Learning Bayesian Networks by Branching on Constraints

Thijs van Ommen T.VANOMMEN@UVA.NL
Informatics Institute, University of Amsterdam (The Netherlands)

Abstract

We consider the Bayesian network structure learning problem, and present a new algorithm for enumerating the k best Markov equivalence classes. This algorithm is score-based, but uses conditional independence constraints as a way to describe the search space of equivalence classes. The techniques we use here can potentially lead to the development of score-based methods that deal with more complex domains, such as the presence of latent confounders or feedback loops. We evaluate our algorithm's performance on simulated continuous data.

Keywords: Bayesian networks; structure learning; model selection; score-based methods; continuous variables.

1. Introduction

Learning a Bayesian network structure that explains the data well has been a topic of interest for a long time. The method we present here combines several desirable properties that have seen increased interest in the literature recently:

- It is an *exact* score-based method, meaning that it is guaranteed to find the network that is optimal according to some score. This contrasts it with heuristic methods, such as those based on local search, which may end up in a local optimum with a suboptimal score.
- It works on Markov equivalence classes rather than on individual Bayesian network structures. Bayesian network structures in the same Markov equivalence class impose the same set of conditional independence relations, and as a result, many scoring criteria will not distinguish between them.
- It can return the top k equivalence classes rather than just the single best equivalence class. If there is not enough signal in the data to distinguish clearly between different equivalence classes, it may be important to know about the equivalence classes that are competing with the top class. Having a list of good equivalence classes allows us for instance to compute an approximate Bayesian posterior over the structures, which can be used to answer many questions, and accompany those answers with measures of uncertainty; see e.g. (Tian et al., 2010). The combination of this point with the previous point is especially valuable: it means that the top k will not be polluted by many structures that are actually equivalent, but that each element in the list will represent new information.

Our method is an example of the branch-and-bound technique. This technique has been applied to the problem of Bayesian network learning before: see e.g. (de Campos and Ji, 2011; Tian, 2000) for two different approaches. While using the same technique, our method is novel: it uses a different branching operator and different score bounds. The branch-and-bound technique offers several useful advantages. The first and third properties of our method mentioned above come directly from this technique (the second property, namely working on equivalence classes rather than individual structures, is not inherent in the technique but is the result of our choice of branching operator). Other advantages are that the algorithm is *anytime* (it can be stopped early and still give an answer, including an upper bound to how much better the optimal answer could be), and parallelizable, so that additional computing power can be used if available to speed up the algorithm's execution.

In this article, we will focus on the linear Gaussian case rather than on the discrete case. An important difference for structure learning is that in the discrete case, the number of parameters grows exponentially with the number of parents of a node, while in the linear case it only grows linearly. This makes it hard for score-based approaches in the linear case to substantially bound the number of parents in advance (using results such as Corollary 3 by de Campos and Ji (2011)).

As the problem of Bayesian structure learning is know to be NP-hard (Chickering et al., 2004), we expect our method to be slow in the worst case. In particular, we expect that our method will be fast if there are few *faithfulness violations* in the data (these are conditional independences that (seem to) hold in the data, but are not implied by the network structure), but will be slowed down if faithfulness violations are common. By contrast, non-exact methods might be fast also under faithfulness violations, but would be much more likely to return a suboptimal answer. Thus by using an exact method like ours, we are giving up some computational efficiency in order to gain greater reliability of the results.

A major motivation for the present work comes from the problem of structure learning beyond Bayesian networks. This is of prime importance for causal discovery, where it is vital to distinguish correlations due to causal relations between observed variables from correlations due to latent confounders. Various classes of graphical models have been defined that can make this distinction. The task of structure learning on these models is typically very difficult, and the most common approach among score-based methods is local search. However, the possibility of wrong results due to local maxima makes any causal conclusions we would then draw less reliable. Another challenge is posed by feedback loops, where the causal relations violate the acyclicity requirement of Bayesian networks; these arise for example in biological cell signalling networks, an important area of application of causal discovery.

