Integer Linear Programming for the Bayesian Network Structure Learning Problem

Mark Bartlett, James Cussens

Department of Computer Science, University of York, York, UK

Abstract

Bayesian networks are a commonly used method of representing conditional probability relationships between a set of variables in the form of a directed acyclic graph (DAG). Determination of the DAG which best explains observed data is an NP-hard problem [1]. This problem can be stated as a constrained optimisation problem using Integer Linear Programming (ILP). This paper explores how the performance of ILP-based Bayesian network learning can be improved through ILP techniques and in particular through the addition of non-essential, implied constraints. There are exponentially many such constraints that can be added to the problem. This paper explores how these constraints may best be generated and added as needed. The results show that using these constraints in the best discovered configuration can lead to a significant improvement in performance and shows significant improvement in speed using a state-of-the-art Bayesian network structure learner.

Keywords: Bayesian networks, Integer Linear Programming, Constrained Optimisation, Cutting planes, Separation

1. Introduction

Bayesian networks (BNs) use a directed acyclic graph (DAG) to represent conditional probability relationships between a set of variables. Each node

Email addresses: mark.bartlett@york.ac.uk (Mark Bartlett), james.cussens@york.ac.uk (James Cussens)

URL: http://www.cs.york.ac.uk/~bartlett (Mark Bartlett), http://www.cs.york.ac.uk/~jc (James Cussens)

in the network corresponds to one of the variables. Edges show conditional dependencies between these variables such that the value of any variable is a probabilistic function of the values of the variables which are its parents in the DAG.

While one can analytically create a BN from expert knowledge, there is considerable interest in learning Bayesian networks in which the relationship between the variables is not known. In this setting, multiple joint observations of the variables are first taken and then a BN structure that best explains the correlations in the data is sought. This is known as the Bayesian network structure learning problem. For any reasonably sized problem. the number of possible structures is far too large to evaluate each individually. Therefore a more intelligent alternative is needed.

In this paper, we tackle the Bayesian network structure learning problem using the score-and-search approach. Each possible parent set of each variable is first given a score based on the correlations between these variables in the observed data. A search algorithm is then used to determine which combination of these parent sets yields the DAG with the optimal overall score. As this search is an NP-hard problem [1], an intelligent search strategy is needed in order to efficiently optimise the BN structure for large numbers of variables.

The search for the best BN can be viewed as a constrained optimisation problem; select the parent sets for variables with the highest combined score subject to the constraint that these form an encoding of a DAG. Specifically, there are two constraints that must be respected. First, there must be exactly one parent set chosen for each variable. Second, there must be no (directed) cycles in the graph. Furthermore, it is possible to write the score which is to be optimised and both of these constraints as linear functions of binary variables, which means that the problem of learning the best BN can be formulated as an Integer Linear Programming (ILP) problem [2, 3]. Formulating the problem in such a way means that highly optimised off-the-shelf ILP solvers can be used and that decades of research in ILP optimisation can be used to improve the speed of the search for the optimal BN.

Encoding the constraint that there is only one parent set for each node is straightforward. However, the constraint that there must be no cycles in the network is relatively complex to encode as a linear inequality and can either be enforced through the introduction of auxiliary variables and constraints [4] or through an exponential number of *cluster constraints* [2]. Previous work has revealed that these cluster constraints perform better in practice and so the current paper focuses on this encoding. As there are so many of these cluster constraints, we do not add them all initially, but rather add them to the problem as additional constraints as needed. That is to say, we first solve a relaxed version of the problem in which most of the acyclicity constraints are not present. We then identify some acyclicity constraints violated by this solution and add them to the problem before resolving. This process is called *separation*, as the added constraints separate the relaxed solution from the space of valid solutions, and the added constraints are known as *cuts* or *cutting planes* as they cut off a portion of the search space containing the relaxed solution. This process repeats until the solution found does not violate any additional acyclicity constraints. By so doing, we typically eliminate the need for most constraints which rule out cycles to ever be explicitly represented in the problem and so increase the solving speed of the problem and simultaneously reduce the memory needed.

In addition to the constraints necessary to define the problem, there are additional implied constraints that can also be added. Doing so may lead to an increase in performance through further constraining the search space or may prove detrimental by increasing the number of constraints that need to be generated and processed at each step.

The contribution of the current paper is to examine several extensions to the existing ILP based method which relate to improving the constraints generated and added during the search. The first extension examines the method by which we search for acyclicity constraints to add, the second introduces additional implied constraints of a different form, and the third attempts to ensure that the constraints found by other methods rule out greater invalid regions of the search space. In addition, the impact of several solver features is assessed.

The rest of this paper is arranged as follows. In Section 2, the problem of Bayesian network learning is addressed in more detail before looking at using Integer Linear Programming for this task. A software platform to carry out this learning is presented in Section 3. The novel contributions of this paper are presented in Section 4 before evaluation of these techniques are given in Section 5. Finally, Section 6 concludes.

2. Background

2.1. Bayesian Network Learning

There are two classes of methods for learning the structure of Bayesian networks. The first takes advantage of the fact that the structure of the network encodes information about conditional independence. One can perform multiple conditional independence tests on subsets of variables and use this information to infer what structure the BN should have.

The alternative method, and the one followed in this paper, is *score-and-search*. In this method, for each node, one computes scores for each possible set of parent nodes for the node and then uses some search algorithm to attempt to maximise a global score formed from the local scores, subject to the resulting network being acyclic.

There are many scores that have been proposed for learning BNs, for example BDeu [5], BIC [6], AIC [7]. These scores have the property of local decomposability, meaning that the global score can be found as a simple function of the score associated with each node. In the current paper, we restrict ourselves to consideration of the BDeu score, though we note that the software presented has been used to learn networks based on other scores [8, 9, 10].

Having produced local scores for the possible parent sets of each node, it is necessary to perform a search for the network with the maximum global score. This can be performed using any search method. These can be divided into heuristic methods that produce a high scoring network but cannot guarantee to produce the best one, and global searches that not only find the best network but also establish that no better network is possible. The work presented in this paper falls into this latter category, alongside recent approaches such as dynamic programming [11], A* search [12] and Branch-and-Bound [13]. As the quality of the learned network is identical for all exact methods, the primary challenge in this case is to produce a search algorithm that runs sufficiently quickly and is sufficiently scalable. Another recent approach [14, 15] also uses Integer Linear Programming to find an optimal Bayesian network, but as this has the added constraint of bounded tree-width, this is not directly comparable to the results presented here.

If there are n nodes in the BN, then the number of possible parent sets for each node is 2^{n-1}. In practice, for even relatively modest n, this is much too large to even score each of them in a practicable time, and probably creates a search space that is too large to explore effectively. In most cases it is possible to show that many parent sets cannot occur in an optimal BN and so can be pruned [16]. This speeds up scoring considerably. However even after pruning there typically remain a very large number of candidate parent sets. To overcome this problem, one must limit the number of parent sets, typically by restricting the maximum size of parent sets for which scores are produced. This in turn limits the search function to only considering BNs in which all nodes have at most a limited number of parents. The networks found using any of the exact methods are therefore no longer globally optimal, but optimal under the additional introduced constraint that nodes have a maximum indegree. This may not be suitable for some applications where large indegrees are expected. In that case, a heuristic method which allows more densely connected networks at the expense of guaranteeing optimality may be more suitable.

2.2. Bayesian Network Learning as an Integer Linear Program

The task of learning the best BN structure given certain observations can immediately be seen to be an optimisation task. Somewhat less obviously, the problem can actually be encoded rather straightforwardly as an Integer Linear Programming (ILP) optimisation problem. That this should be possible should be no surprise given that the decision version of both ILP and BN learning are NP-complete [17, 1].

The major advantage of encoding the problem as an ILP problem rather than directly solving it using a BN specific algorithm is that doing so allows one to take advantage of the decades of research in ILP optimisation. In particular, off-the-shelf solvers have been highly optimised and tuned to give very efficient performance. In addition, any other arbitrary constraint that can be linearly encoded can be simply introduced in to the problem without having to modify the solving method. For example, based on external knowledge of the problem domain, one could easily add an extra constraint to assert that two nodes must have an edge between them in one direction or the other.

The variables in the ILP encoding represent whether or not a node has a given parent set in the network. For every possible node, v and parent set W, a binary variable $I(W \rightarrow v)$ is created which will be assigned the value 1 if W is the parent set of v in the BN or 0 otherwise.

Using this encoding, one must write the objective function to maximise as a linear combination of these variables. The BDeu score (as well as many other locally decomposable scores) defines the score of a BN to be the product of the scores of each of its nodes. However, taking the logarithm of this score turns this into a summation while preserving the ordinality of solution scores. One can then write the score to be optimised (log BDeu) as the following linear expression of the ILP variables, where $c(v, W)$ is the log BDeu score for v having W as parents.

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

Having defined the ILP variables and objective function, it simply remains to define the constraints that must be obeyed by the $I(W \to v)$ variables in order that they encode a valid network. There are two such constraints; each node must have exactly one parent set, and there cannot be any cycles in the network.

The first of these constraints can be written very directly as follows.

$$\forall v \in V : \sum_{W} I(W \to v) = 1 \tag{2}$$

The acyclicity constraint is much less straightforward to encode. Previously [4] considered two schemes to achieve this involving introducing additional auxiliary variables and constraints. In one method, additional binary variables were introduced which recorded whether node u appeared before v in a partial ordering of the tree, along with simple transitivity and non-reflexive constraints on these variables. In the other, generation variables were introduced for each node and the constraint added that parent nodes must have a lower generation value than their children.

Experience however has revealed that an alternative method of enforcing acyclicity, *cluster constraints*, introduced by [2], has superior performance in practice. Observe that for any set of nodes (a cluster), if the graph is acyclic, there must be a node in the set which has no parents in that set, i.e. it either has no parents or all its parents are external to the set. This can be translated into the following linear set of linear inequalities.

$$\forall C \subseteq V : \sum_{v \in C} \sum_{W : W \cap C = \emptyset} I(W \to v) \geq 1 \tag{3}$$

[3] generalised cluster constraints to *k-cluster constraints* by noting that there must be 2 nodes with at most 1 parent in the cluster, 3 nodes with at

most 2 parents in the cluster, etc. However, experiments have failed to reveal any consistent improvement in performance by using these constraints.

Having defined the objective function and the constraints, one can simply enter these into any ILP solver and, given sufficient time and memory, an optimal BN will be produced. However, in order to obtain the best performance, some understanding of the method used to solve ILP problems is needed.

For non-integer LP optimisation, the problem can be solved relatively efficiently. Imagine each variable as corresponding to a dimension; one can then represent any possible assignment to these variables as a point in this space, and a linear equality as a hyperplane in the space. The region containing the valid solutions will be the convex polytope bounded by these equations and an optimal solution can be found at some vertex of this polytope. This solution can be found relatively quickly using the well-known simplex algorithm.

For ILP, this simple algorithm is not in general sufficient. The optimal solution for which the variables are integer may not lay at a vertex of the polytope but may instead be inside the polytope. Therefore, for ILP optimisation a branch-and-bound approach is taken. First, the simplex algorithm is used to solve the *linear relaxation* of the ILP problem (i.e. the ILP problem with the constraint that certain values must be integers removed). If this yields a solution in which the variables happen to be integers, then the true optimum has been found. Otherwise, a variable which has a non-integer value in the relaxed problem is chosen ($v = x$) to branch on and a search tree with two subproblems formed; one in which $v \leq \lfloor x \rfloor$ and one in which $v \geq \lceil x \rceil$. In the case of the binary $I(W \to v)$ variables used in the BN learning problem, this corresponds to branching on whether some $I(W \to v)$ is 1 or 0, i.e. whether some v has W as a parent set or not. Each of these subproblems can be recursively solved in the same way, as part of a standard tree search algorithm.

A common extension to the branch-and-bound algorithm adopted by many ILP solvers is the branch-and-cut approach. In this approach, after the relaxed problem has been solved, additional linear constraints are added to the problem which separate the current relaxed solution from the space of any valid integer solutions, i.e. any integer solution will respect the constraint but the current relaxed solution does not. These constraints are known as cuts. Consider for example the simple case in which it is deduced that an integer $X \leq 7.2$. As we know X to be integer, we can always add a cut of

$X \leq 7$ as this will not remove any possible integer solution from the space, but does usefully remove the parts of the search space where X takes non-integer values between 7 and 7.2. The search for the relaxed solution and the addition of extra constraints alternates until no more cuts can be found. If the variables in the relaxed solution are now integer the problem is solved, otherwise the problem branches as in the branch-and-bound algorithm.

There are well known cuts that can be added in any domain based on deducing implied constraints from the problem and the current relaxed solution, for example Gomory cuts [18] or Chvátal-Gomory cuts [19]. Alternatively (or additionally), one can choose to hold back some necessary known domain-specific constraints from the problem, adding them as cutting planes only if a proposed solution violates them. This approach is adopted with the cluster constraints in the BN learning problem. For a network with $|V|$ nodes, there are $2^{|V-1|} - 1$ cluster constraints; rather than initially adding such a large number of constraints, they are only added explicitly to the problem as cuts if a relaxed solution would violate them. In practice, this means that most cluster constraints are never added to the problem as solutions with cycles involving that cluster of nodes are never proposed.

Adding problem constraints as cutting planes rather than initially is typically done when there are very large numbers of such constraints, as is the case with the cluster constraints. There are two reasons why this may be desirable. First, there may be considerable overhead for the solver in managing the constraints, many of which may never be needed. Second, large numbers of non-redundant constraints lead to a particularly complicated polytope with very many vertices. As the simplex algorithm works by repeatedly considering adjacent vertices, the simpler the polytope, the fewer neighbouring vertices there will be at each step and the fewer vertices there will be on the path between the initial vertex and the optimal one. The improvement in the speed with which the simplex algorithm runs though must of course be weighed against the fact that additional time will be needed to identify the violated constraints to add and the fact that the simplex algorithm must be run repeatedly, rather than once at each node of the search tree.

3. GOBNILP

To investigate BN learning using ILP. we have created the software program GOBNILP (Globally Optimal Bayesian Network learning using Integer Linear Programming) which is freely available and uses the ILP formulation and branch-and-cut method presented in Section 2.2 to learn optimal Bayesian networks [3, 20]. It builds on the SCIP framework [21] to perform the ILP solving along with a further solver for the underlying linear programming solving. In the experiments presented in Section 5, GOBNILP version 1.5 is used with SCIP version 3.1.0 and using CPLEX version 12.5 as the LP solver.

In addition to the basic ILP formulation presented in the previous section, GOBNILP has a number of additional features that improve solving time which are presented below.

3.1. Heuristic Algorithm

During the search process, it is useful to occasionally find sub-optimal heuristic solutions to the problem. This provides a lower bound on the value of the true optimal BN and can be used to eliminate sections of the search tree using branch-and-bound.

In addition, heuristic solutions can be used to turn the solving into an anytime algorithm; the solving can be halted at any point and a valid, but suboptimal solution returned. As the current relaxed LP solution is an upper bound on the optimal solution, the values of the heuristic solution and relaxed LP solution together allow one to see how much better than the current heuristic solution the optimal solution might be. Depending on the application, it may be acceptable to take a sub-optimal solution which is guaranteed to be within a few percent of the optimal one rather than wait much longer for the true optimal solution to be found. Close cuts, which will be introduced in Section 4.4 also rely on a heuristic solution being known.

SCIP features a number of built-in general purpose heuristic finding methods based on the current relaxed solution. For example, one could simply try rounding fractional variables to their nearest integer value and see if this is feasible. SCIP has 6 such *rounding heuristics* which we use to look for valid BNs. It should be noted that often trying to round a relaxed solution will fail to produce a valid BN. In such cases, no new heuristic solution will be produced.

