




## Bayesian Structure Learning and Sampling of Bayesian Networks with the R Package BiDAG

Polina Suter   
ETH Zürich

Jack Kuipers   
ETH Zürich

Giusi Moffa   
University of Basel

Niko Beerenwinkel   
ETH Zürich

---

### Abstract

The R package **BiDAG** implements Markov chain Monte Carlo (MCMC) methods for structure learning and sampling of Bayesian networks. The package includes tools to search for a maximum a posteriori (MAP) graph and to sample graphs from the posterior distribution given the data. A new hybrid approach to structure learning enables inference in large graphs. In the first step, we define a reduced search space by means of the PC algorithm or based on prior knowledge. In the second step, an iterative order MCMC scheme proceeds to optimize the restricted search space and estimate the MAP graph. Sampling from the posterior distribution is implemented using either order or partition MCMC. The models and algorithms can handle both discrete and continuous data. The **BiDAG** package also provides an implementation of MCMC schemes for structure learning and sampling of dynamic Bayesian networks.

*Keywords:* Bayesian networks, dynamic Bayesian networks, structure learning, Bayesian inference, MCMC, R.

---

## 1. Introduction

A Bayesian network is a probabilistic graphical model, which represents conditional independence relationships between a set of random variables by a directed acyclic graph (DAG). The problem of DAG learning from observational data is hard (Chickering 1996), and the number of DAGs grows super-exponentially with the number of nodes. Hence, developing and implementing methods to learn an underlying DAG from observational data in reasonable time continues to be the focus of much research (Bartlett and Cussens 2017; Goudie and Mukherjee 2016; Scanagatta, de Campos, and Corani 2015). Drton and Maathuis (2017) provide

an overview of the approaches for structure learning of graphical models including Bayesian networks.

The R (R Core Team 2022) packages **pcalg** (Kalisch, Mächler, Colombo, Maathuis, and Bühlmann 2012), **bnlearn** (Scutari 2010), **bnstruct** (Franzin, Sambo, and Camillo 2017) and the **TETRAD** toolbox (Ramsey, Zhang, Glymour, Romero, Huang, and Ebert-Uphoff 2018) based in Java (Gosling, Joy, Steele, and Bracha 2000) implement multiple approaches to structure learning, including both constraint-based and search-and-score methods. Constraint-based methods use conditional independence tests to learn the edges of the graph. Search-and-score methods rely on an efficient search strategy in the space of DAGs and a score function to find the graph which best explains the data. Hybrid methods, such as max-min hill climbing (Tsamardinos, Brown, and Aliferis 2006), use a combination of both approaches to learn the optimal graph. A comparison of accuracy and efficiency of different methods for structure learning of Bayesian networks can be found in Scutari, Graafland, and Gutiérrez (2019). Despite a wide variety of available tools, most of them aim to find one best solution. However, especially when the number of observations is modest, relying on one best graph can be inadequate because many graphs may explain the data similarly well (Friedman and Koller 2003). Bayesian methods can help to address this issue. Posterior sampling, in particular, enables Bayesian model averaging and allows us to adequately account for modeling uncertainty when the number of observations is finite. However, only a few tools exist for Bayesian structure learning, probably because Bayesian approaches are computationally demanding and unfeasible in large domains. The R package **mcmcabn** (Kratzer and Furrer 2019) implements a structure MCMC algorithm for sampling DAGs from their posterior distribution given the data. Structure MCMC is only suitable for domains with a limited number of nodes. The R package **deal** implements Bayesian parameter learning, but for structure learning, it uses a greedy search with random restarts (Böttcher and Dethlefsen 2003). The Matlab/C/Java (The MathWorks, Inc. 2011) package **BDagl** (Eaton and Murphy 2007) implements an order MCMC scheme (Friedman and Koller 2003) without any restrictions on the search space, so that it is only feasible for small networks and does not scale well beyond 20 nodes.

Here, we describe the R package **BiDAG**, an implementation of various MCMC schemes, which overcomes the issues previously restricting Bayesian methods to small domains. **BiDAG** implements order (Friedman and Koller 2003) and partition (Kuipers and Moffa 2016) MCMC schemes. These scales to networks with hundreds of nodes when combined with the hybrid approach of Kuipers, Suter, and Moffa (2022). Both order and partition MCMC schemes can sample from the posterior and find a MAP DAG, and both reach convergence much faster than the structure MCMC approach. Simulation studies have shown that the iterative order MCMC scheme (Kuipers *et al.* 2022) displays better accuracy to discover the ground truth DAG compared to other well-established methods such as the PC algorithm (Spirtes, Glymour, and Scheines 2000) or greedy equivalent search (GES) (Chickering 1996).

The **BiDAG** software supports both discrete and continuous data types, and the methods also apply to weighted data as required, for example, in mixture models (Kuipers *et al.* 2018b). Further, all the implemented MCMC schemes handle structure learning and sampling of first-order dynamic Bayesian networks (DBNs). **BiDAG** is available from the Comprehensive R Archive Network (CRAN) at <https://CRAN.R-project.org/package=BiDAG>.

**BiDAG** also imports some methods from other packages. In the first step of a hybrid approach, it uses by default the constraint-based PC algorithm from **pcalg** to define a search

space, complemented by a new implementation of conditional independence tests for discrete and weighted data. **BiDAG** includes a visualization tool, which imports methods from the R packages **graph** (Gentleman, Whalen, Huber, and Falcon 2022) and **Rgraphviz** (Hansen *et al.* 2019).

In Section 2, we describe the methodological background behind the algorithms in **BiDAG**. In Section 3, we describe the **BiDAG** functions, further divided into four subsections on structure learning and sampling, posterior model selection, convergence diagnostics, and model comparison. In Section 4, we describe how to use the package for structure learning of DBNs. Section 5 contains examples of structure learning and sampling on two simulated data sets. In this section we also show how sampling from the posterior can improve model selection. In Section 6, we apply the package to the problem of characterizing cancer sub-types. Finally, in Section 7, we discuss the runtime of the implemented algorithms in different simulation settings.

## 2. Methodological background

A Bayesian network describes a factorization of a joint probability distribution  $P(\mathbf{X})$  of a set of random variables  $\mathbf{X} = (X_1, \dots, X_n)$  by means of a DAG. Specifically we can define a Bayesian network  $\mathcal{B}$  as a pair  $(\mathcal{G}, \Theta)$  where  $\mathcal{G}$  is a DAG whose nodes represent the random variables in  $\mathbf{X}$  and  $\Theta$  are the parameters of the probability distributions  $P(X_i | \mathbf{Pa}_i)$  describing the relationship between each variable  $X_i$  and its parents  $\mathbf{Pa}_i$  in the graph  $\mathcal{G}$ , such that

$$P(\mathbf{X}) = \prod_{i=1}^n P(X_i | \mathbf{Pa}_i).$$

Learning a Bayesian network requires estimating both components: parameters  $\Theta$  and structure  $\mathcal{G}$ . Maximizing or marginalizing the parameters for a given structure will provide a score for each DAG. In **BiDAG**, the score  $S$  of each DAG is proportional to its posterior probability given the data  $D$ . For computational feasibility of the implementation, it is essential that the score function factorizes into a product where each term depends only on one node and its parents:

$$P(\mathcal{G} | D) \propto P(D | \mathcal{G})P(\mathcal{G}) = \prod_{i=1}^n S(X_i, \mathbf{Pa}_i | D).$$

See Geiger and Heckerman (1995) for the technical conditions guaranteeing the desired score factorization. Two score functions  $S$  meeting the conditions for the decomposition in Equation 2 are implemented in **BiDAG**: (1) the Bayesian Dirichlet equivalent score (BDe) (Geiger and Heckerman 1995) with a Dirichlet parameter prior for binary and categorical data, and (2) the Bayesian Gaussian equivalent score (BGe) (Geiger and Heckerman 2002; Consonni and Rocca 2012) with a Wishart prior for continuous data.

Learning the structure component of a network  $\mathcal{B}$  requires finding the DAG  $\mathcal{G}$ , which best fits a data set  $D$ . As far as search-and-score methods are concerned, this means finding a graph with a score larger or equal than any other. In situations where several structures achieve similar scores, focusing on a single structure may be misleading (Friedman and Koller 2003). The MCMC methods in **BiDAG** account for structure uncertainty by sampling DAGs from

the posterior distribution given the data  $D$ . Rather than examining the highest scoring DAG, we can select the model that consists only of edges whose posterior probabilities are higher than a desired threshold. Although not guaranteed to give a DAG, simulation studies show a reduction in the number of false-positive edges with this approach compared to choosing one maximally scoring model (Kuipers *et al.* 2022), while hardly ever resulting in directed cycles.

## 2.1. Order MCMC

The rationale behind MCMC schemes is to construct a Markov chain  $\mathcal{M}$  such that its stationary distribution equals the posterior distribution  $P(\mathcal{G} | D)$  we would like to sample from. One of the schemes implemented in **BiDAG** is order MCMC, which does not operate directly on the space of DAGs but on the smaller space of orders. The posterior landscape is smoother in the space of orders than in the space of DAGs since the likelihood of each order is a sum over multiple DAGs (Friedman and Koller 2003). Consequently order MCMC can achieve faster convergence with respect to structure MCMC (Friedman and Koller 2003).

A permutation  $(i_1, i_2, \dots, i_n)$  of the  $n$  nodes of a DAG defines a linear order  $i_1 \prec i_2 \prec \dots \prec i_n$ . A DAG  $\mathcal{G}$  is compatible with an order  $\prec$  if  $i \prec j$  whenever  $j$  is a parent of  $i$  in  $\mathcal{G}$  (Kuipers *et al.* 2022). We denote with  $\Gamma_{\prec}$  the set of all DAGs compatible with  $\prec$ . Each order is assigned a score that equals the sum of the scores of all DAGs compatible with this order,

$$R(\prec | D) = \sum_{\mathcal{G} \in \Gamma_{\prec}} P(\mathcal{G} | D) \propto \sum_{\mathcal{G} \in \Gamma_{\prec}} \prod_{i=1}^n S(X_i, \mathbf{Pa}_i | D).$$

As discussed in Friedman and Koller (2003) we can exchange the product and sum and get the order score by summing over all parent sets compatible with the order instead of summing over all possible DAGs. Similarly to DAGs we can formulate the compatibility requirement for parent sets. A parent set  $\mathbf{Pa}_i$  of a node  $i$  is compatible with an order  $\prec$  if  $i \prec j$  for all parents  $j \in \mathbf{Pa}_i$ . For each node  $i$ , we denote the set of all parent sets compatible with  $\prec$  by  $\mathbf{U}_{\prec, i}$ . Then

$$R(\prec | D) \propto \prod_{i=1}^n \sum_{\mathbf{Pa}_i \in \mathbf{U}_{\prec, i}} S(X_i, \mathbf{Pa}_i | D). \quad (1)$$

To construct a Markov chain in the space of orders we use the following moves from an order  $\prec$  to a new order  $\prec'$ :

- local move: swapping adjacent nodes in  $\prec$ ;
- global move: swapping two random nodes in  $\prec$ ;
- node relocation: in this move we place a single node  $i_k$  in each possible position  $(1, 2, \dots, n)$  of the current order  $(i_1, i_2, \dots, i_n)$ , while keeping the order of the other nodes in  $\prec$  fixed. All  $n$  orders corresponding to all possible positions of the node  $i_k$  in the order are scored according to Equation 1 and the new order  $\prec'$  is sampled according to these scores.

The Metropolis-Hastings acceptance probability for the first two moves is

$$\rho = \min \left\{ 1, \frac{R(\prec' | D)}{R(\prec | D)} \right\}.$$

The last move is always accepted, but it can return the current order.

Order MCMC produces a sample of orders. Obtaining a sample of graphs from the posterior distribution requires an additional step of sampling DAGs from these orders according to their scores. Due to score decomposability we can do this on a per-node basis: We sample a parent set for each node from the set of parents compatible with the order, independently of other nodes. The exact functional form of score terms in Equation 1 assigned to each parent set depends on the model assumptions, namely parameter and graphical priors. The parameter prior can be chosen between normal-Wishart and Dirichlet corresponding to the BGe and BDe score functions mentioned previously in Section 2. The graphical prior is uniform by default, hence each parent set has the same probability *a priori*. However, **BiDAG** suggests the possibility of imposing a prior that penalizes each additional edge to favor sampling of sparse DAGs. **BiDAG** also suggests the possibility of penalizing each edge with an individual penalization factor to include more detailed prior information. In Section 3.1, we describe how to define a parameter and graphical priors in **BiDAG**. In addition, for large DAGs, all possible parent sets must be limited to a set corresponding to the reduced search space for computational feasibility. The efficient algorithm for defining the reduced search space in a way that it includes the bulk of the posterior weight is described in Section 2.4. In this way, we obtain a sample of DAGs  $\mathcal{G}_1, \dots, \mathcal{G}_M$ , from which we usually exclude the first  $m$  to account for the burn-in period. Assuming that the Markov chain has converged within  $m$  steps, we can approximate the posterior probability of any structural feature  $f$  by the sample average

$$P(f \mid D) \approx \frac{1}{M - m} \sum_{i=m+1}^M f(\mathcal{G}_i), \quad (2)$$

where  $f(\mathcal{G}_i)$  equals 1 if the feature  $f$  is present in structure  $\mathcal{G}_i$  and 0 otherwise.

