these 'extra' cuts by $\mathcal{P}'_{cluster}$. Since the searches for Gomory, strong CG and zero-half cuts are not complete $\mathcal{P}'_{cluster}$ is specific to the problem instance and SCIP parameter settings.

In many cases it is not possible to separate a fractional LP solution x^* even with these extra cuts, so we have $x^* \notin \mathcal{P}$ but $x^* \in \mathcal{P}'_{cluster}$. This raises the question of which inequalities are needed to define $\mathcal{P}$. We have approached this issue by carrying out empirical investigations into $\mathcal{P}$ when there are 3 or 4 nodes.

For 3 nodes $\{1,2,3\}$ there are only 25 DAGs. We eliminated the three $I(\emptyset \rightarrow v)$ variables using the equations (3) and encoded each DAG using the remaining nine $I(W \rightarrow v)$ variables (3 remaining choices of parent set for each node). The lrs algorithm [2] (`http://cgm.cs.mcgill.ca/~avis/C/lrs.html`) was used to find the facets of the convex hull of these 25 vertices in $\mathbb{R}^9$.

Denoting this convex hull as $\mathcal{P}_3$, we find that it has 17 facets. These are: 9 lower bounds on the variables, 3 inequalities corresponding to the original convexity constraints, cluster constraints for the 4 clusters $\{1,2\}$, $\{1,3\}$, $\{2,3\}$, and $\{1,2,3\}$ and one additional constraint:

$$I(\{2,3\} \rightarrow 1) + I(\{1,3\} \rightarrow 2) + I(\{1,2\} \rightarrow 3) \leq 1 \tag{11}$$

Consider the point y^* in $\mathbb{R}^9$ specified by setting $I(\{2,3\} \rightarrow 1) = \frac{1}{2}$, $I(\{1,3\} \rightarrow 2) = \frac{1}{2}$, $I(\{1,2\} \rightarrow 3) = \frac{1}{2}$ and all other variables to zero. It is not difficult to see that this point is on the surface of $\mathcal{P}_{cluster}$ (lying on the hyperplanes defined by the cluster constraints for $\{1,2\}$, $\{1,3\}$ and $\{2,3\}$). However it does not satisfy (11) and so is outside of $\mathcal{P}_3$. Note that there are nine 3-node DAGs where one node has two parents. All of these DAGs lie on the hyperplane defined by (11). It is easy to show that these 9 DAGs (= points in $\mathbb{R}^9$) are affinely independent which establishes that (11) is indeed a facet.

The point y^* was also discussed by Jaakkola *et al* [13]. They considered the *acyclic subgraph polytope* $\mathcal{P}_{dag}$ which is the convex hull of DAGs which results from using binary variables to represent *edges* rather than parent set choices. This polytope has been extensively studied in discrete mathematics [11] and many (but not all) classes of facets are known for it [10]. The parent set representation can be projected onto the edge representation (so that the former is an *extended formulation* of the latter in the language of mathematical programming). As Jaakkola *et al* observe the projection of y^* is a member of $\mathcal{P}_{dag}$.

The inequality (11) can be generalised to give a class of valid *set packing* inequalities:

$$\forall C \subseteq V : \sum_{v \in C} \sum_{W : C \setminus \{v\} \subseteq W} I(W \rightarrow v) \leq 1 \tag{12}$$

We have found that adding all non-trivial inequalities of this sort for $|C| \leq 4$ speeds up solving considerably (see Section 8). This is because the LP relaxation is tighter. Since there are not too many such inequalities they are added directly to the IP rather than being added as cutting planes. Note that making the connection to characteristic imsets with (2) implies these set packing constraints.

We have not found all facets of $\mathcal{P}_4$, which is a polytope in $\mathbb{R}^{28}$ with 543 vertices (for the 543 4-node DAGs). We terminated lrs after a week's computation, by which time it had found 64 facets. We detected 10 different types of facets which we have labelled 4A to 4J. We will provide a full description of these facet classes in a forthcoming technical report. Here we just give a brief overview.