We have also implemented a heuristic method specific to the BN learning application. The algorithm relies on the fact that it is easy to find an optimal BN given a total ordering of nodes. We therefore carry out what is in essence a greedy sink finding algorithm; we first choose a sink node (i.e. one that will have no children), then choose a node that will have no children except possibly the first, then a node that will have no children except possibly the first two, and so on. The algorithm is somewhat akin to the dynamic programming approach to learning Bayesian networks [11], except we commit greedy choices at each stage rather than consider all possible subsets.

To choose the ordering for the sink finding heuristic, we utilise both the scores of each of the $I(W \to v)$ in the objective function (i.e. their associated log BDeu score) and their value in the current relaxed solution. The motivation for using the former is that we wish to choose high scoring parent sets as far as possible, and the latter is used as we believe a good heuristic solution is likely to be found in the vicinity of the current relaxed LP solution.

First, for each node, we arrange the possible parent sets in descending order according to their log BDeu scores. That is to say, for each node, we create a list in which the first parent set is the one with the highest BDeu score and the last parent set is that with the lowest BDeu score. From this point onwards, we do not make further use of the BDeu scores, but rather choose our variable ordering according to the parent set lists just created and the current relaxed LP scores.

On the first iteration of the algorithm, we compute a score for the parent set at the head of each variables' list. This score is one minus the current value of that variable having that parent set in the current relaxed LP solution, i.e. $1 - I(W \to v)$. We then choose the variable with the highest score to be our sink and its best parent set to be its parent set in our heuristic solution. Following this, we remove from all the variables' lists any parent set which contains our chosen sink variable.

At each subsequent iteration, we carry out a similar process of scoring the best remaining option for each as yet unchosen variable, selecting the variable with the highest scoring parent set, and eliminating any remaining parent sets which contain this newly chosen variable. The only difference between the first iteration and subsequent ones is that we calculate the scores slightly differently. In these iterations, we use the sum of the values of all possible remaining parent sets for that variable minus the value of the best remaining parent set for the variable, where the values are the scores in current relaxed LP solution.

Should the algorithm at any stage try to select a parent set for a variable which is impossible (due to user supplied constraints or the current search tree branching choices made, for example), the algorithm simply aborts without returning a heuristic solution.

The sink finding algorithm is very fast, running 9425 times in only 30s in one case we studied. We therefore allow it run after every LP solution along with the built-in SCIP heuristics.

3.2. Value Propagation

Value propagation is a well-known constraint-based technique. Given assignments of values to some variables (such as happens when the problem branches), the constraints that exist between variables can be used to infer reductions to the feasible domains of other variables. As the ILP encoding here contains solely binary variables, this is equivalent to fixing variables to either 1 or 0, i.e. the specified parent set must be selected or cannot be selected respectively. SCIP features built-in propagators which can perform the correct inference for linear equations. However, for the constraint that there must be no cycles in the graph, an extra propagator is needed that will perform propagation based on the constraint as a whole, not just the currently added cluster cuts. GOBNILP includes just such a propagator which attempts to perform this fixing after each branching in the search tree. The propagation uses basic reasoning such as if A is in all remaining possible parent sets of B, then all variables in which B is in the parent set of A must be set to 0.

4. Cutting Planes

Section 2.2 explained how the search for an optimal Bayesian network using ILP required both branching and cutting planes. Despite various attempts, we have been unable to find a method of choosing variables to branch on which outperforms SCIP's default branching strategy. Furthermore, we note that final solution times for problems appear to increase substantially in general when the program finds it necessary to start branching early in the solving process. We therefore focus our attention on finding cutting planes to constrain the problem as much as possible before carrying out any branching.

This paper introduces cutting extensions to the method for learning Bayesian networks using Integer Linear Programming presented in Section 2.2. A major issue identified there was of identifying and adding cutting planes in an effective manner. As with many other aspects of ILP solving, SCIP features a number of built-in general purpose cutting plane finding algorithms some of which are examined in Section 5. In addition to these, the cluster constraints for ruling out cycles in the network are added as cutting planes. Second, new classes of constraints are introduced which any valid integer solution must obey. These form the complete set of tightest possible constraints for the problem with 3 or 4 nodes, but as with the original acyclicity constraints, the issue of which to add and when proves critical to their success in assisting with solving larger problems. Finally, the use of *close cuts* [22, 23] is assessed. These attempt to find cuts that separate more of the search space from the relaxed solution than usual methods.

4.1. Finding Cluster Cuts with a sub-IP

In the standard ILP formulation of the problem, cycles are ruled out through adding cluster cuts, which state that there should be no cycles formed from all elements of that cluster. In practice, it is unnecessary to add all cluster cuts as solutions to the relaxed problem usually do not violate many of these constraints. Even when a relaxed solution does violate a constraint, it may not be desirable to add it, as other constraints may be added that also rule out the relaxed solution.

We now explain how we find cluster cuts. First note that, due to (2), cluster constraints can be written as follows:

$$\forall C \subseteq V : \sum_{v \in C} \sum_{W : W \cap C \neq \emptyset} I(W \to v) \leq |C| - 1 \tag{4}$$

Intuitively, (4) says that not all nodes in the cluster C can have parents in C. Our goal is to find a cluster C such that the LHS of inequality (4) exceeds $|C| - 1$ by as much as possible when the values of the current relaxed solution x^* are used for the variables $I(W \to v)$. We want to find a cut which x^* violates maximally since such cuts are 'deep cuts' leading to tight linear relaxations. We cast this problem as a sub-IP and solve it using SCIP as follows.

For each $I(W \to v)$ in the main problem with non-zero value in the current relaxed solution, create a binary variable $J(W \to v)$ in the cluster constraint finding subproblem. $J(W \to v) = 1$ indicates that the variable $I(W \to v)$ will be included in the cluster cut as formulated in (4). Also create binary variables $I(v \in C)$ which are 1 iff v is in the chosen cluster, Rather straightforwardly, we have for each $J(W \to v)$

$$J(W \to v) \Rightarrow I(v \in C) \tag{5}$$

$$J(W \to v) \Rightarrow \bigvee_{w \in W} I(w \in C) \tag{6}$$

where the first constraint states that $I(v \in C)$ must be 1 if $J(W \to v)$ is 1 and the second makes a similar assertion for the members of W. In other words, if $J(W \to v)$ is 1, v in the cluster along with at least one member of W. These constraints can be posted to SCIP directly as *logicor* constraints, along with a simple summation constraint that $|C| \geq 2$.

For reasons that will become apparent below, we set the objective function to maximise to $-|C| + \sum x^*(W \to v)J(W \to v)$, where $x^*(W \to v)$ is the value of $I(W \to v)$ in the current relaxed LP solution and $|C|$ is shorthand for $\sum_{v \in V} I(v \in C)$. We also use the SCIP function `SCIPsetObjlimit` to declare that only solutions with an objective greater than -1 are feasible.

It follows that any valid solution has

$$-|C| + \sum x^*(W \to v)J(W \to v) > -1 \tag{7}$$

Due to the constraints, for $J(W \to v)$ to be non-zero (and hence equal to 1), v must be in the cluster and at least one element of W must also be in C. So for any feasible solution (7) can be written as

$$-|C| + \sum_{v \in C} \sum_{W : W \cap C \neq \emptyset} x^*(W \to v) > -1 \tag{8}$$

equivalently

$$\sum_{v \in C} \sum_{W : W \cap C \neq \emptyset} x^*(W \to v) > |C| - 1 \tag{9}$$

So x^* violates the cluster constraint for cluster C and we have found a cutting plane. The sub-IP is always solved if possible, so if there is a valid cluster constraint cutting plane, the sub-IP is guaranteed to find it.

On a technical note, we solve the sub-IP using depth first search in the branching, and use any sub-optimal feasible solutions found during the search, as well as the final optimal solution. This means that the routine may sometimes produce multiple cluster constraint based cutting planes that we add to the main problem.

4.2. Finding Cluster Cuts through Cycles

In the previous section, a separate optimisation process is used to search for the cluster cut to add at each stage. The downside to such a strategy is that often only a single cut is added at each separation round whereas it may be more efficient to find several cuts at once.

We therefore outline a different method of identifying cuts to add, based on directly identifying any cycles in the graph encoded by the current relaxed solution and then adding cluster cuts ruling each of them out.

As the relaxed solution, by definition, has the integrality constraint on variable values relaxed, it may contain variables which have fractional values. A solution with fractional variables does not encode a graph structure as described in Section 2.2, therefore the first step must be to extract a graph from these variables.

Let $G = (V, E)$ be a directed graph, where V is the set of nodes involved in the BN. Construct E as follows.[1]

$$E = \left\{ (v_1 \to v_2) : v_1, v_2 \in V, \sum_{W : v_1 \in W} I(W \to v_2) = 1 \right\} \tag{10}$$

This graph is specified in terms of edges, rather than parent sets as in the main problem. An edge $A \to B$ exists in this graph if all parent sets of B with non-zero current LP solution value contain A. Intuitively, this graph is a 'rounded' version of the graph given by the current LP solution with the edges that are 'fractional' removed.

It is straightforward to extract the elementary cycles of G. In the current work, the method of [24] is used. This essentially performs repeated depth-first searches through the graph from each node, blocking any nodes in the current path to prevent paths with sub-cycles.

Having determined the cycles of this graph, one can simply take the set of nodes involved in each cycle and add a cluster cut to the problem for each of these sets. Any cluster cut involving all the nodes in any cycle of G will separate the current relaxed solution. However, the converse does not hold; there exist cluster cuts that separate the current relaxed solution which do not correspond to cycles in G.

Experimentation reveals that the time to identify the cycles of G can be significantly reduced if one only searches for cycles up to a given length. However, the trade off against the possibly reduced number of cycles, and hence the number of cuts, found must be considered. Experimental assessment of the effect of altering the maximum length of cycle to search for is presented in Section 5.2.

[1] In practice, $\sum_{W : v_1 \in W} I(W \to v_2) > 0.99$ is used to permit some tolerance of rounding errors.

As this cycle-based method of finding cuts may not find all valid cluster cuts for a given relaxed solution, it may be worthwhile to use it in conjunction with the sub-IP method of GOBNILP. The sub-IP method is guaranteed to find a valid cluster cut if one is possible but, as noted above, will only typically find a single cut per execution and involves significant overhead in initialising an optimisation process. Section 5.2 considers various methods of combining these two methods for searching for cluster cuts.

4.3. Convex Hull Constraints

It can be useful to think of two polytopes. The first $P_{cluster}$ is the region defined solely by the constraints given in Section 2.2, i.e. the polytope whose vertices are defined by the constraint on one parent set per node, the complete set of all cluster constraints and the upper and lower bounds on the variables. This polytope contains all the valid solutions to the BN learning problem, but many other points as well.

While the inequalities in Section 2.2 are sufficient to correctly classify all integer assignments to the variables as being a valid BN or not, they do not completely specify the smallest possible convex polytope for the problem.

Defining this second polytope, known as the *convex hull*, is desirable as it has the property that each vertex of the polytope is an integer solution. This means that the solution to the linear relaxation is the optimal integer solution and so the simplex algorithm can be used to solve the ILP without any branching needed.

Let us denote the convex hull to be the polytope P. This is contained within the larger polytope $P_{cluster}$. Adding cutting planes has the effect of removing some of the polytope $P_{cluster}$ that lies outside of P, thus reducing the solution space to search without removing any integer solutions. If we were able to add all possible valid constraints to $P_{cluster}$, we would find it reduced to P.

Given this observation, it is interesting to ask what the full set of constraints necessary to define P is. Given this information, we could simply add all these constraints (initially or as cutting planes) and find the BN through the simplex algorithm without branching.

It should be noted that the polytope P is conceptually similar to but distinct from the well studied *acyclic subgraph polytope* [25], P_{dag}, which also represents an acyclic graph as a set of binary variables. In the case of P_{dag}, the variables correspond to the existence of edges in the network, rather than the existence of a particular parent set as in the current work. Many facets of P_{dag} are already known but unfortunately this polytope is of little use in the current application as the scores used in Bayesian networks do not decompose into a linear function based on the existence of edges, but of whole parent sets.

The convex hull P appears to be extremely complicated in the general case, though we have empirically found the complete set of constraints defining the convex hull of the problem when the number of nodes in the network is limited to 3 or 4.

The convex hull of the problem with 3 nodes, P_3, was found using the lrs[2] algorithm. In this case, it transpires that P_3 consists of 17 facets; 9 lower bounds on the variables' values, the 3 limits on each node having one parent set, 4 cluster constraints, and a single extra constraint:

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

This can be generalised to give a class of *set packing* constraints which are valid for any Bayesian network learning problem, not just those with 3 nodes.

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

These will always be obeyed by integer solutions but otherwise acceptable fractional solutions exist which violate these inequalities. GOBNILP therefore adds such inequalities for all $|C| \leq 4$ to the initial formulation, which speeds up solution times.

As the number of nodes in the BN increases, the convex hull becomes more complex. For a BN with just 4 nodes, Matti Järvisalo used cdd[3] to show that there are 64 different facets to the convex hull, P_4. In addition to lower bounds on variable values, limits on the number of parent sets per node and cluster constraints, 7 different classes of facets were identified. We label these constraints *convex4* constraints. A simple generalisation of these constraints are still valid when applied to a subset of 4 nodes in a larger Bayesian network learning task. The problem of finding all facets of the convex hull for a BN of greater than size 4 is, to the best of our knowledge, unsolved.

---

[2] `http://cgm.cs.mcgill.ca/~avis/C/lrs.html`

[3] `http://www.inf.ethz.ch/personal/fukudak/cdd_home/`

Rather than continue searching for all facets of the convex hull for arbitrarily large BNs, we instead utilise those found for BNs with 3 and 4 nodes for subsets of the nodes in larger BNs. Inequality (11) generalises to Inequality (12) and such constraints are added to the initial problem for all subsets of nodes of size 3 and 4. For the convex4 constraints identified, for any reasonable large BN problem, there may be too many to add them all initially to the problem for each subset of 4 nodes. We can therefore add them as cutting planes during the search in much the same way as acyclicity constraints are added.

Adding these constraints initially may lead to better initial solutions. However, it may simply slow down the search by adding constraints that would never be violated if they were not added to the problem. In addition, searching for 4 node convex hull cuts which separate a relaxed solution is itself quite time consuming. As it is difficult to deduce whether these constraints should be added initially, as cutting planes, or not at all, we present experimental results in Section 5.3 assessing the best way to make use of them.

4.4. Close Cuts

The current solution to a relaxed problem will always be a point on the surface of the polytope corresponding to the current relaxed problem. It is possible that a cutting plane added to separate this solution will cut away a significant portion of the polytope, but it is also possible that it will pare away only a small region near the surface. In this latter case, the cut removes little of the solution space and may not assist significantly with finding a solution to the problem.

GOBNILP's usual strategy for dealing with this is to use the sub-IP to search for cutting planes which are efficacious, that is to say, cuts which pass deep into the polytope. However, this cannot be adapted for other methods of finding cutting planes, such as using cycle finding. An alternative to try to ensure arbitrary methods produce deep cuts is needed.

Close cuts [22, 23] are a solution to this problem. Despite the name, they are not actually a different type of cut, but rather they use the same cuts as normally used (general-purpose or domain-specific) but attempt to separate something other than the current relaxed LP solution. Rather than seeking a cut which removes a point on the surface of the polytope and hoping this is deep, close cuts pick a point which is deep in the polytope and then attempt to separate this. Specifically, a point is chosen which lies somewhere on a line between the best known heuristic solution to the problem and the current solution to relaxed problem, and the usual cutting plane finding methods are called to separate this point. Let the best known heuristic solution be $\mathbf{S}^h$ and the current relaxed solution be $\mathbf{S}^r$, then the point to be separated by the cutting plane routine is $\alpha\mathbf{S}^h + (1-\alpha)\mathbf{S}^r$ where the parameter $\alpha \in [0, 1]$ determines how close to the currently known heuristic solution the point to be separated is. As such, the point to separate is a convex combination of the current heuristic solution and the current relaxed solution.