## 2.2. Partition MCMC

One advantage of the order MCMC sampling scheme compared to structure MCMC resides in its increased efficiency. Another characteristic of order MCMC is that it imposes a non-uniform prior over structures by over-representing DAGs that belong to several orders (Friedman and Koller 2003). To achieve unbiased sampling, Kuipers and Moffa (2016) proposed an MCMC scheme in the space of ordered partitions instead. A labeled partition  $\Lambda$  is defined by two components: a node ordering  $\prec$  and a vector of sizes of the parts  $\kappa = (k_1, \dots, k_p)$ , where  $1 \leq p \leq n$  and  $\sum_{i=1}^p k_i = n$ . The vector  $\kappa$  divides the order  $\prec$  into  $p$  parts:  $v_1, \dots, v_p$ , such that  $v_1$  includes the first  $k_1$  nodes of the permutation  $\prec$ ,  $v_2$  includes the following  $k_2$  nodes, etc. A DAG  $\mathcal{G}$  is compatible with a partition  $\Lambda = (\prec, \kappa)$  if the following conditions are satisfied for every node  $X_i$  with, say,  $X_i \in v_j$ :

- if  $j < p$ ,  $X_i$  has at least one parent in the part  $v_{j+1}$ ;
- all nodes in  $\mathbf{Pa}_i$  belong to the parts with indices higher than  $j$ ;
- $\mathbf{Pa}_i = \emptyset$  if and only if  $j = p$ .

**BiDAG** implements the following moves in the space of partitions:

- swap any two nodes from different parts;

- swap any two nodes in adjacent parts;
- split a part or join two parts;
- move a single node into an existing part or form a new part with the single node.

The unbiased sampling with partition MCMC comes at the cost of a higher complexity and slower convergence as compared to order MCMC. However, the bias of order MCMC may not be a strong limitation in practice. [Kuipers \*et al.\* \(2022\)](#) have shown in simulation studies that the models obtained via averaging over the sample of DAGs obtained by the order MCMC scheme are very close to the ground truth structures.

### 2.3. MAP discovery

In addition to sampling from the posterior distribution, we can also use the algorithms implemented in **BiDAG** to search for a MAP graph ([Kuipers \*et al.\* 2022](#)). To do so, we replace the sum in Equation 1 with a maximum. Then, the order score equals the score of a maximum scoring DAG compatible with this order,

$$Q(\prec | D) = \prod_{i=1}^n \max_{\mathbf{Pa}_i \in \mathbf{U}_{i,\prec}} S(X_i, \mathbf{Pa}_i | D) = \max_{\mathcal{G} \in \Gamma_{\prec}} P(\mathcal{G} | D). \quad (3)$$

A non-uniform prior on DAGs imposed by order MCMC sampling does not affect the MAP search since DAG scores are equal in the order and partition schemes. Due to order MCMC being more computationally efficient, we implement it for both sampling and MAP estimation and partition MCMC only for sampling.

### 2.4. Hybrid sampling scheme

Even in the efficient order score decomposition of Equation 1, the number of possible parent sets, which need to be scored, is exponential of order  $O(2^{n-1})$ . To apply the algorithm to networks with, say,  $n > 20$  nodes, we prune the search space. **BiDAG** implements the hybrid approach of [Kuipers \*et al.\* \(2022\)](#) which limits the search space by means of a (possibly undirected) graph  $\mathcal{H}$ , whose maximal parent set size per node is  $K$ , so that the number of possible parent sets reduces to  $O(n2^K)$ . Since we wish to sample DAGs from the posterior distribution, prior knowledge together with evidence from the data drive the pruning process to ensure that the search space  $\mathcal{H}$  captures the bulk of the posterior weight. In **BiDAG**, we used the constraint-based PC algorithm ([Spirtes \*et al.\* 2000](#)) to define the search space.

The PC algorithm starts with a complete undirected graph and deletes edges based on conditional independence tests. After deleting as many edges as possible, we identify the skeleton graph, i.e., a graphical structure where all edges are bi-directional. Inference with the PC algorithm includes steps to direct some edges which yield a completed partially directed acyclic graph (CPDAG), which represents a class of equivalent DAGs. By default, we use a PC-defined skeleton as search space  $\mathcal{H}$  and not the CPDAG to avoid mistakes in directing edges.

An essential feature of **BiDAG** is the possibility to improve the initially defined search space  $\mathcal{H}$  ([Kuipers \*et al.\* 2022](#)). Errors in the statistical tests of the PC algorithm can lead to the deletion of true positive edges or edges appearing in high-scoring DAGs. Simulation studies

---

**Algorithm 1:** Iterative MCMC procedure.

---

**Input:** data  $D$

**Output:** MAP estimate  $\mathcal{G}^{\max}$ , optimized search space  $\mathcal{H}$

- 1 Initiate the search space  $\mathcal{H}$  with the PC algorithm or an arbitrary adjacency matrix
  - 2 Run the order MCMC scheme on the search space  $\mathcal{H}^+$
  - 3 Assign  $\mathcal{G}^{\max}$  the maximally scoring DAG obtained by the MCMC scheme
  - 4 Update  $\mathcal{H}$ ,  $\mathcal{H}^{\text{old}}$ :
    - $\mathcal{H}^{\text{old}} = \mathcal{H}$
    - $\mathcal{H} = \mathcal{H} \cup \mathcal{G}^{\max}$
  - 5 Repeat Steps 2 to 4, till  $\mathcal{H} = \mathcal{H}^{\text{old}}$
- 

show that the true positive rate (TPR) of structures estimated by the PC algorithm decreases when the density of the ground truth DAG, defined as an average number of parents of one node, increases (Kalisch and Bühlmann 2007). If the search space  $\mathcal{H}$  lacks some of the edges from a MAP DAG  $\mathcal{G}^{\text{MAP}}$ , we will not be able to find it when searching in  $\mathcal{H}$ . To address this limitation, Kuipers *et al.* (2022) propose to expand the search to an extended space  $\mathcal{H}^+$  in which the possible parent sets of every node include not only all combinations of parents of  $X_i$  in  $\mathcal{H}$  but also these parent sets joined with any other node that is not a parent of  $X_i$  in  $\mathcal{H}$ . Searching in  $\mathcal{H}^+$  provides the opportunity to correct for any mistakes of the pruning algorithm and yields higher scoring DAGs. We will refer to  $\mathcal{H}$  as the core search space and to  $\mathcal{H}^+$  as the extended search space.

The improvements we can achieve by simply searching in the extended space are limited. For example, if two or more parents are missing in the same node’s parent set, the approach would allow us to recover only one of them. However, if we iterate the procedure we may be able to correct for more than one mistake per parent set. The iterative order MCMC procedure is summarized in Algorithm 1 below:

Simulation studies show that the iterative MCMC procedure can improve even poor search spaces containing only 50 – 60% true positive edges so that the final space contains 90 – 100% true positive edges (Kuipers *et al.* 2022). However, the worse the original search space is, the more expansion iterations we need to optimize it. Defining a reasonable search space  $\mathcal{H}$  to start with can significantly decrease the total runtime of the iterative MCMC scheme. In the current version of **BiDAG**, in addition to the PC algorithm, it is possible to define the search space with an arbitrary adjacency matrix, which may stem from expert knowledge or another algorithm for structure learning. Computational complexity of the MCMC schemes in  $\mathcal{H}^+$  is higher than in  $\mathcal{H}$ . We will discuss differences in runtimes between using  $\mathcal{H}$  and  $\mathcal{H}^+$  in more detail in Section 7.

### 3. BiDAG package

The core functions of the package `learnBN` and `sampleBN` can be used for structure learning and sampling of Bayesian networks accordingly. The remaining functions can be divided into four main groups: convergence diagnostics, model averaging, model comparison, and network visualization. In this section, we describe the most important functions from all groups.

### 3.1. Constructing the score object

All functions for structure learning require an object of class ‘`scoreparameters`’, which stores the data and other quantities needed to score Bayesian networks. We can construct an object of class ‘`scoreparameters`’ using the function

```
scoreparameters(scoretype = c("bge", "bde", "bdecat", "usr"), data,
  bgepar = list(am = 1, aw = NULL, edgepf = 1),
  bdepar = list(chi = 0.5, edgepf = 2),
  bdecatpar = list(chi = 0.5, edgepf = 2),
  dbnpar = list(samestruct = TRUE, slices = 2, b = 0),
  usrpar = list(pctesttype = c("bge", "bde", "bdecat")),
  mixedpar = list(nbin = 0), DBN = FALSE, weightvector = NULL,
  bgnodes = NULL, edgepmat = NULL, nodeslabels = NULL)
```

The `data` should be in the form of a `data.frame` or a `matrix` with  $N$  rows and  $n$  columns, where  $n$  is the number of variables in the Bayesian network and  $N$  the number of observations. The parameter `scoretype` defines which score function is used: `bde` for binary data, `bdecat` for categorical data, `bge` for continuous data and `usr` for a user-defined score. An optional parameter `weightvector` defines the weight of each observation. The need for weighted data may arise, for example, in survey analysis (Kuipers, Moffa, Kuipers, Freeman, and Bebbington 2018a) and Bayesian network-based clustering (Kuipers *et al.* 2018b).

There are several ways to include prior information in structure learning. The parameter `bgnodes` lists root nodes (those who can have children but no parents). For example, we may expect that the gender of a participant in a survey data can have an effect on the answers, but not the opposite. Through the parameter `edgepmat` we can define a graphical prior that penalizes specific edges with individual penalization factors: we do not exclude them completely but simply reduce their chance to be sampled. Excluding the edges from the search space is also possible via the parameter `blacklist` of the structure learning functions, which we will discuss in Section 3.2. For penalizing each additional edge in the parent set and favoring sampling of sparse DAGs, the element `edgepf` of the corresponding score parameter can be used.

### 3.2. Structure learning and sampling

The functions `learnBN` and `sampleBN` implement the described MCMC schemes for learning and sampling of Bayesian networks. The function `sampleBN` implements order, partition and iterative order MCMC schemes for sampling of DAGs from the posterior distribution. The function `learnBN` implements order and iterative order MCMC schemes for finding a MAP DAG. In `sampleBN` the order score is calculated according to Equation 1, while in `learnBN` it is calculated using Equation 3. At each MCMC step, `learnBN` returns the maximally scoring DAG from the order, and `sampleBN` samples DAGs from the orders or partitions according to their scores.

Functions `learnBN` and `sampleBN` return the objects of classes ‘`orderMCMC`’, ‘`partitionMCMC`’ or ‘`iterativeMCMC`’ depending on the value of the parameter `algorithm` with available options being `"order"`, `"partition"` and `"orderIter"` (the corresponding algorithms were described in Sections 2.1, 2.2 and 2.4). The functions `orderMCMC`, `partitionMCMC` and `iterativeMCMC`



corresponding to each of the classes were used for sampling and learning of DAGs in the previous versions of the package (< 2.0.6) instead of `learnBN` and `sampleBN`.

The functions `learnBN` and `sampleBN` share mostly similar syntax. We will describe the main parameters of the function `sampleBN` :

```
sampleBN(scorepar, algorithm = c("order", "orderIter", "partition"),
  chainout = TRUE, scoreout = FALSE, alpha = 0.05, moveprobs = NULL,
  iterations = NULL, stepsave = NULL, gamma = 1, verbose = FALSE,
  compress = TRUE, startspace = NULL, blacklist = NULL, scoretable = NULL,
  startpoint = NULL, plus1 = TRUE, cpdag = FALSE, hardlimit = 12,
  iterpar = list(posterior = 0.5, softlimit = 9, mergetype = "skeleton",
  accum = FALSE, plus1it = NULL, addspace = NULL, alphainit = NULL))
```

and point out important differences with `learnBN`. All parameters, except `scorepar` and `algorithm`, are either optional or have default values. However, the MCMC schemes are very flexible, and the parameters should be consistent with the objectives and constraints of a particular structure learning problem. Parameters fall into four categories:

- parameters defining the search space: `startspace`, `scoretable`, `cpdag`, `plus1`;
- parameters of the Markov chain: `iterations`, `stepsave`, `moveprobs`;
- parameters to include prior information: `blacklist`, `startorder`;
- parameters defining objects included in the output: `chainout`, `scoreout`
- `iterpar` is a list containing additional parameters if the iterative MCMC scheme is chosen (`algorithm = "orderIter"`)

The number of MCMC iterations defined by the parameter `iterations` should be large enough for the MCMC chain to converge, while still controlling the runtime. The number of MCMC iterations required for convergence cannot be calculated analytically. Heuristics (Kuipers and Moffa 2016) and simulation studies (Kuipers *et al.* 2022) suggest that we need  $O(n^2 \log n)$  iterations to reach convergence or discover a maximum DAG. Motivated by this finding, in **BiDAG**, we set the default value of `iterations` to  $6n^2 \log n$  for order MCMC,  $20n^2 \log n$  for partition MCMC and  $3.5n^2 \log n$  for iterative MCMC. The decreased default number of `iterations` for iterative MCMC is motivated by the fact that with each expansion of the search space, a new MCMC chain is constructed. By algorithm definition (see Algorithm 1), two last chains are constructed in the same search space, hence the effective number of MCMC iterations is doubled compared to standard order MCMC.