In these contexts, our algorithm has great potential for being extended to these learning problems, because it is defined in terms of constraints. For many of the graph classes relevant for causal inference, the constraints those graphs impose (which are not necessarily limited to conditional independence constraints) have been well studied; see e.g. (Richardson and Spirtes, 2002; Shpitser et al., 2014; van Ommen and Mooij, 2017; Forré and Mooij, 2018). Due to this good match, our algorithm is in an excellent position to be a starting point for new exact score-based structure learning methods on these graph classes.

2. Preliminaries

We write $v \perp w \,|\, S$ for d-separation of nodes v and w given the set of nodes S. For a DAG G, we will write $[G]$ to denote its Markov equivalence class, i.e. the set of all DAGs imposing the same set of conditional independence constraints (thus having the same d-separations) as G. If any conditional independence constraint imposed by $[G]$ is also imposed by $[H]$, we call $[H]$ a submodel of $[G]$, which we write as $H \prec G$. We write m_G for the number of edges in G; note that for all $H \in [G]$, $m_H = m_G$.

As our algorithm works on equivalence classes, we require the scoring criterion to be *score-equivalent*, meaning that it assigns the same score to all DAGs in the same equivalence class. Another property that is often required of the scoring function is that it is *decomposable*: it can be written as a sum of scores over the nodes, with each node's score determined by the variables associated with that node and its parents. While our method could in principle be used with nondecomposable scores, this would likely make it less efficient, so we will only consider decomposable scores here.

3. The Algorithm

In this section, we will first give an overview of branch-and-bound as a general search technique. Then we will explain our instantiation of it to the problem of learning Bayesian networks.

3.1 Branch-and-Bound

The branch-and-bound search technique is based on the following idea. Suppose we wish to find the element in some set that maximizes some score. Then start with a single *state* that is just the entire set of candidate solutions. The *branch* operation takes a state and splits it into two new, disjoint states whose union is equal to the original state. Further, we need to be able to compute an upper *bound* for the scores in any state. If a state consists of just one element, we will use its actual score as the upper bound. We apply these operations until we are in a situation where we have found an element whose score is larger than the upper bound of each other state. Then we can conclude that this element maximizes the score, without having explicitly computed the score of many individual other elements.

It is straightforward to extend this technique to find not just the single maximizing element, but the top k elements. Define the *threshold* as the score of the kth-best class we have seen, or as $-\infty$ if we have seen fewer than k different classes so far. Now we will continue to branch states until all remaining states have upper bounds below this threshold. Similarly, we might want to find all elements whose score is at most Δ below the maximum score. This can be accomplished by taking the threshold equal to the largest score we have seen, minus Δ.

We will primarily use the *best-first* version of branch-and-bound: in each iteration, we will pick the state with the largest upper bound and split that state into two new states. This approach has the advantage that it leads the algorithm to terminate in as few branches as possible: the state it chooses to branch is one that would need to be branched in any ordering of branchings, because as long as it is not branched, the overall upper bound will be larger than the threshold. A disadvantage may be that we are only working on upper bounds, and so it may take a long time until we start seeing actual solutions, which is important if we want to be able to terminate the algorithm early and still get acceptable results.

3.2 Learning Bayesian Networks: States, Branching, and Bounding

In general, branch-and-bound works on states; in our case, a state is a set of equivalence classes. Our branch operation splits a state into those classes that impose some conditional independence constraint, and those that do not. For any state resulting from these branch operations, we can describe which equivalence classes are in that state by listing the constraints that each of these classes must impose, and the constraints none of them impose. In the context of a single branching operation, the state we start with will be called the *original state*, and the resulting states after the operation will be called the *constrained* state and the *remaining* state respectively. All states must be nonempty.

The other ingredient we need is a way to upper bound the scores of all classes in a state. (We will assume w.l.o.g. that the score must be maximized.) To bound the score $f([G])$, we will decompose it into $f([G]) = g([G]) - h([G])$, where g and h are both monotonically increasing: $G \prec H$ implies $g(G) < g(H)$ and $h(G) < h(H)$. Many commonly used scores can be decomposed like this in a natural way.[1] For example, the BIC score (Schwarz, 1978) decomposes into an increasing log-likelihood term minus an increasing penalty term.Then an upper bound on the likelihood g and a lower bound on the penalty term h together give the required upper bound on the overall score f.