There is no guarantee that the chosen point to separate is actually an invalid solution to the problem. In such a case, the cutting plane finding algorithms will fail to find any cuts, as there are no possible cuts. This does not present a logical problem; one can simply revert to finding separating planes for the solution to the relaxed problem as normal.

Clearly, the larger the parameter α is, the further into the relaxed problem polytope one is attempting to cut but, conversely, the more likely one is to choose a point that is a valid solution to the unrelaxed problem. Setting the value of α correctly is therefore crucial to the success of the technique. Experiments presented in Section 5.4 assess the best value for this parameter in the BN learning problem.

5. Results

To test the effectiveness of each of the aspects of adding cutting planes explored in Section 4, we incorporated them into the GOBNILP software and allowed their behaviour to be set through user supplied parameters. In some cases, for example close cuts, the required behaviour was already available through the SCIP framework.

Experiments were then conducted to assess the impact that each of these constraints had on the optimisation process. The presented methods will always find an optimum BN given sufficient time and computational resources. It therefore makes little sense to evaluate the score of the network found or the similarity of this network to the true one as this is simply evaluating how well optimisation of the score works; the same results would be found for all variants of the ILP technique, as well as for any other exact optimising technique such as [11] or [12]. Rather, the correct form of evaluation here is to analyse the time needed to find an optimum BN.

As the NP-hardness of the problem requires a limit on the size of the parent sets considered, our algorithm may not find the true most likely BN, as it may have larger parent sets than we permit. In cases where nodes with high in-degree are likely to exist, an alternative, heuristic method may be more desirable, which finds a high scoring though not guaranteed optimal network. Such issues, along with choice of scoring function are beyond the scope of the current paper which is restricted to finding the provably optimal BN from a choice of scored parent sets; the accuracy of the final network is solely reliant on the suitability of the scoring function chosen and the validity of assuming a maximum node in-degree.

The experiments presented are divided into two parts. First, a number of features of the solver which can best be described as on or off are examined. Based on prior experience, we believe them all to be generally useful [20]. We begin by turning them all on as a baseline and then conduct experiments in turning each of them off in turn while the others are all on. This provides an estimate of how much each feature helps or hinders the solving process. The features examined in this way are

- The use of Gomory cuts (see Section 2.2)
- The use of Strong Chvátal-Gomory cuts (see Section 2.2)
- The use of Zero-Half cuts (see Section 2.2)
- The sink-based heuristic (see Section 3.1)
- The value propagator (see Section 3.2)
- The use of set packing constraints (see Section 4.3)

It is likely that there are some interactions between these parameters. For example, for some reason the value propagator might turn out to be more effective when Gomory cuts are not being used. However, to assess all possible settings of each of these features together would require a prohibitively large number of experiments to be run, much of which would produce little practical insight; for example, it would be of little use to know if the value propagator was particularly useful when set packing constraints were not used if it turned out that using set packing constraints was always of significant benefit.

Following this set of experiments, the use of a number of extensions which are more parametric are assessed. These three extensions are the method by which cluster cuts are found, the point at which convex4 constraints are added and the use of close cuts. In the previous experiments, we fix their behaviour to using the sub-IP and no cycle finding for identifying cluster cuts, no use of convex 4 cuts and no use of close cuts.

In the second set of experiments, we began with a system in which all 3 extensions are fully disabled, then assessed the incrementally impact of adding each in turn. We first assess how changing the method for discovering cluster cuts to add impacts on the solution time. The best parameter settings discovered in this experiment are then used as a baseline for the convex4 experiments, studying how well adding these constraints alongside the new cluster cut mechanism works. Finally, the best settings for the cluster cuts and convex4 constraints resulting from this experiment are taken forward to the final experiment, in which the close cut parameter is varied.

One might reasonably expect the observed behaviour with different settings for each of these extensions to exhibit interactions with each other in non-trivial ways. The outcome of these experiments therefore cannot be viewed as finding the best parameter settings, though they do represent a greedy search for these best settings. Nevertheless, this scheme provides a reasonable balance between studying the individual effects of each of the extensions and attempting to capture some of the cross-extension interactions.

As an alternative, one could perform a search over the parameter space of all 3 extensions simultaneously, either in a systematic or heuristic manner. This would fully capture the interactions between the parameters associated with the different extensions, but would probably be infeasible to conduct due to the combinatorial increase in the size of the parameter space and would make it difficult to assess the individual impact of each of the extensions. At the other end of the scale, one could neglect the possible interaction between the various parameters entirely and evaluate each extension in turn with the other two disabled. The effects of each extension would be readily apparent but any results would be wholly artificial. Assuming more than one of the extensions to be beneficial, one would not run the system in this state and subsequently combining the best individual settings for each extension may lead to a severely suboptimal system due to previously unseen interaction effects.

Evaluation of the approaches was performed using a number of datasets drawn from Bayesian networks from commonly used sources. The datasets considered are shown in Table 1 and were chosen to test performance on a wide range of different problems. In each case, the BDeu score was computed (using the equivalent sample size listed in the table) external to GOBNILP

Name	Equivalent Sample Size	Number of Variables	Parent Set Limit	Number of Parent Sets
car	1	7	6	35
asia	10	8	2	127
insurance	1	27	6	341
mildew	1	35	3	3520
tic-tac-toe	10	10	3	112
flag	10	30	5	24892
dermatology	10	35	3	5059
hailfinder	1	56	4	4330
kr-vs-kp	10	37	2	12877
soybean-large	2	36	2	10351
sponge	1	46	4	11042
zoo	10	18	4	6461
alarm	1	37	4	8445
diabetes	1	413	2	4441
carpo	1	60	3	16391
lung-cancer	10	57	2	8294

Table 1: Characteristics of the problems studied. "Number of Variables" is the number of variables in the Bayesian network, while "Number of Parent Sets" corresponds to the number of ILP variables.

and this preprocessing time is not included in the reported results. Upper limits on the size of potential parent sets for which scores were calculated were introduced to make the times to compute the scores feasible. Note that the number of parent sets corresponds to the number of variables in the ILP problem. Some pruning of the number of parent sets is possible. For example, if A having a parent set of just B has a better score than A having both B and C as parents, we can prune the latter from the dataset, as there is no situation in which adding the additional parent C would be preferred in an optimal network. The number of parent sets reported in the table refers to the number remaining after this type of pruning has occurred.

Experiments were conducted on a single core of a 2.7GHz dual-core processor computer with 6GB of memory running Linux. All experiments used SCIP version 3.1.0 with CPLEX version 12.5 as the underlying LP solver. A time out limit of 2 hours was set for all experiments. Any experiments which had not terminated in this time limit have the gap between their best relaxed LP solution and their best known heuristic solution shown in the following tables, which provides a proxy for how much more work the search has remaining to do at that point.

As our intention is to evaluate various possible aspects of the ILP BN learning problem, we do not compare the algorithm to other approaches. As previously stated, comparing to heuristic methods or conditional independence methods assesses the quality of the reconstruction which is not our concern here. This factor depends hugely on the score chosen and the validity of the assumption that parent sets are of reasonably small size, the former of which is outside the scope of the current paper and the latter of which is problem dependent. Rather our aim is to assess the speed with which we can find a provably optimal network.

[26] provides an extremely thorough evaluation of the solving times for a number of recent optimal BN learners including GOBNILP on over 700 problem instances. Rather than repeat this exercise, we state the main findings of relevance here. The default configuration of GOBNILP was found to be fastest on over 300 instances, and when combined with various other configurations trialled, GOBNILP was fastest on the majority of problem instances. Various configurations of A*-search [12] were fastest on around 300 instances, but no configuration was fastest on many more than 100 instances. Branch-and-bound [13] is fastest on none. Overall, GOBNILP solves the entire dataset in 661,539 seconds, as compared to 1,917,293 seconds for the best A* configuration and over twice that time for Branch-and-Bound.

5.1. General Purpose Cuts and Solver Features

The results of turning various cuts or solver features off are shown in Table 2. Overall they show a mixed picture for many of the experiments, with everything being useful for some datasets and increasing solving times in others. On the whole Gomory cuts appear to be generally useful (i.e. the solving times rise when they are turned off). The picture for the other two cuts is less clear, with some big improvements seen in some datasets when they are used, but slower solving times witnessed in others.

For the solver features, a similar outcome is observed; some features are particularly helpful sometimes but hinder in other cases. The set packing constraints appear particularly useful in the mid-difficulty problems, but are less useful on the harder problems studied. The sink finding heuristic and value propagation appear more consistently to be useful but there are exceptions to this. Overall, the slowdown caused by not having a solver feature

Network	Baseline	No Cuts of Type			Without Solver Feature		
		G	SCG	ZH	SPH	SPC	VP
car	0.26s	0.01s	0.01s	0.01s	0.01s	0.15s	0.01s
asia	0.36s	0.35s	**0.97s**	0.34s	0.34s	**0.42s**	0.36s
insurance	0.83s	0.76s	**1.00s**	0.74s	0.81s	**1.47s**	0.81s
Mildew	1.20s	1.16s	1.16s	1.12s	**1.21s**	**2.03s**	**1.21s**
tic-tac-toe	9.40s	5.23s	9.18s	2.55s	9.26s	8.41s	9.30s
flag	39.99s	36.79s	17.77s	19.14s	36.09s	**70.85s**	35.95s
dermatology	32.17s	31.19s	21.16s	27.49s	31.63s	28.84s	29.74s
hailfinder	112.56s	79.23s	61.90s	87.56s	**118.97s**	**226.23s**	111.93s
kr-vs-kp	124.37s	80.64s	75.48s	71.91s	**125.81s**	96.75s	122.47s
soybean-large	98.41s	92.83s	**130.14s**	82.02s	89.90s	**110.38s**	97.14s
alarm	200.64s	**280.12s**	112.78s	**244.59s**	**201.50s**	108.71s	**227.45s**
Diabetes	—	—	—	—	—	—	—
sponge	195.03s	**225.24s**	**300.02s**	**231.95s**	**230.15s**	**214.46s**	191.36s
zoo	264.79s	**290.40s**	191.36s	166.54s	**266.65s**	174.74s	214.03s
carpo	483.69s	**528.10s**	**587.44s**	**513.89s**	**577.18s**	**589.40s**	**510.62s**
lung-cancer	670.76s	646.50s	651.57s	583.45s	670.66s	642.77s	627.88s

Table 2: Impact on time to find the best Bayesian network of various features. All times are given in seconds to the nearest whole second. "—" indicates that the solution had not been found after 2 hours. Key: G – Gomory cuts, SCG – Strong CG cuts, ZH – Zero Half cuts, SPH – Sink Primal Heuristic, SPC – Set Packing Constraints, VP – Value Propagator. Results that are worse than the baseline are indicated in bold.

appears to be greater than that for not having one of the cutting algorithms turned on.

5.2. Cycle Finding

The method for deciding which cluster cuts to add in the previous experiment was through a sub-IP. As well as considering simply replacing this with the cycle finding method, we also considered two schemes in which the two methods were both used. In the first, both methods were called every time cuts were searched for. In the second, the cycle finding algorithm was first used and only if this failed to find any cuts was the sub-IP method used. The motivation for considering running both methods is that the cycle finding algorithm only searches for cycles in the portion of the graph which corresponds to edges which are non-fractional in the current LP solution, whereas the sub-IP will also detect violated cluster constraints involving fractional variables. This means that the sub-IP can detect cuts to add that are a superset of those detected by the cycle finding method. This also explains why the idea of using the cycle finding only if the sub-IP failed was not considered.

In addition, the maximum length of cycle to look for by the cycle finding algorithm was investigated. The perceived trade-off here is between choosing a small value which risks missing many longer cycles and choosing a large value which takes a long time to run with possibly few additional cycles found. Preliminary experiments were used to identify a sensible range of values for this parameter over which the presented experiments were conducted.

These preliminary experiments also revealed that using just the cycle finding without the sub-IP was substantially worse than using them both together or just the sub-IP. For example, for some maximum cycle lengths it reached the time out limit on the Mildew problem (which other configurations typically solve in around a second) and consistently took over 5 minutes to solve the Flag problem, instead of about 30 seconds for other settings. The following therefore focuses attention on the techniques involving just the sub-IP, or the sub-IP and the cycle finding together.

The results shown in Tables 3 and 4 demonstrate a fairly consistent benefit from using the cycle finding method with the sub-IP, rather than the latter alone. The results for Diabetes are particularly significant, going from unsolvable in two hours with just the sub-IP to solvable with a wide range of settings when cycle finding was used as well; in the best case, a solution was found in under 30 seconds.

	Sub-IP	3	4	5	6	7	8	9	10	100
car	0.26s	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**
asia	0.36s	1.52s	2.66s	**0.23s**	1.47s	0.40s	0.38s	0.37s	0.38s	0.38s
insurance	0.83s	0.63s	0.60s	0.44s	0.44s	0.67s	**0.26s**	0.30s	0.39s	0.32s
Mildew	1.20s	0.97s	0.62s	0.43s	0.48s	0.44s	0.38s	0.37s	**0.35s**	**0.35s**
tic-tac-toe	9.40s	9.85s	9.64s	9.44s	**9.32s**	9.46s	9.36s	9.42s	9.88s	10.01s
flag	39.99s	27.56s	38.70s	27.18s	23.60s	**17.82s**	22.51s	24.54s	35.68s	32.28s
dermatology	32.17s	24.66s	30.92s	19.22s	**15.50s**	16.32s	19.80s	23.76s	19.98s	19.91s
hailfinder	112.56s	64.28s	61.00s	45.46s	94.18s	65.49s	45.39s	44.93s	44.38s	**44.24s**
kr-vs-kp	124.37s	78.32s	84.10s	62.66s	64.14s	81.18s	76.20s	**59.94s**	55.95s	80.66s
soybean-large	98.41s	104.42s	94.79s	**84.60s**	104.93s	128.98s	99.38s	126.13s	113.29s	101.23s
alarm	200.64s	118.53s	182.08s	**92.52s**	252.18s	224.47s	100.47s	148.26s	205.13s	142.27s
Diabetes	[8.88%]	[3.13 %]	28.04s	68.91s	126.22s	33.99s	39.58s	**23.57s**	35.05s	[3.45 %]
sponge	195.03s	235.87s	189.77s	266.40s	232.30s	199.99s	**175.03s**	207.15s	219.87s	236.56s
zoo	264.79s	171.19s	187.98s	199.36s	174.38s	169.66s	175.07s	207.95s	188.68s	**159.79s**
carpo	483.69s	604.97s	702.44s	605.55s	604.94s	519.70s	553.36s	458.73s	494.55s	**401.82s**
lung-cancer	670.76s	619.15s	**617.57s**	653.24s	625.52s	678.90s	673.10s	672.04s	653.10s	653.10s

Table 3: Times to find an optimal Bayesian network for various maximum cycle lengths when both cycle finding and the sub-IP are used. Results using the sub-IP only are shown for comparison. Percentages in square brackets are shown when the program ran out of time or memory, and indicate the remaining gap between the best discovered solution and the upper bound on the best possible solution at this point. The fastest configuration for each problem is shown in bold.