Consider, as an example, 4B-type facets. They are specified as follows:

$$\begin{aligned} &\sum_{v_4 \in W \wedge \{v_2,v_3\} \cap W \neq \emptyset} I(W \rightarrow v_1) \\ &+ \sum_{v_3 \in W \vee \{v_1,v_4\} \subseteq W} I(W \rightarrow v_2) \\ &+ \sum_{v_2 \in W \vee \{v_1,v_4\} \subseteq W} I(W \rightarrow v_3) \\ &+ \sum_{v_1 \in W \wedge \{v_2,v_3\} \cap W \neq \emptyset} I(W \rightarrow v_4) \quad \leq \quad 2 \end{aligned} \tag{13}$$

Consider the point $z^* \in \mathbb{R}^{28}$ where all variables take zero value except: $I(\{3,4\} \rightarrow 1) = \frac{1}{2}$, $I(\{1,3\} \rightarrow 2) = \frac{1}{2}$, $I(\{1,4\} \rightarrow 2) = \frac{1}{2}$, $I(\{2,4\} \rightarrow 3) = \frac{1}{2}$, $I(\{1,2\} \rightarrow 4) = \frac{1}{2}$. It is easy to check that z^* satisfies all cluster constraints and any constraint of type (12). However, setting $v_i = i$ in (13) we have that the left-hand side is $2\frac{1}{2}$ and so (13) separates (i.e. cuts off) z^*. It follows that adding 4B-type linear inequalities results in a strictly tighter linear relaxation.

We have implemented 6 distinct cutting plane algorithms to search for inequalities of types 4B, 4C, 4E, 4F, 4G and 4H. We call cuts of this sort *convex4* cuts. Type 4A cuts are cluster cuts, and cuts of type 4D, 4I and 4J have not appeared useful in preliminary experiments. We have also experimented with adding a limited number of convex4 cuts directly to the IP rather than finding them 'on the fly' as cutting planes.

In practice we have found LP solutions which violate these constraints but which none of our other

cutting planes can separate (the example z^* was extracted from one such LP solution). Cuts of type 4B appear to be particularly useful. Preliminary experiments indicate that using convex4 cuts is typically but not always beneficial. We have yet to do a controlled evaluation, but using convex4 cuts with a different version of SCIP (SCIP 3.0) and a slightly different machine from that used to present our main results, we do have some partial preliminary results. Using convex4 cuts, problem instances `alarm_3_10000`, `carpo_3_100`, `carpo_3_1000`, `carpo_3_10000` and `Diabetes_2_100` took 54s, 612s, 92s, 660s and 1393s to solve, respectively. All of these times are better than using our properly evaluated system GOBNILP 1.3 (see Section 8). The improvement is particular dramatic for `alarm_3_10000` which takes 298s using GOBNILP 1.3 and took 12872s using the system presented by Cussens [6]. On the other hand, problems `hailfinder_3_1` and `Pigs_2_1000` took 139s and 2809s respectively which is slower than GOBNILP 1.3.

7 Set covering and knapsack representations

If the convexity constraints (3) are all relaxed to set covering constraints: $\forall v : \sum_W I(W \rightarrow v) \geq 1$ then the BN learning problem becomes a pure set covering problem—albeit one with exponentially many constraints—since all the cluster constraints (4) are already set covering constraints. It is not difficult to show that any optimal solution to the set covering relaxation of our BN learning problem is also an optimal solution to the original problem. This opens up the possibility of applying known results concerning the set covering polytope to the BN learning problem. In particular, Balas and Ng [3] provide conditions for set covering inequalities to be facets and also show that for an inequality with integer coefficients and right-hand side of 1 to be a facet it must be one of the set covering inequalities defining the IP. This is a useful result for BN learning. It shows there is no point looking for 'extra' set covering inequalities in the hope they might be facets. Relaxed convexity constraints and cluster constraints are the only set covering inequalities that can be facets.

As Balas and Ng [3] note, Chvátal's procedure [4] can be used to generate the convex hull of integer points satisfying an IP after a finite number of applications. They provide a specialised version of this procedure to find a class of valid inequalities of the form $\alpha^S x \geq 2$ for the set covering polytope. These inequalities dominate all other valid inequalities of the form $\beta x \geq 2$. Each such inequality is defined by taking a set S of set covering inequalities, and combining them to get a new inequality (details omitted for space, see [3]).