For establishing an upper bound on g, it will be helpful if a state contains a greatest element $[G]$ (i.e. for any other class $[H]$ in the state, we have $[H] \prec [G]$). Then each class in the state will have $g([H]) \leq g([G])$, so that $g([G])$ is the upper bound we need. If states could have multiple maximal classes, then to get a bound on g, we would need to compute values of g for each such class (computationally costly), or else bound their scores in some other way, e.g. by the value of g for a class outside the state, which would lead to less tight bounds.

By avoiding certain branching operations, we can ensure that all states we encounter have a greatest element. We will only consider branches of the form specified by the following proposition.

Proposition 1 *Let $[G]$ be the greatest element in the original state. Consider branching on a constraint such that the constrained state that would result from the branch will contain at least one element with $m_G - 1$ edges. The states resulting from this branching will both have a greatest element if and only if the constrained state would consist of exactly one class $[H]$ with $m_H = m_G - 1$ edges, and of all classes from the original state that are subclasses of $[H]$.*

Proof The only way for the remaining state to not contain the class $[G]$ would be by branching on a constraint that leaves the remaining state empty, which we do not allow. Thus the remaining state will always still contain the class $[G]$, and so will have a greatest element regardless of the choice of branching constraint.

Clearly, if the constrained state consists of $[H]$ and its subclasses, $[H]$ will be the constrained state's greatest element. For the other direction: If the constrained state contains another class besides $[H]$ with $m_G - 1$ edges, then $[H]$ is not a greatest element. A constrained state containing $[H]$ could not contain fewer other classes than specified, as the constraint we branch on is imposed by $[H]$ and hence by all its submodels. If the constrained state contains classes that are not subclasses of $[H]$, then $[H]$ would not be a greatest element, contradicting our requirement. ■

We say a branching operation is *well-behaved* if it satisfies these conditions (i.e. if for some $[H]$ with $m_H = m_G - 1$, the constrained state consists of $[H]$ and its subclasses). The requirement that $m_H = m_G - 1$ is a reasonable extra restriction: without such a restriction on $[H]$, the task of choosing from all possible branching operations would become as hard as solving the learning problem in one step.

From a given original state, there may be several constraints that all define a well-behaved branching operation with the same greatest element $[H]$. Then we see that all those constrained

1. In fact, such a decomposition is possible for any score by taking $g([G]) = f([G]) + Mm_G$ and $h([G]) = Mm_G$ for sufficiently large M; however, the algorithm will be efficient only if g and h can be used to give reasonable bounds on the score of sets of classes.

states must in fact be equal, so which of those constraints we choose does not matter. A convenient choice is based on the Delete operator defined by Chickering (2002). This operator transforms an equivalence class into any of its subclasses with one fewer edge, which is exactly what we do to move from $[G]$ to $[H]$. Specifically, provided it satisfies some conditions, the operator $\mathrm{Delete}(x, y, Z)$ removes the edge between x and y, and orients all edges between x or y and a node $z \in Z$ towards z.

Using the theory developed by Chickering (2002), we can generate a list of all valid Delete operators, which tells us all the possible well-behaved branching options available to us. Additionally, using Corollary 18 from that paper, we can find the difference in scores between $[G]$ and $[H]$ for any decomposable scoring function. For a given operator $\mathrm{Delete}(x, y, Z)$, this gives us a natural canonical choice of constraint on which to define our branch, from the set of constraints that lead to the same split: we take the unique constraint imposed by $[H']$, where $[H']$ is the class obtained by applying the $\mathrm{Delete}(x, y, Z)$ operator to the complete graph. This is the constraint $x \perp y \,|\, \mathrm{Pa}_y \cup \mathrm{NA}_{y,x} \setminus Z$, where Pa_y are the parents of y, and $\mathrm{NA}_{y,x}$ are the nodes adjacent to x and having an undirected edge to y (*neighbours* of y).