	Sub-IP	3	4	5	6	7	8	9	10	100
car	0.26s	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**
asia	0.36s	1.19s	0.53s	0.21s	0.87s	**0.16s**	**0.16s**	**0.16s**	**0.16s**	**0.16s**
insurance	0.83s	0.81s	0.46s	0.31s	0.33s	0.19s	**0.16s**	0.35s	0.23s	0.25s
Mildew	1.20s	0.93s	0.57s	0.52s	0.49s	0.58s	**0.25s**	0.26s	**0.25s**	**0.25s**
tic-tac-toe	9.40s	9.38s	9.43s	**9.36s**	9.42s	9.49s	9.48s	9.47s	9.51s	9.42s
flag	39.99s	28.70s	**17.52s**	31.83s	23.47s	19.45s	21.26s	51.73s	35.09s	44.67s
dermatology	32.17s	28.16s	21.32s	**17.17s**	21.70s	18.72s	20.58s	28.17s	18.42s	17.40s
hailfinder	112.56s	50.17s	63.57s	80.85s	51.98s	**40.24s**	41.50s	41.04s	41.79s	41.58s
kr-vs-kp	124.37s	82.81s	73.87s	97.66s	88.55s	84.21s	126.69s	**73.11s**	78.31s	76.31s
soybean-large	98.41s	115.07s	112.34s	**83.93s**	98.58s	97.49s	98.93s	96.81s	108.90s	97.92s
alarm	200.64s	**112.50s**	205.65s	194.60s	177.16s	286.12s	171.58s	187.60s	209.08s	222.24s
Diabetes	[8.88%]	[3.14 %]	3138.39s	1475.24s	66.68s	**18.17s**	172.92s	54.01s	18.61s	[3.45 %]
sponge	195.03s	193.05s	**189.49s**	212.11s	190.22s	262.17s	215.36s	217.67s	208.63s	226.65s
zoo	264.79s	160.25s	184.17s	165.67s	**136.23s**	205.22s	180.08s	165.40s	154.15s	175.48s
carpo	483.69s	724.91s	604.03s	607.85s	510.74s	**402.51s**	678.05s	611.93s	561.09s	531.79s
lung-cancer	670.76s	645.22s	697.98s	**638.32s**	641.71s	673.24s	697.99s	698.94s	716.60s	710.19s

Table 4: Times to find an optimal Bayesian network for various maximum cycle lengths when cycle finding is used followed by the sub-IP only if the former failed to find cuts. Results using the sub-IP only are shown for comparison. Percentages in square brackets are shown when the program ran out of time or memory, and indicate the remaining gap between the best discovered solution and the upper bound on the best possible solution at this point. The fastest configuration for each problem is shown in bold.

The best value for the maximum cycle length varied from dataset to dataset but either 5, 6 or 7 were usually amongst the best settings. In several cases, the difference between the best maximum cycle length and the worst could lead to a halving of the solution time. However, other datasets seemed somewhat insensitive to this parameter. This may reflect the density of the graph on which the cycle-finding algorithm was used; the more edges in the graph, the greater the increase in time that would be needed to search for long cycles though, conversely, the more chance of there being a longer cycle to find.

Using the sub-IP and cycle finding together produced broadly similar behaviour for both the methods studied. The better method depends on whether one considers all examples equally important or the longer ones to be more significant, and on the maximum cycle length chosen.

5.3. Convex4 Constraints

The constraints based on the convex hull of the 4 node BN polytope can be added to the problem initially or used as cutting planes. In the case where they are used as cutting planes, they can be searched for whenever a solution needs separating or only when no efficacious cluster constraints can be found.

In theory, one could consider adding each of the 7 classes of convex4 constraint in a different manner, but we restrict ourselves here to adding them all in the same way in order to make exploration of the space tractable.

For each option, the cluster constraint finding algorithm is fixed to a reasonably good setting as determined by the previous experiment. Specifically, we use cycle-finding initially with the maximum cycle length set to 6 and then call the sub-IP only if this fails to find a useful cluster constraint.

The results, as shown in Table 5, demonstrate a clear pattern amongst the different strategies explored. There are occasional exceptions, but the trend is for the best option to be not using the convex4 cuts or using them as cutting planes only when other efficacious cuts are not found. Overall, adding them as cutting planes in this way seldom does much harm and sometime leads to noticeable improvements.

For a couple of datasets, adding these constraints as cutting planes even when cluster constraints have been found proves very good. However this must be weighed against the fact that in many other case this strategy proves slightly detrimental compared to adding them only when cluster constraints are not found. Adding the constraints initially rather than as cutting planes almost never provides an improvement over leaving them out all together,

	As Cuts Always	As Cuts If Fails	Initially	Never
car	0.01s	0.01s	0.01s	**0.00s**
asia	**0.45s**	0.52s	1.55s	0.82s
insurance	**0.24s**	0.31s	0.32s	0.31s
Mildew	0.82s	**0.46s**	0.62s	**0.46s**
tic-tac-toe	**7.54s**	14.46s	8.36s	9.36s
flag	26.40s	23.31s	29.30s	**22.88s**
dermatology	23.59s	**21.64s**	29.38s	21.70s
hailfinder	76.14s	75.81s	69.88s	**50.61s**
kr-vs-kp	113.11s	101.42s	130.71s	**87.43s**
soybean-large	117.28s	110.79s	102.46s	**92.30s**
alarm	73.65s	**66.05s**	368.55s	175.06s
Diabetes	**55.67s**	67.09s	67.14s	67.16s
sponge	298.49s	257.75s	341.64s	**192.58s**
zoo	325.34s	156.95s	290.39s	**137.67s**
carpo	983.76s	653.13s	725.95s	**519.82s**
lung-cancer	**620.26s**	639.46s	658.59s	638.05s

Table 5: Times to find an optimal Bayesian network for various ways of adding the convex hull constraints. The fastest configuration for each problem is shown in bold.

and only provides a small help in those few cases. This suggests that any advantage they bring to tightening the problem is outweighed by the overhead of having to process all these additional constraints at each LP iteration.

5.4. Close Cuts

A single parameter associated with close cuts is studied. As explained in Section 4.4, α is the parameter that determines how far between the relaxed solution and the currently best known solution the point chosen for separation is. We investigate setting this parameter between 0.1 and 0.9 in increments of 0.1. $\alpha = 0$ corresponds to the case where the relaxed solution is separated (equivalent to close cuts not being used). It should be noted that the heuristic methods for finding valid BNs will also have an effect on how well close cuts work. If a heuristic were able to find the true optimum network consistently, one might expect that cutting near to this optimum (i.e. a large α) might lead to the search space near the solution being quickly pruned away and the problem solved.

As before, cycle finding with a maximum length of 6, followed by the sub-IP if necessary is used with the convex hull constraints added as cutting planes only when efficacious cluster cuts are not found.

The results in Table 6 illustrate that different values of α can have considerable impact on the solution times. In almost all cases, there exists a value of α that can improve the method over not using close cuts. However, there is no value of α that consistently outperforms the others. Furthermore, there is not even a general trend, for example towards larger αs being better or to the best α increasing as problem difficulty does. Worst of all, αs that are very good on one dataset are very poor on another dataset. A particularly extreme example is seen in the Flag dataset, where an α of 0.3 gives an answer in virtually half the time of not using close cuts, but if this α is increased or decreased by 0.1, the time is over twice that of not using close cuts.

Overall, it is correct to say that the use of close cuts can make a considerable improvement to the solving time. However, the choice of an α value that leads to this performance improvement cannot be deduced from these experiments. The ramifications of this result are returned to in the following section.

6. Conclusions

In this paper, the Bayesian network learning problem has been explored as an ILP problem. In particular, attention has been focused on the problem of finding and adding appropriate cutting planes to speed the solution times.

Various aspects were studied and some found to be generally beneficial across a range of problem instances. For others, impact was either modest or erratic on different instances.

Using cycle finding to determine cutting planes was highly effective when used in conjunction with a sub-IP approach. The impact of adding constraints based on the convex hull of smaller BNs was much less evident. When used in the best way, they improved performance more often than they degraded it, though for most datasets the difference in runtime was negligible.

The modest improvement associated with the additional classes of cut could call into question whether further work on additional cutting planes based on implied constraints truly merits attention. [20] showed that including implied constraints based on the convex hull of of a 3 node BN led to a measurable improvement, while [4] states inclusion of an implied constraint

	Off	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
car	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**
asia	0.53s	0.35s	0.55s	0.61s	0.58s	0.54s	0.37s	**0.27s**	0.40s	0.58s
insurance	0.31s	0.28s	0.31s	0.29s	0.19s	0.19s	**0.18s**	0.27s	0.26s	0.19s
Mildew	**0.46s**	0.63s	0.66s	0.65s	0.55s	0.61s	0.63s	0.71s	0.54s	0.58s
tic-tac-toe	14.62s	10.37s	10.74s	10.12s	10.45s	15.19s	**9.99s**	10.65s	13.37s	11.37s
flag	23.10s	47.43s	62.26s	14.17s	64.02s	57.73s	15.62s	14.59s	**11.75s**	28.88s
dermatology	21.49s	21.62s	15.93s	26.38s	17.86s	**15.67s**	21.18s	22.59s	15.73s	19.42s
hailfinder	75.32s	127.84s	**41.08s**	89.22s	66.41s	56.34s	56.46s	55.36s	47.63s	58.83s
kr-vs-kp	104.46s	86.22s	62.98s	**53.07s**	66.08s	83.16s	68.31s	58.92s	64.35s	66.76s
soybean-large	110.96s	116.13s	95.38s	**79.96s**	98.80s	107.94s	91.28s	105.42s	116.81s	93.23s
alarm	66.38s	78.29s	70.62s	59.08s	48.64s	46.51s	70.90s	67.35s	**46.16s**	76.39s
Diabetes	67.46s	26.25s	**24.73s**	202.65s	58.37s	36.51s	28.94s	36.79s	48.31s	49.40s
sponge	257.89s	**163.28s**	320.99s	180.75s	222.62s	299.64s	326.30s	232.73s	239.47s	276.13s
zoo	**154.34s**	181.41s	203.72s	221.17s	177.98s	237.12s	227.44s	202.42s	199.60s	154.56s
carpo	663.75s	676.93s	663.79s	667.87s	622.24s	604.76s	**586.35s**	787.34s	620.65s	640.97s
lung-cancer	642.65s	635.13s	**592.56s**	726.35s	639.21s	618.50s	622.12s	710.86s	603.47s	605.03s

Table 6: Times to find an optimal Bayesian network for various values of close cut parameter α. Off is equivalent to $\alpha = 0$. The fastest configuration for each problem is shown in bold.

that at least one node has no parents led to a dramatic decrease in solving time. On the other hand, a generalisation of cluster constraints to k-cluster constraints proposed by [3] has failed to prove to be of any notable benefit. Clearly, different types of implied constraints vary vastly in their usefulness and further theoretical work is needed to understand why some are beneficial for this problem and others are not.

The results of the close cuts experiments are particularly interesting. Close cuts could almost always provide an improvement in solving time, but only for the correct value of the α parameter, and there was no consistently good value for this parameter across datasets. A similar observation could also be made for the maximum length of cycle to search for in the earlier experiment. Though the technique was of clear benefit, the solution times for similar maximum lengths varied quite considerably and no value was best across the whole dataset. These findings suggest that rather than fix the solver's method to some compromise 'best' configuration, a future approach may be to change the settings for individual problems. The issue then becomes predicting appropriate solver settings for a previously unseen problem instance. [26] provides a step in this direction. Based on very simple characteristics of a problem instance they are able to determine quite accurately which of two Bayesian network learning algorithms will be quicker. However, further work is clearly needed for this to be applicable here, where there are larger numbers of options from which to choose and where one might reasonably expect choosing from amongst various configurations of a single solver to be more complex than deciding between two entirely separate solvers.

Acknowledgements

This work has been supported by the UK Medical Research Council (Project Grant G1002312).

[1] D. M. Chickering, Learning Bayesian networks is NP-complete, in: D. Fisher, H. Lenz (Eds.), Learning from Data: Artificial Intelligence and Statistics V, Springer-Verlag, 1996, pp. 121–130.

[2] T. Jaakkola, D. Sontag, A. Globerson, M. Meila, Learning Bayesian network structure using LP relaxations, in: Proceedings of 13th International Conference on Artificial Intelligence and Statistics (AISTATS 2010), Vol. 9, Journal of Machine Learning Research Workshop and Conference Proceedings, 2010, pp. 358–365.

[3] J. Cussens, Bayesian network learning with cutting planes, in: F. G. Cozman, A. Pfeffer (Eds.), Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI 2011), AUAI Press, 2011, pp. 153–160.

[4] J. Cussens, Maximum likelihood pedigree reconstruction using integer programming, in: Proceedings of the Workshop on Constraint Based Methods for Bioinformatics (WCB-10), 2010.

[5] W. L. Buntine, Theory refinement on Bayesian networks, in: B. D'Ambrosio, P. Smets, P. Bonissone (Eds.), Proceedings of the 7th Conferenece on Uncertainty in Artificial Intelligence (UAI 1991), Morgan Kaufmann, 1991, pp. 52–60.

[6] G. E. Schwarz, Estimating the dimension of a model, Annals of Statistics 6 (2) (1978) 461–464.

[7] H. Akaike, A new look at the statistical model identification, IEEE Transactions on Automatic Control 19 (6) (1974) 716–723.

[8] E. Brenner, D. Sontag, SparsityBoost: A new scoring function for learning Bayesian network structure, in: Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence (UAI 2013), 2013.

[9] M. Scanagatta, C. P. de Campos, M. Zaffalon, Min-BDeu and max-BDeu scores for learning Bayesian networks, in: Proceedings of Probabilistic Graphical Models 2014, Vol. 8754 of Lecture Notes in Artificial Intelligence, Springer, 2014.

[10] N. Sheehan, M. Bartlett, J. Cussens, Improved maximum likelihood reconstruction of complex multi-generational pedigrees, Theoretical Population Biology 97 (2014) 11–19.

[11] T. Silander, P. Myllymäki, A simple approach for finding the globally optimal Bayesian network structure, in: Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence (UAI 2006), AUAI Press, 2006.

[12] C. Yuan, B. Malone, Learning optimal Bayesian networks: A shortest path perspective, Journal of Artificial Intelligence Research 48 (2013) 23–65.

[13] C. de Campos, Q. Ji, Efficient learning of Bayesian networks using constraints, Journal of Machine Learning Research 12 (2011) 663–689.

[14] P. Parviainen, H. S. Farahani, J. Lagergren, Learning bounded treewidth Bayesian networks using Integer Linear Programming, in: Proceedings of the 17th International Conference on AI and Statistics (AISTATS), 2014, pp. 751–759.

[15] S. Nie, D. D. Mauá, C. P. de Campos, Q. Ji, Advances in learning Bayesian networks of bounded treewidth, in: Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, K. Weinberger (Eds.), Advances in Neural Information Processing Systems 27 (NIPS), Vol. abs/1406.1411, 2014, pp. 2285–2293.

[16] C. de Campos, Q. Ji, Properties of Bayesian Dirichlet scores to learn Bayesian network structures, in: AAAI-10, 2010, pp. 431–436.

[17] R. M. Karp, Reducibility among combinatorial problems, in: R. E. Miller, J. W. Thatcher (Eds.), Complexity of Computer Computations, Plenum, 1972, pp. 85–103.

[18] R. E. Gomory, Outline of an algorithm for integer solutions to linear programs, Bulletin of the American Mathematical Society 64 (5) (1958) 275–278.

[19] V. Chvátal, Edmonds polytopes and a hierarchy of combinatorial problems, Discrete Mathematics 4 (1973) 305–337.

[20] M. Bartlett, J. Cussens, Advances in Bayesian network learning using Integer Programming, in: A. Nicholson, P. Smyth (Eds.), Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence (UAI 2013), AUAI Press, 2013, pp. 182–191.

[21] T. Achterberg, SCIP: Solving constraint integer programs, Mathematical Programming Computation 1 (1) (2009) 1–41.

[22] D. L. Applegate, R. E. Bixby, V. Chvatal, W. J. Cook, The Traveling Salesman Problem: A Computational Study, Princeton University Press, 2006.

[23] W. Ben-Ameur, J. Neto, Acceleration of cutting-plane and column generation algorithms: Applications to network design, Networks 49 (1) (2007) 3–17.

[24] D. Johnson, Finding all the elementary circuits of a directed graph, SIAM Journal on Computing 4 (1) (1975) 77–84.

[25] M. Grötschel, M. Jnger, G. Reinelt, On the acyclic subgraph polytope, Mathematical Programming 33 (1) (1985) 28–42.

[26] B. Malone, K. Kangas, M. Järvisalo, M. Koivisto, P. Myllymäki, Predicting the hardness of learning Bayesian networks, in: Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI 2014), AAAI Press, 2014.