We can use the procedure to get new inequalities for the BN learning problem by combining cluster constraints. For example combining the constraints for $C = \{a, b\}, C = \{a, c\}, C = \{a, d\}$ produces: $\sum_{W:W\cap\{b,c,d\}=\emptyset} 2I(W \rightarrow a) + \sum_{W:0<|W\cap\{b,c,d\}|<3} I(W \rightarrow a) + \sum_{a\notin W} I(W \rightarrow b) + \sum_{a\notin W} I(W \rightarrow c) + \sum_{a\notin W} I(W \rightarrow d) \geq 2$.

Our BN learning problem can also be reformulated as a multi-dimensional 0-1 knapsack problem. Due to the convexity constraints (4) we can eliminate each $I(\emptyset \rightarrow v)$ variable by replacing it with the linear expression $1 - \sum_{W\neq\emptyset} I(W \rightarrow v)$. Due to pre-pruning, the objective value of $I(\emptyset \rightarrow v)$ will be lower than that for all other $I(W \rightarrow v)$ variables and so once the $I(\emptyset \rightarrow v)$ variables have been eliminated the remaining variables will all have positive objective coefficient. Eliminating $I(\emptyset \rightarrow v)$ variables from the cluster constraints produces the knapsack constraints (5) previously mentioned.

8 Results

The system GOBNILP 1.3 described in the previous sections was implemented using C, with SCIP 2.1.1 used as the constraint solver. The underlying LP solver was CPLEX 12.5. Both SCIP and CPLEX are available for free under academic licences. All experiments were performed using a single core of a 64-bit Linux machine with a 2.80 GHz 4 core processor and 7.7 GB of RAM. A time out limit of 2 hours was imposed across all experiments after which runs were aborted. Our results can be reproduced by going to `http://www.cs.york.ac.uk/aig/sw/gobnilp`.

Experiments were performed on data taken from a variety of Bayesian networks, with different numbers of observations, N, and with different limits, m, on the maximum number of nodes considered as the parent set of each node. The problem sets used are shown in Table 3. Several of these problem sets were used in [6], with additional larger networks and parent set sizes being added to assess performance on harder problems. Local BDeu scores were computed for all experiments external to the systems tested and the times taken to compute and filter these are not included in the presented results. Score computation times ranged from 1 second to 5497 seconds in the longest case, `diabetes` with $N = 10000$.

The primary experiment in this paper is to assess how long GOBNILP 1.3 takes to find the BN with the highest score, and rule out the possibility of finding a BN with a higher score for each dataset. In particular, we compare how the system with all features introduced

in previous sections compared to the earlier IP-based BN learning system presented by [6] (henceforth referred to as Cussens 2011).

Additionally, we examine the behaviour of the systems in those situations in which they failed to find the highest scoring BN within the 2 hour time limit. Integer Programming can be used as an any-time learning algorithm, where a current best solution can be taken at any point during the search, though this may not turn out to be the eventual best solution. Specifically, our aim here is to examine the examples which reached the two hour solving time limit and determine how close to finding a provably best BN they are at that point. This gives an idea of how good that system would be for use as an any-time algorithm, and acts as an (imperfect) proxy for comparing how much longer the search process would take to reach completion.

The Cussens 2011 system is not publicly available. However, GOBNILP 1.0 is available and is closely based on Cussens 2011 with some inefficiencies taken out. Therefore, comparisons were performed against GOBNILP 1.0 using exactly the same machine and SCIP and CPLEX versions as those used to run GOBNILP 1.3. These results are shown in Table 3. As the results reported in [6] are performed on a broadly similar machine to that used for the current experiments, times taken directly from that paper are also shown in the table for illustrative purposes. Results are not shown in the table for some data sets, as [6] did not study and report them.

The results show that in the vast majority of cases, GOBNILP 1.3 outperforms GOBNILP 1.0, often being 2–3 times faster. GOBNILP 1.3 is also never slower than Cussens 2011 and for the larger examples is usually an order of magnitude quicker. For example, the `alarm 3 10000` data set takes over 3.5 hours to solve using Cussens 2011, but less than 5 minutes using GOBNILP 1.3. Some of the difference in run times between GOBNILP 1.3 and Cussens 2011 may be due to different machines being used, however the vastly improved performance on the larger examples undoubtedly reflects an overwhelming improvement.