To avoid excessively long runtimes, the algorithm does not sample DAGs at each MCMC iteration but once every `stepsave` steps. The idea of `stepsave` is that the number of iterations needed for the MCMC chain to converge is large and can be tens of thousands or even millions, while the required number of DAGs sampled from the posterior is usually much smaller. Sampling DAGs from the orders after each  $l$  steps significantly reduces the runtime without having a negative effect on convergence. By default, we define `stepsave` in such a way that the algorithm samples 1001 DAGs. This procedure is also known as thinning and results in reduced autocorrelation between the resulting samples. Less correlated samples provide

a better representation of the posterior distribution. However, it was shown that unthinned chains offer more precise estimates (Link and Eaton 2011) in the majority of cases because they contain more samples and hence more information about the posterior distribution. Thinning is still recommended to avoid the need for large computational and memory resources (Link and Eaton 2011).

The function `learnBN` implements the MAP search and does not save the list of DAGs found at each MCMC step. By default, it returns only one maximally scoring DAG. The list of DAGs can be saved for inspection by setting the parameter `chainout` to `TRUE`. However, it is not recommended to use for model averaging since `learnBN` does not yield a random sample from the posterior distribution. In contrast, the function `sampleBN` targets obtaining a sample of DAGs from the posterior distribution and its default output includes a trace of sampled DAGs.

The parameter `startspace` can define the search space via a sparse or binary adjacency matrix of size  $n \times n$ . An entry  $[i, j]$  in the adjacency matrix is 1 to indicate the presence of an edge from  $i$  to  $j$ . The search space can be an arbitrary graph without the acyclicity requirement. When edges are bidirectional both entries  $[i, j]$  and  $[j, i]$  should be equal to 1. Unit entries in column  $j$  determine the permissible parent sets for node  $j$ . Another way to pass a search space is the parameter `scoretable`, which has to be an object of class ‘`scorespace`’. The parameter `scoretable` can be used to pass an adjacency matrix together with the score tables and decrease the overall runtime (see Section 7 for details). Objects of class ‘`scorespace`’ can be extracted from the objects of classes ‘`orderMCMC`’, ‘`partitionMCMC`’ or ‘`iterativeMCMC`’ returned by the `sampleBN` and `learnBN` functions using the function `getSpace`. When neither `startspace` nor `scoretable` is specified, we define the search space by the skeleton estimated by the PC algorithm or by an equivalence class represented by a CPDAG if the parameter `cpdag` equals `TRUE`.

The parameter `alpha` defines the significance level  $\alpha$  used in the conditional independence tests of the PC algorithm. Larger  $\alpha$  values lead to larger search spaces, which decrease the risk that true positive edges are absent as a result of errors in the statistical tests. By the same principle though, high  $\alpha$  values will also increase the number of false-positive edges. While a higher number of false positive edges in the search space does not affect the goodness of fit of the resulting structures, it can negatively affect the runtime. Moreover, larger  $\alpha$  values also imply longer runtimes for the PC algorithm, which is worst-case exponential. By default  $\alpha = 0.05$ .

For the order MCMC scheme, the parameter `plus1` indicates whether the algorithm should perform the search/sampling in the core search space  $\mathcal{H}$  or in the extended space  $\mathcal{H}^+$ . When `plus1` equals `TRUE`, then the chain is constructed in  $\mathcal{H}^+$  instead of  $\mathcal{H}$ , as described in Section 2.4.

The parameter `blacklist` defines all single edges we wish to remove from the search space, and hence they will not appear in any of the sampled DAGs. If a node is not allowed to have any parents, it is computationally more efficient to define it as a root node via the parameter `bgnodes` in the `scorepar` object instead of specifying all edges from any other node in a `blacklist`. The parameter `edgepmat` of the function `scoreparameters` mentioned above can be regarded as a soft version of the blacklist.

In Section 2.4, we noted that the maximum number of parents has to be limited to ensure feasibility of the schemes for larger DAGs. The parameter `hardlimit` sets a limit on the

number of parents any node may have. If the initial search space is higher than this limit, the functions `sampleBN` and `learnBN` stop execution and warn the user to redefine the search space or increase `hardlimit`. The runtimes for computing the scores of all possible parent sets for a node with  $K$  possible parents in the search space  $\mathcal{H}$  and its extension  $\mathcal{H}^+$  are given in Section 7.

When the parameter `algorithm = "orderIter"`, the parameter `iterpar` includes additional parameters to define the iterative expansions of the search space. The element `pluslit` defines the number of iterations of expansion of the search space; In the function `learnBN`, when `pluslit` is not specified, the search space expands until no edges can be added to the search space to improve the score of a maximally scoring DAG  $\mathcal{G}^{\max}$ . When `sampleBN` is used, the expansion graph  $\mathcal{G}^*$  is estimated at each iteration on the basis of a sample of graphs and a posterior probability threshold given by the element `posterior` of the parameter `iterpar`.  $\mathcal{G}^*$  includes all edges with posterior probabilities higher than the threshold. When extending the search space, the maximal parent set size  $K$  may increase as well. The parameter `hardlimit` sets a limit on the number of parents any node may have. When we hit the limit for one node, the algorithm prevents adding further elements to that node's parent set, but it can still expand the parent sets of other nodes until they all reach the limit or the score does not improve further. Another element of `iterpar` controlling the expansion of the search space is `mergetype`. The possible values of `mergetype`, namely `dag`, `cpdag`, and `skeleton`, correspond to merging the core space  $\mathcal{H}$  with a maximally scoring graph  $\mathcal{G}^{\max}$  (or posterior-based  $\mathcal{G}^*$  in `sampleBN`), its equivalence class or a skeleton accordingly.

### 3.3. Bayesian model averaging and posterior model selection

To calculate posterior probabilities of single edges based on a sample of graphs from MCMC schemes we can use the function

```
edgep(MCMCchain, pdag = FALSE, burnin = 0.2, endstep = 1)
```

where the parameter `MCMCchain` is an object of one of the classes `'orderMCMC'`, `'partitionMCMC'` or `'iterativeMCMC'`. The parameter `burnin` defines the proportion of samples to discard as burn-in. We can also perform posterior model selection by constructing a graph consisting only of edges with posterior probability higher than a certain threshold with the function

```
modelp(MCMCchain, p, pdag = FALSE, burnin = 0.2)
```

which however is not guaranteed to result in a DAG. When building a consensus graph from a sample of DAGs it is possible to account for the uncertainty related to equivalence class by setting the parameter `pdag` to `TRUE`. In this case, we first convert all DAGs in the sample to CPDAGs corresponding to their equivalence classes. In this case, the resulting graph is not guaranteed to be either a DAG or a CPDAG. However, this option is still recommended since it results in more stable posterior probabilities of non-directed edges. In model averaging, the focus is on selecting individual edges rather than on specific structure requirements. For example, model selection based on posterior probabilities of individual edges was beneficial for small datasets, where estimated MAP DAGs contained more than half of false-positive edges (Kuipers *et al.* 2022).

### 3.4. Diagnostic plots

The convergence of the MCMC schemes is essential both for sampling from the posterior distribution as well as for MAP discovery. It is generally impossible to prove that the Markov chain has converged. However, diagnostic plots may help analyzing convergence and spotting cases when convergence was not reached. Trace plots are the basic tool for convergence diagnostics. For objects of classes ‘`orderMCMC`’, ‘`partitionMCMC`’ and ‘`iterativeMCMC`’, the method `plot` is available, which plots the trace of log scores of sampled DAGs.

To plot the changes in posterior probabilities of all single edges with the addition of new graphs from the sample according to Equation 2 we can use the function

```
plotpedges(MCMCtrace, cutoff = 0.2, pdag = FALSE, onlyedges = NULL,
  highlight = NULL, ...)
```

Large fluctuations of posterior probabilities are possible at the beginning, but while approaching convergence posterior probabilities should also reach stable levels.

Convergence diagnostic plots based on a single chain may be misleading. For a better understanding of convergence we can examine jointly several independent MCMC runs with random starting points. If all chains converge, the DAGs in each chain should represent the posterior distribution in a similar way. Posterior probabilities of single edges calculated on the basis of each sample should then be close to each other. If some chains do not converge, we are likely to see significant differences between posterior probabilities of single edges. We can plot the concordance between pairs of MCMC runs using the function

```
plotpcor(pmat, highlight = 0.3, printedges = FALSE, cut = 0.05, ...)
```

where the parameter `pmat` is a list of matrices containing posterior probabilities of single edges; such a list can be created by applying the function `edgep` to a list of objects of class ‘`orderMCMC`’ or ‘`partitionMCMC`’. We can also inspect the edges whose posterior probabilities differ by more than `highlight` in the first two matrices by setting `printedges` to `TRUE`.

The functions `bidag2coda` and `bidag2codalist` transform the objects of classes ‘`orderMCMC`’ and ‘`partitionMCMC`’ (or a list of such objects) into objects of classes ‘`mcmc`’ and ‘`mcmc.list`’ from the R-package `coda` (Plummer, Best, Cowles, and Vines 2006). The package `coda` contains standard tools for convergence diagnostics of MCMC chains of continuous or categorical variables. However, samples of DAGs usually cannot be analyzed by standard convergence diagnostic tools. For this reason, the MCMC objects constructed by `bidag2coda` and `bidag2codalist` contain the traces of DAG scores from a single or multiple MCMC chains that can be visualized in one plot for mixing diagnostics using `coda` functions `traceplot` and `densplot`. When the parameter `edges` is set to `TRUE`, instead of score traces, these functions compute the traces of posterior probabilities of single edges, that are continuous and can be used to perform convergence diagnostics based on the potential scale reduction factor (PRSF) that compares within- and between-chain means and variances (Gelman and Rubin 1992; Goudie and Mukherjee 2016). The function `gelman.diag` from the package `coda` can be used to compute PRSF. However, as noted by Goudie and Mukherjee (2016) this diagnostics failed to produce reliable results for DAGs. The most common convergence diagnostics of MCMC schemes for DAGs is implemented in the **BiDAG** function `plotpcor`, hence it is the recommended option.

### 3.5. Model comparison

The function `DAGscore` computes the score of a single DAG. When the goal is MAP discovery, we can use this function to compare structures estimated by different algorithms. We can also compare scores of the estimated structures to the score of the ground truth DAG when the latter is known.

To compare the performance of structure learning algorithms it is useful to assess how close the estimated structure is to the ground truth DAG on the basis of a certain distance measure. The function `compareDAGs` allows several measures: the number of true-positive edges (TP), the number of false-positive edges (FP), the number of false-negative edges (FN), the structural Hamming distance (SHD) and others; see `?compareDAGs` for a detailed list of measures and formulas. All measures apart from SHD refer to differences in the skeletons of two DAGs, i.e., the directions of the edges are disregarded. SHD equals the sum of all types of mistakes: false negatives, false positives, and edges with erroneous directions. The functions `plotdiffs`, `plotdiffsDBN` and `plot2in1` can be used to visualize the differences and similarities between two graphs.

## 4. Structure learning of dynamic Bayesian networks

A dynamic Bayesian network (DBN) is a graphical model that encodes temporal relationships between random variables in  $\mathbf{X}$ . A DBN defines a joint probability distribution over  $\mathbf{X}^t = (X_1^t, \dots, X_n^t)$  for all discrete time points  $t = 1, \dots, T$ . The random variable  $X_i^t$  describes feature  $i$  at time point  $t$ . In **BiDAG**, we consider first-order homogeneous DBNs, where the conditional probability distributions  $P(\mathbf{X}^t \mid \mathbf{X}^{t-1})$  are assumed to be the same for all time points  $t$ . In a first-order DBNs, variables in time slice  $t$  can only depend on other variables in the same time slice or on variables in the previous time slice  $t - 1$ . The structure of a first-order homogeneous DBN  $\mathcal{G}$  is fully specified by the initial structure  $\mathcal{G}_0$  and the transition structure  $\mathcal{G}_{\rightarrow}$ .  $\mathcal{G}_0$  represents the structure of the first slice, and  $\mathcal{G}_{\rightarrow}$  represents the structure for transitioning between any pair of consecutive time slices. Specifically,  $\mathcal{G}_{\rightarrow}$  includes intra-slice edges, i.e., edges between the nodes within the same time slice and inter-slice edges, i.e., edges from the nodes in a previous time slice to the current time slice. Intra-slice edges are also called instantaneous or contemporaneous to characterize the time interval in which interactions between the corresponding variables occur. Similarly, inter-slice edges are also called time-delayed or temporal. The unfolded DBN structure  $\mathcal{G}$  shown in Figure 1 can be more compactly represented as the two structures  $\mathcal{G}_0$  and  $\mathcal{G}_{\rightarrow}$  shown in Figure 2.