Integer Linear Programming for the Bayesian Network Structure Learning Problem

Mark Bartlett, James Cussens

Department of Computer Science, University of York, York, UK

Abstract

Bayesian networks are a commonly used method of representing conditional probability relationships between a set of variables in the form of a directed acyclic graph (DAG). Determination of the DAG which best explains observed data is an NP-hard problem [1]. This problem can be stated as a constrained optimisation problem using Integer Linear Programming (ILP). This paper explores how the performance of ILP-based Bayesian network learning can be improved through ILP techniques and in particular through the addition of non-essential, implied constraints. There are exponentially many such constraints that can be added to the problem. This paper explores how these constraints may best be generated and added as needed. The results show that using these constraints in the best discovered configuration can lead to a significant improvement in performance and shows significant improvement in speed using a state-of-the-art Bayesian network structure learner.

Keywords: Bayesian networks, Integer Linear Programming, Constrained Optimisation, Cutting planes, Separation

1. Introduction

Bayesian networks (BNs) use a directed acyclic graph (DAG) to represent conditional probability relationships between a set of variables. Each node

Email addresses: `mark.bartlett@york.ac.uk` (Mark Bartlett), `james.cussens@york.ac.uk` (James Cussens)

URL: `http://www.cs.york.ac.uk/~bartlett` (Mark Bartlett), `http://www.cs.york.ac.uk/~jc` (James Cussens)

in the network corresponds to one of the variables. Edges show conditional dependencies between these variables such that the value of any variable is a probabilistic function of the values of the variables which are its parents in the DAG.

While one can analytically create a BN from expert knowledge, there is considerable interest in learning Bayesian networks in which the relationship between the variables is not known. In this setting, multiple joint observations of the variables are first taken and then a BN structure that best explains the correlations in the data is sought. This is known as the Bayesian network structure learning problem. For any reasonably sized problem. the number of possible structures is far too large to evaluate each individually. Therefore a more intelligent alternative is needed.

In this paper, we tackle the Bayesian network structure learning problem using the score-and-search approach. Each possible parent set of each variable is first given a score based on the correlations between these variables in the observed data. A search algorithm is then used to determine which combination of these parent sets yields the DAG with the optimal overall score. As this search is an NP-hard problem [1], an intelligent search strategy is needed in order to efficiently optimise the BN structure for large numbers of variables.

The search for the best BN can be viewed as a constrained optimisation problem; select the parent sets for variables with the highest combined score subject to the constraint that these form an encoding of a DAG. Specifically, there are two constraints that must be respected. First, there must be exactly one parent set chosen for each variable. Second, there must be no (directed) cycles in the graph. Furthermore, it is possible to write the score which is to be optimised and both of these constraints as linear functions of binary variables, which means that the problem of learning the best BN can be formulated as an Integer Linear Programming (ILP) problem [2, 3]. Formulating the problem in such a way means that highly optimised off-the-shelf ILP solvers can be used and that decades of research in ILP optimisation can be used to improve the speed of the search for the optimal BN.

Encoding the constraint that there is only one parent set for each node is straightforward. However, the constraint that there must be no cycles in the network is relatively complex to encode as a linear inequality and can either be enforced through the introduction of auxiliary variables and constraints [4] or through an exponential number of *cluster constraints* [2]. Previous work has revealed that these cluster constraints perform better in practice and so the current paper focuses on this encoding. As there are so many of these cluster constraints, we do not add them all initially, but rather add them to the problem as additional constraints as needed. That is to say, we first solve a relaxed version of the problem in which most of the acyclicity constraints are not present. We then identify some acyclicity constraints violated by this solution and add them to the problem before resolving. This process is called *separation*, as the added constraints separate the relaxed solution from the space of valid solutions, and the added constraints are known as *cuts* or *cutting planes* as they cut off a portion of the search space containing the relaxed solution. This process repeats until the solution found does not violate any additional acyclicity constraints. By so doing, we typically eliminate the need for most constraints which rule out cycles to ever be explicitly represented in the problem and so increase the solving speed of the problem and simultaneously reduce the memory needed.

In addition to the constraints necessary to define the problem, there are additional implied constraints that can also be added. Doing so may lead to an increase in performance through further constraining the search space or may prove detrimental by increasing the number of constraints that need to be generated and processed at each step.

The contribution of the current paper is to examine several extensions to the existing ILP based method which relate to improving the constraints generated and added during the search. The first extension examines the method by which we search for acyclicity constraints to add, the second introduces additional implied constraints of a different form, and the third attempts to ensure that the constraints found by other methods rule out greater invalid regions of the search space. In addition, the impact of several solver features is assessed.

The rest of this paper is arranged as follows. In Section 2, the problem of Bayesian network learning is addressed in more detail before looking at using Integer Linear Programming for this task. A software platform to carry out this learning is presented in Section 3. The novel contributions of this paper are presented in Section 4 before evaluation of these techniques are given in Section 5. Finally, Section 6 concludes.

2. Background

2.1. Bayesian Network Learning

There are two classes of methods for learning the structure of Bayesian networks. The first takes advantage of the fact that the structure of the network encodes information about conditional independence. One can perform multiple conditional independence tests on subsets of variables and use this information to infer what structure the BN should have.

The alternative method, and the one followed in this paper, is *score-and-search*. In this method, for each node, one computes scores for each possible set of parent nodes for the node and then uses some search algorithm to attempt to maximise a global score formed from the local scores, subject to the resulting network being acyclic.

There are many scores that have been proposed for learning BNs, for example BDeu [5], BIC [6], AIC [7]. These scores have the property of local decomposability, meaning that the global score can be found as a simple function of the score associated with each node. In the current paper, we restrict ourselves to consideration of the BDeu score, though we note that the software presented has been used to learn networks based on other scores [8, 9, 10].

Having produced local scores for the possible parent sets of each node, it is necessary to perform a search for the network with the maximum global score. This can be performed using any search method. These can be divided into heuristic methods that produce a high scoring network but cannot guarantee to produce the best one, and global searches that not only find the best network but also establish that no better network is possible. The work presented in this paper falls into this latter category, alongside recent approaches such as dynamic programming [11], A* search [12] and Branch-and-Bound [13]. As the quality of the learned network is identical for all exact methods, the primary challenge in this case is to produce a search algorithm that runs sufficiently quickly and is sufficiently scalable. Another recent approach [14, 15] also uses Integer Linear Programming to find an optimal Bayesian network, but as this has the added constraint of bounded tree-width, this is not directly comparable to the results presented here.

If there are n nodes in the BN, then the number of possible parent sets for each node is 2^{n-1}. In practice, for even relatively modest n, this is much too large to even score each of them in a practicable time, and probably creates a search space that is too large to explore effectively. In most cases it is possible to show that many parent sets cannot occur in an optimal BN and so can be pruned [16]. This speeds up scoring considerably. However even after pruning there typically remain a very large number of candidate parent sets. To overcome this problem, one must limit the number of parent sets, typically by restricting the maximum size of parent sets for which scores are produced. This in turn limits the search function to only considering BNs in which all nodes have at most a limited number of parents. The networks found using any of the exact methods are therefore no longer globally optimal, but optimal under the additional introduced constraint that nodes have a maximum indegree. This may not be suitable for some applications where large indegrees are expected. In that case, a heuristic method which allows more densely connected networks at the expense of guaranteeing optimality may be more suitable.

2.2. Bayesian Network Learning as an Integer Linear Program

The task of learning the best BN structure given certain observations can immediately be seen to be an optimisation task. Somewhat less obviously, the problem can actually be encoded rather straightforwardly as an Integer Linear Programming (ILP) optimisation problem. That this should be possible should be no surprise given that the decision version of both ILP and BN learning are NP-complete [17, 1].

The major advantage of encoding the problem as an ILP problem rather than directly solving it using a BN specific algorithm is that doing so allows one to take advantage of the decades of research in ILP optimisation. In particular, off-the-shelf solvers have been highly optimised and tuned to give very efficient performance. In addition, any other arbitrary constraint that can be linearly encoded can be simply introduced in to the problem without having to modify the solving method. For example, based on external knowledge of the problem domain, one could easily add an extra constraint to assert that two nodes must have an edge between them in one direction or the other.

The variables in the ILP encoding represent whether or not a node has a given parent set in the network. For every possible node, v and parent set W, a binary variable $I(W \to v)$ is created which will be assigned the value 1 if W is the parent set of v in the BN or 0 otherwise.

Using this encoding, one must write the objective function to maximise as a linear combination of these variables. The BDeu score (as well as many other locally decomposable scores) defines the score of a BN to be the product of the scores of each of its nodes. However, taking the logarithm of this score turns this into a summation while preserving the ordinality of solution scores. One can then write the score to be optimised (log BDeu) as the following linear expression of the ILP variables, where $c(v, W)$ is the log BDeu score for v having W as parents.

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

Having defined the ILP variables and objective function, it simply remains to define the constraints that must be obeyed by the $I(W \to v)$ variables in order that they encode a valid network. There are two such constraints; each node must have exactly one parent set, and there cannot be any cycles in the network.

The first of these constraints can be written very directly as follows.

$$\forall v \in V : \sum_{W} I(W \to v) = 1 \tag{2}$$

The acyclicity constraint is much less straightforward to encode. Previously [4] considered two schemes to achieve this involving introducing additional auxiliary variables and constraints. In one method, additional binary variables were introduced which recorded whether node u appeared before v in a partial ordering of the tree, along with simple transitivity and non-reflexive constraints on these variables. In the other, generation variables were introduced for each node and the constraint added that parent nodes must have a lower generation value than their children.

Experience however has revealed that an alternative method of enforcing acyclicity, *cluster constraints*, introduced by [2], has superior performance in practice. Observe that for any set of nodes (a cluster), if the graph is acyclic, there must be a node in the set which has no parents in that set, i.e. it either has no parents or all its parents are external to the set. This can be translated into the following linear set of linear inequalities.

$$\forall C \subseteq V : \sum_{v \in C} \sum_{W : W \cap C = \emptyset} I(W \to v) \geq 1 \tag{3}$$

[3] generalised cluster constraints to *k-cluster constraints* by noting that there must be 2 nodes with at most 1 parent in the cluster, 3 nodes with at

most 2 parents in the cluster, etc. However, experiments have failed to reveal any consistent improvement in performance by using these constraints.

Having defined the objective function and the constraints, one can simply enter these into any ILP solver and, given sufficient time and memory, an optimal BN will be produced. However, in order to obtain the best performance, some understanding of the method used to solve ILP problems is needed.

For non-integer LP optimisation, the problem can be solved relatively efficiently. Imagine each variable as corresponding to a dimension; one can then represent any possible assignment to these variables as a point in this space, and a linear equality as a hyperplane in the space. The region containing the valid solutions will be the convex polytope bounded by these equations and an optimal solution can be found at some vertex of this polytope. This solution can be found relatively quickly using the well-known simplex algorithm.

For ILP, this simple algorithm is not in general sufficient. The optimal solution for which the variables are integer may not lay at a vertex of the polytope but may instead be inside the polytope. Therefore, for ILP optimisation a branch-and-bound approach is taken. First, the simplex algorithm is used to solve the *linear relaxation* of the ILP problem (i.e. the ILP problem with the constraint that certain values must be integers removed). If this yields a solution in which the variables happen to be integers, then the true optimum has been found. Otherwise, a variable which has a non-integer value in the relaxed problem is chosen ($v = x$) to branch on and a search tree with two subproblems formed; one in which $v \leq \lfloor x \rfloor$ and one in which $v \geq \lceil x \rceil$. In the case of the binary $I(W \rightarrow v)$ variables used in the BN learning problem, this corresponds to branching on whether some $I(W \rightarrow v)$ is 1 or 0, i.e. whether some v has W as a parent set or not. Each of these subproblems can be recursively solved in the same way, as part of a standard tree search algorithm.

A common extension to the branch-and-bound algorithm adopted by many ILP solvers is the branch-and-cut approach. In this approach, after the relaxed problem has been solved, additional linear constraints are added to the problem which separate the current relaxed solution from the space of any valid integer solutions, i.e. any integer solution will respect the constraint but the current relaxed solution does not. These constraints are known as cuts. Consider for example the simple case in which it is deduced that an integer $X \leq 7.2$. As we know X to be integer, we can always add a cut of

$X \leq 7$ as this will not remove any possible integer solution from the space, but does usefully remove the parts of the search space where X takes non-integer values between 7 and 7.2. The search for the relaxed solution and the addition of extra constraints alternates until no more cuts can be found. If the variables in the relaxed solution are now integer the problem is solved, otherwise the problem branches as in the branch-and-bound algorithm.

There are well known cuts that can be added in any domain based on deducing implied constraints from the problem and the current relaxed solution, for example Gomory cuts [18] or Chvátal-Gomory cuts [19]. Alternatively (or additionally), one can choose to hold back some necessary known domain-specific constraints from the problem, adding them as cutting planes only if a proposed solution violates them. This approach is adopted with the cluster constraints in the BN learning problem. For a network with $|V|$ nodes, there are $2^{|V-1|} - 1$ cluster constraints; rather than initially adding such a large number of constraints, they are only added explicitly to the problem as cuts if a relaxed solution would violate them. In practice, this means that most cluster constraints are never added to the problem as solutions with cycles involving that cluster of nodes are never proposed.

Adding problem constraints as cutting planes rather than initially is typically done when there are very large numbers of such constraints, as is the case with the cluster constraints. There are two reasons why this may be desirable. First, there may be considerable overhead for the solver in managing the constraints, many of which may never be needed. Second, large numbers of non-redundant constraints lead to a particularly complicated polytope with very many vertices. As the simplex algorithm works by repeatedly considering adjacent vertices, the simpler the polytope, the fewer neighbouring vertices there will be at each step and the fewer vertices there will be on the path between the initial vertex and the optimal one. The improvement in the speed with which the simplex algorithm runs though must of course be weighed against the fact that additional time will be needed to identify the violated constraints to add and the fact that the simplex algorithm must be run repeatedly, rather than once at each node of the search tree.

3. GOBNILP

To investigate BN learning using ILP. we have created the software program GOBNILP (Globally Optimal Bayesian Network learning using Integer Linear Programming) which is freely available and uses the ILP formulation and branch-and-cut method presented in Section 2.2 to learn optimal Bayesian networks [3, 20]. It builds on the SCIP framework [21] to perform the ILP solving along with a further solver for the underlying linear programming solving. In the experiments presented in Section 5, GOBNILP version 1.5 is used with SCIP version 3.1.0 and using CPLEX version 12.5 as the LP solver.

In addition to the basic ILP formulation presented in the previous section, GOBNILP has a number of additional features that improve solving time which are presented below.

3.1. Heuristic Algorithm

During the search process, it is useful to occasionally find sub-optimal heuristic solutions to the problem. This provides a lower bound on the value of the true optimal BN and can be used to eliminate sections of the search tree using branch-and-bound.

In addition, heuristic solutions can be used to turn the solving into an anytime algorithm; the solving can be halted at any point and a valid, but suboptimal solution returned. As the current relaxed LP solution is an upper bound on the optimal solution, the values of the heuristic solution and relaxed LP solution together allow one to see how much better than the current heuristic solution the optimal solution might be. Depending on the application, it may be acceptable to take a sub-optimal solution which is guaranteed to be within a few percent of the optimal one rather than wait much longer for the true optimal solution to be found. Close cuts, which will be introduced in Section 4.4 also rely on a heuristic solution being known.

SCIP features a number of built-in general purpose heuristic finding methods based on the current relaxed solution. For example, one could simply try rounding fractional variables to their nearest integer value and see if this is feasible. SCIP has 6 such *rounding heuristics* which we use to look for valid BNs. It should be noted that often trying to round a relaxed solution will fail to produce a valid BN. In such cases, no new heuristic solution will be produced.