3.3 Representation of States

Before we turn to the topic of establishing a lower bound on h, we will briefly turn to the question of how a state can be represented during the operation of the algorithm. It is clear from the above that the greatest element $[G]$ plays an important role here, as having a representation of $[G]$ allows us for example to find all valid Delete operators. Further, to describe a state, we need to know the list of constraints its classes impose. Because $[G]$ is the greatest element, it contains all this information.

To represent $[G]$, we can use a PDAG or completed PDAG (CPDAG); see e.g. (Chickering, 2002). Because several operations we perform on states require the CPDAG, we will store that as part of the state representation.

This takes care of the constraints imposed by all classes in a state. To complete the description of a state, we also need to know which constraints may *not* be imposed by any of its members. We can not in general represent this 'bottom' of a state by a single class as for the top of a state. For example, on three nodes a, b, c, after two well-behaved branches, a remaining state may exist that consists of all equivalence classes for which $a \not\perp b$ and $b \not\perp c$. This state contains no classes with fewer than two edges, but four classes with two edges (and the saturated class with three edges). Thus, to represent the bottom boundary of a state, we will keep a list of all d-connections required in its members.

The list of valid Delete operators we obtain from $[G]$ does not take into account that some subclasses of $[G]$ may not actually belong to the state, because they have been ruled out by a required d-connection. Let $[H]$ be the result of applying a given Delete operator to $[G]$. Then we can use this operator to branch if and only if $[H]$ is a member of the current state, which is the case if and only if $[H]$ obeys all required d-connections.

3.4 Lower Bounds on the Penalty Term

To get a lower bound on h, we want to know the smallest penalty among all classes in the state. The score we are primarily interested in is the BIC score for linear Gaussian models. The BIC penalty is determined by the number of parameters of a model. In the linear case, the number of parameters is itself determined by the number of edges, and so will be equal for those equivalence classes in a state that have the same number of edges. Hence we do not pay a large price for allowing states that do not have a least element in the $\prec$ ordering.

The computational problem we are concerned with is, given a representation of a state, to find the smallest number of edges among all classes in that state. We did not find an efficient algorithm for this task in the literature, so we will use a lower bound instead. If this bound is slack, the algorithm may waste time on states that look more promising than they really are, but its output will still be correct.

Here we sketch an approach to this problem, based on two ideas. First, suppose that for all classes in a state, there is an edge between two nodes v and w. We will call this a *required adjacency*. The number of required adjacencies is a lower bound for the number of edges in any class in the state. We refer to Appendix A for more details of how required adjacencies can be determined.

The above works well when many d-connections are enforced in a state, or when $[G]$ is already fairly sparse so that few branching operations remain. When the algorithm starts out, $[G]$ will be a complete graph and no d-connections are enforced yet, so it would take many branching operations from there before we could say anything about the minimum number of edges in a state. To improve the lower bound in the earlier stages of the algorithm, we use the following observation. If even a single required d-connection exists between two nodes that are not connected via a path of required adjacencies, then at least one more edge will be needed, in addition to the edges covering required adjacencies, to satisfy this d-connection. In fact, each edge we add will reduce the number of connected components in a graph by at most one, and the number of connected components must be brought down to the number of connected components from required d-connections in order to satisfy all of them. So as lower bound on the number of edges, we use the number of required adjacencies plus the difference in numbers of these two types of connected components.

3.5 Branch-and-Bound Equivalence Search

Pseudocode for the algorithm is given in Algorithm 1. We have called it Branch-and-Bound Equivalence Search, because it is similar to Greedy Equivalence Search (Chickering, 2002), but using branch-and-bound instead of a greedy approach. Some brief remarks about its implementation are mentioned here. First, we can use a cache to store scores that have been previously computed, to avoid computing them again. The question of how to choose a constraint to branch on will be investigated in the next section. Finally, whenever a new state is enqueued, its bound must be computed, which involves computing the likelihood of that state's greatest element. That makes it a convenient time to compare this class to the current list of results and add it if it is good enough (increasing the threshold for further classes to be added to the list). A different implementation could do this at a different time.