**BiDAG** can also be used for learning DBNs from data. When we initialize the score object with the function `scoreparameters` we set the parameter `DBN` to `TRUE` (the default is `FALSE`). The `data` object must adhere to a special DBN format to perform structure learning correctly. The number of columns must equal the number of variables in all time slices  $b + n \cdot T$ , where  $b$  is the number of static variables,  $n$  the number of dynamic variables in one time slice and  $T$  the number of time points. All  $b$  static variables, if present, have to be in the first  $b$  columns of the `data` and  $b$  should be specified via the parameter `dbnpar`, containing a list of variables specific to DBNs. We assume that static variables are present in every time slice, but since they do not change over time we need to store their values only once. The next  $n \cdot T$  columns should store the observations of the dynamic variables over all time slices. They need to be ordered in such a way that for each group of  $n$  variables, the  $i^{\text{th}}$  column of group  $t$  contains the observations of the variable  $X_i^t$ .

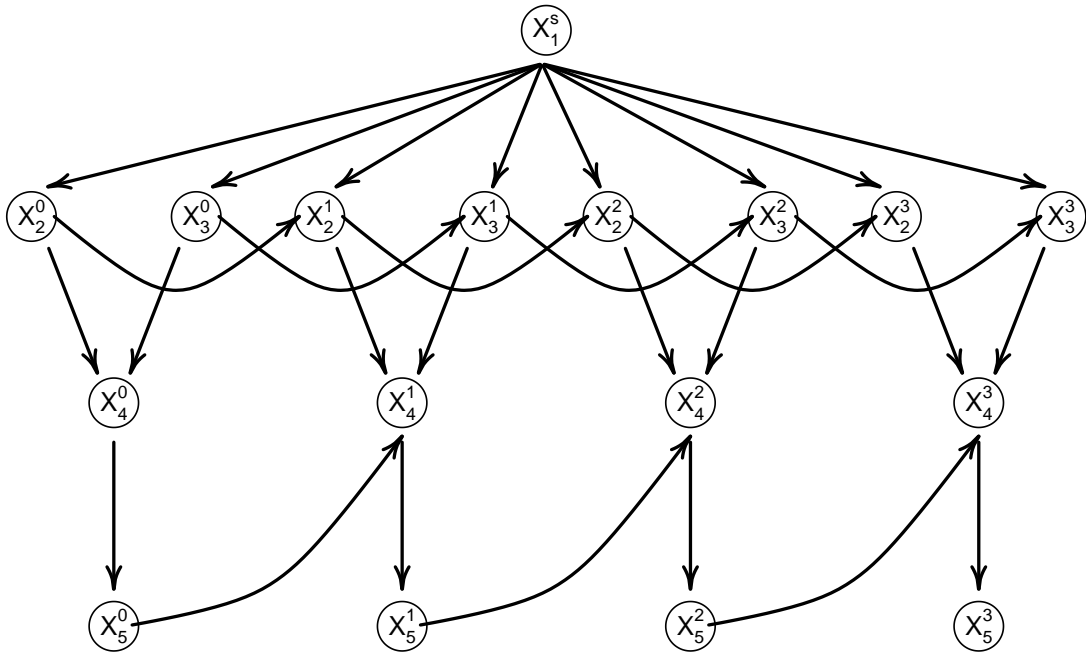


Figure 1: Unfolded structure of a first-order DBN consisting of four time slices. Each time slice includes one static variable  $X_1^S$  and four dynamic variables  $X_2^t, X_3^t, X_4^t, X_5^t$ , for  $t = 0, 1, 2, 3$ .

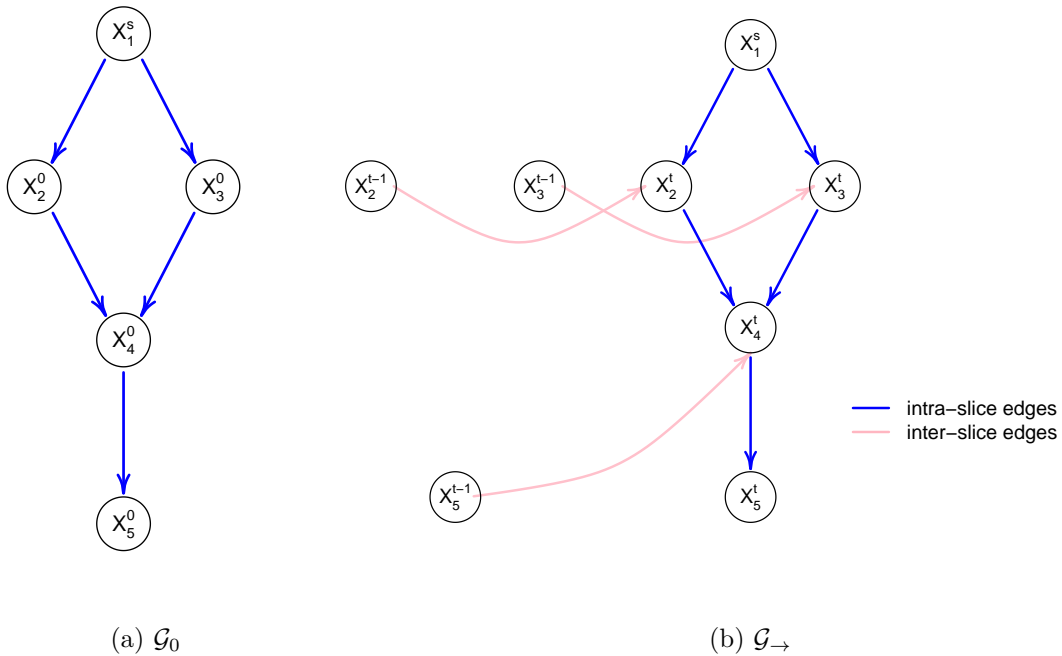


Figure 2: Initial  $\mathcal{G}_0$  and transition  $\mathcal{G}_{\rightarrow}$  structures representing the first-order DBN whose unfolded structure is depicted in Figure 1.

In **BiDAG**, we consider a special case when the structure within the first time slice is the same as the internal structure in all other time slices (including the edges from static nodes), which we indicate by setting to `TRUE` the slot `samestruct` in the parameter `dbnpar`. Otherwise we learn the initial and transition structures independently.

## 5. Examples on simulated data

We consider two data sets to demonstrate possible ways of working with the **BiDAG** package. The first simulated dataset is `gsim100`; it includes 100 observations generated from a randomly generated DAG with  $n = 100$  nodes, corresponding to Gaussian random variables. The second simulated dataset, `DBNdata`, contains observations from five consecutive time points of a DBN consisting of 12 dynamic 3 static variables.

### 5.1. MAP discovery

We first demonstrate how to use the algorithms in **BiDAG** for MAP discovery, which we can perform via the function `learnBN`. The parameter `algorithm` can be either `order` or `orderIter` and defines the approach to construct the search space.

To run any of the implemented MCMC schemes, we need to construct an object of class `'scoreparameters'`.

```
R> set.seed(4419)
R> library("BiDAG")
R> data("gsim100", package = "BiDAG")
R> score100 <- scoreparameters("bge", gsim100)
```

We first learn the MAP network from `gsim100` dataset by running `order` MCMC on a core search space  $\mathcal{H}$  (`plus1 = FALSE`) defined by the PC algorithm. It is the least computationally expensive of all options to define a search space but also prone to mistakes.

```
R> basefit <- learnBN(scorepar = score100, algorithm = "order", plus1 = FALSE)
```

The score of the maximum DAG found in the core search space is lower than the score of the ground truth structure stored as the adjacency matrix `gsimmat`:

```
R> getMCMCscore(basefit)
```

```
[1] -17952.42
```

```
R> data("gsimmat", package = "BiDAG")
R> DAGscore(scorepar = score100, incidence = gsimmat)
```

```
[1] -15239.79
```

By looking at structural differences, we can see that most differences in the estimated equivalence class come from the low number of discovered true-positive edges:

```
R> compareDAGs(getDAG(basefit), gsimmat,
+   cpdag = TRUE)[c("TPR", "FPRn", "SHD")]
```

```
   TPR  FPRn  SHD
0.58  0.03 94.00
```

The TPR of the highest scoring graph found in the core search space is only 58%. In an attempt to improve the search space and estimate a better DAG, we use the iterative MCMC by setting `algorithm = "orderIter"`:

```
R> iterativefit <- learnBN(score100, algorithm = "orderIter",
+   scoreout = TRUE, verbose = FALSE)
```

For each expansion iteration, the algorithm constructs a new MCMC chain, and the scheme may take a while to run. When the parameter `verbose` equals `TRUE`, messages in the output indicates the iteration currently running.

```
R> summary(iterativefit)
```

```
object of class 'iterativeMCMC'
```

```
Results:
```

```
maximum score DAG with 100 nodes and 199 edges:
maximum DAG score= -15195.67
```

```
algorithm: iterative order MCMC
number of search space expansion steps: 7
number of edges in the intial search space: 204
number of added edges: 196
total number of MCMC iterations: 1127000
total number of MCMC sampling steps (length of trace): 7007
number of MCMC iterations per expansion step: 161000
number of MCMC sampling steps per expansion step: 1001
initial search space: PC
sample/MAP: MAP
```

```
Additional output:
```

```
scoretable, object of class 'scorepace'
```

The iterative order MCMC scheme added 196 edges to the initial PC-defined search space in 7 iterations. We can observe how the score improved with each search space expansion step by looking at the trace plot depicted in Figure 3:

```
R> plot(iterativefit)
```

The scores of the DAGs sampled at the final expansion step improved significantly compared to the initial step. Moreover, the score of a MAP DAG found in the last iteration by iterative



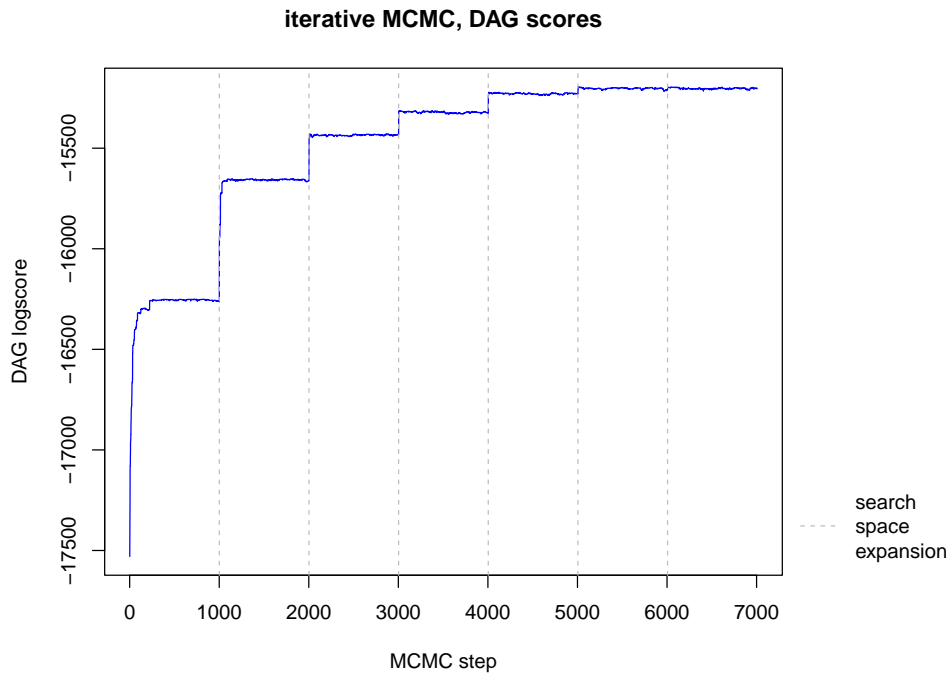


Figure 3: Trace plot of saved DAG scores obtained by iterative MCMC.

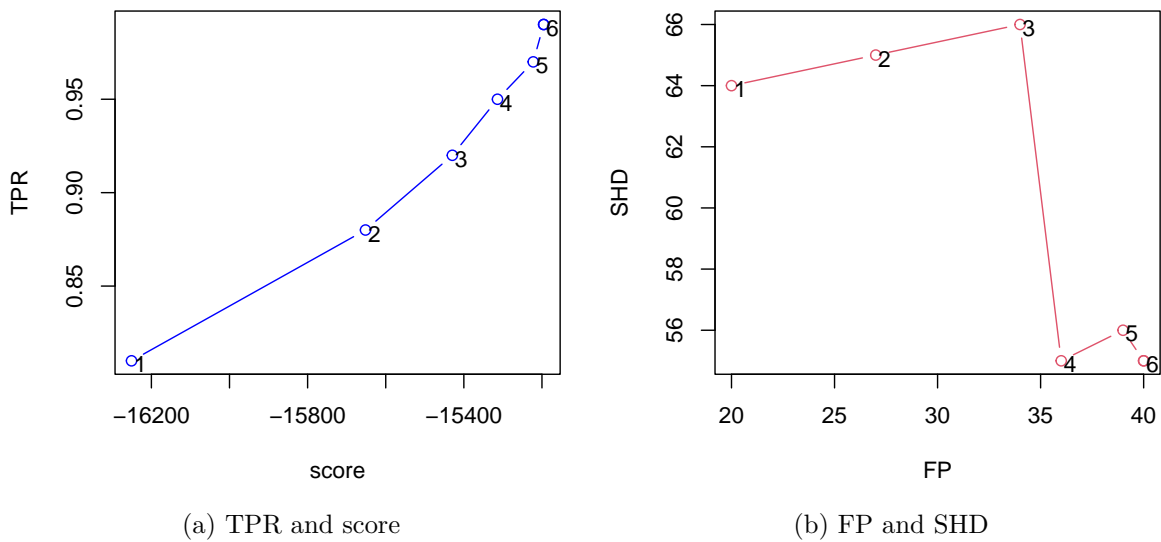


Figure 4: Structure fit changes through iterative expansions of the search space: the iterative MCMC scheme was applied to the simulated dataset `gsim100`. At each search space expansion, the MAP DAG is stored together with its score and compared to the ground truth structure with the function `itercomp`. For (a), TPR and DAG score were used, while for (b), FP and SHD were used to visualize the changes.