We have also implemented a heuristic method specific to the BN learning application. The algorithm relies on the fact that it is easy to find an optimal BN given a total ordering of nodes. We therefore carry out what is in essence a greedy sink finding algorithm; we first choose a sink node (i.e. one that will have no children), then choose a node that will have no children except possibly the first, then a node that will have no children except possibly the first two, and so on. The algorithm is somewhat akin to the dynamic programming approach to learning Bayesian networks [11], except we commit greedy choices at each stage rather than consider all possible subsets.

To choose the ordering for the sink finding heuristic, we utilise both the scores of each of the $I(W \to v)$ in the objective function (i.e. their associated log BDeu score) and their value in the current relaxed solution. The motivation for using the former is that we wish to choose high scoring parent sets as far as possible, and the latter is used as we believe a good heuristic solution is likely to be found in the vicinity of the current relaxed LP solution.

First, for each node, we arrange the possible parent sets in descending order according to their log BDeu scores. That is to say, for each node, we create a list in which the first parent set is the one with the highest BDeu score and the last parent set is that with the lowest BDeu score. From this point onwards, we do not make further use of the BDeu scores, but rather choose our variable ordering according to the parent set lists just created and the current relaxed LP scores.

On the first iteration of the algorithm, we compute a score for the parent set at the head of each variables' list. This score is one minus the current value of that variable having that parent set in the current relaxed LP solution, i.e. $1 - I(W \to v)$. We then choose the variable with the highest score to be our sink and its best parent set to be its parent set in our heuristic solution. Following this, we remove from all the variables' lists any parent set which contains our chosen sink variable.

At each subsequent iteration, we carry out a similar process of scoring the best remaining option for each as yet unchosen variable, selecting the variable with the highest scoring parent set, and eliminating any remaining parent sets which contain this newly chosen variable. The only difference between the first iteration and subsequent ones is that we calculate the scores slightly differently. In these iterations, we use the sum of the values of all possible remaining parent sets for that variable minus the value of the best remaining parent set for the variable, where the values are the scores in current relaxed LP solution.

Should the algorithm at any stage try to select a parent set for a variable which is impossible (due to user supplied constraints or the current search tree branching choices made, for example), the algorithm simply aborts without returning a heuristic solution.

The sink finding algorithm is very fast, running 9425 times in only 30s in one case we studied. We therefore allow it run after every LP solution along with the built-in SCIP heuristics.

3.2. Value Propagation

Value propagation is a well-known constraint-based technique. Given assignments of values to some variables (such as happens when the problem branches), the constraints that exist between variables can be used to infer reductions to the feasible domains of other variables. As the ILP encoding here contains solely binary variables, this is equivalent to fixing variables to either 1 or 0, i.e. the specified parent set must be selected or cannot be selected respectively. SCIP features built-in propagators which can perform the correct inference for linear equations. However, for the constraint that there must be no cycles in the graph, an extra propagator is needed that will perform propagation based on the constraint as a whole, not just the currently added cluster cuts. GOBNILP includes just such a propagator which attempts to perform this fixing after each branching in the search tree. The propagation uses basic reasoning such as if A is in all remaining possible parent sets of B, then all variables in which B is in the parent set of A must be set to 0.

4. Cutting Planes

Section 2.2 explained how the search for an optimal Bayesian network using ILP required both branching and cutting planes. Despite various attempts, we have been unable to find a method of choosing variables to branch on which outperforms SCIP's default branching strategy. Furthermore, we note that final solution times for problems appear to increase substantially in general when the program finds it necessary to start branching early in the solving process. We therefore focus our attention on finding cutting planes to constrain the problem as much as possible before carrying out any branching.

This paper introduces cutting extensions to the method for learning Bayesian networks using Integer Linear Programming presented in Section 2.2. A major issue identified there was of identifying and adding cutting planes in an effective manner. As with many other aspects of ILP solving, SCIP features a number of built-in general purpose cutting plane finding algorithms some of which are examined in Section 5. In addition to these, the cluster constraints for ruling out cycles in the network are added as cutting planes. Second, new classes of constraints are introduced which any valid integer solution must obey. These form the complete set of tightest possible constraints for the problem with 3 or 4 nodes, but as with the original acyclicity constraints, the issue of which to add and when proves critical to their success in assisting with solving larger problems. Finally, the use of *close cuts* [22, 23] is assessed. These attempt to find cuts that separate more of the search space from the relaxed solution than usual methods.

4.1. Finding Cluster Cuts with a sub-IP

In the standard ILP formulation of the problem, cycles are ruled out through adding cluster cuts, which state that there should be no cycles formed from all elements of that cluster. In practice, it is unnecessary to add all cluster cuts as solutions to the relaxed problem usually do not violate many of these constraints. Even when a relaxed solution does violate a constraint, it may not be desirable to add it, as other constraints may be added that also rule out the relaxed solution.

We now explain how we find cluster cuts. First note that, due to (2), cluster constraints can be written as follows:

$$\forall C \subseteq V : \sum_{v \in C} \sum_{W : W \cap C \neq \emptyset} I(W \to v) \leq |C| - 1 \tag{4}$$

Intuitively, (4) says that not all nodes in the cluster C can have parents in C. Our goal is to find a cluster C such that the LHS of inequality (4) exceeds $|C|-1$ by as much as possible when the values of the current relaxed solution x^* are used for the variables $I(W \to v)$. We want to find a cut which x^* violates maximally since such cuts are 'deep cuts' leading to tight linear relaxations. We cast this problem as a sub-IP and solve it using SCIP as follows.

For each $I(W \to v)$ in the main problem with non-zero value in the current relaxed solution, create a binary variable $J(W \to v)$ in the cluster constraint finding subproblem. $J(W \to v) = 1$ indicates that the variable $I(W \to v)$ will be included in the cluster cut as formulated in (4). Also create binary variables $I(v \in C)$ which are 1 iff v is in the chosen cluster, Rather straightforwardly, we have for each $J(W \to v)$

$$J(W \to v) \Rightarrow I(v \in C) \tag{5}$$

$$J(W \to v) \Rightarrow \bigvee_{w \in W} I(w \in C) \tag{6}$$

where the first constraint states that $I(v \in C)$ must be 1 if $J(W \to v)$ is 1 and the second makes a similar assertion for the members of W. In other words, if $J(W \to v)$ is 1, v in the cluster along with at least one member of W. These constraints can be posted to SCIP directly as *logicor* constraints, along with a simple summation constraint that $|C| \geq 2$.

For reasons that will become apparent below, we set the objective function to maximise to $-|C| + \sum x^*(W \to v)J(W \to v)$, where $x^*(W \to v)$ is the value of $I(W \to v)$ in the current relaxed LP solution and $|C|$ is shorthand for $\sum_{v \in V} I(v \in C)$. We also use the SCIP function `SCIPsetObjlimit` to declare that only solutions with an objective greater than -1 are feasible.

It follows that any valid solution has

$$-|C| + \sum x^*(W \to v)J(W \to v) > -1 \tag{7}$$

Due to the constraints, for $J(W \to v)$ to be non-zero (and hence equal to 1), v must be in the cluster and at least one element of W must also be in C. So for any feasible solution (7) can be written as

$$-|C| + \sum_{v \in C} \sum_{W : W \cap C \neq \emptyset} x^*(W \to v) > -1 \tag{8}$$

equivalently

$$\sum_{v \in C} \sum_{W : W \cap C \neq \emptyset} x^*(W \to v) > |C| - 1 \tag{9}$$

So x^* violates the cluster constraint for cluster C and we have found a cutting plane. The sub-IP is always solved if possible, so if there is a valid cluster constraint cutting plane, the sub-IP is guaranteed to find it.

On a technical note, we solve the sub-IP using depth first search in the branching, and use any sub-optimal feasible solutions found during the search, as well as the final optimal solution. This means that the routine may sometimes produce multiple cluster constraint based cutting planes that we add to the main problem.

4.2. Finding Cluster Cuts through Cycles

In the previous section, a separate optimisation process is used to search for the cluster cut to add at each stage. The downside to such a strategy is that often only a single cut is added at each separation round whereas it may be more efficient to find several cuts at once.

We therefore outline a different method of identifying cuts to add, based on directly identifying any cycles in the graph encoded by the current relaxed solution and then adding cluster cuts ruling each of them out.

As the relaxed solution, by definition, has the integrality constraint on variable values relaxed, it may contain variables which have fractional values. A solution with fractional variables does not encode a graph structure as described in Section 2.2, therefore the first step must be to extract a graph from these variables.

Let $G = (V, E)$ be a directed graph, where V is the set of nodes involved in the BN. Construct E as follows.[1]

$$E = \left\{ (v_1 \to v_2) : v_1, v_2 \in V, \sum_{W : v_1 \in W} I(W \to v_2) = 1 \right\} \tag{10}$$

This graph is specified in terms of edges, rather than parent sets as in the main problem. An edge $A \to B$ exists in this graph if all parent sets of B with non-zero current LP solution value contain A. Intuitively, this graph is a 'rounded' version of the graph given by the current LP solution with the edges that are 'fractional' removed.

It is straightforward to extract the elementary cycles of G. In the current work, the method of [24] is used. This essentially performs repeated depth-first searches through the graph from each node, blocking any nodes in the current path to prevent paths with sub-cycles.

Having determined the cycles of this graph, one can simply take the set of nodes involved in each cycle and add a cluster cut to the problem for each of these sets. Any cluster cut involving all the nodes in any cycle of G will separate the current relaxed solution. However, the converse does not hold; there exist cluster cuts that separate the current relaxed solution which do not correspond to cycles in G.

Experimentation reveals that the time to identify the cycles of G can be significantly reduced if one only searches for cycles up to a given length. However, the trade off against the possibly reduced number of cycles, and hence the number of cuts, found must be considered. Experimental assessment of the effect of altering the maximum length of cycle to search for is presented in Section 5.2.

[1]In practice, $\sum_{W : v_1 \in W} I(W \to v_2) > 0.99$ is used to permit some tolerance of rounding errors.

As this cycle-based method of finding cuts may not find all valid cluster cuts for a given relaxed solution, it may be worthwhile to use it in conjunction with the sub-IP method of GOBNILP. The sub-IP method is guaranteed to find a valid cluster cut if one is possible but, as noted above, will only typically find a single cut per execution and involves significant overhead in initialising an optimisation process. Section 5.2 considers various methods of combining these two methods for searching for cluster cuts.

4.3. Convex Hull Constraints

It can be useful to think of two polytopes. The first $P_{cluster}$ is the region defined solely by the constraints given in Section 2.2, i.e. the polytope whose vertices are defined by the constraint on one parent set per node, the complete set of all cluster constraints and the upper and lower bounds on the variables. This polytope contains all the valid solutions to the BN learning problem, but many other points as well.

While the inequalities in Section 2.2 are sufficient to correctly classify all integer assignments to the variables as being a valid BN or not, they do not completely specify the smallest possible convex polytope for the problem.

Defining this second polytope, known as the *convex hull*, is desirable as it has the property that each vertex of the polytope is an integer solution. This means that the solution to the linear relaxation is the optimal integer solution and so the simplex algorithm can be used to solve the ILP without any branching needed.

Let us denote the convex hull to be the polytope P. This is contained within the larger polytope $P_{cluster}$. Adding cutting planes has the effect of removing some of the polytope $P_{cluster}$ that lies outside of P, thus reducing the solution space to search without removing any integer solutions. If we were able to add all possible valid constraints to $P_{cluster}$, we would find it reduced to P.

Given this observation, it is interesting to ask what the full set of constraints necessary to define P is. Given this information, we could simply add all these constraints (initially or as cutting planes) and find the BN through the simplex algorithm without branching.

It should be noted that the polytope P is conceptually similar to but distinct from the well studied *acyclic subgraph polytope* [25], P_{dag}, which also represents an acyclic graph as a set of binary variables. In the case of P_{dag}, the variables correspond to the existence of edges in the network, rather than the existence of a particular parent set as in the current work. Many facets of P_{dag} are already known but unfortunately this polytope is of little use in the current application as the scores used in Bayesian networks do not decompose into a linear function based on the existence of edges, but of whole parent sets.

The convex hull P appears to be extremely complicated in the general case, though we have empirically found the complete set of constraints defining the convex hull of the problem when the number of nodes in the network is limited to 3 or 4.

The convex hull of the problem with 3 nodes, P_3, was found using the lrs[2] algorithm. In this case, it transpires that P_3 consists of 17 facets; 9 lower bounds on the variables' values, the 3 limits on each node having one parent set, 4 cluster constraints, and a single extra constraint:

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

This can be generalised to give a class of *set packing* constraints which are valid for any Bayesian network learning problem, not just those with 3 nodes.

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

These will always be obeyed by integer solutions but otherwise acceptable fractional solutions exist which violate these inequalities. GOBNILP therefore adds such inequalities for all $|C| \leq 4$ to the initial formulation, which speeds up solution times.

As the number of nodes in the BN increases, the convex hull becomes more complex. For a BN with just 4 nodes, Matti Järvisalo used cdd[3] to show that there are 64 different facets to the convex hull, P_4. In addition to lower bounds on variable values, limits on the number of parent sets per node and cluster constraints, 7 different classes of facets were identified. We label these constraints *convex4* constraints. A simple generalisation of these constraints are still valid when applied to a subset of 4 nodes in a larger Bayesian network learning task. The problem of finding all facets of the convex hull for a BN of greater than size 4 is, to the best of our knowledge, unsolved.

[2] http://cgm.cs.mcgill.ca/~avis/C/lrs.html
[3] http://www.inf.ethz.ch/personal/fukudak/cdd_home/

Rather than continue searching for all facets of the convex hull for arbitrarily large BNs, we instead utilise those found for BNs with 3 and 4 nodes for subsets of the nodes in larger BNs. Inequality (11) generalises to Inequality (12) and such constraints are added to the initial problem for all subsets of nodes of size 3 and 4. For the convex4 constraints identified, for any reasonable large BN problem, there may be too many to add them all initially to the problem for each subset of 4 nodes. We can therefore add them as cutting planes during the search in much the same way as acyclicity constraints are added.

Adding these constraints initially may lead to better initial solutions. However, it may simply slow down the search by adding constraints that would never be violated if they were not added to the problem. In addition, searching for 4 node convex hull cuts which separate a relaxed solution is itself quite time consuming. As it is difficult to deduce whether these constraints should be added initially, as cutting planes, or not at all, we present experimental results in Section 5.3 assessing the best way to make use of them.

4.4. Close Cuts

The current solution to a relaxed problem will always be a point on the surface of the polytope corresponding to the current relaxed problem. It is possible that a cutting plane added to separate this solution will cut away a significant portion of the polytope, but it is also possible that it will pare away only a small region near the surface. In this latter case, the cut removes little of the solution space and may not assist significantly with finding a solution to the problem.

GOBNILP's usual strategy for dealing with this is to use the sub-IP to search for cutting planes which are efficacious, that is to say, cuts which pass deep into the polytope. However, this cannot be adapted for other methods of finding cutting planes, such as using cycle finding. An alternative to try to ensure arbitrary methods produce deep cuts is needed.

Close cuts [22, 23] are a solution to this problem. Despite the name, they are not actually a different type of cut, but rather they use the same cuts as normally used (general-purpose or domain-specific) but attempt to separate something other than the current relaxed LP solution. Rather than seeking a cut which removes a point on the surface of the polytope and hoping this is deep, close cuts pick a point which is deep in the polytope and then attempt to separate this. Specifically, a point is chosen which lies somewhere on a line between the best known heuristic solution to the problem and the current solution to relaxed problem, and the usual cutting plane finding methods are called to separate this point. Let the best known heuristic solution be $\mathbf{S}^h$ and the current relaxed solution be $\mathbf{S}^r$, then the point to be separated by the cutting plane routine is $\alpha\mathbf{S}^h + (1-\alpha)\mathbf{S}^r$ where the parameter $\alpha \in [0, 1]$ determines how close to the currently known heuristic solution the point to be separated is. As such, the point to separate is a convex combination of the current heuristic solution and the current relaxed solution.

