Approximation Algorithms for Bayesian Network Structure Learning

Adhanet Kiflemariam & Aurora Ingebrigtsen

December 5, 2025

1 Introduction

A Bayesian network is a *probabilistic graphical model* that represent a joint probability distribution over a set of random variables [3]. There are two main components of a Bayesian network: the structure and the parameters [4]. The structure is represented by a directed acyclic graph (DAG), where each node corresponds to a random variable, and each edge indicates a direct influence from one variable to another [4] [3]. While the parameters define the conditional probability distribution for each node, they describe how likely each value of a variable is, given the values of its parent variables [4][3]. This organization means that the joint distribution can be broken down into a collection of local, parent–child relationships, enabling efficient reasoning and interpretation of complex probabilistic systems.

In many real-world problems, the structure of the network is unknown. *Bayesian network structure learning* (BNSL) aims to discover the DAG that best explains the observed data [3]. There are several families of methods for BNSL, one of these are *score-based methods* [3]. Score-based methods assigns each DAG a score based on how well it fits the data and finds the DAG that maximizes the score.

The optimization problem in the score-based approach is NP-hard , even under strong structural restrictions, which means that exact algorithms have to use exponential time in the worst case [4, 6]. This motivates the use of *approximation algorithms*, which give up exact optimality in return for lower computational coast. However, there has been relatively little research on approximation algorithms with provable guarantees on how close their solutions are to optimal. At the time of this project, only Ziegler [7] had proposed two such algorithms, and more recently Kundu, Parviainen and Saurabh [4] introduced two approximation algorithms with explicit time-approximation trade-offs.

The goal of this project is to evaluate the performance of the moderately exponential–time approximation algorithm introduced by Kundu and Parvianen [4]. To do so, we first implement the two exact structure-learning algorithms that the approximation method builds upon. We then implement the approximation algorithm and assess its performance by running it on a set of known benchmark networks, as described in the experiments presented in Section 3.

2 Methodology

In this section we describe the different algorithms we implemented for the project. The code can be found on GitHub.

2.1 Dynamic Programming

Several papers have suggested using dynamic programming for BNSL. These approaches build on the fact that the scoring function is *decomposable*, so score of a DAG factorizes as

$$\text{Score}(G) = \sum_i s(X_i, \text{Pa}(X_i)).$$

This means that the total score of the graph equals the sum of the local scores of each node given its parents. In this project we implement dynamic programming (DP) algorithm by Silander and Myllymäki [6].

Their algorithm works by first computing all local scores. For every variable, and possible parent set, we compute the local score from the data. This is the only step that actually touches the data.

Second, for each variable v and every possible set of candidate parents C, we compute the best parent set *bps* within C. That is, the best parent set may be the entire candidate set or a subset of it, which allows us to compute this by a simple recursion:

$$score_i(g_i^*(C)) = \max\big(score_i(C),\, score_1(C)\big)$$

where

$$score_1(C) = \max_{c \in C} score_i\big(g_i^*(C \setminus \{c\})\big).$$

In other words, the best score for C is either the score of using all of C as parents, or the best score from a smaller candidate set where we remove on element.

Third, we compute the best sink for every subset $W \in V$. This step builds on the fact that every DAG must have at least one sink, that is, a node with no outgoing arcs. If s is the sink, the total score of W is:

- the total score of the smaller subset $W \setminus s$

- plus the local score of s with its best possible parents in $W \setminus s$.

We can thus use the best parent sets from step 2 to recurse over subsets and for each subset pick the sink that gives the highest total score.

Once we know the optimal sink for every subset of variables, we can recover the optimal variable ordering. We build the order from the end. At each step, we look at the current set of remaining variables and pick the best sink for that set, place it last,remove it from set and repeat until all positions are filled.