In order to discover which aspects of GOBNILP 1.3 were leading to this improvement in performance, a second set of experiments was conducted in which the performance of the full system was compared to that resulting from removing parts of the system one at a time. Three features were identified as being suitable to remove while still leaving a system that would still result in the best BN being found, albeit potentially not as efficiently. These three features were

Set Packing Constraints (Section 6) As these (12) are logically implied by the basic problem, without them the IP for finding the BN is still correct.

Sink Primal Heuristic (Section 5) The algorithm for finding feasible solutions through sink finding is not necessary, but may improve the search process through tightening the lower bound on the best BN.

Value Propagator Explicitly determining which values must be fixed at zero or one at each node of the search tree is not necessary as this information will eventually be discovered through search. However, by performing this propagation as early as possible, a significant amount of search may be saved.

In addition, three cutting plane algorithms which are built into SCIP are used within GOBNILP 1.3. These three were chosen from those available in SCIP based on preliminary experiments to determine which potential cuts would be added reasonably often and reasonably quickly. Each of these was also turned off in turn in order to assess whether it was positively contributing to the improved performance.

Six modified versions of the GOBNILP 1.3 resulted from this; three which each had one feature turned off and three which each had one type of cutting plane turned off. Each of the data sets was run on each of these systems and the time taken to find the optimal solution recorded. The results of these experiments are shown in Table 3.

The results show the biggest change in solution time occurs when the set packing constraints are removed. In nearly every case, this leads to an increase in solution time. In fact the situation can be even more extreme than the table suggests; for the `pigs 1000` data set, the system without the set packing constraints was allowed to continue running beyond the time out limit and had still not finished after more than 30 hours, when the full system finished in about 30 minutes.

Furthermore, in cases in which the two hour time limit was reached, the gap between the upper and lower bounds at that point was much larger in the version without set packing constraints than that for the full system. Closer examination revealed that this was because the score of the best BN found so far was the same as in the full system, but significantly less progress had been made in reducing the upper (dual) bound.

It is not immediately clear from this table if using the sinks heuristic aids the system or not. There are a number of problems for which it decreases solution

Network	m	p	N	Families	GOBNILP 1.3	GOBNILP 1.0	Cussens 2011	Without Solver Feature			No Cuts of Type			New VP
								SPC	SPH	VP	G	SCG	ZH	
			100	244	1	*3*	1	1	1	1	1	1	1	1
hailfinder	3	56	1000	761	5	*14*	5	*11*	5	**4**	**4**	**4**	**4**	5
			10000	3708	100	*361*	*169*	102	**56**	102	*558*	**75**	**83**	98
			100	4106	18	*270*		*34*	18	*34*	18	**13**	**13**	19
hailfinder	4	56	1000	767	4	*14*		*10*	4	4	*5*	*6*	4	4
			10000	4330	68	*934*		62	*128*	*587*	*216*	71	71	70
			100	907	2	*6*	*4*	*3*	2	**1**	2	2	2	2
alarm	3	37	1000	1928	5	*14*	*15*	*12*	**4**	5	*7*	**4**	5	**4**
			10000	6473	289	*792*	*12872*	*479*	*394*	*397*	*739*	*1049*	*710*	298
			100	1293	2	*7*		*9*	2	*7*	*3*	2	2	2
alarm	4	37	1000	2097	7	*15*		*21*	**6**	*8*	**6**	**3**	**6**	7
			10000	8445	398	*839*		*1253*	*806*	*1421*	*633*	*1567*	*1052*	398
			100	5068	756	*887*	*15176*	742	**628**	716	690	**642**	740	**651**
carpo	3	60	1000	3827	106	*171*	*593*	*143*	*134*	115	*117*	104	110	99
			10000	16391	1311	**566**	*42275*	*2158*	*1574*	**1071**	*4350*	**1032**	*2057*	1286
			100	13185	[0%]	[*1*%]		6649	[0%]	[0%]	[0%]	7014	[0%]	[0%]
carpo	4	60	1000	4722	151	*406*		*252*	*168*	*188*	*240*	*208*	140	149
			10000	34540	[0%]	4065		[0%]	[0%]	[0%]	[0%]	[0%]	[0%]	[0%]
			100	4441	2982	[*31*%]		[*39*%]	3082	3040	3036	**1506**	3212	2745
diabetes	2	413	1000	21493	[17%]	[*23*%]		[*168*%]	[*199*%]	[16%]	[17%]	[**15**%]	[17%]	[18%]
			10000	262129	[44%]	[**17**%]		[*378*%]	[*380*%]	[44%]	[44%]	[44%]	[44%]	[44%]
			100	2692	89	[*0*%]		*5103*	87	**32**	88	85	89	**32**
pigs	2	441	1000	15847	1818	[7%]		[*8*%]	1788	1715	1714	1802	1822	1657
			10000	304219	[3%]	[*9*%]		[*13*%]	[*42*%]	[3%]	[3%]	[3%]	[3%]	[3%]