4. Experiments

We performed several experiments where we searched for the equivalence class with the highest BIC score (Schwarz, 1978), with linear Gaussian models.[2]

The performance of the branch-and-bound algorithm will depend heavily on the way in we decide which of potentially many branches will be performed. To study the behaviour of many different branching heuristics, we implemented a version of the algorithm that first precomputes

2. The code for reproducing these results is available online at `https://github.com/caus-am/bbes`.

Algorithm 1: BBES

```
Input: Data $\mathcal{D}$, number $k$
Output: A list Res consisting of the $k$ equivalence classes with the highest scores on $\mathcal{D}$
Let S be the state consisting of all classes;
Add the greatest element of S to the list of results Res, and update threshold accordingly;
Create a priority queue Q containing only the state S;
while Q is not empty do
    Let S be the state in Q with the largest upper bound, and remove it from Q;
    If the upper bound of S is below the threshold, exit the while loop;
    if S contains two or more classes then
        Pick a conditional independence constraint constraint corresponding to a valid
          Delete operator on the greatest element of S and such that the class resulting from
          applying the operator is also in S;
        Create a new state T consisting of those classes in S that impose constraint;
        Consider the greatest element in T as a solution, updating Res and threshold;
        Let R be S $\setminus$ T;
        Add T and R to the queue Q;
    end
end
```

the entire search space. This is feasible up to $n = 6$, where the number of equivalence classes is 1,067,825 (Gillispie and Perlman, 2001). This allows us to represent a state as a vector of that many bits, indicating for each equivalence class whether it is included. This representation effectively provides an oracle for queries that would otherwise be hard to answer, such as for the exact minimum penalty among all classes contained in a state, or for the number of classes in the state having that number of parameters. This way, we can easily test whether such quantities are valuable as part of a branching heuristic, before putting effort into approximating them in an implementation of the algorithm that does not have access to a precomputed search space.

For the following experiments, data was generated from linear Gaussian Bayesian networks randomly chosen as follows. First, independently for each pair of nodes with $v < w$, an arrow is added from v to w with probability p. Then the nodes are shuffled using a uniformly chosen permutation. The edge coefficients are sampled independently from the standard Gaussian distribution, and the variances of the noise terms from $\Gamma(1/2, 1/5)$. Finally, N data points are sampled from the distribution defined by this linear Gaussian model. Each heuristic was evaluated on the same sequence of datasets. The results are displayed in Table 1 and 2.

As the tables show, picking a branch at random gives much worse performance of the algorithm than other choices. This supports our claim that a good heuristic is a deciding factor for the algorithm's performance.

Of the branching heuristics that look only at the maximum log-likelihood of the constrained state, the best performance is obtained by taking the branch for which this quantity is smallest (heuristic 's'). An explanation is that taking a branch for which this quantity is larger will create two new states that will typically both have high priority: the constrained state will have a lower likelihood but this will be a small difference, and reductions in priority due to changes in penalty are usually small. This heuristic avoids fragmenting the search space like this as much as possible.

	$N = 100$			$N = 10000$		
Heuristic	$p = .2$	$p = .4$	$p = .8$	$p = .2$	$p = .4$	$p = .8$
random	1655.5	2976.4	3843.5	546.9	1278.6	990.3
$-\|s - s_{-1}\|$	1931.3	1282.7	586.8	204.0	210.3	241.2
s	183.2	221.5	259.9	93.4	158.5	231.4
$(s < s_{-1}, -\|s - s_{-1}\|)$	867.7	575.0	349.0	182.0	189.6	234.7
$-s$	13673.3	24496.2	24970.3	6824.7	14008.5	3835.3
$(\|S\|, s)$	781.7	1398.4	1652.6	242.2	603.3	551.4
$(-\|S\|, s)$	335.4	683.4	1410.8	115.2	429.4	687.0
(b, s)	169.0	407.0	703.8	63.2	259.9	420.1
$(b, \max(s,t), \|S\|)$	163.1	370.0	674.4	67.9	230.6	407.4
$(\lfloor \max(s,t)/\delta \rfloor, b, s)$	131.4	170.5	225.9	83.5	150.3	227.8
$\max(s,t) + \delta b$	105.4	145.7	202.7	79.7	145.2	225.8
$\max(s,t) + 2\delta b$	95.3	134.3	189.2	78.0	142.6	222.3
$(b, \max(s,t), \|S\|)$*	162.7	370.1	674.5	67.9	230.6	407.4
$(\lfloor \max(s,t)/\delta \rfloor, b, s)$*	115.5	155.1	204.0	68.1	114.0	194.9
$\max(s,t) + \delta b$*	94.4	131.7	184.4	46.2	80.2	183.5
$\max(s,t) + 2\delta b$*	**86.4**	**124.6**	**177.0**	**34.5**	**65.2**	**165.1**