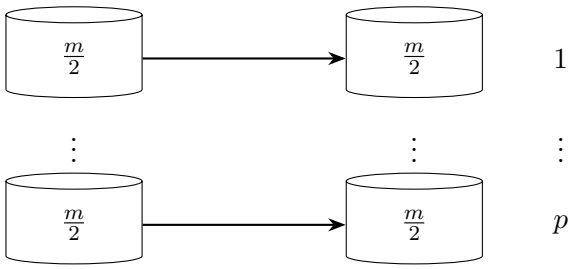
Finally, given the optimal order,we build the optimal DAG. We go through the variables from left to right, and for each variable only those that appear earlier in the order can be its parents. From those, we look up the best parent set from step 2 and assign it. Then we move on to the next variable.

2.2 Partial Order Approach

Although the dynamic programming find the globally optimal DAG, its main drawback is the memory requirement. The DP tables stores one value for every subset of variables, which means that both the running time and space usage grow on the order of 2^n [6]. In practice this limits the algorithm to networks of roughly 30 variables, even with optimized implementations.

To address this, [5] introduced precedence constraints to the problem. That is, instead of enumerating all possible parent sets for all possible subsets, the search is restricted using partial orders. A *partial order* P on a set M is a relation that is reflexive, antisymmetric, and transitive. A subset I of M is called an *ideal* of P if $y \in \mathcal{I}$ and $xy \in P$ imply that $x \in \mathcal{I}$. The general idea of their approach is to use DP over all ideals of a partial order P.

They also propose a scheme for selecting the partial orders, called the *Two-Bucket Scheme*. The idea behind this scheme is to partition nodes into front/back buckets, where every node in the front bucket must precede every node in the back bucket. The algorithm uses two user-defined parameters, m and p, which determine the size of each bucket order and the number of disjoint bucket orders to generate, respectively. One can then generate partial orders from all the different bucket configurations.



Using the partial order approach with the two-bucket schemes guarantees to find the find globally optimal DAG. It also gives a space and time trade-off by the parameters m and p, enables parallelism, allows exploitation of prior knowledge of variable ordering.

2.3 Moderately Exponential-Time Algorithm

Kundu, Parviainen & Saurabh proposes an algorithm that allows a trade off between speed and approximation ratio $\frac{\ell}{k}$ where ℓ and k is user defined parameters [4]. The idea is to control how much of the search space is explored, large values of $\frac{\ell}{k}$ gives a better approximation but also increase the running time.

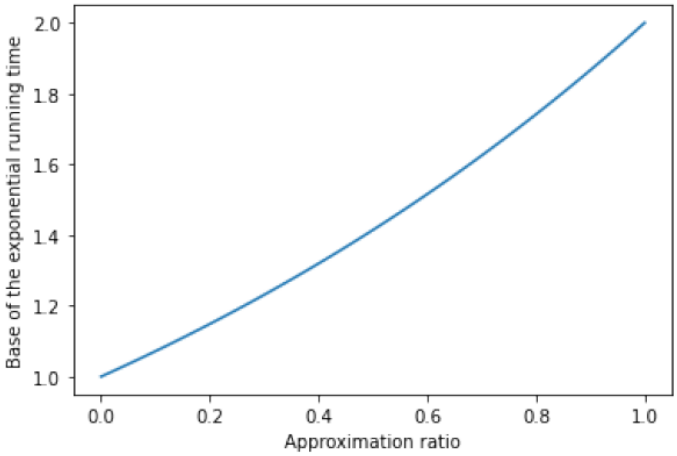


Figure 1: Trade off between approximation ratio and running time

Their approach suggests to randomly partition nodes into k equally sized sets, and then pick every combination of ℓ sets as the last bucket order. For all $\binom{k}{\ell}$ choices we run dynamic programming on the partial order and finally return the best DAG found.

Kundu et al. show that this approach provides an upper bound on the score, guaranteeing that the returned DAG archives at least a $\frac{\ell}{k}$ fraction of the score of an optimal DAG [4]. The results assume that the local scores are non-negative, even though most scoring metrics are negative [4]. However, this is technically no problem, as negative scores can be transformed into non-negative by adding a sufficiently large constant [4]. Details on how the upper bound is computed, as well as an example can be found in [4].