Table 3: Comparison of GOBNILP 1.3 with older systems and impact of various features. p is the number of variables in the data set. m is the limit on the number of parents of each variable. N is the number of observations in the data set. Families is the number of family variables in the data set after pruning. All times are given in seconds (rounded). "[—]" indicates that the solution had not been found after 2 hours — the value given is the gap, rounded to the nearest percent, between the score of the best found BN and the upper bound on the score of the best potential BN, as a percentage of the score of the best found BN. Entries in italics are at least 10% worse than GOBNILP 1.3, while those in bold are at least 10% better. Key: SPC – Set Packing Constraints, SPH – Sink Primal Heuristic, VP – Value Propagator, G – Gomory cuts, SCG – Strong CG cuts, ZH – Zero-half cuts.

time and a number for which it increases it. For the most difficult problems, it appears to have little impact on solving time. As with the version without set packing constraints, the system without the primal heuristic also resulted in much larger gaps between the bounds in cases where it reached the time out limit. For the version without the primal heuristic, the difference is due to the best BN found so far being significantly worse after the time had elapsed, while the upper bound on the best possible BN that could be found was virtually identical to the full system. This latter point suggests that, while the sinks heuristic was of questionable value when solving problems to completion, on larger problems for which the algorithm may run out of time or resources before successfully finding the provably best BN, the heuristic plays an import role in ensuring that the best BN found so far is of high score.

The evidence for the effectiveness of the propagation is also mixed. In some cases, the system performs faster without the propagation, though study of the log files reveals this is almost exclusively down to the time spent directly carrying out propagation. Having noted this, a new faster propagator was created and used to replace the existing one. The result of running the full system with this faster propagator was to achieve a minor improvement over both the full system and the system without a propagator, as shown in the final column of Table 3.

The usefulness of each of the cutting plane algorithms is much clearer. Without Gomory cuts, the system is often slower and frequently by a large amount. On the other hand, removing Strong CG cuts usually improves the system's performance with one notable exception. Though checking log files, it can be confirmed that little time is spent generating Strong CG cuts and hence the improvement without them is due to a better search strategy rather than just a saving on the time spent adding these cuts. Zero-half cuts appear to have less effect than the other two studied but reduce solving time noticeably on two fairly large problems.

9 Conclusions and future work

Our principal conclusion is that IP is an attractive framework for exact BN learning from complete discrete data. However, as our comparative experiments demonstrate, some care (and empirical investigation) is required to properly exploit its potential. We have shown considerable advances over the results presented by Cussens [6] who himself presented faster solving times on four problems than [13]. However, it would clearly be desirable to compare GOBNILP 1.3 against further exact BN learning systems, not necessarily IP-based. We intend to compare against the URLearning system [17] in the immediate future.

In this paper we have focused on efficiently finding BNs with maximal score subject to constraints on parent set size. This raises the question of whether it is worth the effort to do this if one's ultimate goal is to return a DAG with high structural accuracy. In the context of 'pedigree reconstruction', work by Cussens *et al* [8] answers this question in the positive. In that paper an exact learning approach led to high structural accuracy. However, in a pedigree (a 'family tree') no node can have more than two parents. In other applications where the 'true' structure may have nodes with many parents our current restriction on parent set size may lead to poor structural accuracy. This is a clear limitation which we intend to address by the IP technique of *delayed column generation* where IP variables (i.e. parent sets) are created during solving [7].