MCMC ( $-15195.67$ ) is higher than the score of the ground truth structure ( $-15239.79$ ), and much higher than the score of a MAP graph found in the core PC-defined search space ( $-17952.42$ ). Outside of simulation studies, DAG score is the most used criterion that informs model selection and iterative MCMC has shown great performance in maximizing the score (Kuipers *et al.* 2022).

Since we know the ground truth DAG, we can use the function `itercomp` to assess how close the estimated MAP structures are to the true DAG with each expansion of the search space:

```
R> it100 <- itercomp(iterativefit, gsimmat)
R> plot(it100, vars = c("score", "TPR"), showit = 1:6)
R> plot(it100, vars = c("FP", "SHD"), col = 2, showit = 1:6)
```

As visualized in Figure 4a, the results of this comparison show that the TPR grows as the score increases with each search space expansion and is very close to 1 in the last iteration. However, FP grows as well, and thus, the improvement of SHD is not as impressive as of TPR (Figure 4b).

Simulation studies help us by guiding what we can expect from applying a particular method in a specific simulation setting, e.g., low sample size. In our example, using the PC-defined search space results in a low TPR. While iterative MCMC helps with the TPR and the DAG score, it does not necessarily result in the best structure fit, as previously mentioned, due to a possible increase in false positives. In the next section, we will describe how to use **BiDAG** to obtain consensus graphs that help mitigate this problem.

## 5.2. Sampling graphs from posterior distribution

So far, we focused on finding one maximally scoring DAG. For sampling from the posterior distribution, we can use the function `sampleBN`. In addition to two algorithms available for MAP discovery, `sampleBN` suggests the option `algorithm = "partition"` implementing partition MCMC.

For sampling, it is important that the search space includes as many true positives as possible. The iterative MCMC scheme has been shown to optimize the search space in multiple simulation settings successfully (Kuipers *et al.* 2022). Therefore, we can pass the search space optimized in Section 5.1 to the sampling function via the parameters `startspace` or `scoretable`. The parameter `startspace` allows passing an arbitrary adjacency matrix. In contrast, `scoretable` requires an object of class ‘`scorespace`’ which in addition to the adjacency matrix includes the score tables described in Section 2 and hence decreases the runtime.

```
R> iterSpace <- getSpace(iterativefit)
R> orderfit <- sampleBN(score100, algorithm = "order",
+   scoretable = iterSpace)
R> plot(orderfit)
```

For MCMC sampling schemes, it is important to check if the chain has converged, and we can look at diagnostic plots, which may highlight lack of convergence. The trace plot in Figure 5 shows the scores of all sampled DAGs. When a random order is used as a starting point, typically, the scores increase sharply in the beginning, reflecting the burn-in period of the chain. A sharp increase is visible on the left subgraph, while the right subgraph shows

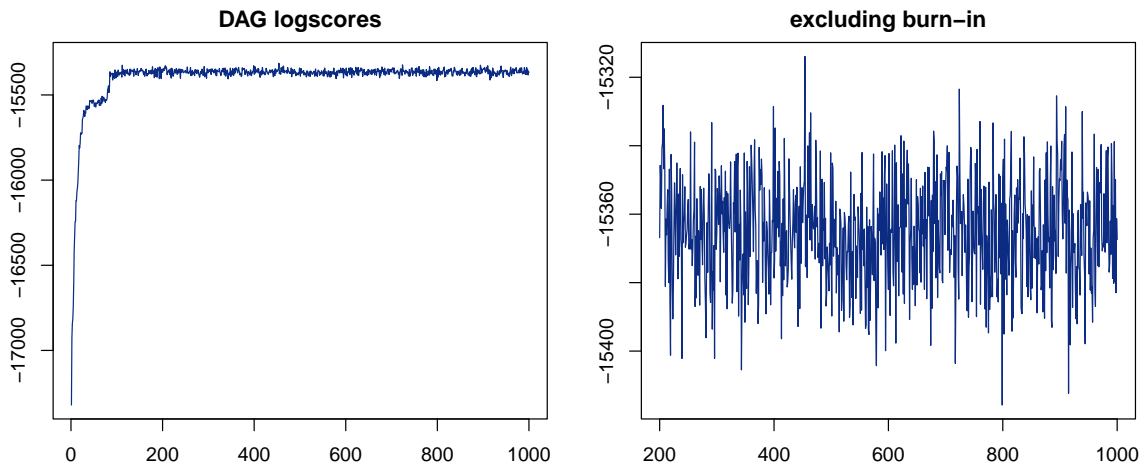


Figure 5: Trace plot of DAGs sampled by the order MCMC scheme. The sampling is performed on a search space that was previously optimized by iterative MCMC.

the trace plots of scores after excluding the burn-in period. If we choose the burn-in period adequately, the scores on the right will stay in a narrow stable range. To modify the default burn-in period of 20% we can set the parameter `burnin` to another value.

To demonstrate a more rigorous convergence diagnostic, we need to run MCMC two (or more) times for the same data but with different starting points. By default, the starting point of each run is random unless the parameter `startorder` is set to a specific value.

We proceed with computing posterior probabilities of edges based on each of the two chains with the function `edgep` and visualize the results with the function `plotpcor`, Figure 6.

```
R> orderfit2 <- sampleBN(score100, algorithm = "order",
+   scoretable = iterSpace)
R> epd <- lapply(list(orderfit, orderfit2), edgep, pdag = TRUE)
R> plotpcor(epd, xlab = "run 1", ylab = "run 2")
```

The concordance plot in Figure 6a does not indicate any convergence issues. All points are close to diagonal, meaning that posterior probabilities of single edges based on two samples of DAGs produced by order MCMC are close to each other.

We repeat the same procedure for a pair of runs of partition MCMC. The concordance plot in Figure 6b indicates convergence issues. Several edges have a high posterior probability in one run and low in the other. When the concordance plot looks concerning, increasing the number of `iterations` and repeating sampling is recommended. However, even with the default number of iterations, partition sampling takes longer than order sampling, and any further increase in runtime might be unwanted. It is important to be aware of the convergence and complexity properties of both approaches in order to choose the right trade-off between runtime and prior over structures. As mentioned in Section 2.2, partition MCMC was designed in order to impose a uniform prior over structures. However, this feature comes at the cost of slower convergence and longer runtimes. In this example, we proceed with the sample obtained by the order MCMC scheme for model selection. Section 6 will demonstrate an example using real data, where the convergence diagnostics of the partition MCMC sample does not indicate any issues.

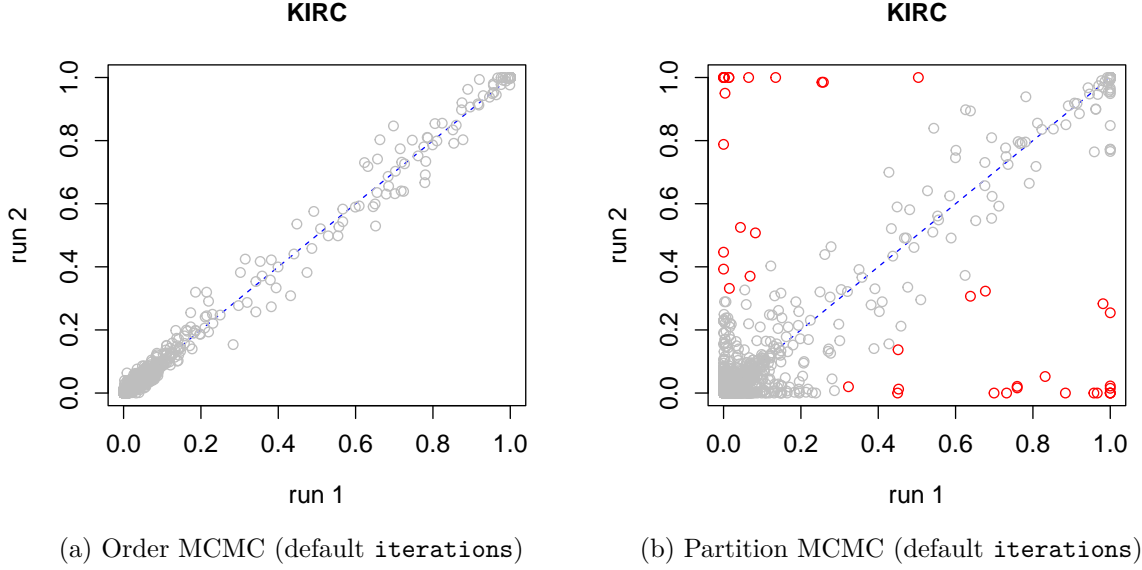


Figure 6: Convergence diagnostic plot: concordance of posterior probabilities estimates of single edges between pairs of MCMC runs. For each subfigure two pairs of samples of DAGs were obtained by order/partition MCMC schemes. Posterior probabilities of all edges were calculated based on each sample with the function `edgep` and visualized pair-wise with `plotpcor`. Red points correspond to edges whose posterior probabilities differ by more than 0.3 between two samples.

In Section 5.1, we have compared an estimated MAP DAG to the ground truth structure by comparing their scores and skeletons. The estimated MAP DAG contains 39 false-positive edges, around 20% of all edges in the discovered DAG. Simulation studies show that in cases when the number of observations is low, even high scoring structures may contain a lot of false-positive edges (Kuipers *et al.* 2022). With a sample of DAGs from the posterior distribution, we can apply an alternative approach to model selection based on posterior probability estimates of single edges, by using the function `modelp` and setting the parameter `p` to a desired value. Since we also know the ground truth structure in this example, we can use the function `samplecomp` to demonstrate how models selected based on a range of posterior thresholds compare to the ground truth. We set the parameter `pdag` to `TRUE` to account for equivalence class uncertainty and accordingly, in each case, compare the chosen model to an equivalence class corresponding to the ground truth DAG:

```
R> samplecomp(orderfit, gsimmat, pdag = TRUE, p = c(0.5, 0.7, 0.9, 0.95))
```

```
object of class 'samplecomp'
```

	TP	FP	FN	TPR	FPR	FPRn	FDR	SHD	p
1	159	29	2	0.99	0.01	0.18	0.15	38	0.50
2	157	16	4	0.98	0.00	0.10	0.09	25	0.70
3	144	3	17	0.89	0.00	0.02	0.02	20	0.90
4	141	1	20	0.88	0.00	0.01	0.01	21	0.95

Each row in the table corresponds to the result of comparing a consensus graph based on

	$N = 100$			$N = 1000$		
	TP	FP	SHD	TP	FP	SHD
MAP	159	40	55	161	8	12
$p = 0.50$	159	29	38	161	4	6
$p = 0.90$	144	3	20	154	1	8
$p = 0.95$	141	1	21	146	1	18

Table 1: Comparison between MAP and posterior threshold-based models for the two data sets `gsim100` and `gsim` generated from the same graph and containing 100 and 1000 observations accordingly.

the posterior threshold in the last column to the ground truth CPDAG. For example, a graph consisting only of edges with an estimated probability higher than 0.90 contains only three false-positive edges, while maintaining a rather high TPR of 89%. To further show in which settings posterior model selection based on a threshold may provide an advantage over choosing one highest scoring model, we also applied a similar MCMC scheme to the dataset generated from the same network but with a larger number of observations (dataset `gsim`,  $N = 1000$ ). The results of comparing the estimated models with the ground truth structure are summarized in Table 1.

For both sample sizes, a posterior threshold-based model for  $p = 0.5$  has as many true edges as the MAP estimate while reducing the number of false-positive edges. A more stringent threshold can further reduce false positives. For example, for  $N = 100$  and  $p = 0.95$  there was only one false-positive edge in the estimated model compared to 20% in the MAP DAG. We can also observe that for  $N = 100$ , the reduction in false-positive edges is more pronounced than for  $N = 1000$ . As a result, SHDs between consensus and ground truth models are smaller than SHDs between MAP and the ground truth models for all posterior thresholds and  $N = 100$ , while for  $N = 1000$ , the lower SHD is only observed for  $p = 0.5$ . Similar results were observed in larger-scale simulation studies by Kuipers *et al.* (2022).