3 Experiments

This section describes the experiments carried out to evaluate the moderately exponential time approximation algorithm. The experiments are divided into subsections based on the size of the benchmark networks. For smaller networks, we can compare the approximation result with the dynamic programming result, giving us an understanding of how close the approximation is to the optimal score. It allows allow us to compare the theoretical upper bound on score given by the local scores and the approximation ratio to the actual optimal score. For medium and large networks, the results are more informative in terms of running time and scalability.

3.1 Smaller Networks

For the first set of experiments, we evaluate the algorithms on a collection of small benchmark networks from the bnlearn repository [1]. These networks are commonly used in structure-learning research and allow us to compare the approximation algorithm directly against the dynamic programming baseline, since both methods can be run to completion.

Name	**Nodes**	**Arcs**	**Parameters**
ASIA	8	8	18
CANCER	5	4	10
EARTHQUAKE	5	4	10
SACHS	11	17	178
SURVEY	6	6	21

Table 1: bnlearn-networks used for experiments[1]

To compute local scores we use the pygobnilp implementation [2], with its default scoring parameters. We use BIC score as the scoring-metric. For each network, we generate datasets of three different sizes that is used to compute local scores. We also evaluate three random seeds and four different approximation ratios, specified through the (l, k) parameter pairs.

Setting	**Values**
Sample sizes	100, 1000, 10000
Random seeds	42, 43, 44
Parameter pairs (l, k)	$(1, 2)$, $(2, 3)$, $(3, 4)$, $(4, 5)$

Table 2: Overview of the experimental setup.

3.2 Medium and Large Networks

For the medium and large networks, we can no longer generate the local scores ourselves, since pygobnilp only supports models up to a certain size with the available Gurobi license. Instead, we use the pre-computed local scores provided in the zip archive of the files associated with the paper Bayesian network learning with cutting planes Proc. UAI 2011 by James Cussens: https://jcussens.github.io/. We also included the medium sized "child" network, from bnlearn [1], as this is small enough to still be able to generate local scores locally.

Name	Nodes	Arcs	Parameters
ALARM	37	46	509
CARPO	60	–	–
CHILD	20	25	230
HAILFINDER	56	66	2656
INSURANCE	27	52	1008
MILDEW	35	46	540150
WATER	32	66	10083

Table 3: Selected bnlearn networks (medium and large).

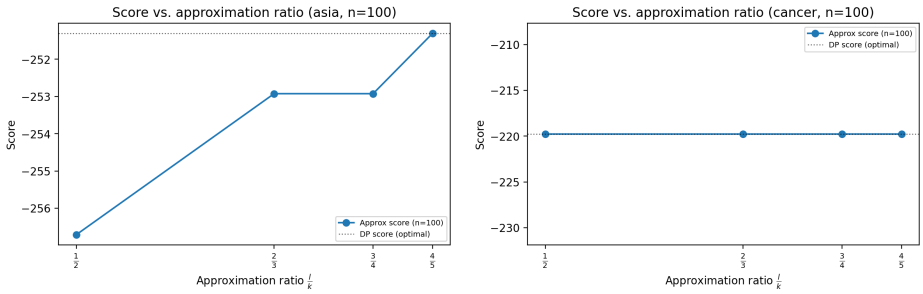
The approximation ratios we tested vary across the networks, as high approximation ratios is not feasible for the larger networks.

4 Results

This section presents the main findings from the experiments reported in Section 3.

4.1 Effect of the Approximation Ratio on the Score

In principle, we expect the score to increase linearly with the approximation ratio, since a higher ratio preserves more information and should find solutions closer to the DP optimum. For the small networks this trend is partly visible, see 2a, where the score increase as the approximation ratio increases. However, in many cases the optimal score is already obtained at the $\frac{1}{2}$ approximation ratio, as in 2b. The reason is that several of these small networks are structurally simple, which might explain why relatively coarse approximations are sufficient to recover the optimal parent sets.