Table 1: Number of branching operations performed. The heuristics are specified by sort keys, and will select the branching operation with the smallest key (this representation is used because it gives fewer minus signs). Tuples are compared lexicographically, and for booleans, true comes before false. The quantities used are: s is the upper bound of the new constrained state taking into account its likelihood but not any changes in the number of parameters; s_{-1} is the bound of a fictional state with the same likelihood as the original but one more parameter; δ is the difference in score from having one more parameter (this is a constant for BIC); t is the threshold score; S is the conditioning set of the branching constraint; and b is the fraction of classes with the minimum number of parameters in the original state that are still in the remaining state. A star denotes that the algorithm looked at small classes in a state to possibly improve the threshold. Each entry in the table is an average over 100 datasets; entries with the same N and p were evaluated over the same sequence of 100 datasets. The best entry in each column is bolded.

The second phase of Greedy Equivalence Search (Chickering, 2002) also works by repeatedly picking a constraint of the same type as we look for, and 'moving down' in the search space to the class with one fewer edge that imposes this constraint. GES picks the constraint for which the resulting log-likelihood will be *largest*. An important difference is that GES does not allow backtracking: it keeps no record of the set of classes that have become unreachable by taking this step. The prime consideration is thus to avoid imposing a constraint that is not imposed by the optimal class. As we can see in the table, GES's strategy of picking the largest log-likelihood (heuristic '$-s$') is not a good choice at all for our branch-and-bound algorithm, as it maximally fragments the search space.

	$N = 100$			$N = 10000$		
Heuristic	$p = .2$	$p = .4$	$p = .8$	$p = .2$	$p = .4$	$p = .8$
random	629.97	859.37	725.40	212.46	309.55	85.75
$-\|s - s_{-1}\|$	938.17	556.72	171.42	90.81	48.74	12.03
s	50.08	39.55	25.12	15.34	11.53	4.67
$(s < s_{-1}, -\|s - s_{-1}\|)$	454.71	250.51	82.88	76.36	35.97	8.40
$-s$	5594.77	7779.62	4863.41	2855.38	3956.40	352.58
$(\|S\|, s)$	246.64	360.90	309.27	73.67	135.47	42.03
$(-\|S\|, s)$	88.39	121.64	153.00	31.40	61.49	28.91
(b, s)	68.18	112.16	120.20	26.83	58.38	26.90
$(b, \max(s,t), \|S\|)$	66.48	105.82	117.91	28.11	55.28	25.82
$(\lfloor \max(s,t)/\delta \rfloor, b, s)$	38.96	30.37	19.74	14.57	10.86	4.51
$\max(s,t) + \delta b$	**35.38**	**27.34**	**18.18**	**14.44**	**10.60**	4.46
$\max(s,t) + 2\delta b$	35.72	27.76	18.77	14.50	10.84	**4.43**
$(b, \max(s,t), \|S\|)$*	66.36	105.78	117.74	28.10	55.28	25.85
$(\lfloor \max(s,t)/\delta \rfloor, b, s)$*	40.87	33.08	21.78	14.87	11.70	5.00
$\max(s,t) + \delta b$*	37.70	29.83	20.02	14.77	11.46	4.78
$\max(s,t) + 2\delta b$*	37.75	31.15	20.92	15.65	13.36	5.41

Table 2: Number of states that were visited: taken from the queue at least once. This table is structured as Table 1.

We experimented with several heuristics that also look at other quantities. Some algorithms, such as PC and FCI (Spirtes et al., 2000), start with testing conditional independences for small conditioning sets S. However, our results suggest that this can have a significant negative impact on the performance of our algorithm. Curiously, of the two options, results were better when starting with *large* sets S. This should not be taken as a recommendation, as larger sets have important drawbacks: the score differences are more expensive to compute, and typically less reliable.