### 5.3. Learning DBNs

Both functions `learnBN` and `sampleBN` can also be applied to structure learning and sampling of DBNs. Here we consider a simulated example of a DBN consisting of 12 dynamic and 3 static nodes. The data includes samples generated from five consecutive time slices. The syntax of structure learning functions then is the same as for usual Bayesian networks. Here we again learn the MAP estimate and optimize the search space with the function `learnBN` and `algorithm = "orderIter"`.

```
R> set.seed(4419)
R> data("DBNdata", package = "BiDAG")
R> data("DBNmat", package = "BiDAG")
R> DBNscore <- scoreparameters("bge", DBNdata, DBN = TRUE,
+   dbnpar = list(samestruct = TRUE, slices = 5, b = 3))
R> DBNfit <- learnBN(DBNscore, algorithm = "orderIter", verbose = FALSE,
+   scoreout = TRUE)
R> plotdiffsDBN(getDAG(DBNfit), DBNmat, "trans", b = 3)
```

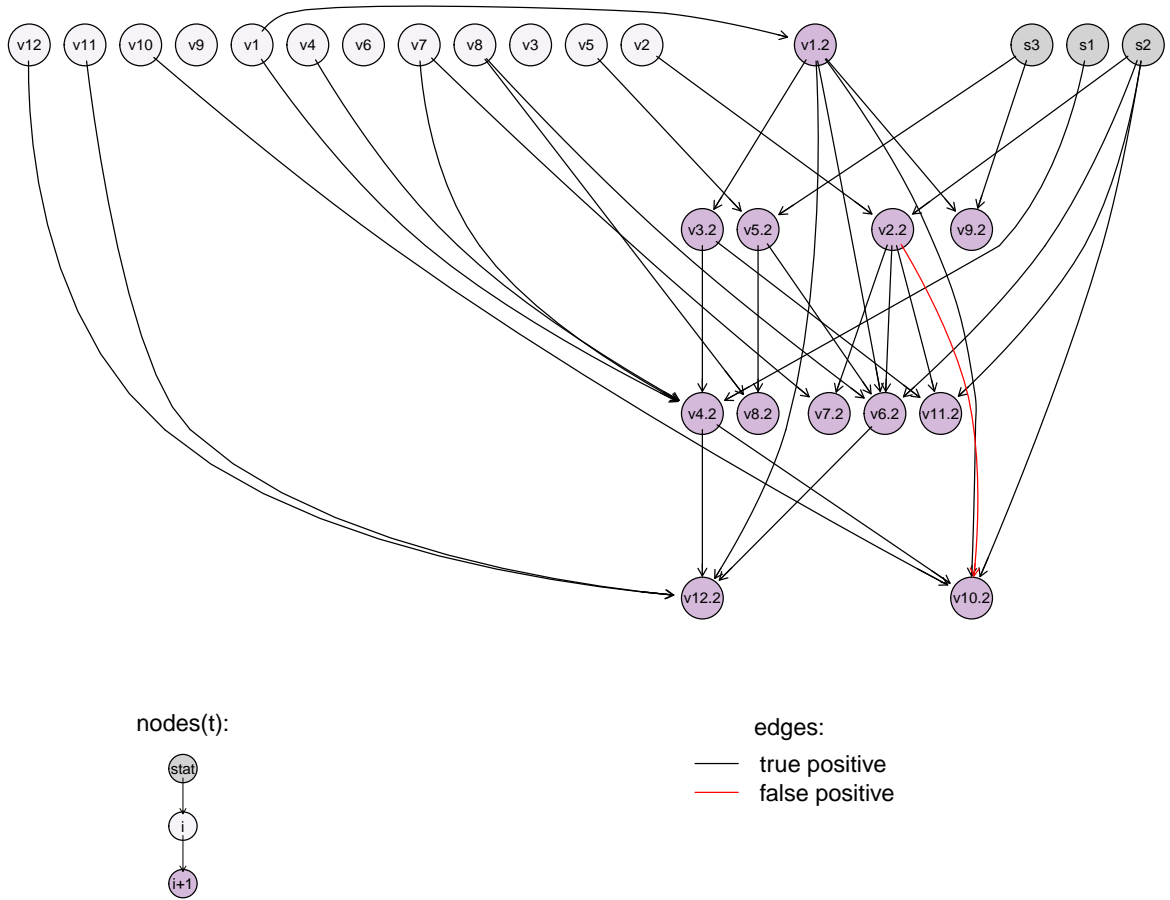


Figure 7: Transition structure of MAP estimate of a 15-node DBN. Edge colors highlight differences/similarities between the maximum scoring structure found by the iterative MCMC scheme and the ground truth transition structure.

Figure 7 demonstrates the transition structure of MAP estimate of a DBN learned from `DBNdata` and its differences with the ground truth structure (when the ground truth structure is unknown the function `plotDBN` can be used instead of `plotdiffsDBN` for visualization). The MAP transition structure found by iterative MCMC is very close to the ground truth structure with just two false-positive edges. Typically for DBNs, we observe many transitional edges connecting the same variable in neighboring time points  $i$  and  $i + 1$ .

## 6. Applications

Kuipers *et al.* (2018b) used **BiDAG** for learning structures of Bayesian networks that characterize mutation profiles across cancer types and novel subtypes. The dataset included mutational profiles of  $N = 8198$  tumor samples across 22 cancer types. For  $n = 201$  significantly mutated genes, a Bayesian network-based clustering approach was used to define clusters of tumor samples, such that a Bayesian network represented each cluster center. Structure learning was performed in two steps. In the first step, the function `learnBN` with the param-

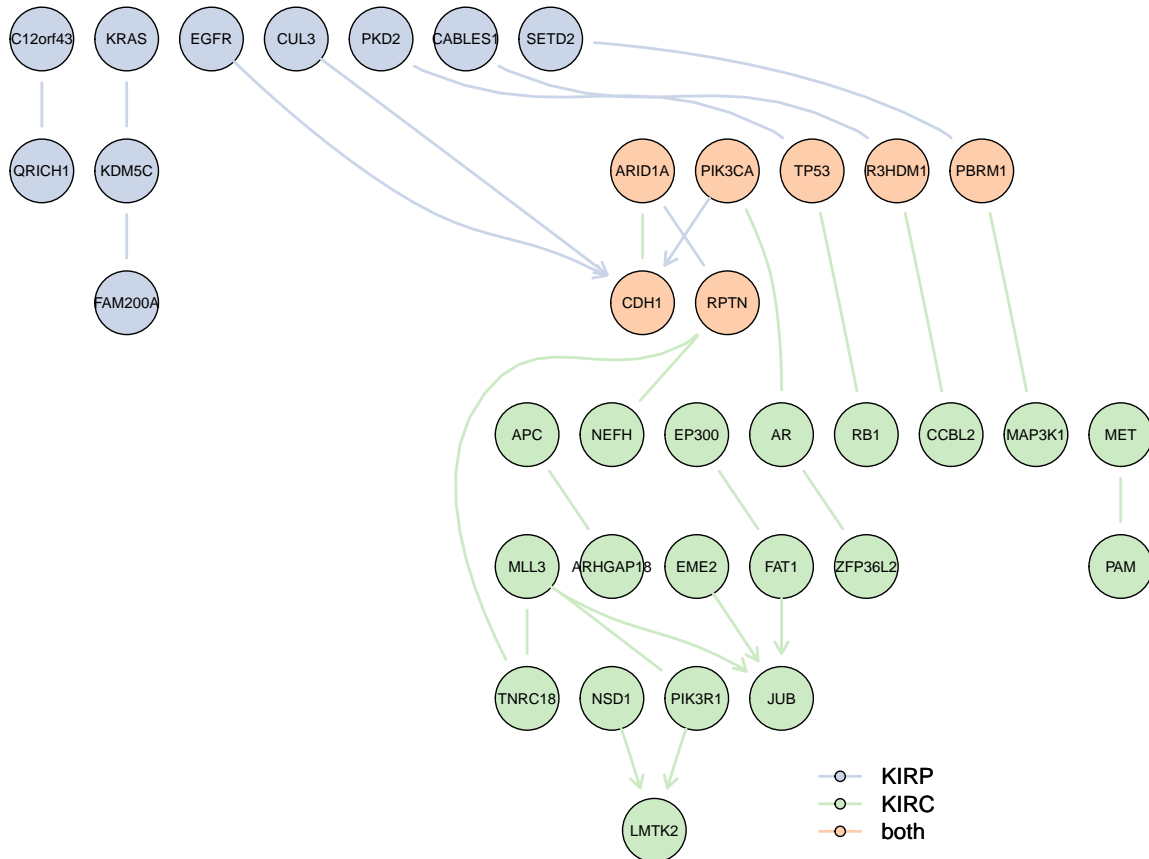


Figure 8: Joint graph representing KIRC and KIRP MAP CPDAGs obtained by iterative MCMC. Only nodes that have at least one connection are shown. Blue and green nodes/edges are specific to KIRP and KIRC graphs, respectively. Orange nodes have connections in both graphs.

eter algorithm = "orderIter" was used to optimize the search space. In the second step, sampling was performed with `sampleBN` with the parameter `algorithm = "partition"` on the optimized search space. Posterior model selection was performed based on the sample of 100 DAGs from the posterior distribution with a posterior threshold of 0.5. The code for unsupervised clustering as well as the example of using **BiDAG** for Bayesian network-based clustering can be found at <https://github.com/cbg-ethz/pancancer-clustering>. Kuipers *et al.* (2018b) discovered networks both in supervised and unsupervised settings. As a demonstration here we show how **BiDAG** can be used to characterize cancer subtypes in a supervised setting and follow the learning steps described in Kuipers *et al.* (2018b).

We analyze non-silent mutation data from two cohorts from the cancer genome atlas (TCGA, <https://www.cancer.gov/tcga>). The cohorts represent two kidney cancer subtypes: renal papillary cell carcinoma (KIRP) and kidney renal clear cell carcinoma (KIRC). We include the most significantly mutated genes ( $q < 0.1$ ) from both cohorts. Mutation data and corresponding lists of significantly mutated genes were obtained from Broad Institute TCGA Genome Data Analysis Center (2016a) and Broad Institute TCGA Genome Data Analysis Center (2016b). Additionally, we have included connected genes from KIRP and KIRC net-

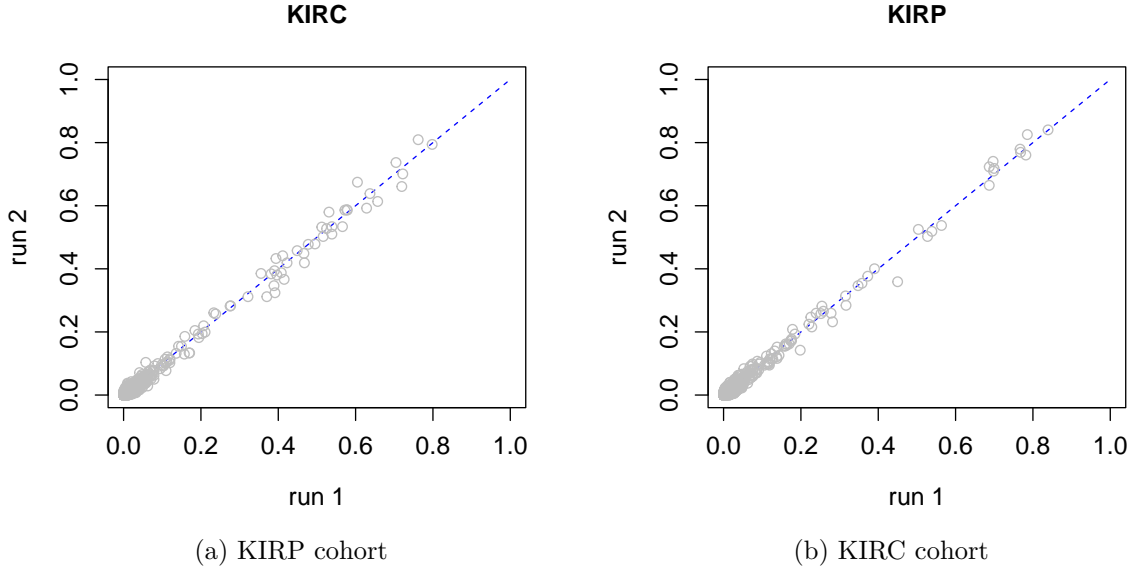


Figure 9: Convergence diagnostic plot: concordance of posterior probabilities estimates of single edges between pairs of partition MCMC runs.

works discovered by [Kuipers \*et al.\* \(2018b\)](#). Both pre-processed datasets are accessible in the **BiDAG** package.

Following the methods in [Kuipers \*et al.\* \(2018b\)](#), we use a prior derived from the protein-protein interaction database STRING ([Szklarczyk \*et al.\* 2019](#)). The edges that are not among interactions in the STRING database are penalized by a factor of 2 for graph inference. The database is being constantly updated, and known interactions between genes have changed considerably since the analysis reported by [Kuipers \*et al.\* \(2018b\)](#) was performed. Here, we use the most recent version 11.0 of the database. In **BiDAG**, the function `string2mat` transforms the downloaded list of interactions from STRING into a matrix, which can be used for blacklisting or penalizing of edges in **BiDAG**.