(a) Scores for the Asia network (b) Scores for the Cancer network

Figure 2: Score for networks against approximation ratio. The dashed line represents the DP score, obtained by running the Silander Myllymäki algorithm on the same local scores.

For the larger network the effect is clearer. Figure 3 illustrate the relationship between runtime and score for different approximation ratios for two medium sized networks. Increasing $\frac{\ell}{k}$ consistently improves the score while causing a high increase in runtime (visible on the log scale). Unlike the small networks, the score does not flatten out at the optimum within the tested ratios as in Figure 2b. This may be due to the increased complexity of the networks. For Water (32 variables), we used smaller approximation ratios than we did in 2 because of memory overloads, which may also explain the difference.

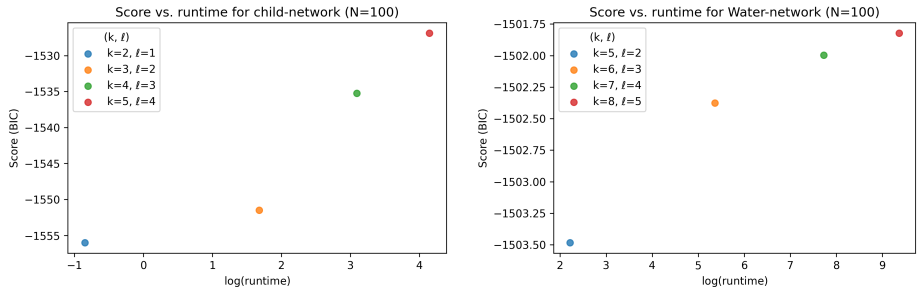


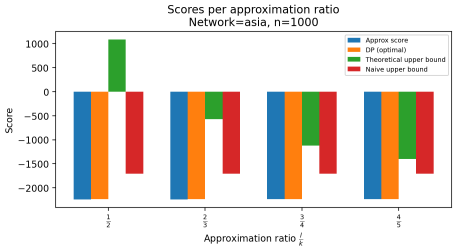
Figure 3: Score vs running time for a) the child network and b) the water network.

4.2 Approximation Guarantees

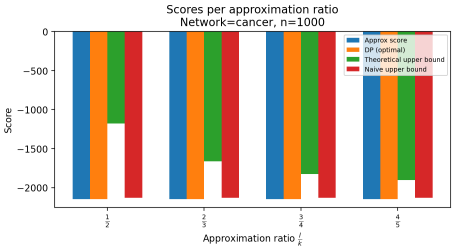
An interesting question is whether the theoretical upper bound computed by the local scores and the approximation ratio is useful in practice. To evaluate this, we compare it against two reference quantities: the DP score, which represents the true optimal network score, and a naive upper bound obtained by selecting, for each node, the parent set with the highest local

score without constraints on acyclicity.

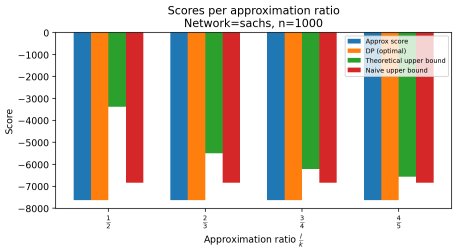
The comparison for each of the small networks is shown in Figure 4. From these results we observe that the theoretical upper bound tends to be far from the actual optimal score: it is frequently much larger than the naive upper bound, and in some cases even becomes positive, as seen in Figure 4a.



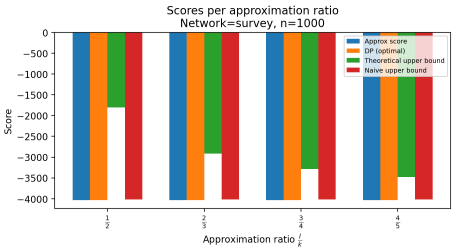
(a) Scores for the Asia network



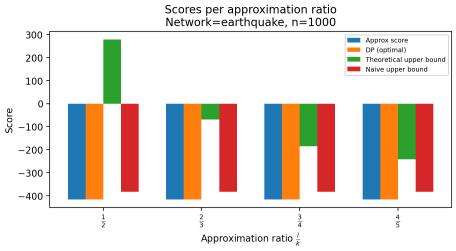
(b) Scores for the Cancer network



(c) Scores for the Sachs network



(d) Scores for the Survey network



(e) Scores for the earthquake network

Figure 4: Approximation guarantee compared to true optimal score, approximation score and naive upper bound.