There is no guarantee that the chosen point to separate is actually an invalid solution to the problem. In such a case, the cutting plane finding algorithms will fail to find any cuts, as there are no possible cuts. This does not present a logical problem; one can simply revert to finding separating planes for the solution to the relaxed problem as normal.

Clearly, the larger the parameter α is, the further into the relaxed problem polytope one is attempting to cut but, conversely, the more likely one is to choose a point that is a valid solution to the unrelaxed problem. Setting the value of α correctly is therefore crucial to the success of the technique. Experiments presented in Section 5.4 assess the best value for this parameter in the BN learning problem.

5. Results

To test the effectiveness of each of the aspects of adding cutting planes explored in Section 4, we incorporated them into the GOBNILP software and allowed their behaviour to be set through user supplied parameters. In some cases, for example close cuts, the required behaviour was already available through the SCIP framework.

Experiments were then conducted to assess the impact that each of these constraints had on the optimisation process. The presented methods will always find an optimum BN given sufficient time and computational resources. It therefore makes little sense to evaluate the score of the network found or the similarity of this network to the true one as this is simply evaluating how well optimisation of the score works; the same results would be found for all variants of the ILP technique, as well as for any other exact optimising technique such as [11] or [12]. Rather, the correct form of evaluation here is to analyse the time needed to find an optimum BN.

As the NP-hardness of the problem requires a limit on the size of the parent sets considered, our algorithm may not find the true most likely BN, as it may have larger parent sets than we permit. In cases where nodes with high in-degree are likely to exist, an alternative, heuristic method may be more desirable, which finds a high scoring though not guaranteed optimal network. Such issues, along with choice of scoring function are beyond the scope of the current paper which is restricted to finding the provably optimal BN from a choice of scored parent sets; the accuracy of the final network is solely reliant on the suitability of the scoring function chosen and the validity of assuming a maximum node in-degree.

The experiments presented are divided into two parts. First, a number of features of the solver which can best be described as on or off are examined. Based on prior experience, we believe them all to be generally useful [20]. We begin by turning them all on as a baseline and then conduct experiments in turning each of them off in turn while the others are all on. This provides an estimate of how much each feature helps or hinders the solving process. The features examined in this way are

- The use of Gomory cuts (see Section 2.2)
- The use of Strong Chvátal-Gomory cuts (see Section 2.2)
- The use of Zero-Half cuts (see Section 2.2)
- The sink-based heuristic (see Section 3.1)
- The value propagator (see Section 3.2)
- The use of set packing constraints (see Section 4.3)

It is likely that there are some interactions between these parameters. For example, for some reason the value propagator might turn out to be more effective when Gomory cuts are not being used. However, to assess all possible settings of each of these features together would require a prohibitively large number of experiments to be run, much of which would produce little practical insight; for example, it would be of little use to know if the value propagator was particularly useful when set packing constraints were not used if it turned out that using set packing constraints was always of significant benefit.

Following this set of experiments, the use of a number of extensions which are more parametric are assessed. These three extensions are the method by which cluster cuts are found, the point at which convex4 constraints are added and the use of close cuts. In the previous experiments, we fix their behaviour to using the sub-IP and no cycle finding for identifying cluster cuts, no use of convex 4 cuts and no use of close cuts.

In the second set of experiments, we began with a system in which all 3 extensions are fully disabled, then assessed the incrementally impact of adding each in turn. We first assess how changing the method for discovering cluster cuts to add impacts on the solution time. The best parameter settings discovered in this experiment are then used as a baseline for the convex4 experiments, studying how well adding these constraints alongside the new cluster cut mechanism works. Finally, the best settings for the cluster cuts and convex4 constraints resulting from this experiment are taken forward to the final experiment, in which the close cut parameter is varied.

One might reasonably expect the observed behaviour with different settings for each of these extensions to exhibit interactions with each other in non-trivial ways. The outcome of these experiments therefore cannot be viewed as finding the best parameter settings, though they do represent a greedy search for these best settings. Nevertheless, this scheme provides a reasonable balance between studying the individual effects of each of the extensions and attempting to capture some of the cross-extension interactions.

As an alternative, one could perform a search over the parameter space of all 3 extensions simultaneously, either in a systematic or heuristic manner. This would fully capture the interactions between the parameters associated with the different extensions, but would probably be infeasible to conduct due to the combinatorial increase in the size of the parameter space and would make it difficult to assess the individual impact of each of the extensions. At the other end of the scale, one could neglect the possible interaction between the various parameters entirely and evaluate each extension in turn with the other two disabled. The effects of each extension would be readily apparent but any results would be wholly artificial. Assuming more than one of the extensions to be beneficial, one would not run the system in this state and subsequently combining the best individual settings for each extension may lead to a severely suboptimal system due to previously unseen interaction effects.

Evaluation of the approaches was performed using a number of datasets drawn from Bayesian networks from commonly used sources. The datasets considered are shown in Table 1 and were chosen to test performance on a wide range of different problems. In each case, the BDeu score was computed (using the equivalent sample size listed in the table) external to GOBNILP

Name	Equivalent Sample Size	Number of Variables	Parent Set Limit	Number of Parent Sets
car	1	7	6	35
asia	10	8	2	127
insurance	1	27	6	341
mildew	1	35	3	3520
tic-tac-toe	10	10	3	112
flag	10	30	5	24892
dermatology	10	35	3	5059
hailfinder	1	56	4	4330
kr-vs-kp	10	37	2	12877
soybean-large	2	36	2	10351
sponge	1	46	4	11042
zoo	10	18	4	6461
alarm	1	37	4	8445
diabetes	1	413	2	4441
carpo	1	60	3	16391
lung-cancer	10	57	2	8294

Table 1: Characteristics of the problems studied. "Number of Variables" is the number of variables in the Bayesian network, while "Number of Parent Sets" corresponds to the number of ILP variables.

and this preprocessing time is not included in the reported results. Upper limits on the size of potential parent sets for which scores were calculated were introduced to make the times to compute the scores feasible. Note that the number of parent sets corresponds to the number of variables in the ILP problem. Some pruning of the number of parent sets is possible. For example, if A having a parent set of just B has a better score than A having both B and C as parents, we can prune the latter from the dataset, as there is no situation in which adding the additional parent C would be preferred in an optimal network. The number of parent sets reported in the table refers to the number remaining after this type of pruning has occurred.

Experiments were conducted on a single core of a 2.7GHz dual-core processor computer with 6GB of memory running Linux. All experiments used SCIP version 3.1.0 with CPLEX version 12.5 as the underlying LP solver. A time out limit of 2 hours was set for all experiments. Any experiments which had not terminated in this time limit have the gap between their best relaxed LP solution and their best known heuristic solution shown in the following tables, which provides a proxy for how much more work the search has remaining to do at that point.

As our intention is to evaluate various possible aspects of the ILP BN learning problem, we do not compare the algorithm to other approaches. As previously stated, comparing to heuristic methods or conditional independence methods assesses the quality of the reconstruction which is not our concern here. This factor depends hugely on the score chosen and the validity of the assumption that parent sets are of reasonably small size, the former of which is outside the scope of the current paper and the latter of which is problem dependent. Rather our aim is to assess the speed with which we can find a provably optimal network.

[26] provides an extremely thorough evaluation of the solving times for a number of recent optimal BN learners including GOBNILP on over 700 problem instances. Rather than repeat this exercise, we state the main findings of relevance here. The default configuration of GOBNILP was found to be fastest on over 300 instances, and when combined with various other configurations trialled, GOBNILP was fastest on the majority of problem instances. Various configurations of A*-search [12] were fastest on around 300 instances, but no configuration was fastest on many more than 100 instances. Branch-and-bound [13] is fastest on none. Overall, GOBNILP solves the entire dataset in 661,539 seconds, as compared to 1,917,293 seconds for the best A* configuration and over twice that time for Branch-and-Bound.

5.1. General Purpose Cuts and Solver Features

The results of turning various cuts or solver features off are shown in Table 2. Overall they show a mixed picture for many of the experiments, with everything being useful for some datasets and increasing solving times in others. On the whole Gomory cuts appear to be generally useful (i.e. the solving times rise when they are turned off). The picture for the other two cuts is less clear, with some big improvements seen in some datasets when they are used, but slower solving times witnessed in others.

For the solver features, a similar outcome is observed; some features are particularly helpful sometimes but hinder in other cases. The set packing constraints appear particularly useful in the mid-difficulty problems, but are less useful on the harder problems studied. The sink finding heuristic and value propagation appear more consistently to be useful but there are exceptions to this. Overall, the slowdown caused by not having a solver feature

Network	Baseline	No Cuts of Type			Without Solver Feature		
		G	SCG	ZH	SPH	SPC	VP
car	0.26s	0.01s	0.01s	0.01s	0.01s	0.15s	0.01s
asia	0.36s	0.35s	**0.97s**	0.34s	0.34s	**0.42s**	0.36s
insurance	0.83s	0.76s	**1.00s**	0.74s	0.81s	**1.47s**	0.81s
Mildew	1.20s	1.16s	1.16s	1.12s	**1.21s**	**2.03s**	**1.21s**
tic-tac-toe	9.40s	5.23s	9.18s	2.55s	9.26s	8.41s	9.30s
flag	39.99s	36.79s	17.77s	19.14s	36.09s	**70.85s**	35.95s
dermatology	32.17s	31.19s	21.16s	27.49s	31.63s	28.84s	29.74s
hailfinder	112.56s	79.23s	61.90s	87.56s	**118.97s**	**226.23s**	111.93s
kr-vs-kp	124.37s	80.64s	75.48s	71.91s	**125.81s**	96.75s	122.47s
soybean-large	98.41s	92.83s	**130.14s**	82.02s	89.90s	**110.38s**	97.14s
alarm	200.64s	**280.12s**	112.78s	**244.59s**	**201.50s**	108.71s	**227.45s**
Diabetes	—	—	—	—	—	—	—
sponge	195.03s	**225.24s**	**300.02s**	**231.95s**	**230.15s**	**214.46s**	191.36s
zoo	264.79s	**290.40s**	191.36s	166.54s	**266.65s**	174.74s	214.03s
carpo	483.69s	**528.10s**	**587.44s**	**513.89s**	**577.18s**	**589.40s**	**510.62s**
lung-cancer	670.76s	646.50s	651.57s	583.45s	670.66s	642.77s	627.88s

Table 2: Impact on time to find the best Bayesian network of various features. All times are given in seconds to the nearest whole second. "—" indicates that the solution had not been found after 2 hours. Key: G – Gomory cuts, SCG – Strong CG cuts, ZH – Zero Half cuts, SPH – Sink Primal Heuristic, SPC – Set Packing Constraints, VP – Value Propagator. Results that are worse than the baseline are indicated in bold.

appears to be greater than that for not having one of the cutting algorithms turned on.

5.2. Cycle Finding

The method for deciding which cluster cuts to add in the previous experiment was through a sub-IP. As well as considering simply replacing this with the cycle finding method, we also considered two schemes in which the two methods were both used. In the first, both methods were called every time cuts were searched for. In the second, the cycle finding algorithm was first used and only if this failed to find any cuts was the sub-IP method used. The motivation for considering running both methods is that the cycle finding algorithm only searches for cycles in the portion of the graph which corresponds to edges which are non-fractional in the current LP solution, whereas the sub-IP will also detect violated cluster constraints involving fractional variables. This means that the sub-IP can detect cuts to add that are a superset of those detected by the cycle finding method. This also explains why the idea of using the cycle finding only if the sub-IP failed was not considered.

In addition, the maximum length of cycle to look for by the cycle finding algorithm was investigated. The perceived trade-off here is between choosing a small value which risks missing many longer cycles and choosing a large value which takes a long time to run with possibly few additional cycles found. Preliminary experiments were used to identify a sensible range of values for this parameter over which the presented experiments were conducted.

These preliminary experiments also revealed that using just the cycle finding without the sub-IP was substantially worse than using them both together or just the sub-IP. For example, for some maximum cycle lengths it reached the time out limit on the Mildew problem (which other configurations typically solve in around a second) and consistently took over 5 minutes to solve the Flag problem, instead of about 30 seconds for other settings. The following therefore focuses attention on the techniques involving just the sub-IP, or the sub-IP and the cycle finding together.

The results shown in Tables 3 and 4 demonstrate a fairly consistent benefit from using the cycle finding method with the sub-IP, rather than the latter alone. The results for Diabetes are particularly significant, going from unsolvable in two hours with just the sub-IP to solvable with a wide range of settings when cycle finding was used as well; in the best case, a solution was found in under 30 seconds.

	Sub-IP	3	4	5	6	7	8	9	10	100
car	0.26s	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**
asia	0.36s	1.52s	2.66s	**0.23s**	1.47s	0.40s	0.38s	0.37s	0.38s	0.38s
insurance	0.83s	0.63s	0.60s	0.44s	0.44s	0.67s	**0.26s**	0.30s	0.39s	0.32s
Mildew	1.20s	0.97s	0.62s	0.43s	0.48s	0.44s	0.38s	0.37s	**0.35s**	**0.35s**
tic-tac-toe	9.40s	9.85s	9.64s	9.44s	**9.32s**	9.46s	9.36s	9.42s	9.88s	10.01s
flag	39.99s	27.56s	38.70s	27.18s	23.60s	**17.82s**	22.51s	24.54s	35.68s	32.28s
dermatology	32.17s	24.66s	30.92s	19.22s	**15.50s**	16.32s	19.80s	23.76s	19.98s	19.91s
hailfinder	112.56s	64.28s	61.00s	45.46s	94.18s	65.49s	45.39s	44.93s	44.38s	**44.24s**
kr-vs-kp	124.37s	78.32s	84.10s	62.66s	64.14s	81.18s	76.20s	**59.94s**	55.95s	80.66s
soybean-large	98.41s	104.42s	94.79s	**84.60s**	104.93s	128.98s	99.38s	126.13s	113.29s	101.23s
alarm	200.64s	118.53s	182.08s	**92.52s**	252.18s	224.47s	100.47s	148.26s	205.13s	142.27s
Diabetes	[8.88%]	[3.13 %]	28.04s	68.91s	126.22s	33.99s	39.58s	**23.57s**	35.05s	[3.45 %]
sponge	195.03s	235.87s	189.77s	266.40s	232.30s	199.99s	**175.03s**	207.15s	219.87s	236.56s
zoo	264.79s	171.19s	187.98s	199.36s	174.38s	169.66s	175.07s	207.95s	188.68s	**159.79s**
carpo	483.69s	604.97s	702.44s	605.55s	604.94s	519.70s	553.36s	458.73s	494.55s	**401.82s**
lung-cancer	670.76s	619.15s	**617.57s**	653.24s	625.52s	678.90s	673.10s	672.04s	653.10s	653.10s

Table 3: Times to find an optimal Bayesian network for various maximum cycle lengths when both cycle finding and the sub-IP are used. Results using the sub-IP only are shown for comparison. Percentages in square brackets are shown when the program ran out of time or memory, and indicate the remaining gap between the best discovered solution and the upper bound on the best possible solution at this point. The fastest configuration for each problem is shown in bold.