In practical applications one often has prior knowledge concerning the (likely) structure of the 'true' BN. Because our GOBNILP 1.3 system is an example of *declarative machine learning* it is very easy to allow the user to declare constraints on BN structure when these can be encoded as linear constraints. Although we have not exploited it in the experiments reported here, GOBNILP 1.3 allows the user to declare the absence/presence of (i) particular directed edges, (ii) particular adjacencies and (iii) particular immoralities. In addition upper and lower bounds on the number of edges and founder nodes are possible. It is also possible to rule out specific BNs with a linear constraint. This allows GOBNILP 1.3 to not only find the optimal BN but also the top k BNs in decreasing order of score.

Acknowledgements

Thanks to Milan Studený, Raymond Hemmecke and David Haws for useful discussions concerning the imset representation. We would like to thank three anonymous referees for their valuable comments. This work has been supported by the UK Medical Research Council (Project Grant G1002312). This work benefited from a visit by the second author to the Helsinki Institute for Information Technology supported by the Finnish Centre of Excellence in Computational Inference Research (COIN).

References

[1] Tobias Achterberg. *Constraint Integer Programming.* PhD thesis, TU Berlin, July 2007.

[2] David Avis. A revised implementation of the reverse search vertex enumeration algorithm. In Gil

Kalai and Günter M. Ziegler, editors, *Polytopes Combinatorics and Computation*, volume 29 of *DMV Seminar*, pages 177–198. Birkhuser Basel, 2000.

[3] Egon Balas and Shu Ming Ng. On the set covering polytope: 1. All the facets with coefficients in {0,1,2}. *Mathematical Programming*, 43:57–69, 1989.

[4] Vacláv Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4:305–337, 1973.

[5] James Cussens. Bayesian network learning by compiling to weighted MAX-SAT. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence (UAI 2008)*, pages 105–112, Helsinki, 2008. AUAI Press.

[6] James Cussens. Bayesian network learning with cutting planes. In Fabio G. Cozman and Avi Pfeffer, editors, *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI 2011)*, pages 153–160, Barcelona, 2011. AUAI Press.

[7] James Cussens. Column generation for exact BN learning: Work in progress. In *Proc. ECAI-2012 workshop on COmbining COnstraint solving with MIning and LEarning (CoCoMile 2012)*, 2012.

[8] James Cussens, Mark Bartlett, Elinor M. Jones, and Nuala A. Sheehan. Maximum likelihood pedigree reconstruction using integer linear programming. *Genetic Epidemiology*, 37(1):69–83, Janary 2013.

[9] Cassio de Campos and Qiang Ji. Properties of Bayesian Dirichlet scores to learn Bayesian network structures. In *AAAI-10*, pages 431–436, 2010.

[10] Michel X. Goemans and Leslie A. Hall. The strongest facets of the acyclic subgraph polytope are unknown. In *Integer Programming and Combinatorial Optimization*, volume 1084 of *Lectures Notes in Computer Science*, pages 415–429. Springer, 1996.

[11] Martin Grötschel, Michael Jünger, and Gerhard Reinelt. On the acyclic subgraph polytope. *Mathematical Programming*, 33(1):28–42, 1985.

[12] Raymond Hemmecke, Silvia Lindner, and Milan Studený. Characteristic imsets for learning Bayesian network structure. *International Journal of Approximate Reasoning*, 53:1336–1349, 2012.

[13] Tommi Jaakkola, David Sontag, Amir Globerson, and Marina Meila. Learning Bayesian network structure using LP relaxations. In *Proceedings of 13th International Conference on Artificial Intelligence and Statistics (AISTATS 2010)*, volume 9, pages 358–365, 2010. Journal of Machine Learning Research Workshop and Conference Proceedings.

[14] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[15] Tomi Silander and Petri Myllymäki. A simple approach for finding the globally optimal Bayesian network structure. In *Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence (UAI-06)*, pages 445–45, 2006.

[16] Laurence A. Wolsey. *Integer Programming*. John Wiley, 1998.

[17] Changhe Yuan and Brandon Malone. An improved admissible heuristic for learning optimal Bayesian networks. In *Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence (UAI-12)*, Catalina Island, CA, 2012.