4.3 Structural Hamming Distance

We also examined the structural hamming distance (SHD) between the learned network and the true network.[1] In general, the SHD tends to decrease as the approximation ratio increases, which aligns with the expectation that larger ratios preserve more information.

[1]Specifically, we compute the SHD between the CPDAGs of the learned and true graphs in order to compare markov equivalence classes.

However, because we work with finite data, the learned score-optimal DAG is not always markov equivalent to the true network. A clear example of this appears in the Survey network in figure 5: the higher approximation ratios gets a higher SHD than the lower ratios. At first this seems counter-intuitive.

Upon closer inspection, however, we found that for these ratios the learned parent maps were identical to those learned by the exact DP algorithm. In other words, the approximation algorithm successfully recovered the optimal network given the local scores. The higher SHD is therefore not due to approximation error but rather reflects that, with limited data, the score-optimal DAG itself may not be Markov equivalent to the true network.

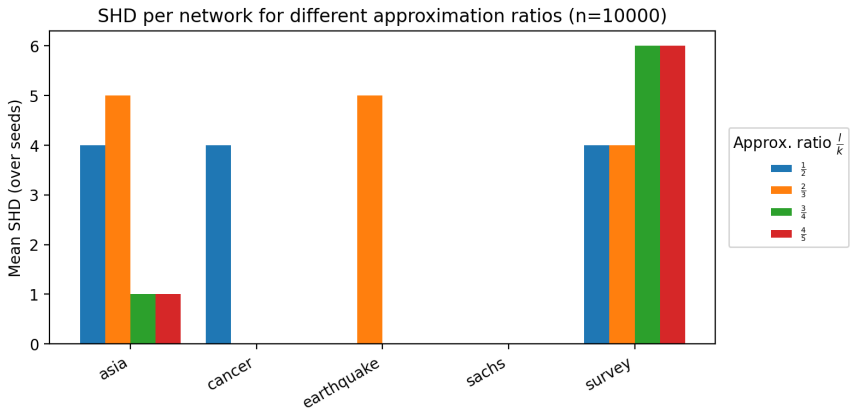


Figure 5: SHD to the true network from the small networks with approximation ratio $\frac{1}{2}$, $\frac{2}{3}$, $\frac{3}{4}$ and $\frac{4}{5}$

4.4 Running Time

To analyse the running time of the moderately exponential–time algorithm, we consider the larger networks, where differences in complexity become more clear. As discussed earlier, the algorithm provides a tradeoff between approximation ratio and computational cost (see Figure 1).

Ideally, the empirical runtime should scale consistently with the theoretical time complexity $\mathcal{O}(2^{\frac{\ell}{k}n})$. Figure 6 illustrates that this predicted scaling behaviour is indeed observed in practice. The horizontal axis shows the theoretical log-runtime exponent $(\ell/k)\,n$, while the vertical axis shows the empirical log-runtime measured by the implementation.

For each choice of (k, ℓ), multiple points appear along a vertical line.

These correspond to different sample sizes. Higher sample sizes influence the complexity of the local score, and thus tends to have a slightly higher empirical runtime.