	Sub-IP	3	4	5	6	7	8	9	10	100
car	0.26s	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**
asia	0.36s	1.19s	0.53s	0.21s	0.87s	**0.16s**	**0.16s**	**0.16s**	**0.16s**	**0.16s**
insurance	0.83s	0.81s	0.46s	0.31s	0.33s	0.19s	**0.16s**	0.35s	0.23s	0.25s
Mildew	1.20s	0.93s	0.57s	0.52s	0.49s	0.58s	**0.25s**	0.26s	**0.25s**	**0.25s**
tic-tac-toe	9.40s	9.38s	9.43s	**9.36s**	9.42s	9.49s	9.48s	9.47s	9.51s	9.42s
flag	39.99s	28.70s	**17.52s**	31.83s	23.47s	19.45s	21.26s	51.73s	35.09s	44.67s
dermatology	32.17s	28.16s	21.32s	**17.17s**	21.70s	18.72s	20.58s	28.17s	18.42s	17.40s
hailfinder	112.56s	50.17s	63.57s	80.85s	51.98s	**40.24s**	41.50s	41.04s	41.79s	41.58s
kr-vs-kp	124.37s	82.81s	73.87s	97.66s	88.55s	84.21s	126.69s	**73.11s**	78.31s	76.31s
soybean-large	98.41s	115.07s	112.34s	**83.93s**	98.58s	97.49s	98.93s	96.81s	108.90s	97.92s
alarm	200.64s	**112.50s**	205.65s	194.60s	177.16s	286.12s	171.58s	187.60s	209.08s	222.24s
Diabetes	[8.88%]	[3.14 %]	3138.39s	1475.24s	66.68s	**18.17s**	172.92s	54.01s	18.61s	[3.45 %]
sponge	195.03s	193.05s	**189.49s**	212.11s	190.22s	262.17s	215.36s	217.67s	208.63s	226.65s
zoo	264.79s	160.25s	184.17s	165.67s	**136.23s**	205.22s	180.08s	165.40s	154.15s	175.48s
carpo	483.69s	724.91s	604.03s	607.85s	510.74s	**402.51s**	678.05s	611.93s	561.09s	531.79s
lung-cancer	670.76s	645.22s	697.98s	**638.32s**	641.71s	673.24s	697.99s	698.94s	716.60s	710.19s

Table 4: Times to find an optimal Bayesian network for various maximum cycle lengths when cycle finding is used followed by the sub-IP only if the former failed to find cuts. Results using the sub-IP only are shown for comparison. Percentages in square brackets are shown when the program ran out of time or memory, and indicate the remaining gap between the best discovered solution and the upper bound on the best possible solution at this point. The fastest configuration for each problem is shown in bold.

The best value for the maximum cycle length varied from dataset to dataset but either 5, 6 or 7 were usually amongst the best settings. In several cases, the difference between the best maximum cycle length and the worst could lead to a halving of the solution time. However, other datasets seemed somewhat insensitive to this parameter. This may reflect the density of the graph on which the cycle-finding algorithm was used; the more edges in the graph, the greater the increase in time that would be needed to search for long cycles though, conversely, the more chance of there being a longer cycle to find.

Using the sub-IP and cycle finding together produced broadly similar behaviour for both the methods studied. The better method depends on whether one considers all examples equally important or the longer ones to be more significant, and on the maximum cycle length chosen.

5.3. Convex4 Constraints

The constraints based on the convex hull of the 4 node BN polytope can be added to the problem initially or used as cutting planes. In the case where they are used as cutting planes, they can be searched for whenever a solution needs separating or only when no efficacious cluster constraints can be found.

In theory, one could consider adding each of the 7 classes of convex4 constraint in a different manner, but we restrict ourselves here to adding them all in the same way in order to make exploration of the space tractable.

For each option, the cluster constraint finding algorithm is fixed to a reasonably good setting as determined by the previous experiment. Specifically, we use cycle-finding initially with the maximum cycle length set to 6 and then call the sub-IP only if this fails to find a useful cluster constraint.

The results, as shown in Table 5, demonstrate a clear pattern amongst the different strategies explored. There are occasional exceptions, but the trend is for the best option to be not using the convex4 cuts or using them as cutting planes only when other efficacious cuts are not found. Overall, adding them as cutting planes in this way seldom does much harm and sometime leads to noticeable improvements.

For a couple of datasets, adding these constraints as cutting planes even when cluster constraints have been found proves very good. However this must be weighed against the fact that in many other case this strategy proves slightly detrimental compared to adding them only when cluster constraints are not found. Adding the constraints initially rather than as cutting planes almost never provides an improvement over leaving them out all together,

	As Cuts Always	As Cuts If Fails	Initially	Never
car	0.01s	0.01s	0.01s	**0.00s**
asia	**0.45s**	0.52s	1.55s	0.82s
insurance	**0.24s**	0.31s	0.32s	0.31s
Mildew	0.82s	**0.46s**	0.62s	**0.46s**
tic-tac-toe	**7.54s**	14.46s	8.36s	9.36s
flag	26.40s	23.31s	29.30s	**22.88s**
dermatology	23.59s	**21.64s**	29.38s	21.70s
hailfinder	76.14s	75.81s	69.88s	**50.61s**
kr-vs-kp	113.11s	101.42s	130.71s	**87.43s**
soybean-large	117.28s	110.79s	102.46s	**92.30s**
alarm	73.65s	**66.05s**	368.55s	175.06s
Diabetes	**55.67s**	67.09s	67.14s	67.16s
sponge	298.49s	257.75s	341.64s	**192.58s**
zoo	325.34s	156.95s	290.39s	**137.67s**
carpo	983.76s	653.13s	725.95s	**519.82s**
lung-cancer	**620.26s**	639.46s	658.59s	638.05s

Table 5: Times to find an optimal Bayesian network for various ways of adding the convex hull constraints. The fastest configuration for each problem is shown in bold.

and only provides a small help in those few cases. This suggests that any advantage they bring to tightening the problem is outweighed by the overhead of having to process all these additional constraints at each LP iteration.

5.4. Close Cuts

A single parameter associated with close cuts is studied. As explained in Section 4.4, α is the parameter that determines how far between the relaxed solution and the currently best known solution the point chosen for separation is. We investigate setting this parameter between 0.1 and 0.9 in increments of 0.1. $\alpha = 0$ corresponds to the case where the relaxed solution is separated (equivalent to close cuts not being used). It should be noted that the heuristic methods for finding valid BNs will also have an effect on how well close cuts work. If a heuristic were able to find the true optimum network consistently, one might expect that cutting near to this optimum (i.e. a large α) might lead to the search space near the solution being quickly pruned away and the problem solved.

As before, cycle finding with a maximum length of 6, followed by the sub-IP if necessary is used with the convex hull constraints added as cutting planes only when efficacious cluster cuts are not found.

The results in Table 6 illustrate that different values of α can have considerable impact on the solution times. In almost all cases, there exists a value of α that can improve the method over not using close cuts. However, there is no value of α that consistently outperforms the others. Furthermore, there is not even a general trend, for example towards larger αs being better or to the best α increasing as problem difficulty does. Worst of all, αs that are very good on one dataset are very poor on another dataset. A particularly extreme example is seen in the Flag dataset, where an α of 0.3 gives an answer in virtually half the time of not using close cuts, but if this α is increased or decreased by 0.1, the time is over twice that of not using close cuts.

Overall, it is correct to say that the use of close cuts can make a considerable improvement to the solving time. However, the choice of an α value that leads to this performance improvement cannot be deduced from these experiments. The ramifications of this result are returned to in the following section.

6. Conclusions

In this paper, the Bayesian network learning problem has been explored as an ILP problem. In particular, attention has been focused on the problem of finding and adding appropriate cutting planes to speed the solution times.

Various aspects were studied and some found to be generally beneficial across a range of problem instances. For others, impact was either modest or erratic on different instances.

Using cycle finding to determine cutting planes was highly effective when used in conjunction with a sub-IP approach. The impact of adding constraints based on the convex hull of smaller BNs was much less evident. When used in the best way, they improved performance more often than they degraded it, though for most datasets the difference in runtime was negligible.

The modest improvement associated with the additional classes of cut could call into question whether further work on additional cutting planes based on implied constraints truly merits attention. [20] showed that including implied constraints based on the convex hull of of a 3 node BN led to a measurable improvement, while [4] states inclusion of an implied constraint

	Off	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
car	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**	**0.01s**
asia	0.53s	0.35s	0.55s	0.61s	0.58s	0.54s	0.37s	**0.27s**	0.40s	0.58s
insurance	0.31s	0.28s	0.31s	0.29s	0.19s	0.19s	**0.18s**	0.27s	0.26s	0.19s
Mildew	**0.46s**	0.63s	0.66s	0.65s	0.55s	0.61s	0.63s	0.71s	0.54s	0.58s
tic-tac-toe	14.62s	10.37s	10.74s	10.12s	10.45s	15.19s	**9.99s**	10.65s	13.37s	11.37s
flag	23.10s	47.43s	62.26s	14.17s	64.02s	57.73s	15.62s	14.59s	**11.75s**	28.88s
dermatology	21.49s	21.62s	15.93s	26.38s	17.86s	**15.67s**	21.18s	22.59s	15.73s	19.42s
hailfinder	75.32s	127.84s	**41.08s**	89.22s	66.41s	56.34s	56.46s	55.36s	47.63s	58.83s
kr-vs-kp	104.46s	86.22s	62.98s	**53.07s**	66.08s	83.16s	68.31s	58.92s	64.35s	66.76s
soybean-large	110.96s	116.13s	95.38s	**79.96s**	98.80s	107.94s	91.28s	105.42s	116.81s	93.23s
alarm	66.38s	78.29s	70.62s	59.08s	48.64s	46.51s	70.90s	67.35s	**46.16s**	76.39s
Diabetes	67.46s	26.25s	**24.73s**	202.65s	58.37s	36.51s	28.94s	36.79s	48.31s	49.40s
sponge	257.89s	**163.28s**	320.99s	180.75s	222.62s	299.64s	326.30s	232.73s	239.47s	276.13s
zoo	**154.34s**	181.41s	203.72s	221.17s	177.98s	237.12s	227.44s	202.42s	199.60s	154.56s
carpo	663.75s	676.93s	663.79s	667.87s	622.24s	604.76s	**586.35s**	787.34s	620.65s	640.97s
lung-cancer	642.65s	635.13s	**592.56s**	726.35s	639.21s	618.50s	622.12s	710.86s	603.47s	605.03s

Table 6: Times to find an optimal Bayesian network for various values of close cut parameter α. Off is equivalent to $\alpha = 0$. The fastest configuration for each problem is shown in bold.

that at least one node has no parents led to a dramatic decrease in solving time. On the other hand, a generalisation of cluster constraints to k-cluster constraints proposed by [3] has failed to prove to be of any notable benefit. Clearly, different types of implied constraints vary vastly in their usefulness and further theoretical work is needed to understand why some are beneficial for this problem and others are not.

The results of the close cuts experiments are particularly interesting. Close cuts could almost always provide an improvement in solving time, but only for the correct value of the α parameter, and there was no consistently good value for this parameter across datasets. A similar observation could also be made for the maximum length of cycle to search for in the earlier experiment. Though the technique was of clear benefit, the solution times for similar maximum lengths varied quite considerably and no value was best across the whole dataset. These findings suggest that rather than fix the solver's method to some compromise 'best' configuration, a future approach may be to change the settings for individual problems. The issue then becomes predicting appropriate solver settings for a previously unseen problem instance. [26] provides a step in this direction. Based on very simple characteristics of a problem instance they are able to determine quite accurately which of two Bayesian network learning algorithms will be quicker. However, further work is clearly needed for this to be applicable here, where there are larger numbers of options from which to choose and where one might reasonably expect choosing from amongst various configurations of a single solver to be more complex than deciding between two entirely separate solvers.

Acknowledgements

This work has been supported by the UK Medical Research Council (Project Grant G1002312).

[1] D. M. Chickering, Learning Bayesian networks is NP-complete, in: D. Fisher, H. Lenz (Eds.), Learning from Data: Artificial Intelligence and Statistics V, Springer-Verlag, 1996, pp. 121–130.

[2] T. Jaakkola, D. Sontag, A. Globerson, M. Meila, Learning Bayesian network structure using LP relaxations, in: Proceedings of 13th International Conference on Artificial Intelligence and Statistics (AISTATS 2010), Vol. 9, Journal of Machine Learning Research Workshop and Conference Proceedings, 2010, pp. 358–365.

[3] J. Cussens, Bayesian network learning with cutting planes, in: F. G. Cozman, A. Pfeffer (Eds.), Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI 2011), AUAI Press, 2011, pp. 153–160.

[4] J. Cussens, Maximum likelihood pedigree reconstruction using integer programming, in: Proceedings of the Workshop on Constraint Based Methods for Bioinformatics (WCB-10), 2010.

[5] W. L. Buntine, Theory refinement on Bayesian networks, in: B. D'Ambrosio, P. Smets, P. Bonissone (Eds.), Proceedings of the 7th Conferenece on Uncertainty in Artificial Intelligence (UAI 1991), Morgan Kaufmann, 1991, pp. 52–60.

[6] G. E. Schwarz, Estimating the dimension of a model, Annals of Statistics 6 (2) (1978) 461–464.

[7] H. Akaike, A new look at the statistical model identification, IEEE Transactions on Automatic Control 19 (6) (1974) 716–723.

[8] E. Brenner, D. Sontag, SparsityBoost: A new scoring function for learning Bayesian network structure, in: Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence (UAI 2013), 2013.

[9] M. Scanagatta, C. P. de Campos, M. Zaffalon, Min-BDeu and max-BDeu scores for learning Bayesian networks, in: Proceedings of Probabilistic Graphical Models 2014, Vol. 8754 of Lecture Notes in Artificial Intelligence, Springer, 2014.

[10] N. Sheehan, M. Bartlett, J. Cussens, Improved maximum likelihood reconstruction of complex multi-generational pedigrees, Theoretical Population Biology 97 (2014) 11–19.

[11] T. Silander, P. Myllymäki, A simple approach for finding the globally optimal Bayesian network structure, in: Proceedings of the 22nd Conference on Uncertainty in Artificial Intelligence (UAI 2006), AUAI Press, 2006.

[12] C. Yuan, B. Malone, Learning optimal Bayesian networks: A shortest path perspective, Journal of Artificial Intelligence Research 48 (2013) 23–65.

[13] C. de Campos, Q. Ji, Efficient learning of Bayesian networks using constraints, Journal of Machine Learning Research 12 (2011) 663–689.

[14] P. Parviainen, H. S. Farahani, J. Lagergren, Learning bounded treewidth Bayesian networks using Integer Linear Programming, in: Proceedings of the 17th International Conference on AI and Statistics (AISTATS), 2014, pp. 751–759.

[15] S. Nie, D. D. Mauá, C. P. de Campos, Q. Ji, Advances in learning Bayesian networks of bounded treewidth, in: Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, K. Weinberger (Eds.), Advances in Neural Information Processing Systems 27 (NIPS), Vol. abs/1406.1411, 2014, pp. 2285–2293.

[16] C. de Campos, Q. Ji, Properties of Bayesian Dirichlet scores to learn Bayesian network structures, in: AAAI-10, 2010, pp. 431–436.

[17] R. M. Karp, Reducibility among combinatorial problems, in: R. E. Miller, J. W. Thatcher (Eds.), Complexity of Computer Computations, Plenum, 1972, pp. 85–103.

[18] R. E. Gomory, Outline of an algorithm for integer solutions to linear programs, Bulletin of the American Mathematical Society 64 (5) (1958) 275–278.

[19] V. Chvátal, Edmonds polytopes and a hierarchy of combinatorial problems, Discrete Mathematics 4 (1973) 305–337.

[20] M. Bartlett, J. Cussens, Advances in Bayesian network learning using Integer Programming, in: A. Nicholson, P. Smyth (Eds.), Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence (UAI 2013), AUAI Press, 2013, pp. 182–191.

[21] T. Achterberg, SCIP: Solving constraint integer programs, Mathematical Programming Computation 1 (1) (2009) 1–41.

[22] D. L. Applegate, R. E. Bixby, V. Chvatal, W. J. Cook, The Traveling Salesman Problem: A Computational Study, Princeton University Press, 2006.

[23] W. Ben-Ameur, J. Neto, Acceleration of cutting-plane and column generation algorithms: Applications to network design, Networks 49 (1) (2007) 3–17.

[24] D. Johnson, Finding all the elementary circuits of a directed graph, SIAM Journal on Computing 4 (1) (1975) 77–84.

[25] M. Grötschel, M. Jnger, G. Reinelt, On the acyclic subgraph polytope, Mathematical Programming 33 (1) (1985) 28–42.

[26] B. Malone, K. Kangas, M. Järvisalo, M. Koivisto, P. Myllymäki, Predicting the hardness of learning Bayesian networks, in: Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI 2014), AAAI Press, 2014.