Besides the likelihood, the main consideration that should affect the choice of branching operator is the bound on the penalty. We observed that the heuristics that only look at the likelihood may often spend a long time on a single state, repeatedly branching off using a constraint that has a large effect on the likelihood. But as long as the bound on the penalty does not change, the remaining state will again be at the top of the queue for the next iteration. Heuristics that look at the parameter b are trying to choose branches that will lead to a refinement of the remaining state's bound on the penalty term. The heuristics that put this consideration first did not perform very well.

The best results overall were obtained by a variety of similar heuristics that look primarily at s, but use b to decide between branching operations for which the difference in s is small. These heuristics use the term $\max(s,t)$ (where t is the threshold, defined in Section 3.1 as the minimum score a class must have in order to be included in the set of results). This term reflects that the precise bound of the constrained state is irrelevant if it is below the threshold t.

The heuristics that depend on the value of the threshold t may suffer from the fact that a pure best-first branch-and-bound strategy does not put any particular effort into improving the value of the threshold. Therefore, we tested a version of the algorithm that, as part of each branching operation, looks at an arbitrary class with the minimum number of parameters from the remaining state, and adds it to the result if it meets the threshold. Especially if this state has no branching options that result in large drops of the log-likelihood, it will often be the case that these classes with the minimum number of parameters are the best scoring classes in the state. In that case, this strategy will allow the threshold to be improved much faster than otherwise. In terms of the number of branching operations, the performance of the algorithm benefits from such updates to the threshold if a heuristic is used that depends on the threshold. However, the opposite is true for the number of states visited in Table 2.

These experimental results will hopefully provide a useful guide for the development of heuristics that do not have access to an oracle. In particular, the value b proved to be valuable in our experiments. If a heuristic does not have access to the value b, it could instead prioritize (sets of) branching operations that will lead to a change in the lower bound on the penalty term. This is expected to lead to similar behaviour of the algorithm.

5. Discussion and Conclusion

If the best class is sparse, we might be able to find it faster by not trying out all available branches from the start. We could instead, for instance, initially only evaluate the log-likelihoods that results from imposing a constraint $v \perp w \mid S$ for $|S| \leq k$ for some k. If many branching options are already found this way that would be good choices according to the branching heuristic, then we can pick one of those; otherwise, we increase k and evaluate additional branching options. As long as we allow the algorithm to consider and take such branches eventually, we are still guaranteed to find the true optimal class. This trades off the amount of work done with the quality of the chosen branch, and further experimentation would be needed to find a good way to do this.

While our Python implementation is currently not fast enough to deal with much larger numbers of variables, our main vision with this algorithm is that this idea can be extended to other classes of graphs, so that it can deal with latent confounders, and even feedback loops. As most existing algorithms for these tasks are not exact, advancement in this area would be invaluable for reliable causal inference, and the algorithm we presented here may prove to be a good starting point.

Acknowledgments

I would like to thank Joris Mooij and the anonymous reviewers for useful feedback. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement n$^{\text{o}}$ 639466).

Appendix A. Computing a Lower Bound on the Penalty Term

In this appendix, we provide more details about the approach we sketched in Section 3.4; in particular, how it can be determined that an adjacency between two nodes is required to hold in all equivalence classes in a state. Due to space limitations, we do not give the precise graph-theoretical rules that constitute the algorithm, but rather explain its operation on a slightly higher level.