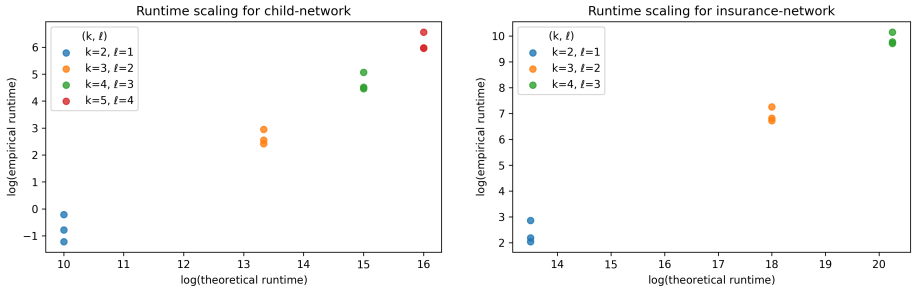


Figure 6: Runtime scaling for a) child and b) insurance. X-axis is the analytical runtime, while the Y-axis represents the logarithm of the actual runtime elapsed when running the algorithm with this ratio (averaged over seeds).

4.5 Number of Ideals

Running experiments on larger networks often caused memory overloads, especially for high approximation ratios. This happens because larger approximation ratios create larger final buckets, which in turn drastically increase the number of ideals that must be enumerated for each partial order, as shown in Table 4. The number of ideals grows very quickly as ℓ/k increases, and for the largest networks the counts become too large to fit in memory.

Network	**Nodes** n	$(\mathbf{k=2}, \ell=\mathbf{1})$	$(\mathbf{k=3}, \ell=\mathbf{2})$	$(\mathbf{k=4}, \ell=\mathbf{3})$	$(\mathbf{k=5}, \ell=\mathbf{4})$
CHILD	20	2.05×10^3	1.64×10^4	3.28×10^4	6.56×10^4
INSURANCE	27	2.46×10^4	2.63×10^5	2.10×10^6	4.19×10^6
WATER	32	1.31×10^5	4.20×10^6	1.68×10^7	6.71×10^7
MILDEW	35	3.93×10^5	1.68×10^7	1.34×10^8	2.68×10^8
ALARM	37	7.86×10^5	3.36×10^7	2.68×10^8	1.07×10^9
HAILFINDER	56	5.37×10^8	2.75×10^{11}	4.40×10^{12}	3.52×10^{13}
CARPO	60	2.15×10^9	1.10×10^{12}	3.52×10^{13}	2.81×10^{14}

Table 4: Number of ideals generated by the first partial order for different approximation ratios

5 Future Work

A natural next step would be to implement the approximation algorithms proposed by Ziegler [7]. This would make it possible to compare the different approximation methods directly and provide a more detailed analysis of the moderately exponential–time algorithm.

The implementations could also be optimized further in several ways. Potential improvements include replacing python dictionaries with `ndarrays` structures, using bitmasks to represent sets more compactly, or rewriting the implementations to C++ to achieve more efficient memory allocation. These optimizations would mainly improve constant factors rather than asymptotic running time but they may still make it feasible to analyse larger networks with higher approximation ratios, which we were unable to run within our current memory constraints.

There are also several other experiments that would be useful to carry out. Cover size (the number of partial orders in the cover) is not yet included in the analysis. This would be interesting to compute during runtime and include in the result file. A more thorough analysis could be done of the results of the larger networks. Moreover, comparing the solutions found by the approximation algorithm on large networks such as Hailfinder and Carpo to those obtained by standard heuristics would be useful for understanding its practical performance.

References

[1] Bn learn repository. https://www.bnlearn.com/bnrepository/. Accessed: 2025-12-01.

[2] pygobnilp documentation. https://pygobnilp.readthedocs.io/en/latest/, 2025. Accessed: 2025-12-01.

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

[4] Madhumita Kundu, Pekka Parviainen, and Saket Saurabh. Time–approximation trade-offs for learning bayesian networks. 2024.

[5] Pekka Parviainen and Mikko Koivisto. Finding optimal bayesian networks using precedence constraints. *The Journal of Machine Learning Research*, 14(1):1387–1415, 2013.

[6] Tomi Silander and Petri Myllymaki. A simple approach for finding the globally optimal bayesian network structure. *arXiv preprint arXiv:1206.6875*, 2012.

[7] Valentin Ziegler. Approximation algorithms for restricted bayesian network structures. *Information Processing Letters*, 108(2):60–63, 2008.