We run iterative MCMC to find MAP DAGs representing the KIRC and KIRP subtypes and corresponding equivalence classes. Figure 8, produced by the function `plot2in1`, shows edges from both discovered CPDAGs in one graph. The genes TP53, PIK3CA, ARID1A, PBRM1, CDH1, RPTN, R3HDM1 are connected to other nodes in both subgraphs representing KIRP and KIRC cohorts. However, the subgraphs do not share any edges.

We proceed with partition MCMC to understand how confident we can be about the discovered mutational interactions. To check convergence we use the `edgep` and `plotpcor` functions as in the previous section. The result is shown in Figure 9. There are no visible convergence problems, and all the points are close to the diagonal.

Another useful plot for checking convergence depicts how the posterior probabilities of single edges change through MCMC iterations. In Figure 10 we can see that posteriors of the vast majority of edges stabilize in both cases after a short burn-in period. Edges of the MAP CPDAG (highlighted in green) reach higher posterior probabilities than almost all other edges. However, many edges of MAP structures converge to posterior levels below 0.5. The posterior probability of an edge can be interpreted as a measure of confidence in the edge based on the data.



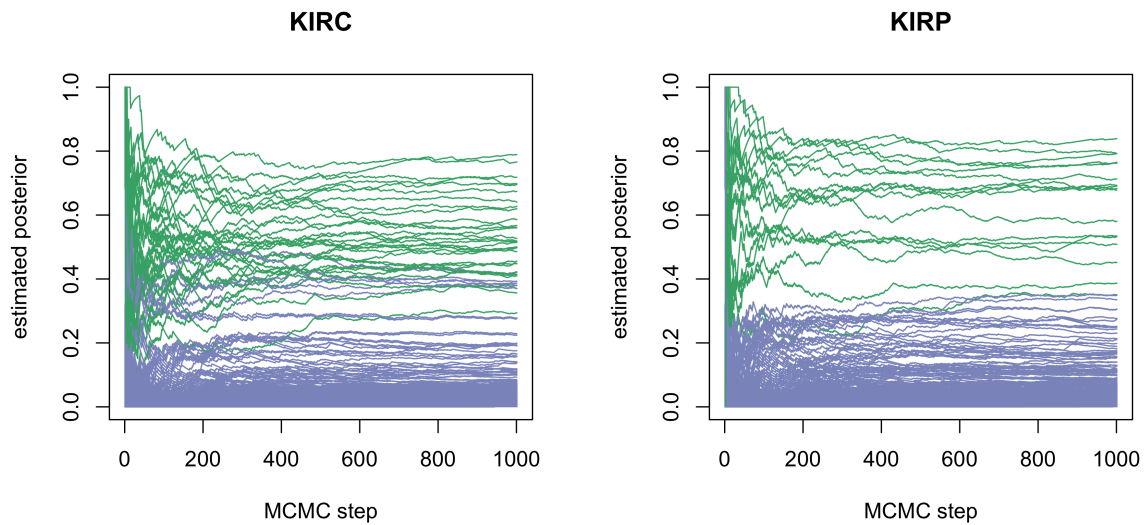
(a) KIRC, function `plotpedges`(b) KIRP, function `plotpedges`

Figure 10: Convergence of posterior probabilities of single edges: a pair of MCMC chains were run for each of KIRP and KIRC datasets. Posterior probability traces were obtained by applying Equation 2 ( $m = 1$ ) at each MCMC step ( $M = 1, 2, \dots, 1001$ ). Green lines correspond to traces of posterior probabilities of the edges of the estimated MAP CPDAG.

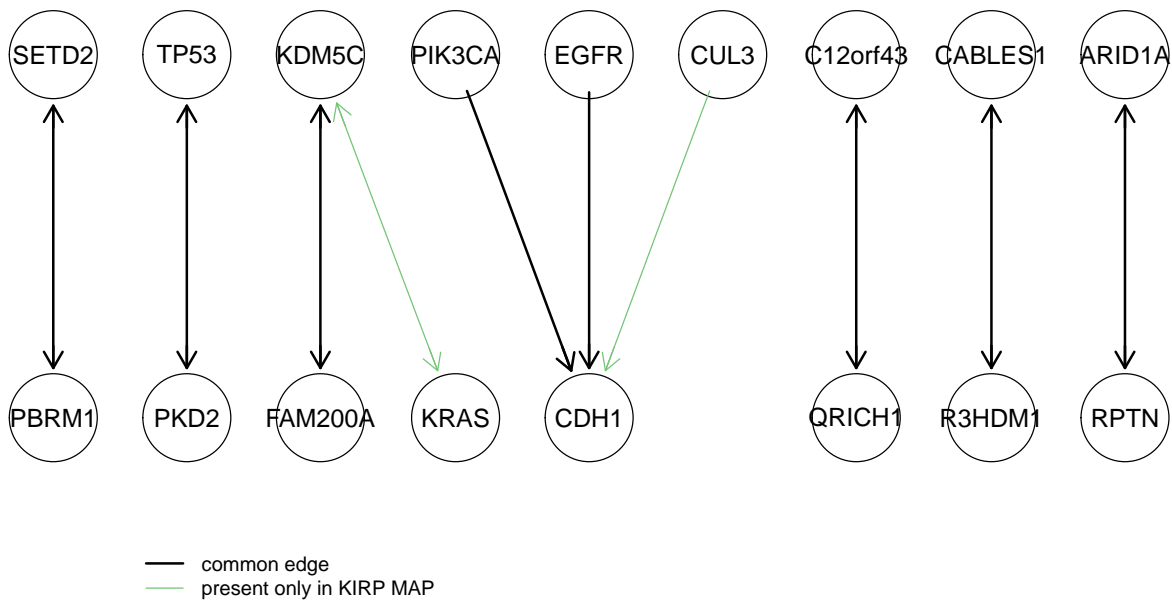


Figure 11: Comparison between MAP CPDAG and consensus model learned from the `kirp` dataset. MAP CPDAG was obtained with iterative MCMC. Consensus model was obtained by averaging over a sample of DAGs obtained by partition MCMC and converted to CPDAGs and keeping the edges whose posterior is bigger than 0.5.

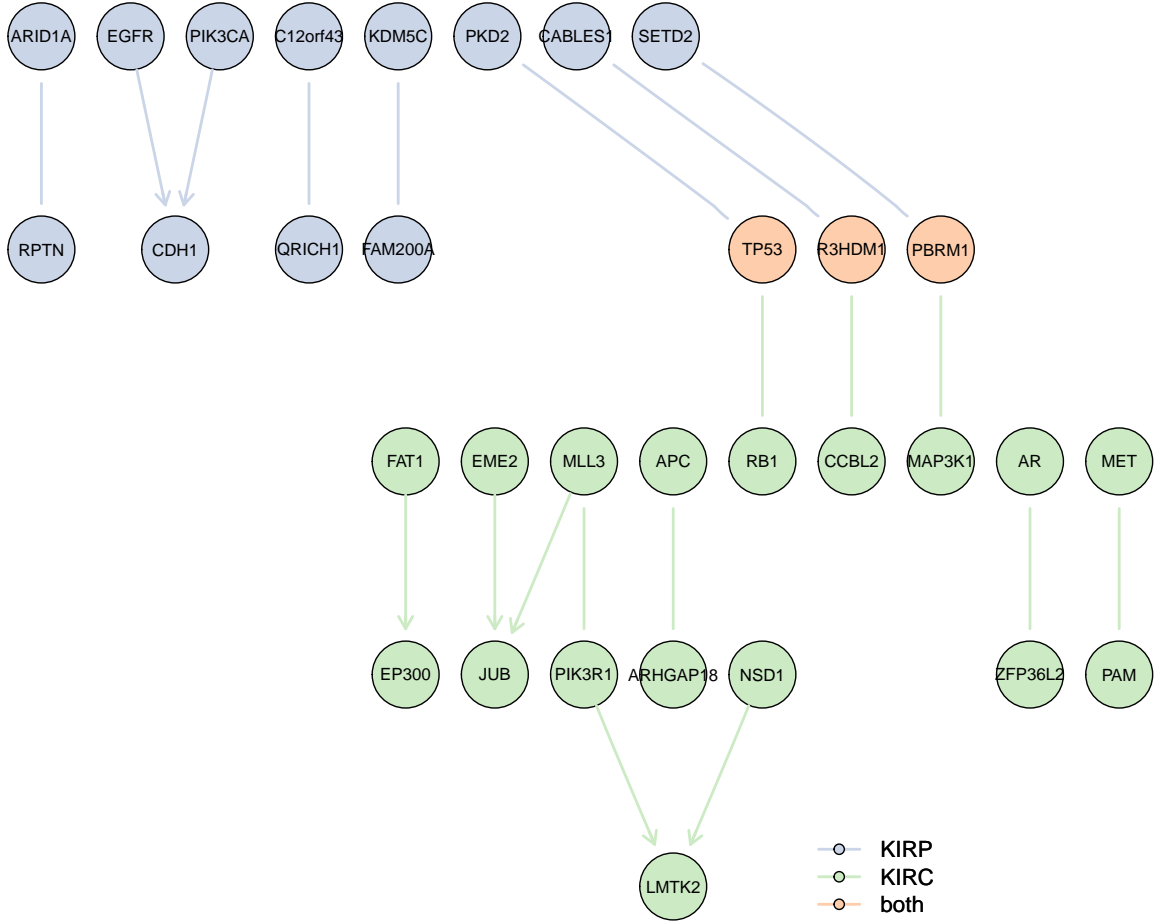


Figure 12: Joint graph for KIRC and KIRP cohorts obtained averaging over a sample from posterior distribution obtained by partition MCMC. Only nodes that have at least one connection and whose posterior probability is higher than 0.5 are shown. Blue and green nodes/edges are specific to KIRP and KIRC. Orange nodes have connections in both graphs.

Kuipers *et al.* (2022) have shown that when the data is noisy or scarce, the MAP graph may include a lot of edges with low posterior probability, many of which turn out to be false positives. We perform model averaging based on a sample of DAGs obtained by partition MCMC and remove the edges with a posterior less than 0.5. As we have seen in Section 5, this approach can help to remove false-positive edges, while keeping most of the true positives.

Figure 11 depicts differences between MAP and consensus graphs for the KIRP cohort. Eight edges out of ten passed the posterior threshold of 0.5. In the KIRC cohort, similarly, 12 out of 18 edges of MAP CPDAG passed the threshold of 0.5.

Figure 12 visualizes the consensus models for KIRP and KIRC in one graph. Many of the discovered edges correspond to those found previously by Kuipers *et al.* (2018b) for the respective cancer subtypes. Although we have followed the analysis steps from Kuipers *et al.* (2018b) on the same set of tumors, we used an updated prior and genes that are specific to analyzed cohorts. Consequently, the discovered networks display some differences. For example, we discovered an edge between CCBL2 and R3HDM1, but the gene CCLB2 was not included in the list of genes analyzed by Kuipers *et al.* (2018b).

## 7. Runtime

The number of MCMC iterations used in the order MCMC scheme by default is  $6n^2 \log n$ , for a network with  $n$  nodes. Each MCMC iteration requires the computation of the score  $R(\prec' | D)$  of at least one proposed order. When implemented naively, the complexity of scoring an order is exponential,  $O(n^{K+1})$ , where  $K$  is the maximum number of parents allowed in the scheme. This brings the total chain complexity to  $O(n^K n^2 \log n)$ . For efficient implementation, we use the approach and computational optimizations described by [Kuipers \*et al.\* \(2022\)](#) and pre-compute the quantities needed to score an order at each iteration of the MCMC scheme. We refer to this step further as pre-computing the score tables. This reduces the complexity of the chain by a factor of  $n^K$  to  $O(n^2 \log n)$  ([Kuipers \*et al.\* \(2022\)](#)). Of course, the complexity of computing the score tables remains exponential, but now it is independent of the number of MCMC iterations as it has to be done only once. In addition, using the search space  $\mathcal{H}$  instead of restricting the number of parents to a hard limit  $K$  reduces the complexity of computing the score tables to  $O(K^3 2^K)$  for order MCMC. In  $\mathcal{H}^+$  the complexity grows by a factor of  $n$  to  $O(nK^3 2^K)$  for order MCMC and to  $O(nK^2 3^K)$  for partition MCMC.

When the score tables are pre-computed, the complexity of the MCMC scheme is polynomial in the size of the number of nodes of the network,  $n$ , such that the algorithm is applicable to large networks with hundreds of nodes. The computation of the score tables is exponential in the maximal parent set size  $K$ , so  $K$  imposes a feasibility limit on the implemented algorithms. While no hard limit for  $K$  is required in **BiDAG**, for large  $K$ , the score table computations can become prohibitive.