Algorithm 2: Compute a lower bound on h.

```
Input: A state, specified by the CPDAG $G$ of its greatest element and a list $L$ of its
       required d-connections
Output: A lower bound on the number of edges among classes in the state
Initially mark all adjacencies in $G$ as required;
For each edge that has available branching operators in the state (i.e. not conflicting with
  any required d-connection in $L$), mark that adjacency as not required;
while required adjacency marks were changed in the last round do
    Mark all directed edges in $G$ as not fixed;
    For each directed edge in $G$ that is part of a v-structure for all deletions of non-required
      adjacencies (other than itself), mark it as fixed;
    Apply Meek's rules (Meek, 1995) where they are applicable for all deletions of
      non-required edges / changes of non-fixed directed edges, to determine what other
      directed edges are fixed;
    For each required adjacency $x, y$ for which $\mathrm{Pa}_y$ might have lost a node, or $\mathrm{Pa}_y \cup \mathrm{NA}_{y,x}$
      might have gained a node, compared to the original $G$, mark it as not required;
end
Return the number of required adjacencies, plus the number of required adjacency
  components, minus the number of required d-connection components;
```

Our algorithm for this task (Algorithm 2) proceeds by starting from the state's greatest element G, and repeatedly considering what edges might be removed by an available branch operation. We only track for individual edges whether or not they might be removed; keeping track of this for all combinations of edges would be much too computationally expensive. Similarly, for each individual directed edge from v to w, we may ask if its orientation is *fixed*, meaning that in all equivalence classes considered so far where v and w are adjacent, the edge is directed with the same orientation.

Because the goal is to find a lower bound on the number of edges in a state, it is acceptable if our algorithm marks a required adjacency as not required. However, mistakes in the other direction would violate the requirement of being a lower bound. So to guarantee correctness, we will mark an adjacency as not required if a Delete operator could be applied to it after some subset of non-required edges was already removed and some set of non-fixed directed edges replaced by other edges — even if no sequence of Delete operators would actually yield a graph in which precisely those sets of edges are missing or changed. The same type of reasoning is used to determine which directed edges are fixed: an edge is only marked as fixed if we can guarantee its orientation after any subset of non-required edges is removed. Also, the list L of required d-connections is ignored for branching operators that might become applicable, to save computation time. Instead, as soon as the graph might change in such a way that a branching operation exists on it that did not exist in

G, we consider the effect of performing this operation. Using the correspondence between branches and Deletes (see Section 3.2) and the conditions for applicability of Delete operators (Chickering, 2002), we can tell when this happens from changes in Pa_y and $\mathrm{NA}_{y,x}$.

References

D. M. Chickering. Optimal structure identification with greedy search. *Journal of Machine Learning Research*, 3:507–554, 2002.

D. M. Chickering, D. Heckerman, and C. Meek. Large-sample learning of Bayesian networks is NP-hard. *Journal of Machine Learning Research*, 5:1287–1330, 2004.

C. P. de Campos and Q. Ji. Efficient structure learning of Bayesian networks using constraints. *Journal of Machine Learning Research*, 12:663–689, 2011.

P. Forré and J. M. Mooij. Constraint-based causal discovery for non-linear structural causal models with cycles and latent confounders. In *Proceedings of the 34th Annual Conference on Uncertainty in Artificial Intelligence (UAI 2018)*, 2018.

S. B. Gillispie and M. D. Perlman. Enumerating Markov equivalence classes of acyclic digraph models. In *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence (UAI 2001)*, pages 171–177, 2001.

C. Meek. Causal inference and causal explanation with background knowledge. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI 1995)*, pages 403–411, 1995.

T. S. Richardson and P. Spirtes. Ancestral graph Markov models. *The Annals of Statistics*, 30(4): 962–1030, 2002.

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

I. Shpitser, R. J. Evans, T. S. Richardson, and J. M. Robins. Introduction to nested Markov models. *Behaviormetrika*, 41(1):3–39, 2014.

P. Spirtes, C. Glymour, and R. Scheines. *Causation, Prediction, and Search*. MIT Press, Cambridge, MA, second edition, 2000.

J. Tian. A branch-and-bound algorithm for MDL learning Bayesian networks. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI 2000)*, pages 580–588, 2000.

J. Tian, R. He, and L. Ram. Bayesian model averaging using the k-best Bayesian network structures. In P. Grünwald and P. Spirtes, editors, *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence (UAI 2010)*, pages 589–597, 2010.

T. van Ommen and J. M. Mooij. Algebraic equivalence of linear structural equation models. In *Proceedings of the 33rd Annual Conference on Uncertainty in Artificial Intelligence (UAI 2017)*, 2017.