Figure 13 shows how much time is needed to compute the score tables for a node with  $K$  par-

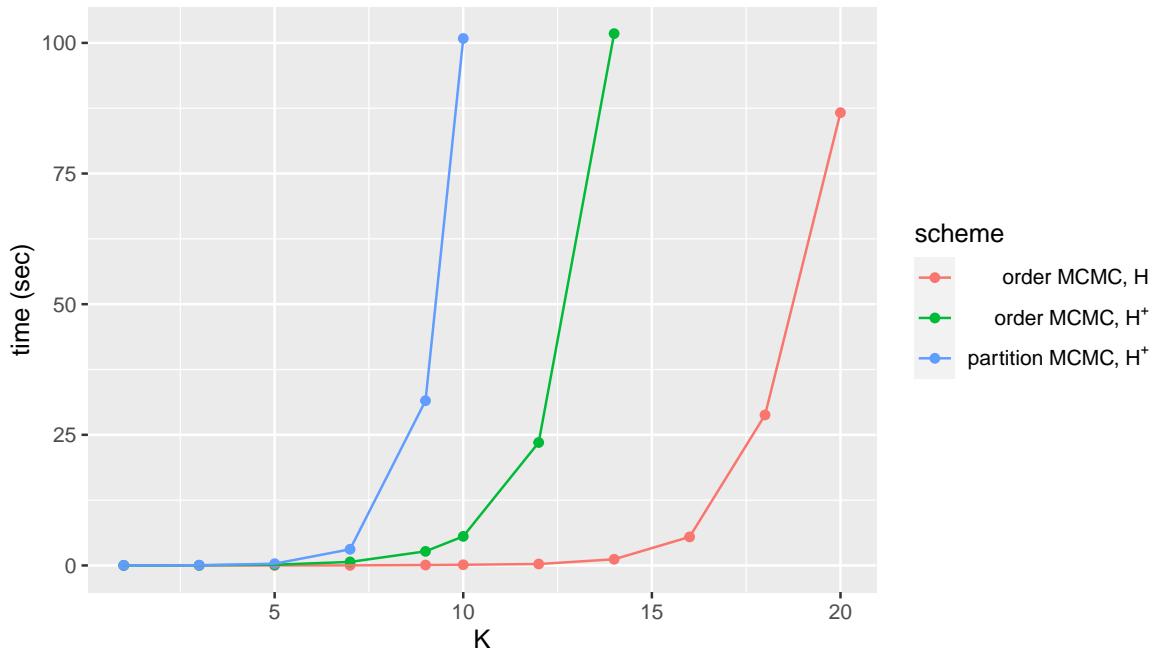


Figure 13: Time needed to compute a score table for a node with  $K$  parents for a network with  $n = 100$  nodes.

ents for a network with  $n = 100$  nodes. All timing measurements were carried out on a 2.3 GHz Intel Core i5 processor. For  $K > 7$ , the differences in runtimes of different MCMC schemes become substantial. As expected, the runtime is lowest for order MCMC sampling in  $\mathcal{H}$  and highest for partition MCMC. Building score tables in the core search space  $\mathcal{H}$  (`plus1=FALSE`) requires shorter time than in its extended version  $\mathcal{H}^+$  (`plus1=TRUE`), but scoring a node with up to 14 parents is feasible in both cases for the order MCMC scheme. Note, however, that most real-world networks are much sparser than that. For example, in 30 networks found in the Bayesian network repository (<http://www.bnlearn.com/bnrepository/>) the average parent set size is 1.4, while the maximum parent set size is 13.

The parameter `hardlimit` of structure learning functions `sampleBN` and `learnBN` ensures that the search space contains only nodes with parent set sizes not exceeding this limit. As mentioned in Section 3.2, the iterative MCMC scheme stops extending a node’s parent set when the `hardlimit` has been hit for this node, but it can still expand the parent sets of other nodes until they all reach the limit or the score does not improve further.

## 8. Discussion

The R package **BiDAG** implements flexible MCMC schemes for structure learning and sampling of Bayesian networks. The iterative MCMC scheme can be used to search for a MAP graph and to optimize the search space, while partition and order MCMC can be used for sampling from the posterior distribution. Order MCMC converges faster and is computationally less demanding than partition MCMC, but only the latter provides an unbiased sample of the posterior. Other tools for structure learning either focus on finding one best solution or implement Bayesian approaches, which are not feasible for large networks due to computational costs or slow convergence. **BiDAG** is the first package available for efficient sampling of DAGs with hundreds of nodes. In the future, we plan to implement features that could potentially reduce the runtimes of partition and iterative MCMC schemes. In the iterative MCMC scheme, we consider adding algorithms other than PC for defining the initial search space. Furthermore, the convergence of partition MCMC could be improved with the addition of new moves.

## References

- Bartlett M, Cussens J (2017). “Integer Linear Programming for the Bayesian Network Structure Learning Problem.” *Artificial Intelligence*, **244**, 258–271. doi:10.1016/j.artint.2015.03.003.
- Böttcher SG, Dethlefsen C (2003). “**deal**: a Package for Learning Bayesian Networks.” *Journal of Statistical Software*, **8**(20). doi:10.18637/jss.v008.i20.
- Broad Institute TCGA Genome Data Analysis Center (2016a). “Mutation Analysis (MutSig 2CV V3.1).” doi:10.7908/c19c6wtf.
- Broad Institute TCGA Genome Data Analysis Center (2016b). “Mutation Analysis (MutSig 2CV V3.1).” doi:10.7908/c10864rm.

- Chickering DM (1996). “Learning Bayesian Networks is NP-Complete.” In *Learning from Data: Artificial Intelligence and Statistics V*, 1st edition, pp. 121–130. Springer-Verlag. doi:10.1007/978-1-4612-2404-4\_12.
- Consonni G, Rocca LL (2012). “Objective Bayes Factors for Gaussian Directed Acyclic Graphical Models.” *Scandinavian Journal of Statistics*, **39**(4), 743–756. doi:10.1111/j.1467-9469.2011.00785.x.
- Drton M, Maathuis MH (2017). “Structure Learning in Graphical Modeling.” *Annual Review of Statistics and Its Application*, **4**(1), 365–393. doi:10.1146/annurev-statistics-060116-053803.
- Eaton D, Murphy K (2007). **BDAGL: Bayesian DAG Learning**. URL <https://www.cs.ubc.ca/~murphyk/Software/BDAGL/>.
- Franzin A, Sambo F, Camillo BD (2017). “**bnstruct**: An R Package for Bayesian Network Structure Learning in the Presence of Missing Data.” *Bioinformatics*, **33**(8), 1250–1252. doi:10.1093/bioinformatics/btw807.
- Friedman N, Koller D (2003). “A Bayesian Approach to Structure Discovery in Bayesian Networks.” *Machine Learning*, **50**, 95–125. doi:10.1023/a:1020249912095.
- Geiger D, Heckerman D (1995). “Learning Bayesian Networks: A Unification for Discrete and Gaussian Domains.” In *Proceedings of Eleventh Conference on Uncertainty in Artificial Intelligence*, pp. 274–284.
- Geiger D, Heckerman D (2002). “Parameter Priors for Directed Acyclic Graphical Models and the Characterization of Several Probability Distributions.” *The Annals of Statistics*, **30**(5), 1412–1440. doi:10.1214/aos/1035844981.
- Gelman A, Rubin DB (1992). “Inference from Iterative Simulation Using Multiple Sequences.” *Statistical Science*, **7**(4). doi:10.1214/ss/1177011136.
- Gentleman R, Whalen E, Huber W, Falcon S (2022). **graph: A Package to Handle Graph Data Structures**. R package version 1.76.0, URL <http://www.bioconductor.org/packages/release/bioc/html/graph.html>.
- Gosling J, Joy B, Steele G, Bracha G (2000). *The Java Language Specification*. Addison-Wesley Professional.
- Goudie RJB, Mukherjee S (2016). “A Gibbs Sampler for Learning DAGs.” *Journal of Machine Learning Research*, **17**(30). URL <https://www.jmlr.org/papers/volume17/14-486/14-486.pdf>.
- Hansen KD, Gentry J, Long L, Gentleman R, Falcon S, Hahne F, Sarkar D (2019). **Rgraphviz: Provides Plotting Capabilities for R Graph Objects**. URL <https://www.bioconductor.org/packages/release/bioc/html/Rgraphviz.html>.
- Kalisch M, Bühlmann P (2007). “Estimating High-Dimensional Directed Acyclic Graphs with the PC-Algorithm.” *Journal of Machine Learning Research*, **8**, 613–636. URL <https://www.jmlr.org/papers/v8/kalisch07a.html>.

- Kalisch M, Mächler M, Colombo D, Maathuis M, Bühlmann P (2012). “Causal Inference Using Graphical Models with the R Package **pcalg**.” *Journal of Statistical Software*, **47**(11), 1–26. doi:10.18637/jss.v047.i11.
- Kratzer G, Furrer R (2019). **mcmcabn**: A Structural MCMC Sampler for DAGs Learned from Observed Systemic Datasets. R package version 3.18.17, URL <https://CRAN.R-project.org/package=mcmcabn>.
- Kuipers J, Moffa G (2016). “Partition MCMC for Inference on Acyclic Digraphs.” *Journal of the American Statistical Association*, **1**(517), 1–15. doi:10.1080/01621459.2015.1133426.
- Kuipers J, Moffa G, Kuipers E, Freeman D, Bebbington P (2018a). “Links between Psychotic and Neurotic Symptoms in the General Population: An Analysis of Longitudinal British National Survey Data Using Directed Acyclic Graphs.” *Psychological Medicine*, **49**(3), 388–395. doi:10.1017/s0033291718000879.
- Kuipers J, Suter P, Moffa G (2022). “Efficient Sampling and Structure Learning of Bayesian Networks.” *Journal of Computational and Graphical Statistics*, **31**(3), 639–650. doi:10.1080/10618600.2021.2020127.
- Kuipers J, Thurnherr T, Moffa G, Suter P, Behr J, Goosen R, Christofori G, Beerenwinkel N (2018b). “Mutational Interactions Define Novel Cancer Subgroups.” *Nature Communications*, **9**(1). doi:10.1038/s41467-018-06867-x.
- Link WA, Eaton MJ (2011). “On Thinning of Chains in MCMC.” *Methods in Ecology and Evolution*, **3**(1), 112–115. doi:10.1111/j.2041-210x.2011.00131.x.
- Plummer M, Best N, Cowles K, Vines K (2006). “**coda**: Convergence Diagnosis and Output Analysis for MCMC.” *R News*, **6**(1), 7–11. URL <http://CRAN.R-project.org/doc/Rnews/>.
- Ramsey JD, Zhang K, Glymour M, Romero RS, Huang B, Ebert-Uphoff I (2018). “**TETRAD** – A Toolbox for Causal Discovery.” In *8th International Workshop on Climate Informatics*. Boulder.
- R Core Team (2022). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL <https://www.R-project.org/>.
- Scanagatta M, de Campos CP, Corani G (2015). “Learning Bayesian Networks with Thousands of Variables.” *NIPS: Proceedings of the 28th International Conference on Neural Information Processing Systems*. URL <https://papers.nips.cc/paper/5803-learning-bayesian-networks-with-thousands-of-variables.pdf>.
- Scutari M (2010). “Learning Bayesian Networks with the **bnlearn** R Package.” *Journal of Statistical Software*, **35**(3), 1–22. doi:10.18637/jss.v035.i03.
- Scutari M, Graafland CE, Gutiérrez JM (2019). “Who Learns Better Bayesian Network Structures: Accuracy and Speed of Structure Learning Algorithms.” *International Journal of Approximate Reasoning*, **115**, 235–253. doi:10.1016/j.ijar.2019.10.003.

- Spirtes P, Glymour C, Scheines R (2000). *Causation, Prediction, and Search*. 2nd edition. MIT Press. doi:10.7551/mitpress/1754.001.0001.
- Szklarczyk D, Gable AL, Lyon D, Junge A, Wyder S, Huerta-Cepas J, Simonovic M, Doncheva NT, Morris JH, Bork P, Jensen LJ, von Mering C (2019). “STRING V11: Protein-Protein Association Networks with Increased Coverage, Supporting Functional Discovery in Genome-Wide Experimental Datasets.” *Nucleic Acids Research*, **47**(D1), D607–D613. doi:10.1093/nar/gky1131.
- The MathWorks, Inc (2011). *MATLAB – The Language of Technical Computing, Version R2011b*. The MathWorks, Inc., Natick. URL <http://www.mathworks.com/products/matlab/>.
- Tsamardinos I, Brown LE, Aliferis CF (2006). “The Max-Min Hill-Climbing Bayesian Network Structure Learning Algorithm.” *Machine Learning*, **65**(1), 31–78. doi:10.1007/s10994-006-6889-7.

**Affiliation:**

Polina Suter, Jack Kuipers, Niko Beerenwinkel  
Department of Biosystems Science and Engineering  
ETH Zürich  
Mattenstrasse 26  
4058 Basel, Switzerland  
E-mail(s): [polina.suter@gmail.com](mailto:polina.suter@gmail.com), [jack.kuipers@bsse.ethz.ch](mailto:jack.kuipers@bsse.ethz.ch),  
[niko.beerenwinkel@bsse.ethz.ch](mailto:niko.beerenwinkel@bsse.ethz.ch)

and

SIB Swiss Institute of Bioinformatics  
4058 Basel, Switzerland

Giusi Moffa  
Department of Mathematics and Computer Science  
University of Basel  
Spiegelgasse 1  
4051 Basel, Switzerland  
E-mail: [giusi.moffa@unibas.ch](mailto:giusi.moffa@unibas.ch)